Universität Ulm

Sektion Angewandte
Informationsverarbeitung

Zur Erweiterung von Programmiersprachen durch Bibliotheken

Konzeption und Realisierung der Ulmer Oberon-Bibliothek

Dissertation zur Erlangung des Doktorgrades
Dr. rer. nat.
der Fakultät für Mathematik und Wirtschaftswissenschaften
der Universität Ulm

vorgelegt von
Andreas Franz Borchert
aus
Nümbrecht

Ulm, im Mai 1994

Amtierender Dekan:	Prof. Dr. Joachim Voeller
1. Gutachter:	Prof. Dr. Franz Schweiggert
2. Gutachter:	Prof. Dr. Heiko Vogler
Tag der Promotion:	24. Juni 1994

Meinen Eltern

Die Software-Industrie steht vor dem Dilemma,
entweder ihre bisherigen Investitionen in der Form
von Programmen und Systemen beizubehalten
oder von den modernen Paradigmen und Methoden
der Software-Entwicklung zu profitieren.
[Wirth94]

Vorwort

Einige gängige Werke über objekt-orientierte Techniken stellen auf ihren ersten Seiten Komplexität als das herausragende Problem heraus:

- “[...] complexity is an essential property of all large software systems.” [Booch91]
- “Software applications are complex because they model the complexity of the real world.” [Wirfs-Brock90]

Die Komplexität — oder, um genauer zu sein, die Kompliziertheit — ergibt sich jedoch häufig nicht nur aus dem zugrundeliegenden Problem, sondern auch aus den zu verwendenden Werkzeugen. Wenn die Entwicklungsmethoden, CASE-Tools, Programmiersprachen oder die wiederzuverwendende Software bereits sehr kompliziert sind, dann wird es schwieriger, zu einer qualitativ hochwertigen Problemlösung in vertretbarer Zeit zu gelangen.

Dies läßt sich besonders deutlich im Bereich der Programmiersprachen sehen. Auch wenn die Kodierphase in vielen Software-Entwicklungsprozessen zu einer sehr kurzen, fast unbedeutenden Phase geworden ist, kommt dem Programmtext eine entscheidende Bedeutung zu. Zwar wird er nach langer Vorplanung relativ rasch niedergeschrieben, jedoch muß er später unzählige Male von Menschen gelesen und verstanden werden, sei es zur Fehlerbehebung oder zur Weiterentwicklung des Systems. Parallel existierende Dokumente und Hilfsmittel können diesen Prozeß erleichtern, aber nicht vermeiden.

Die Verständlichkeit eines Programmtexts hängt nicht nur vom Programmierstil ab, sondern auch von der verwendeten Programmiersprache und ihren Schutzmechanismen sowohl gegen Flüchtigkeitsfehler als auch bezüglich versteckter Abhängigkeiten zu anderen Teilen des Software-Systems. Je umfangreicher eine Programmiersprache ist, umso schwieriger wird generell das Lesen und Verstehen von Texten in dieser Sprache.

Trotz dieser Erkenntnis gibt es einen Trend zur ständigen Erweiterung verbreiteter Programmiersprachen. Während zum Beispiel das in [Kernighan78] veröffentliche C-Referenz-Manual mit 42 Seiten auskommt, benötigt C++ 413 Seiten (siehe [Ellis90]) mit steigender Tendenz, wie Concurrent C++, O++ und E (C++ mit Persistenz) belegen.

Diese Arbeit zeigt am Beispiel von Oberon und der Ulmer Oberon-Bibliothek, daß es auch bei einer Programmiersprache mit ungewöhnlich geringem Umfang möglich ist, viele der bei größeren Software-Systemen notwendigen Mechanismen und Strukturen in eine zugehörige Bibliothek auszulagern, ohne daß dabei ein Verlust an Ausdruckskraft oder Eleganz hingenommen werden muß. Auf diese Weise läßt sich die Komplexität reduzieren, da lokal nur die Sprachmittel eine Rolle spielen, die benötigt und daher importiert werden.

Das erste Kapitel bewertet kritisch eine Reihe von Techniken und Sprachmitteln in Programmiersprachen, die für die Produktion größerer Software-Systeme entwickelt worden sind. In Abkehr von der klassischen objekt-orientierten Sichtweise und dem traditionellen Modulverständnis wird dabei eine Technik eingeführt, bei der Objekte zu Bezugspunkten werden und Module entweder Abstraktionen oder Implementierungen sind, die bestimmte Teilsichten auf ihnen bekannte Objekte haben.

Oberon weicht sehr von der Linie der klassischen objekt-orientierten Sprachen ab. Im zweiten Kapitel ist daher eine Einführung in Oberon enthalten, die eine Reihe von Techniken an konkreten Beispielen zeigt.

Das dritte Kapitel liefert eine kurze Übersicht der Bibliothek und ihrer Konventionen, die das Lesen der weiteren Kapitel erleichtert. Mit dabei ist ein Überblick über die kooperativen Techniken der Bibliothek, die in den folgenden Kapiteln weiter verfeinert werden.

Ereignisse gehören zu den wichtigsten und grundlegendsten Abstraktionen der Bibliothek und werden im vierten Kapitel vorgestellt.

Ausnahmenbehandlungen sind Bestandteil einer Vielzahl von Programmiersprachen. Im fünften Kapitel wird gezeigt, wie auf der Basis von Ereignissen Ausnahmenbehandlungen auch in Oberon elegant durchgeführt werden können.

Eine erste größere Anwendung der im einführenden Kapitel entwickelten Prinzipien ist im sechsten Kapitel über Ein- und Ausgabe enthalten. Zwar sind auch die Abstraktionen für Ein- und Ausgabe in anderen Programmiersprachen in die Bibliothek verlagert worden — trotzdem wurde durch eine ungewöhnliche Aufteilung mehr Einfachheit und Effizienz gewonnen.

Zu den Schwerpunkten dieser Arbeit gehören die Abstraktionen für Parallelisierungen, die im siebten Kapitel behandelt werden. Hierzu wurde ein neues Sprachmittel zur Synchronisierung entwickelt, dessen Semantik auf der Basis von CSP erläutert wird. Anschließend wird gezeigt, wie sich aufgrund dieser Synchronisierungstechnik die weiteren Abstraktionen im Umfeld von Parallelisierungen aufteilen und vereinfachen lassen. Die neue Technik wird dann anhand einiger Beispiele diskutiert und mit anderen Programmiersprachen verglichen.

Im achten Kapitel wird Persistenz auf der Basis der Abstraktionen für Ein- und Ausgabe eingeführt. Damit wird nicht nur Persistenz im Sinne von Langlebigkeit, sondern auch von Mobilität möglich. Entsprechend wird gezeigt, wie sich aufgrund dieser Techniken verteilte Systeme und damit die Forderungen aus dem ersten Kapitel in vollem Umfang erfüllen lassen.

Danksagungen

Das Ulmer Oberon-System und diese Arbeit entstanden in den Jahren 1989 bis 1994 in der Sektion Angewandte Informationsverarbeitung an der Universität Ulm. Sie wäre undenkbar gewesen ohne den ungewöhnlich umfangreichen Einsatz von Herrn Prof. Dr. Franz Schweiggert. Mich erstaunte immer wieder sein unerschöpfliches Talent, neue Stellen und Mittel zu beschaffen.

Der Anfang meiner Arbeit wurde erst möglich durch seine guten Kontakte zum HDI Haftpflichtverband der Deutschen Industrie V.a.G., der nicht nur meine Stelle für anderthalb Jahre finanzierte, sondern auch noch die Kosten des Rechners übernahm, auf dem die Arbeit beginnen konnte. Mein besonderer Dank gebührt hier Herrn Dr. Manfred Mücke, der zu dieser Zeit Vorstandsmitglied beim HDI war und diese ungewöhnliche Partnerschaft ermöglicht hat.

Weiter möchte ich mich bei Herrn Prof. Dr. Franz Schweiggert für die vielen anregenden Gespräche bedanken. Unter meinen Kollegen darf ich besonders Herrn Dipl.Math. Werner Stanglow hervorheben, der sich zeitweise mit der Ulmer Oberon-Bibliothek beschäftigte, und dem ich daher zahlreiche Verbesserungsvorschläge zu verdanken habe.

Von Anfang an ließen sich einige Studenten auf das beginnende Experiment des Ulmer Oberon-Systems ein und erduldeten zahllose inkompatible Änderungen und Umbauten der Ulmer Oberon-Bibliothek. Besonders hervorzuheben ist hier Herr Martin Hasch, der sich 1989 nicht nur spontan an der Entwicklung des Übersetzers beteiligte, sondern auch später in vielen Diskussionen zur ständigen Verbesserung der Bibliothek beigetragen hat. Weiter zu danken sind hier Herr Dipl.Math.oec. Andreas Haupt, der einen ersten Debugger entwickelte, Herr Dipl.Math.oec. Detlef Birkholz, der den Grundstein für die persistenten Objekte legte, Herr Till Franke, der zur Beschleunigung der Interprozeßkommunikation beitrug, und Herr Hansjörg Nägele, dem die automatische Speicherverwaltung zu verdanken ist.

Weiter gilt mein Dank der Firma Sun Microsystems GmbH, die das Ulmer Oberon-Projekt 1993 mit einem Arbeitsplatzrechner sponserte.

Abschließend gilt mein Dank Herrn Reinhard Schön und Herrn Martin Hasch, die diese Arbeit korrekturgelesen haben.

Inhaltsverzeichnis

The tools we use have a profound (and devious!) influence
on our thinking habits, and, therefore, on our thinking abilities.
[Dijkstra82]

1. Zielsetzungen und Techniken

1.1. Oberon — In Zürich nichts Neues?

1988 erschien der erste Artikel über Oberon [Wirth88a], der zunächst in der Literatur über objekt-orientierte Sprachen kaum Widerhall fand. Einerseits fehlte völlig die Erwähnung von ‘‘Objekten’’, zum anderen schien trotz des vielversprechenden Zitats von Albert Einstein

> ‘‘Make it as simple as possible, but not simpler.’’

eine Sprachbeschreibung von weniger als zwanzig Seiten, die gegenüber dem Vorgänger Modula-2 eine signifikante Reduzierung darstellt, völlig unzureichend zu sein. Statt der gewohnten Terminologie in der Literatur über objekt-orientierte Sprachen fand man als einzige Erweiterung zu Modula-2 nur den Begriff der Typerweiterung, der bereits früher vorgestellt worden ist [Wirth87, Wirth88b].

Entsprechend fielen auch einige der Kommentare aus. So schreibt Sakkinen in seinem kritischen Artikel über C++ [Sakkinen92]:

> ‘‘I suspect that some recent entrants in the camp of statically-typed object-oriented languages are already too Spartan to be convenient for general-purpose programming. I am in doubt about Modula-3, but rather sure about Oberon [...]’’

Ähnliche Gründe werden angeführt von Mössenböck und Templ [Mössenböck89] bei ihrem Artikel über Object Oberon — einem Nachfolger von Oberon und Vorläufer von Oberon-2 [Mössenböck93]:

> ‘‘However, Oberon itself is intentionally not a fully object-oriented language, i.e., it does not allow the declaration of classes with methods that can be overridden in subclasses (known as dynamic binding).’’

Dabei wird teilweise übersehen, daß die Programmiersprache Oberon in Zusammenhang mit einem gleichnamigen Projekt einer integrierten Benutzeroberfläche entstanden ist, die in ihrer Kombination von Einfachheit und Erweiterbarkeit auf Basis objekt-orientierter Techniken kaum Konkurrenten findet [Wirth88c, Wirth89a, Wirth89b, Wirth92]. Eine ähnliche Entwicklung mit der Tendenz zur Vereinfachung gab es auch auf dem Wege von Simula zu BETA [Madsen93].

Während im Bereich von Prozessorarchitekturen der Übergang von CISC (*complex instruction set computer*) zu RISC (*reduced instruction set computer*) erfolgreich gelang, scheint eine ähnliche Entwicklung im Bereich von Programmiersprachen undenkbar. Im Bereich der Prozessoren könnte man möglicherweise noch argumentieren, daß die Komplexität sich von dem Prozessor zum Compiler verlagerte, jedoch beweisen auch einige neuere Architekturen im Bereich der Betriebssysteme (z.B. Mach), daß Vereinfachungen keinen Verlust an Ausdruckskraft nach sich ziehen müssen.

Man kann eine Programmiersprache als Kern betrachten, der ausreicht, um weitere Sprachmittel in Form von Bibliotheken hinzuzufügen. Dieses Konzept ist nicht neu, insbesondere Smalltalk [Goldberg89] hat bewiesen, daß man praktisch alle Sprachmittel in die Bibliothek verlagern kann. Allerdings macht dieser Ansatz eine Reihe spezieller Henkel zu nicht in Smalltalk geschriebenen Teilen notwendig. Dies erfordert entweder einen Interpreter oder eine entsprechende Laufzeitbibliothek.

Im Gegensatz zu Smalltalk ist Oberon zunächst eine Programmiersprache, die der Tradition von Modula-2 folgt. Alle Sprachmittel lassen sich einfach und effizient auf die gängigen Prozessoren umsetzen und erfordern keine Laufzeitunterstützung.

1.2. Produktivität und Qualität

Ein fairer Vergleich zwischen Oberon und anderen Programmiersprachen macht es notwendig, die Techniken zur Produktion von Software und die damit verbundenen Sprachmittel zu betrachten. Letztendlich ist eine Programmiersprache ein Werkzeug, und alle Sprachmittel müssen an den Anforderungen der Software-Produktion gemessen werden:

Produktivität
: Es sollte möglich sein, mit einem verhältnismäßig geringen Aufwand an Zeit und Personal ein Software-System zu erstellen. Wichtig sind hier das Teile-Und-Herrsche-Prinzip, das eine weitgehend unabhängige

Entwicklung von Teilen ermöglicht, und die Wiederverwendbarkeit vorhandener Software.

Qualität
: Es sollte eine möglichst hohe Sicherheit dafür gegeben sein, daß das erstellte Software-System auch der Spezifikation entspricht. Zumindest nach deutschem Recht unterliegt Software der Produkthaftung [Koch92], und nur durch eine Qualitätssicherung kann eine Haftungsentlastung erreicht werden. Sicherheiten, die sich durch statische und dynamische Überprüfungsmöglichkeiten bei einer Programmiersprache ergeben, können dabei eine wichtige Teilfunktion übernehmen.

Sind beide Punkte ausreichend gewährleistet, dann reduziert sich auch das Problem der Wartbarkeit. Einerseits ist wegen der Sicherheit der Übereinstimmung mit der Spezifikation dafür gesorgt, daß mit wenigen Ausfällen und entsprechenden Korrekturen zu rechnen ist. Zum anderen können Änderungs- und Erweiterungswünsche leichter berücksichtigt werden, wenn Software grundsätzlich unter dem Aspekt der größtmöglichen Unabhängigkeit und Wiederverwendbarkeit hergestellt wird.

1.3. Modulkonzepte

Module als Bezeichnung für getrennt übersetzbare Teile eines Programms gibt es seit den frühen Anfängen der Programmierung. Man erwartet von einer Modularisierung

- daß die Entwicklungszeit verkürzt wird, da getrennte Gruppen jeweils einzelne Module bearbeiten und der Kommunikationsaufwand gering ist;
- Freiheit in der Implementierung eines Moduls: Es sollte möglich sein, größere Änderungen durchzuführen, ohne daß andere Module davon betroffen werden.
- Übersichtlichkeit: Um ein Modul zu verstehen, sollte es ausreichen, den zugehörigen Programmtext zu lesen und die verwendeten Schnittstellen zu kennen.

Wie Parnas darlegt, reicht es nicht, ein System nur in eine Reihe von zusammenhängenden Funktionen zu zerlegen, da dann Änderungswünsche sehr viele Module auf einmal betreffen können [Parnas72]. Statt dessen ist es notwendig, das Prinzip des ''*information hiding*'' anzuwenden. Das führt dazu, daß alle Teile, die sich ändern könnten, konsequent in separate Module untergebracht werden. Die Schnittstellen stellen dann nur noch eine Abstraktion zur Verfügung, ohne damit die interne Repräsentierung offen zu legen. Auf diese Weise läßt sich das Prinzip der Stetigkeit realisieren: Geringe Änderungen in den Anforderungen sollten nur zu geringen Änderungen in den Implementierungen führen.

Verbunden mit dem Begriff der Abstraktionen sind die abstrakten Datentypen. Ein abstrakter Datentyp ist nur durch die zur Verfügung stehenden Operationen und eine zugehörige Spezifikation charakterisiert. Häufig sind jedoch mehrere abstrakte Datentypen so eng miteinander verbunden, daß sie zu einer Abstraktion gehören. Von daher ist ein Modulkonzept allgemeiner, das mehrere abstrakte Datentypen zusammenfassen kann, als das häufig vorgefundene Klassenkonzept, das Module und Typen unifiziert, wie es z.B. bei CLU [Liskov77] und den meisten objekt-orientierten Sprachen der Fall ist.

Einige Autoren, wie z.B. Wegner, betrachten Sprachen, die auf dem Modulkonzept basieren (z.B. Modula-2 und Ada), nicht als klassenbasiert [Wegner90, S. 26]:

> ''But Ada is not class-based because its objects (packages) do not have a type and cannot therefore be passed as parameters, [...]''

Zwar vergißt Wegner nicht zu erwähnen, daß es auch private Datentypen in Ada gibt. Jedoch sieht er sie nicht als gleichberechtigt im Vergleich zu CLU, das Klassen als Typen betrachtet, obwohl es nicht mehrere parallele Implementierungen im gleichen Programm zuläßt. Wesentlich ist die mögliche Trennung von Abstraktion und Implementierungen, und dies ist völlig unabhängig davon, ob Module Typen sind oder nicht.

1.3.1. Schnittstellen für Module

Um die Sicherheit zu erreichen, daß Module auch nachher gemäß der Schnittstellen zusammenpassen, und um die Unabhängigkeit zwischen Schnittstelle und Implementierung zu erhöhen, ist es notwendig, die Schnittstelle von den möglichen Implementierungen zu trennen. Abbildung 1 zeigt, wie auf diese Weise trotz der getrennten Übersetzung eine statische Überprüfung der Schnittstellen möglich ist. Dies steht im Gegensatz zur *unabhängigen* Übersetzung, wie sie z.B. bei Fortran realisiert ist, die die Schnittstellensicherheit der Verantwortung des Programmierers überläßt.

Es gibt drei gängige Techniken für die Repräsentierung von Schnittstellen. Die erste Technik (siehe Abbildung 2) kopiert die Schnittstelle textuell in die Implementierung und die Klienten. Dies geschieht mit Hilfe eines

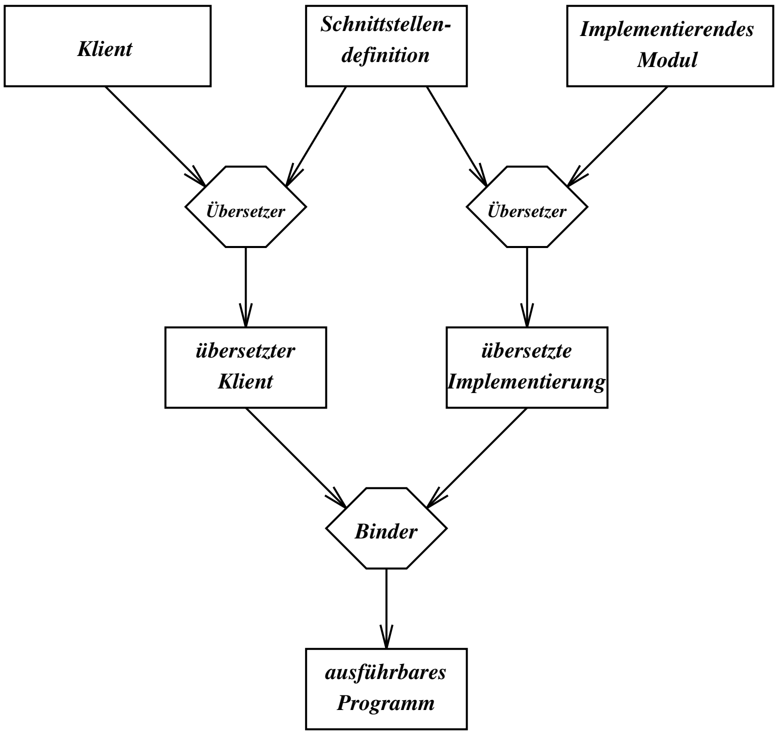

Abb. 1: Getrennte Übersetzung mit statischer Überprüfung

Makro-Präprozessors und wird in erster Linie von C [Kernighan78] und C++ [Stroustrup86] realisiert. Diese Technik setzt die Disziplin des Programmierers voraus, da die entsprechenden *include*-Anweisungen auch vergessen werden können, ohne daß es deshalb notwendigerweise zu Fehlermeldungen des Compilers kommt. In der Praxis sind die notwendigen Richtlinien kaum einzuhalten oder zu überprüfen [Moylan93]. Eine weitere Fehlerquelle ergibt sich daraus, daß notwendige Neuübersetzungen nach Schnittstellenänderungen nur bei Verwendung entsprechender Werkzeuge garantiert erfolgen (z.B. *make* und *makedepend*).

Sichere Techniken lassen sich nur auf der Basis von Schnittstellendefinitionen erreichen, die vom Übersetzer generiert werden. Im folgenden werden generierte Schnittstellen als Symboldateien bezeichnet, um sie von den Schnittstellen zu unterscheiden, die in einer Programmiersprache gefaßt sind. Für die Generierung von Symboldateien gibt es zwei gängige Sichtweisen und damit verbundene Techniken:

Vertrags-Modell:
: Eine Schnittstelle stellt ein eigenständiges Dokument dar, das einen Kontrakt darstellt zwischen möglichen Klienten und Implementierungen. Analog zu einem notariellen Vertrag kann man sich vorstellen, daß dieser Kontrakt zu Beginn der Design-Phase entwickelt und dann eingefroren wird. Spätere Änderungen sind dann nur mit Einverständnis aller betroffenen Parteien möglich oder können zumindest nicht unbemerkt vorgenommen werden.

Sicht-Modell:
: Jedes Modul ermöglicht verschiedene Sichten. Eine Sicht eröffnet den vollständigen Quelltext, andere Sichten erlauben nur den Einblick auf eine Reihe von Definitionen. Da alle Sichten aus dem vollständigen Quelltext generiert werden, muß bei jeder Änderung festgestellt werden, ob sich irgendwelche Sichten ebenfalls geändert haben.

Während das Vertrags-Modell zur frühzeitigen Festlegung von Schnittstellen führt, entsteht beim Sicht-Modell die Schnittstelle erst mit der Implementierung. Somit entstehen Änderungen an der Schnittstelle bei dem Sicht-Modell implizit und sind kaum von Änderungen an der Implementierung zu unterscheiden.

Die klassischen Vertreter des Vertrags-Modells sind Modula-2 [Wirth82] und Ada [ANSI83], während beispielsweise Eiffel [Meyer92] das Sicht-Modell verwendet. Dies erfolgt mit Hilfe der zugehörigen Kommandos **short** und **flat**, die einen Auszug der exportierten Definitionen liefern (**flat** expandiert dabei die Ausgabe um die übergeordneten Klassen). Da diese Werkzeuge auch die Kommentare der exportierten Deklarationen liefern, ist es damit möglich, eine komplette Dokumentation aus dem Quelltext zu extrahieren.

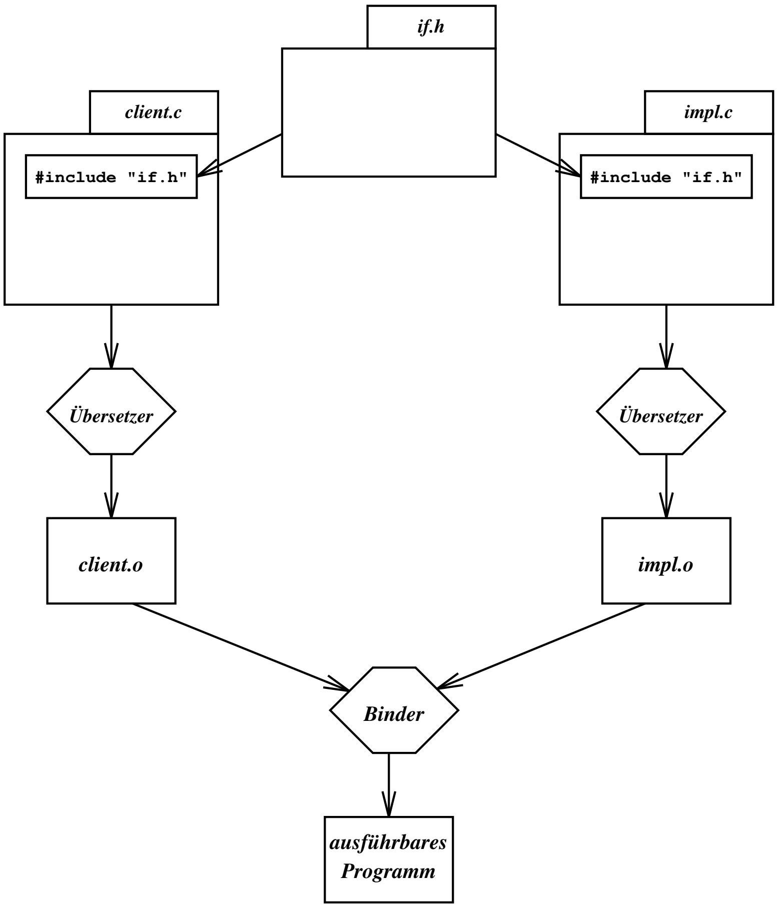

Abb. 2: Getrennte Übersetzung bei C und C++

Meyer betont dabei ausdrücklich [Meyer88, §9.5.1],

> “[...] that the purpose of information hiding is abstraction, not protection. We do not necessarily wish to prevent client programmers from accessing secret class elements, but rather to *relieve* them from having to do so.”

Hierbei wird jedoch übersehen, daß das Lesen von fremden Implementierungen zu Annahmen verleitet, die möglicherweise bei einer späteren Version nicht mehr zutreffen. Außerdem ist es gerade bei kommerziell verkauften Bibliotheken durchaus üblich, nur die Schnittstellen und die übersetzten Module auszuliefern — nicht jedoch die Quellen der zugehörigen Implementierungen.

Eines der Hauptprobleme bei der festen Verknüpfung von Schnittstelle und Implementierung ist die schlechte Trennbarkeit — schließlich sind mehrere Implementierungen einer Abstraktion denkbar. Einige objektorientierte Sprachen lösen dieses Problem über das Sprachmittel einer *abstract class* (oder *deferred class*), die effektiv einer reinen Schnittstellenbeschreibung entspricht. Andere Sprachen, wie z.B. Emerald [Raj91], führen den Begriff der Konformität ein: eine Abstraktion A ist im wesentlichen dann konform zu B, wenn sie eine Obermenge der Operationen von B zur Verfügung stellt.

Andererseits erhöht das Vertrags-Modell die Sicherheit, daß die Schnittstelle nicht unbeabsichtigt oder unbemerkt verändert wird. Sie entsteht zeitlich vor den möglichen Implementierungen und erlaubt damit die Übersetzbarkeit von Klienten ohne die Existenz von Implementierungen. Sie hat ihren eigenen Zeitstempel und geht entsprechend auch in die Versionsverwaltung ein. Der Zeitstempel einer Symboldatei kann erheblich jünger sein, wenn darin zusätzliche Informationen abgelegt werden, wie z.B. die Größe privater Datenstrukturen.

Bei dem Vertrags-Modell kann man allenfalls den vernachlässigbaren Nachteil finden, daß es notwendig ist, die Deklarationen aus der Schnittstelle in jeder der Implementierungen zu duplizieren. Hinzu kommt der zusätzliche Aufwand beim Übersetzer für die Überprüfung der Implementierung in Bezug auf die Schnittstelle.

Es ist bemerkenswert, daß Oberon zunächst dem Vertrags-Modell folgte [Wirth88a], aber seit der ersten Revision das Sicht-Modell unterstützt wird [Wirth89c]. Wirth [Wirth89d] begründet dies analog zu oben:

> "It appeared as desirable to have a module's specification contained in a single document, both from the view of the programmer and the compiler."

Da aus unserer Sicht die Vorteile einer getrennten Schnittstelle überwiegen, unterstützt unser Übersetzer bislang nur das Vertrags-Modell, da sich das Vertrags-Modell als wesentliche Hilfe bei der Entwicklung der Ulmer Oberon-Bibliothek erwiesen hat. Allerdings wäre eine Unterstützung beider Modelle denkbar.

Analog zu Eiffel bietet das Züricher Oberon-System ein Kommando namens *Browser* [Reiser92], das alle exportierten Deklarationen in Form eines Auszugs liefert, der in der Syntax der ursprünglichen Schnittstellenbeschreibung entspricht. Allerdings fallen dabei (zumindest in der vorliegenden Version) alle Kommentare weg und der Auszug ist nicht notwendigerweise konsistent. Ein Beispiel dafür ist in Abbildung 3 gegeben. Alle Namen, die mit einem Stern gekennzeichnet sind, sind nach außen sichtbar. So sind in dem Beispiel *Counter* und *Incr* sichtbar, jedoch nicht *Increment*. Die zugehörige Ausgabe des Browsers findet sich in Abbildung 4. Diese Sicht ist unzureichend, da zwar *Incr* exportiert wird, nicht jedoch alle zugehörigen Parametertypen. Beim Vertrags-Modell würde eine solche Schnittstelle vom Übersetzer abgelehnt werden.

```
MODULE Counters;

   TYPE
      Counter* = INTEGER; (* exportiert *)
      Increment = INTEGER; (* nicht exportiert *)

   PROCEDURE Incr*(VAR count: Counter; increment: Increment);
   BEGIN
      INC(count, increment);
   END Incr;

END Counters.
```

Abb. 3: Beispiel für inkonsistente Exportierungen

```
DEFINITION Counters;

   TYPE
      Counter = INTEGER;

   PROCEDURE Incr(VAR count: Counter; increment: Increment);

END Counters.
```

Abb. 4: Inkonsistente Schnittstelle

Wenn man abschließend den Aufwand für einen *Browser*, der eine konsistente Schnittstelle mit zugehöriger Kommentierung liefert, mit dem zusätzlichen Aufwand beim Übersetzer zur Unterstützung des Vertrags-Modells vergleicht, dann relativiert sich diese vermeintliche Vereinfachung rasch.

1.3.2. Erweiterungen des Vertrags- und des Sicht-Modells

Beide Modelle lassen Verallgemeinerungen zu, die nicht die Bijektivität zwischen Schnittstelle und Implementierung annehmen. Beim Vertrags-Modell ist es denkbar, daß sowohl mehrere Implementierungen an einen Vertrag gebunden sind, als auch, daß eine Implementierung mehrere Verträge unterstützt. Letzteres wird beispielsweise von Modula-3 [Nelson91] realisiert. Bei ersterem handelt es sich um Polymorphismus, der üblicherweise durch objekt-orientierte Techniken (*late binding*) unterstützt wird.

Das Sicht-Modell kann dahingehend erweitert werden, daß mehrere eingeschränkte Sichten zur Verfügung stehen. Weitere Sichten werden typischerweise nur für benannte Klienten zur Verfügung gestellt, wie z.B. in C++ über die *friends*-Deklaration oder in Eiffel durch restriktive Export-Anweisungen. Diese zusätzlichen Sichten werden insbesondere notwendig, wenn Module den Typen gleichgesetzt werden und deswegen nicht mehrere abstrakte Datentypen zu einer Abstraktion zusammengefaßt werden können.

1.3.3. Die Importbeziehung als Ordnungsrelation

Auf der Basis von Schnittstellen lassen sich Importbeziehungen zwischen Übersetzungseinheiten (Schnittstelle oder Implementierung) präzise definieren. Fehlen sie (wie beispielsweise in C oder C++) so bleiben sie anonym, da die exportierende Übersetzungseinheit beim Übersetzen der importierenden Übersetzungseinheit unbekannt bleibt. Ist jeder Schnittstelle (in Quelltextform) genau eine Implementierung zugeordnet und umgekehrt (oder liegt überhaupt keine Trennung vor), so kann man bei Importbeziehungen das Paar Schnittstelle und Implementierung auch als zusammengehörig betrachten.

Auf der Basis von Importbeziehungen lassen sich die irreflexiven Ordnungsrelationen $\rightarrow$ und $\rightarrow^+$ zwischen Modulen definieren, wobei $\rightarrow^+$ strikt und partiell ist, wenn auf Zyklen verzichtet wird:

$$A \rightarrow B, \textit{ wenn } B \textit{ das Modul } A \textit{ importiert}$$

$$A \rightarrow^+ B, \textit{ wenn } A \rightarrow B \textit{ oder } \exists\, M_1, \ldots, M_n \textit{ (mit } n > 0\textit{), so daß } A \rightarrow M_1 \rightarrow \cdots \rightarrow M_n \rightarrow B$$

In den folgenden Diagrammen, die Modulhierarchien darstellen, beziehen sich die Pfeile immer auf die Relation $\rightarrow$. Wenn zwischen zwei Modulen A und B die Relation $A \rightarrow^+ B$ definiert ist, wird A unterhalb von B angeordnet.

1.4. Wiederverwendbarkeit und Erweiterbarkeit

Im weiteren Sinne kann man unter Wiederverwendung von Software alle Vorgänge zusammenfassen, die aufgrund früherer Software-Projekte zur Reduzierung des Aufwands der Software-Entwicklung und der Wartung bei einem neuen Software-System führen. Hierzu kann man erworbenes Fachwissen, Entwicklungserfahrung, Design-Entscheidungen, Programmtext und Dokumentation zählen [Biggerstaff89]. In engeren Kontext von Programmiersprachen sind insbesondere Techniken zur Wiederverwendung interessant, die durch Sprachkonstrukte, Bibliotheken und geeignete Konventionen realisiert werden.

Im industriellen Sinne kann man versuchen, eine Parallele zwischen Hardware- und Software-Herstellung zu ziehen. Wenn es selbstverständlich ist, daß jedes neue technische Gerät fast nur aus Standardteilen besteht, die aus Katalogen bestellt werden, die ihre Spezifikationen enthalten, warum ließe sich nicht eine ähnliche Technik in der Software-Industrie verfolgen? Diese Frage wurde zuerst von McIlroy auf der berühmten NATO-Konferenz 1968 in Garmisch gestellt und bleibt bis heute nur teilweise beantwortet [McIlroy76].

Folgt man Krueger, so bilden Abstraktionen die gemeinsame Grundlage für die verschiedenen Techniken der Wiederverwendung von Software [Krueger92]. Um Abstraktionen zu bewerten, führt Krueger den Begriff der *kognitiven Distanz* ein. Die kognitive Distanz definiert sich intuitiv als der intellektuelle Aufwand, der von Software-Entwicklern aufgewendet werden muß, um auf Basis eines vorhandenen Software-Systems ein neues System zu entwickeln. Um die kognitive Distanz zu reduzieren, ist es notwendig, schlüssige und ausdrucksstarke Abstraktionen zu entwickeln, die zugleich minimal sind.

Wenn die kognitive Distanz eines Software-Elements so niedrig ist, daß der Aufwand bei der Wiederverwendung geringer ist als der Aufwand, es völlig neu zu entwickeln, so kann man von Wiederverwendbarkeit sprechen.

Da Module prinzipiell ein Werkzeug zur Trennung von Abstraktionen und Implementierungen sind, wären sie im Sinne von McIlroy die geeigneten Einheiten für die Wiederverwendung, obwohl typischerweise Module nicht ganz isoliert zu betrachten sind, sondern in Einbettung einer Bibliothek. Dies führt zu einem sehr engen Begriff der Wiederverwendung, da "eingekaufte" Standardteile nur "eingebaut" und nicht verändert werden. Und dies ist das Problem: Module, die im Rahmen eines früheren Software-Projekts entwickelt worden sind, lassen sich in vielen Fällen nicht ohne Änderungen in neue Systeme integrieren.

Erweiterbarkeit läßt sich als die Möglichkeit definieren, ein bestehendes Modul an neue Anforderungen anpassen zu können, ohne es zu verändern. Der Begriff der Wiederverwendung schließt hingegen auch die Änderung vorhandener Module ein, nachdem sie möglicherweise dupliziert worden sind. Erweiterbarkeit setzt Wiederverwendbarkeit voraus und stellt zusätzliche Anforderungen, die das Zusammenleben von alten unveränderten Modulen mit neuen Erweiterungen ermöglichen.

Da einzelne Bibliotheken häufig nicht isoliert leben, sondern entweder mit anderen koexistieren müssen (z.B. dadurch, daß sie zu einem Programm zusammengebunden werden) oder mit anderen Software-Systemen im Verbund gepflegt werden (unzählige Varianten eines Moduls wären ein Problem), ist Erweiterbarkeit eine zentrale Voraussetzung für Produktivität. Traditionelle Ansätze, die eine Abstraktion bijektiv einer Implementierung zuordnen, sind dafür unzureichend, wie sich an folgendem Beispiel ersehen läßt.

Angenommen, ein Statistik-Paket wurde entwickelt mit der Vorgabe, daß resultierende Tabellen in formatierter Form auf einem Terminal ausgegeben werden können. Die traditionelle Technik würde Abbildung 5 entsprechen. *InOut* repräsentiert die Abstraktion für einfache Ein- und Ausgabe im Kontakt mit dem Benutzer in Verbindung mit einigen Operationen für die formatierte Ausgabe von Zahlenwerten. *InOut* basiert auf *Terminal*, das die Abstraktion eines alphanumerischen Terminals offeriert.

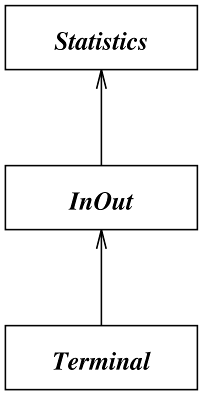

Abb. 5: Traditionelles Modul-Layout

Folgende Änderungswünsche wären denkbar:

- Zusätzlich zu der Ausgabe auf dem Terminal sollte auch die Möglichkeit bestehen, die Tabellen in einer Datei abzulegen.
- Das alphanumerische Terminal soll durch ein bestimmtes Fenster innerhalb eines Fenstersystems auf einem Graphik-Bildschirm ersetzt werden.

Es ist offensichtlich, daß die Abstraktion, die das Statistik-Paket bezüglich der Ausgabe verwendet, mehrere Implementierungen benötigt, die parallel im gleichen Programm zur Verfügung stehen. Bei einer einzigen Implementierung ergäbe sich die Notwendigkeit, sie um jede weitere Anforderung entsprechend aufzublähen. Abbildung 6 zeigt das konkrete Beispiel. Statt *InOut* wird eine Abstraktion *Streams* verwendet, für die es die drei parallelen Implementierungen *Terminal*, *Files* und *Windows* gibt. Abbildung 7 zeigt die zugehörige Verallgemeinerung.

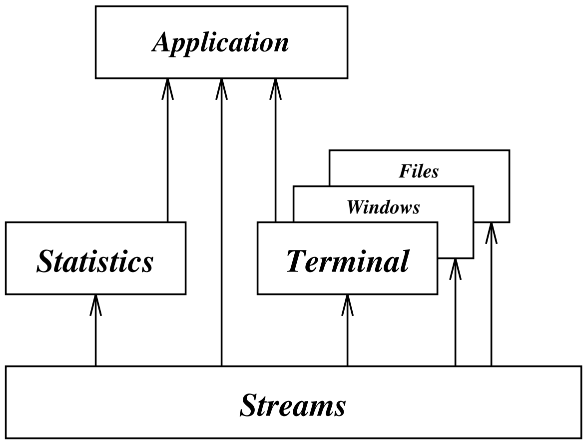

Abb. 6: Abstraktion *Streams* mit drei Implementierungen

1.5. Erweiterbarkeit von Abstraktionen

Es ist sehr wichtig, daß die Abstraktionen selbst auch erweiterbar sind. Eine Erweiterung ist dabei als Sonderfall oder Spezialisierung zu betrachten. Wenn die Abstraktion *B* eine Erweiterung von *A* ist, so sollte eine Implementierung von *B* auch der Abstraktion von *A* genügen. Entsprechend sollte es möglich sein, einem Modul, das die Abstraktion *A* kennt und verwendet, auch alternativ eine Implementierung, die der Abstraktion *B* folgt, übergeben zu können.

Dies ist deshalb so wichtig, da durch die Trennung zwischen Abstraktionen und Implementierungen Prozeduraufrufe entgegen der Importhierarchie erfolgen können (*up calls*). Auf diese Weise kommt es zu

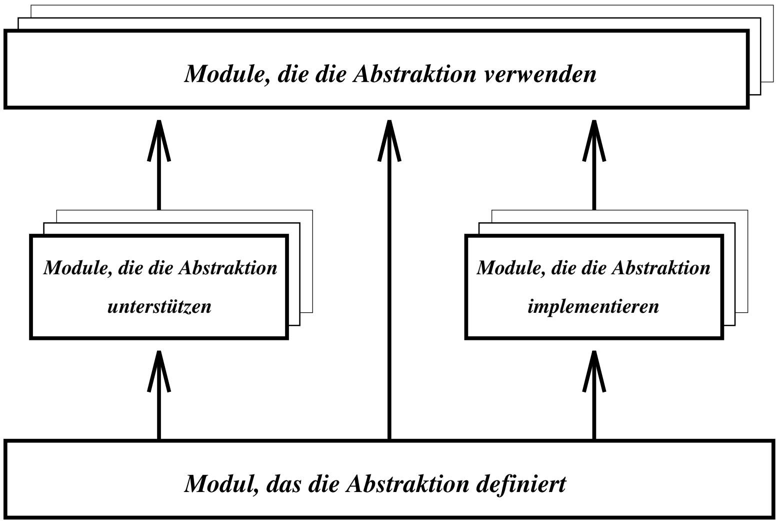

Abb. 7: Modulhierarchie bei Abstraktionen mit mehreren Implementierungen

Situationen, in denen ein aufrufendes Modul von der Abstraktion *A* ausgeht, während das aufgerufene Modul eine Implementierung von *B* ist.

Als Beispiel dafür läßt sich die Abstraktion *Viewers* aus [Reiser91] entnehmen. Von *Viewers* sind einige Erweiterungen denkbar, wie z.B. *TextViewers*, *GraphicViewers* und *PictureViewers*, wofür jeweils verschiedene Implementierungen denkbar sind. Die Module jedoch, die daran beteiligt sind, das Bewegen und Benutzen der Maus in entsprechende Ereignisse zu verwandeln, die an einen *Viewer* weiterzugeben sind, sollten von denkbaren Erweiterungen der *Viewer* unbehelligt bleiben.

1.6. Objekt-orientierte Techniken

Wenngleich es keine genaue Definition für "objekt-orientiert" gibt und dieser Begriff sogar gerne mißbraucht wird [King89], kann man die ersten objekt-orientierten Programmiersprachen Simula [Birtwistle73] und Smalltalk [Goldberg89] als Ausgangspunkt nehmen.

Sowohl Simula als auch Smalltalk erlauben die Definition von Klassen, die einem abstrakten Datentyp nahekommen. Neben der Deklaration von Variablen (*instance variables*) finden sich auch zugehörige Operationen (*instance methods*), die auf diese Variablen zugreifen dürfen. Klassen sind statische Typbeschreibungen, während Objekte zur Laufzeit entstehende Instanzen sind.

Klassen können als Erweiterung einer (*single inheritance*) oder mehrerer (*multiple inheritance*) Klassen definiert werden. In diesem Fall werden alle von der übergeordneten Klasse definierten Variablen und Operationen übernommen. Es besteht dabei die Möglichkeit, Operationen überzudefinieren und ggf. innerhalb einer überdefinierten Operation auf die Operationen der Basisklasse zurückzugreifen.

Wie Snyder beschreibt, sind Klassen als Sprachmittel nicht notwendigerweise mit dem wünschenswerten Zugriffsschutz ausgestattet [Snyder86]. So gibt es beispielsweise in Simula67 und CLOS [Moon89] überhaupt keine Mechanismen, die den Zugriff auf Variablenkomponenten und interne Operationen ausschließen. Moon begründet dies folgendermaßen:

> "The Lisp culture tends to oppose protection and enforcement, so CLOS does not contain a real protection mechanism. Instead it relies on informal conventions [...]"

Dabei wird übersehen, daß Zugriffsschutz nicht zur Gängelung von Programmierern dient, sondern vielmehr hilft, Abhängigkeiten zu verringern und versehentliche Zugriffe auf private Komponenten zu verhindern.

Beim Klassenkonzept gibt es mindestens zwei Sorten von Klienten: importierende Klienten und erweiternde Klienten. Obwohl z.B. Smalltalk und Eiffel zwar einen Schutz gegenüber importierenden Klienten anbieten, so fehlt er völlig gegenüber Erweiterungen. Diese Freiheit führt zu Problemen, da Änderungen an einer Basisklasse zu unvorhersehbaren Problemen bei den Erweiterungen führen können. Aus der Sicht der Software-Wartung ist dies ein Alptraum.

Die beliebten Überdefinitionen bei Methoden können in der Praxis die Lesbarkeit und damit die Wartbarkeit erheblich erschweren [Wilde92]. Der Trend, selbst wenige Anweisungen einer Operation über mehrere

Methoden zu verteilen, führt zu unglaublich kleinen Methoden, wie die Tabelle 1 von Wilde und Huitt zeigt, die um die beiden Ulmer Bibliotheken erweitert wurde.[1]

Bibliothek	Sprache	Anzahl der Methoden	Median der Größe der Methoden
Bellcore: Interactive network design aid	C++	1280	1 ausführbare Anweisung
Bellcore: Prototype of noninteractive task planning system	Smalltalk	477	2 unkommentierte Zeilen
Smalltalk/V: Sample from the environment's class library	Smalltalk	2224	3 unkommentierte Zeilen
Ulmer Oberon-Bibliothek	Oberon	1288	6 unkommentierte Zeilen
zum Vergleich: Ulmer Modula-Bibliothek	Modula-2	807	5 unkommentierte Zeilen

Tab. 1: Größe von Methoden in objekt-orientierten Sprachen

Die Möglichkeit, Methoden überzudefinieren, ohne dabei auf die entsprechende Variante der übergeordneten Klasse im Sinne der Wiederverwendung zu verzichten, führt zur Konkurrenz von einander teilweise ausschließenden Zielen. Auf der einen Seite wäre es angemessen, eine Hierarchie (oder bei *multiple inheritance* einen gerichteten antizyklischen Graphen) von Abstraktionen zu haben, und andererseits Implementierungen, die selbst wieder entweder andere Implementierungen verwenden oder auch auf Abstraktionen basieren.

Wie LaLonde, Thomas und Pugh an dem Beispiel der Klassenhierarchie von Smalltalk zeigen, wurde hier zugunsten der günstigeren Wiederverwendung die **is-a**-Beziehung der zugehörigen Abstraktionen verletzt, als *Dictionary* als Erweiterung von *Set* definiert worden ist [LaLonde86]. Sie schlagen deswegen die Unterstützung zwei verschiedener Sorten von Hierarchien vor: eine für die logische Sicht der Abstraktionen und die andere für die physische Sicht. Sie weisen nach, daß alle Bemühungen, mit einer Hierarchie auszukommen, Defizite aufweisen und führen deswegen ein weiteres Sprachmittel in Smalltalk ein (*exemplar inheritance*). Die Möglichkeit, auf Basis vorhandener Sprachmittel eine zweite Hierarchie aufzubauen, bleibt allerdings unberücksichtigt. Eine entsprechende Lösung gibt es in einer Klassenbibliothek von C++ [Kofler93].

Ein ähnliches Beispiel, in dem Abstraktionen und Implementierungsaspekte unauflöslich miteinander verbunden sind, liefert Meyer in den Auszügen seiner Eiffel-Bibliothek [Meyer88]. Abbildung 8 zeigt die Klassenhierarchie der Eiffel-Bibliothek, wobei die Pfeile für Erweiterungen stehen. Es gibt dabei zwei Fälle von kombinierten Erweiterungen (*multiple inheritance*): *FIXED_LIST* ist eine Kombination aus *ARRAY* und *LIST*, während *TREE* auf *LINKABLE* und *LINKED_LIST* aufbaut und somit Mehrwegbäume unterstützt.

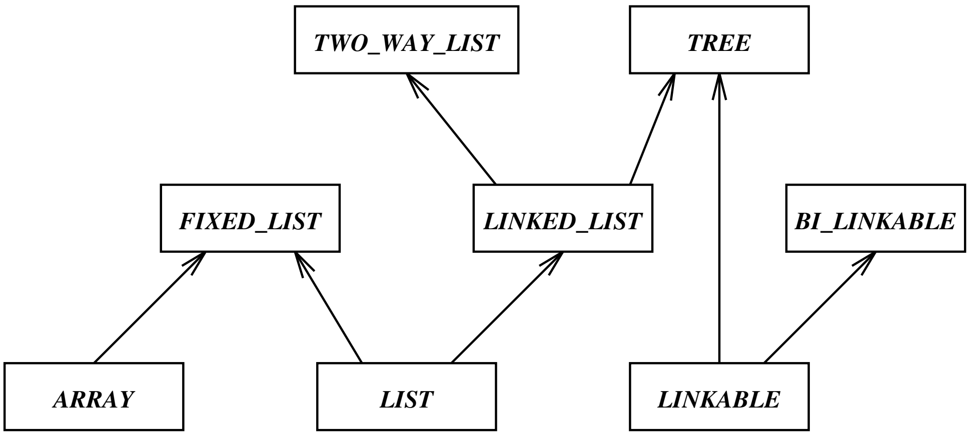

Abb. 8: Klassenhierarchie der Eiffel-Bibliothek

[1] Als Größe der Methode wurde der reine Anweisungsteil in Betracht gezogen. Während bei C++ die Anweisungen selbst gezählt worden sind, wurden bei den anderen Sprachen die Zeilen gezählt, die Anweisungen enthielten. Bei Oberon und Modula-2 wurden lokale Prozeduren der umgebenden Prozedur zugerechnet.

Der Aufbau orientiert sich an den zugehörigen Implementierungen, wobei die Möglichkeit von kombinierten Erweiterungen hauptsächlich ausgenutzt wird, um vorhandenen Programmtext wiederzuverwenden. Schließlich dürfte es Klienten normalerweise kaum interessieren, ob es sich um einen Mehrwegbaum handelt, wenn sie eine geordnete Kollektion benötigen. Abbildung 9 zeigt eine Alternative, die teilweise an Smalltalk erinnert. Eine allgemeine Abstraktion *COLLECTION*, die das Einbringen, Durchlaufen und Entfernen von Daten unterstützt, wird um zwei Sonderfälle erweitert. *ORDERED_COLLECTION* steht für alle Kollektionen, in denen die Elemente einer Reihenfolge unterliegen: so ist es beispielsweise möglich, ein Element an einer bestimmten Stelle einzufügen. Denkbare Implementierungen sind *ARRAY*, *LINKED_LIST* und *TWO_WAY_LIST*. *SortedCollection* geht davon aus, daß eine totale Ordnung über den Elementtyp vorliegt, und ordnet alle eingefügten Elemente entsprechend an. Hier wäre *TREE* eine mögliche Implementierung.

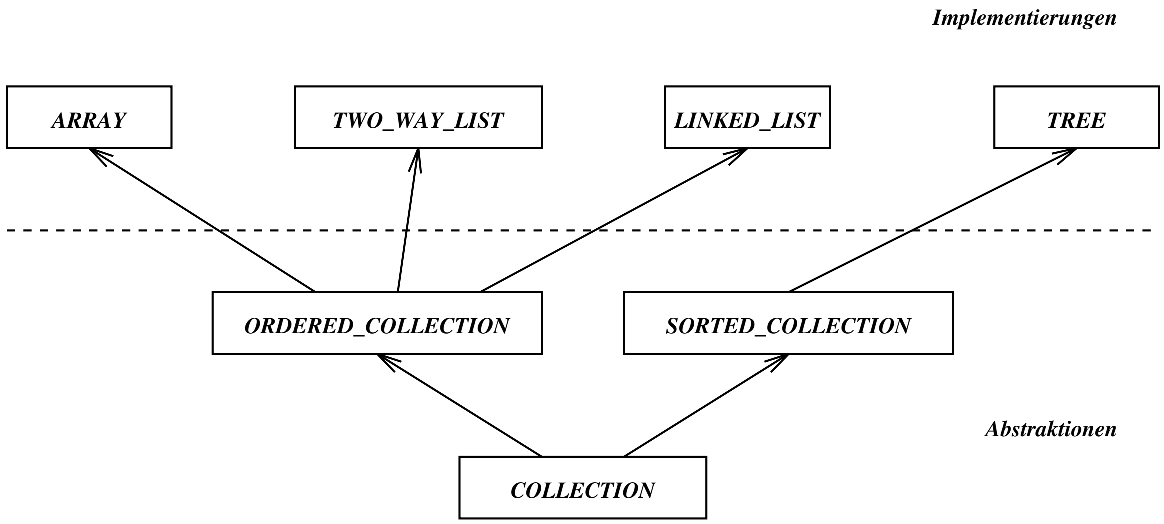

Abb. 9: Alternative mit zwei Hierarchien

Auf der anderen Seite können *TREE* und *TWO_WAY_LIST* durch normale Importbeziehungen von der Implementierung von *LINKED_LIST* Gebrauch machen. Diese Alternative ist nicht nur viel klarer, sie kommt auch überraschenderweise völlig ohne kombinierte Erweiterungen aus.

Wirfs-Brock und Wilkerson bezeichnen diese Entwurfstechnik der ersten Variante als "*data-driven design*", da sie sich unmittelbar an der Struktur der zugrunde liegenden Daten richtet und die zugehörigen Operationen entsprechend modelliert [Wirfs-Brock89]. Auf diese Weise gelangen mehr Informationen und Abhängigkeiten nach außen als notwendig. Alternativ propagieren sie eine Entwurfstechnik, die danach fragt, wofür Objekte verantwortlich sind und welche Informationen sie tragen sollen. Im Beispiel der Eiffel-Bibliothek orientierte sich *TREE* an der Struktur eines Mehrwegbaumes, während *SORTED_COLLECTION* dafür verantwortlich ist, eine Sammlung von Objekten sortiert zu verwalten.

Allerdings gibt es auch bei den klassischen objekt-orientierten Techniken die Möglichkeit, Abstraktionen und Implementierungen strenger zu trennen. Viele objekt-orientierte Sprachen (z.B. C++ [Stroustrup91] und Eiffel) unterstützen die Deklaration einer abstrakten Klasse, die selbst noch nicht instanziiert werden kann. Erst wenn davon Erweiterungen definiert werden, die alle offengelassenen Implementierungen realisieren (konkrete Klassen), dürfen davon Instanzen kreiert werden.

Inzwischen ist die Eiffel-Bibliothek revidiert worden und wurde um eine zugehörige Hierarchie von Abstraktionen ergänzt [Meyer90, Meyer94].

In einigen Fällen kann es allerdings sehr nützlich sein, eine abstrakte Klasse mit einigen Operationen anzureichern, die auf Basis der von Implementierungen beizusteuernden Operationen formuliert werden. Auch ist es hilfreich, für bestimme Operationen eine möglicherweise ineffiziente Implementierung mitzuliefern, die im konkreten Fall durch eine verbesserte Version ersetzt werden kann. In beiden Fällen findet keine inkrementelle Überdefinition statt, die die eigentliche Methode über mehrere Hierarchien verteilt. Deutsch zeigt [Deutsch89], daß auf der Basis dieser Techniken ein Gerüst gebaut werden kann, das als Grundlage für die Wiederverwendung eines Designs dienen kann. Eines seiner Beispiele ist eine Benutzerschnittstelle, bei der die Klasse *View* die Mechanismen für die Berechnung des sichtbaren Bereiches, die Konvertierung der Koordinaten und das Neuzeichnen bei Veränderungen unterstützt, jedoch die Operation offenläßt, die den eigentlichen Inhalt anzeigen soll. Ein weiteres Beispiel liefert die Ulmer Oberon-Bibliothek mit dem Modul *Streams*, das, wenn die jeweilige Implementierung es wünscht, einen Cache benutzt, um die Anzahl der Operationen, die an die Implementierung weitergegeben werden, zu reduzieren.

1.7. Kombinierte Erweiterungen

Viele der objekt-orientierten Sprachen unterstützen *multiple inheritance* (z.B. Eiffel) oder haben sie zumindest nachträglich eingeführt (z.B. C++). Wenn man die zugehörigen Beispiele betrachtet, findet man jedoch relativ häufig Fälle, in denen dieses Sprachmittel eher mißbraucht wird.

So werden, wie vorher gezeigt, im Zuge der Wiederverwendung alle benötigten Teile auf diese Weise aufgenommen. Meyer ermutigt auch dazu, Module, die sich nicht um einen abstrakten Datentyp ranken, wie z.B. mathematische Funktionen oder Sammlungen von Konstanten, auf diese Weise zu importieren [Meyer88, §13.2].

Ein anderes Beispiel (siehe Abbildung 10) für *multiple inheritance* gibt Shopiro [Shopiro89]. Hier wird *multiple inheritance* zur verbesserten statischen Typüberprüfung eingesetzt. Die virtuelle Klasse *ios* enthält alle gemeinsamen Teile und kann selbst nicht instanziiert werden. Je nachdem, ob man nur einen Eingabekanal, Ausgabekanal oder einen Kanal zur Ein- und Ausgabe benötigt, kann man den entsprechenden Typ angeben. Dies funktioniert, da alle Eingabe-Operationen bei *istream* residieren und alle Ausgabe-Operationen bei *ostream*. Die Klasse *iostream* ist nur eine Verknüpfung von *istream* und *ostream* und bietet ansonsten keine weiteren Operationen.

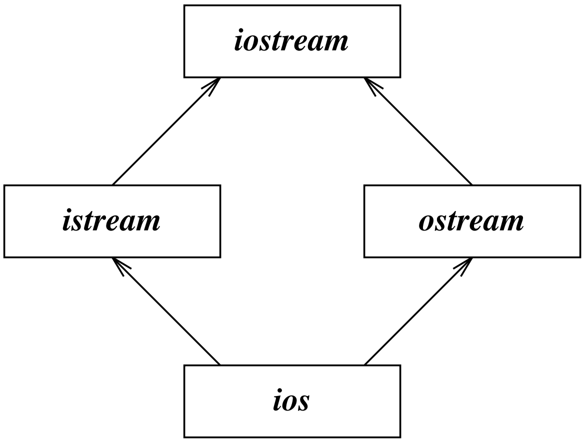

Abb. 10: *iostream* Klassenstruktur in C++

In diesem Fall wurde *multiple inheritance* verwendet, um unterschiedliche und zueinander orthogonale Fähigkeiten voneinander zu trennen und ihre Verwendung statisch überprüfbar zu machen. Zwei Fähigkeiten (im Beispiel Ein- und Ausgabe) sind natürlich nur ein Spezialfall — in der Praxis gibt es typischerweise viel mehr. Weitere denkbare Fähigkeiten im obigen Beispiel wäre die Möglichkeit der Positionierung und der nicht blockierenden Ein- und Ausgabe. Da alle diese Varianten zueinander orthogonal sind, möchte man natürlich auf keine der denkbaren Kombinationen verzichten. Dies führt dazu, daß bei n Varianten 2^n Klassen statisch definiert werden müssen.

Völlig unübersichtlich wird es, wenn Erweiterungen definiert werden, die einige Varianten unterstützen und ggf. die zugehörigen Methoden überdefinieren, jedoch die Auswahl der Varianten nicht festlegen. Dies führt dazu, daß entsprechend viele Erweiterungen definiert werden müssen, da jede unterstützte Kombination berücksichtigt werden muß.

Ein gutes Beispiel dafür sind die *Dribble-Streams*, die Lieberman in seinem Artikel über Delegation angibt [Lieberman86]. Ein *Dribble-Stream* soll die Ein- und Ausgabe eines gegebenen Kanals in eine separate Datei protokollieren und ansonsten alle Operationen weitergeben an die eigentliche Implementierung. Ein *Dribble-Stream* macht sowohl Sinn für reine Ein- oder Ausgabekanäle als auch für bidirektionale Kanäle. Selbst bei der vereinfachten Fassung, die Shopiro veröffentlich hat, müßten nicht weniger als vier neue Klassen vereinbart werden mit mindestens drei Konstruktoren.

Außerdem steht die Auswahl der Varianten nicht immer statisch fest. Ein Beispiel dafür wäre eine Operation bei einem Editor, die eine Datei eröffnet. Es ist sinnvoll, wenn ein Editor nicht nur Dateien mit Lese- und

Schreibberechtigung unterstützt, sondern auch solche, die sich nur zum Lesen öffnen lassen. Welchen Typ soll eine solche Operation zurückliefern? *istream* wäre zu wenig, *iostream* würde in einigen Fällen zuviel versprechen. Die pragmatische Lösung in C++ besteht darin, einen *iostream* zurückzuliefern und sich den tatsächlichen Stand separat zu merken. Das Beispiel zeigt, daß man sich auch bei einem *iostream* nicht darauf verlassen kann, daß Schreibzugriffe legal sind.

Eine einfachere Lösung besteht darin, daß man eine Klasse für alle Varianten definiert, die zur Laufzeit überprüft, ob die gewünschte Operation implementiert ist. Wie sich zumindest in der Ulmer Oberon-Bibliothek erwiesen hat, ist der zusätzliche zeitliche Aufwand vernachlässigbar. Zusätzlich kann die Möglichkeit offeriert werden, die Menge der unterstützten Fähigkeiten abzufragen.

Im Beispiel der *Dribble-Streams* ist dies ideal, da die zugehörige Implementierung auf diese Weise die Fähigkeiten propagieren kann: Wenn der zugrunde liegende Kanal Schreib-Operationen nicht unterstützt, dann kann der *Dribble-Stream* dies ebenfalls untersagen. Auf diese Weise wird verhindert, daß eine versehentlich akzeptierte Ausgabe protokolliert wird.

Ein sinnvolleres Beispiel für *multiple inheritance* liefert Schaffert in seiner Einführung zu Trellis/Owl [Schaffert86], das man als kombinierte Erweiterung betrachten kann (siehe Abbildung 11). Auf Basis der abstrakten Datentypen *Window* und *IO_Stream* läßt sich *TextWindow* definieren. Wie jedoch Templ zeigt, kann man auch auf Basis von einfacher Erweiterbarkeit beide Seiten unterstützen, wenn man statt einem Objekt zwei erzeugt: ein *TextWindow*, das nur ein Fenster repräsentiert und ein *TextWindow_Stream*, das die Ein- und Ausgabe ermöglicht [Templ93]. Dabei gehören beide Objekte unauflöslich zusammen: das *TextWindow*-Objekt verweist auf das *TextWindow_Stream*-Objekt und umgekehrt. Wenn eine Prozedur beispielsweise einen Parameter vom Typ *TextWindow* erhält und etwas darauf ausgeben möchte, kann es jederzeit auf den zugehörigen *IO_Stream* zugreifen.

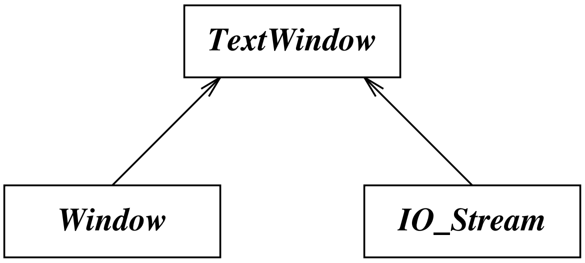

Abb. 11: *TextWindow* als kombinierte Erweiterung von *Window* und *IO_Stream*

Man kann kombinierte Erweiterungen auf unterschiedliche Weise mit *information hiding* verbinden. Bei Eiffel muß jeweils eine Klasse den Überblick über sämtliche Basisklassen inklusive aller privaten Teile wahren. Somit fällt jedes Objekt in die Zuständigkeit von genau einer Klasse. Wenn jedoch der Zugriffschutz nicht nur für Klienten, sondern auch für Erweiterungen gilt, dann haben wir eine verteilte Verantwortung für ein Objekt, die genauso über Kontrakte geregelt ist wie die normalen Beziehungen. In Erweiterung dieses Modells läßt es sich auch vorstellen, daß zur Laufzeit Module private Informationen zu Objekten unterhalten, die nicht von ihnen stammen. Dann dienen Objekte in erster Linie als Bezugspunkte, zu denen jedes Modul seine private Statusinformation unterhalten kann. Dieses Modell wird von der Ulmer Oberon-Bibliothek unterstützt durch das Modul *Disciplines*.

Ein Beispiel für zur Laufzeit entstehende private Zusatzinformationen sind die Parametrisierungen für Ein- und Ausgabe-Operationen. Bei C++ wurden diese Parameter und die zugehörigen Operationen alle in die *iostream*-Klassen aufgenommen. Damit ist zwar das Problem für die gängigen Operationen gelöst, jedoch nicht für denkbare Erweiterungen. Hier wäre man gezwungen, weitere Klassen zu definieren und dann im Wege von *multiple inheritance* eine konkrete Implementierung und die zusätzlichen Operationen zusammen mit ihren Parametern zu verknüpfen. Abgesehen von der Komplexität dieses Ansatzes wegen der vier einzelnen Klassen wird die Möglichkeit verwehrt, die Erweiterungen auf bereits geöffneten Kanälen anzuwenden.

1.8. Unabhängigkeit und Dezentralisierung

Madsen und Møller-Pedersen geben eine ungewöhnlich kurze und nicht technisch orientierte Definition für objekt-orientierte Programmierung [Madsen88]:

> “Object-oriented programming. A program execution is regarded as a *physical model*, simulating the behavior of either a real or imaginary part of the world.”

Das Ziel dieses Prinzips ist Stetigkeit. Wenn Änderungen in der realen Welt zu neuen oder veränderten Anforderungen führen, sollten die notwendigen Anpassungen den gleichen Umfang besitzen. Dies kann nur erreicht werden, wenn man sich am Modell der realen Welt orientiert und Unabhängigkeiten, die dort existieren, auch im Software-System bestehen.

Um diesem Prinzip gerecht zu werden, sollte man sich von der Vorstellung verabschieden, daß ein Objekt notwendigerweise vollständig in den Sichtbarkeitsbereich eines oder gar mehrerer Module fällt. Um die reale Welt als Vorbild zu nehmen, könnte man als Beispiel einen Studenten einer Universität betrachten. Abbildung 12 zeigt, daß die Daten, die zu einem Studenten gehören, in drei Sichtbarkeitsbereiche fallen könnten: allgemeine Daten, die zum größten Teil einem Basistyp für natürliche Personen gehören könnten, Daten, die aus der Sicht des Studentensekretariats von Bedeutung sind, und der Bereich des Prüfungsamtes, das die abgelegten Prüfungen aufzeichnet.

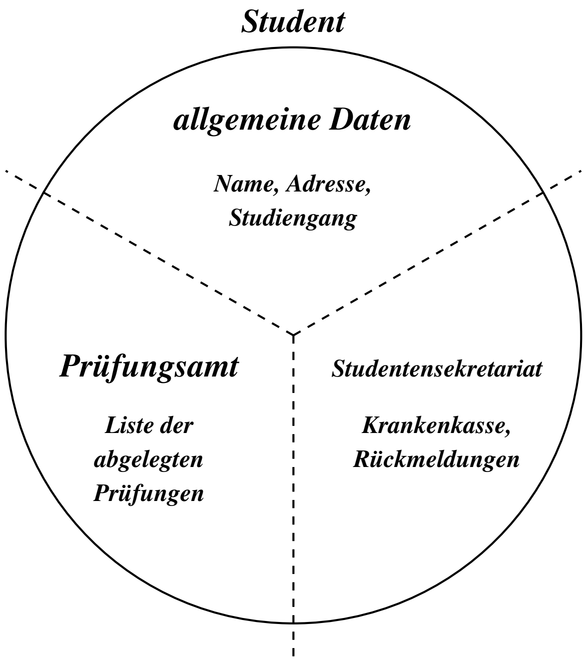

Abb. 12: Unterschiedliche Datenbereiche bei einem Studenten

Alle Datenbereiche im Beispiel beziehen sich zweifellos auf einen Studenten und sollten direkt mit ihm verbunden sein. Traditionelles objekt-orientiertes Design würde zu folgender Klassendefinition führen (die Notation folgt keiner konkreten Sprache und wurde von [Szyperski92a] verwendet):

```
CLASS Student;
   INHERITS Person;

   studiengang: Studiengang;
   pruefungen: PruefungsListe;
   rueckmeldungen: RueckmeldeDaten;
END Student;
```

Dieser Ansatz legt jedoch genau fest, welche Daten zu einem Studenten gehören, und legt sie in einen Sichtbarkeitsbereich, der sich möglicherweise (wie in Eiffel) auf alle abgeleiteten Klassen von *Student* erstreckt. Die Gefahr würde insbesondere dann vergrößert werden, wenn *Student* auch die Datenstrukturen von *PruefungsListe* und *RueckmeldeDaten* offenlegen und/oder festlegen würde.

Entsprechend wird das Gesetz der Stetigkeit verletzt. Wenn beispielsweise die Prüfungsordnung in einer Weise geändert werden würde, die die Datenstruktur von *PruefungsListe* verändert, sollten andere Bereiche (wie z.B. die Sicht des Studentensekretariats) davon nicht betroffen werden.

Wenn die zur Verfügung stehende Sprache die Sichtbarkeit auch gegenüber abgeleiteten Klassen unterstützt (wie z.B. C++), dann könnten die privaten Datenstrukturen zur Prüfungsliste und den Rückmeldedaten in separate Klassen herausgetrennt werden, die dann in Form einer kombinierten Erweiterung wieder zusammengefaßt werden. Abbildung 13 zeigt den entsprechenden Klassenaufbau. Dieser Ansatz wird von Snyder als *submodularity* genannt und gehört für ihn zu den wichtigen Techniken, die *multiple inheritance* rechtfertigen [Cook88].

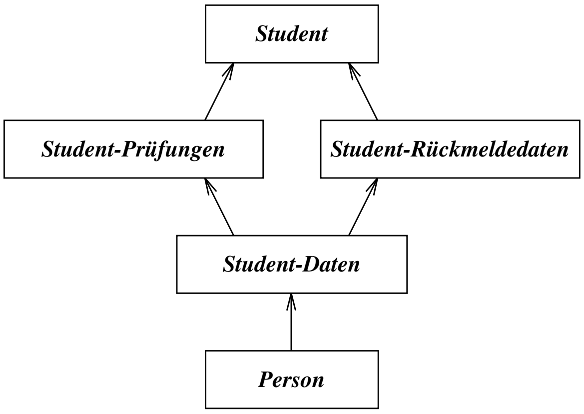

Abb. 13: Student als kombinierte Erweiterung von Klassen mit privaten Datenstrukturen

Während die Privatheit der einzelnen Datenstrukturen möglicherweise sicherer geworden ist, setzt sich *Student* nach wie vor aus einer gegebenen Menge von Komponenten zusammen, die insbesondere auch zum Zeitpunkt der Instanziierung feststehen müssen. Dabei könnten jederzeit weitere Aspekte hinzukommen:

- Studenten können sich am Hochschulsport beteiligen und sich dort für Veranstaltungen anmelden.
- Studenten können als wissenschaftliche Hilfskräfte eingestellt werden und müssen dafür vom Personalreferat erfaßt werden.

Hier bleibt bei traditionellen Ansätzen nur die Möglichkeit, völlig unabhängige Klassen zu definieren, wobei die direkte Verbindung verloren geht und nur analog zu relationalen Datenbanken über Primärschlüssel erreicht werden kann. Dies führt jedoch zu möglichen Inkonsistenzen: wenn ein Student sich exmatrikuliert, wird nicht sichergestellt, daß die anderen Informationen auch verschwinden — dieses Problem ist unabhängig von dem möglichen Einsatz einer automatischen Speicherverwaltung (*garbage collection*).

Eine relativ umständliche Alternative ist die Verwendung von sogenannten *mix-and-match*-Klassen, die statisch zu einem Studenten hinzugebunden werden können. Das Problem ist hier, daß bei n *mix-and-match*-Klassen insgesamt 2^n zusätzliche Klassen statisch definiert werden müssen, die alle möglichen Kombinationen repräsentieren. Eine Änderung der Zusammensetzung, die im obigen Beispiel völlig naturgemäß ist, könnte dann nur durch eine Neu-Kreierung des Studenten erfolgen. Dies dürfte jedoch kaum praktikabel sein.

Hendler hat für dieses Problem ein Sprachmittel vorgeschlagen, das es ermöglicht, sogenannte *enhancements*, die zusätzliche Funktionalitäten repräsentieren, dynamisch zu einem Objekt hinzuzufügen [Hendler86]. Diese Technik wird von der Ulmer Oberon-Bibliothek unterstützt durch das Modul *Disciplines*, das es ermöglicht, weitere Datenstrukturen dynamisch an ein Objekt zu heften und nachher wiederzufinden. Während Hendler nur das Hinzufügen erwähnt, erlaubt *Disciplines* auch das Entfernen dieser Datenstrukturen. Ein weiterer Unterschied besteht darin, wer das Hinzufügen veranlassen kann. In Hendlers Erweiterung von Flavor [Moon86] wird hierfür eine Nachricht an das Objekt geschickt. Damit steht fest, daß einerseits jeder das Zusammenlegen veranlassen kann und daß zum anderen dies unter Kontrolle des Objekts geschieht.

Ein alternativer Ansatz wurde bei den KNOs (*knowledge acquisition, dissemination, and manipulation objects*) verfolgt, die im Rahmen von Büroinformationssystemen entwickelt worden sind: Hier entscheiden die Objekte selbst darüber, welche weiteren Operationen sie aufnehmen [Tsichritzis87]. Aufgenommene Operationen haben dabei vollen Zugriff auf alle zum Objekt gehörenden Daten. Um sich gegen “trojanische Pferde” zu schützen, können neue Operationen zunächst in einer sicheren Umgebung ausprobiert werden. Klassen dienen bei diesem Modell nur als Prototypen — die einzelnen Objekte sind frei, sowohl die neu aufgenommenen als auch die alten Operationen zu entfernen. Dementsprechend gibt es auch keine **is-a**-Beziehungen und die Autoren konstatieren selbst, daß das Verhalten der Objekte nicht immer einfach zu verstehen ist, und regen daher an, zu den Objekten parallel entsprechende graphische Animationen zu entwickeln.

Bei den *Disciplines* erfolgt das Zusammenlegen unter Kontrolle des Moduls, dem die (normalerweise völlig private) Datenstruktur gehört. Damit wird die Unabhängigkeit vergrößert, da das Hinzufügen nur implizit durch exportierte Operationen erfolgen kann. In vielen Fällen bleibt die Tatsache des Hinzufügens völlig verborgen gegenüber der Außenwelt.

Einerseits durch die Trennung von Modulen und Typen und zum anderen durch die Möglichkeit, private Datenstrukturen dynamisch zu Objekten hinzufügen zu können, erhält man ein gegenüber den klassischen Ansätzen von Simula, Smalltalk und Eiffel verallgemeinertes Konzept, das es erlaubt, Sichtbarkeitsgrenzen zu ziehen, die unabhängig von Objekten und Klassen sind. So kann ein Objekt in mehrere private Bereiche eingeteilt werden, und andererseits können Objekte selbst unterschiedlichen Typs in einen Sichtbarkeitsbereich fallen. Dies erhöht die Unabhängigkeit der einzelnen Module voneinander und vermeidet Ersatzmechanismen, wie z.B. *friends* in C++ oder restriktive Exports in Eiffel. Szyperski zeigt dies an einer Reihe von Beispielen [Szyperski92a].

Abbildung 14 zeigt ein vereinfachtes Beispiel aus der Ulmer Oberon-Bibliothek. Im konkreten Beispiel unterhalten sowohl *StreamDisciplines* als auch *RelatedEvents* eigene Bereiche zum durch *s* referenzierten Objekt. *StreamDisciplines* beherbergt die Standardparameter zu komplexeren Ein- und Ausgabe-Operationen und *RelatedEvents* ist zuständig für Ausnahmenbehandlungen (*exception handling*). *Streams* verwaltet optional einen Cache zu einem *Stream*, der natürlich im Sichtbarkeitsbereich von *Streams* liegen sollte. *UnixFiles* ist eine mögliche Implementierung von *Streams*, die private Zusatzinformationen benötigt.

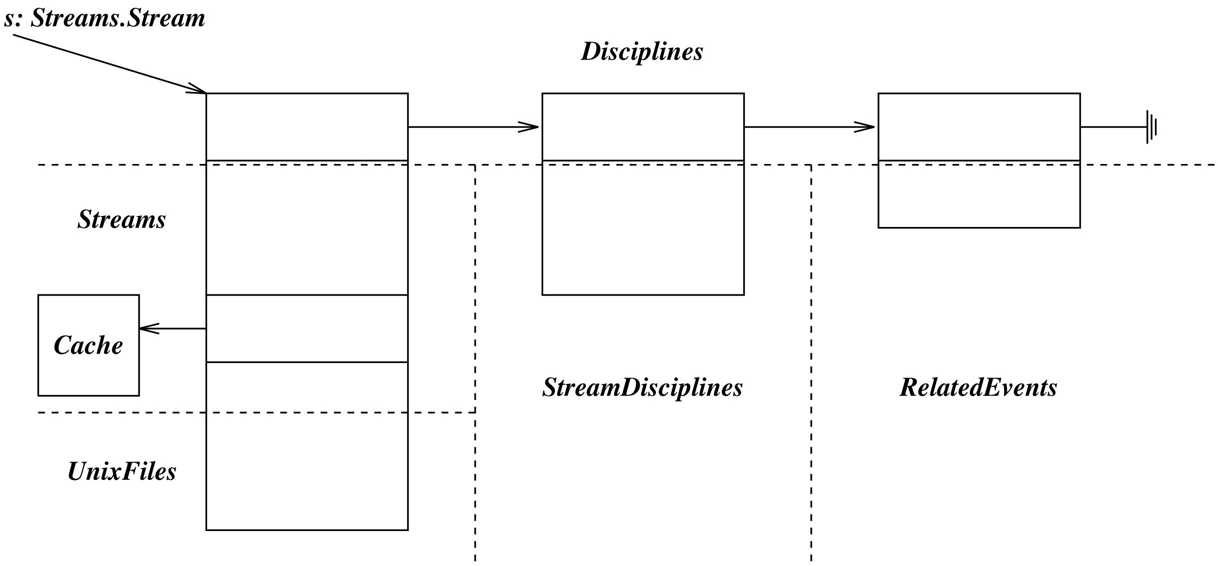

Abb. 14: Sichtbarkeitsbereiche an einem Beispiel der Ulmer Oberon-Bibliothek

Dieses Beispiel wäre in Eiffel nicht einfach umzusetzen. So würden *UnixFiles* alle Interna von *Streams* und *Disciplines* offenliegen. Damit hätte man die Gefahr unerwünschter Abhängigkeiten: Bei Änderungen der privaten Datenstruktur von *Streams* müßten alle Implementierungen von *Streams* erneut überprüft und übersetzt werden. Auf der anderen Seite müßte der Cache außerhalb von *Streams* deklariert werden. Der Zugang von *Streams* zum Cache wäre dann nur über zusätzliche Prozeduren möglich, die der Cache dann exportieren würde. Damit würde der Zugriff auf den Cache unnötig ineffizient werden.

Folgt man Wirth, dann ist das Schlüsselkonzept objekt-orientierter Programmierung die dezentralisierte Kontrolle [Wirth89e]. Unter diesem Gesichtspunkt erweist es sich als bedeutend, daß Module und Klassen als orthogonale Konzepte beibehalten werden, auch wenn man dabei die Linie der klassischen Vertreter der objekt-orientierten Sprachen verläßt.

1.9. Delegationen

1986 verursachte Lieberman auf der OOPSLA-Konferenz mit seinem Beitrag über Prototyp-Objekte einiges Aufsehen, da er durch die Elimination von Klassen einen Zuwachs an Ausdruckskraft gegenüber den klassischen objekt-orientierten Programmiersprachen erhielt [Lieberman86]. Auf der nächsten Konferenz wies Stein [Stein87] nach, daß Delegation auch auf Basis von *inheritance* formuliert werden kann, wenn man Klassen als Prototypen betrachtet, wobei diese Klassen dann nicht notwendigerweise den gewohnten statischen Klassen entsprechen. Die auf den ersten Blick unterschiedlichen Standpunkte wurden von Lieberman, Stein und Ungar zu einem konsensfähigen Modell zusammengefaßt, das als *Treaty of Orlando* in die Literatur einging [Lieberman88, Stein89].

Die Hauptmotivation, Klassen durch Prototypen zu ersetzen, liegt bei Lieberman in der Repräsentierung von Wissen mit Hilfe von objekt-orientierten Systemen. Im entsprechenden experimentellen Umfeld sind Klassen und die zugehörigen Abstraktionen nicht sofort erkennbar, so daß es naheliegt, Objekte auf Basis vorhandener Objekte zu beschreiben.

Der dafür notwendige Mechanismus ist die Delegation. Ein Objekt kann entscheiden, welche Aufträge (*messages*) es selbst bearbeitet und welche es an andere Objekte weiterleitet. Dabei ist das Weiterleiten nicht statisch festgelegt: In Abhängigkeit vom aktuellen Status kann ein Auftrag an unterschiedliche Objekte weitergegeben werden. Insbesondere ist es auch möglich, Aufträge an mehrere Objekte weiterzugeben.

Das Besondere an der Delegation von Lieberman ist das dynamische Zusammenlegen des Prototyps mit dem Objekt, das den Auftrag ursprünglich erhalten hat. So muß das abgeleitete Objekt nicht von allen Daten eine Kopie unterhalten, und sowohl die Methoden des Prototyps als auch des abgeleiteten Objekts können beide Objekte als ein gemeinsames betrachten. Wenn beispielsweise das abgeleitete Objekt eine Operation überdefiniert hat, die intern in einer der Methoden im Prototyp aufgerufen wird, so wird korrekt die geänderte Operation aufgerufen.

Liebermans Delegationen in der reinen Form unterstützen nicht mehr die Modellierung in Form von Abstraktionen, da abgeleitete Objekte keine **is-a**-Beziehung mehr zu ihrem Prototyp unterhalten und durch die gemeinsame Datenhaltung die Abhängigkeit ungewöhnlich groß ist. Allerdings eröffnet das Beispiel des *dribble stream* (abgekürzt *dstream*) die Frage, ob man Delegation in einer abgemilderten Form benötigt, die keine Abstraktionsgrenzen bricht. Eine Lösung wäre Abbildung 15, die voraussetzt, daß unbekannte Operationen weitergegeben werden können. Pfeile stehen in der Abbildung für den Fluß von Aufträgen.

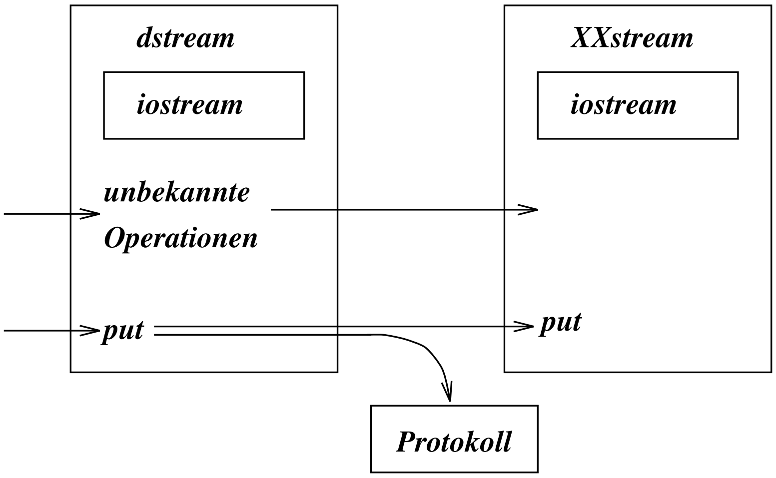

Abb. 15: *dstream*-Lösung mit Delegation bei getrennten Objekten

In diesem Beispiel hat die *Streams*-Klasse analog zu der *iostream*-Bibliothek in C++ nicht nur Operationen für die Ein- und Ausgabe von einzelnen Zeichen, sondern auch für alle Standardtypen. Bei der Implementierung eines *dstream* ergibt sich somit folgendes Dilemma: Da jede andere Erweiterung von *iostream* die Freiheit hat, diese Operationen überzudefinieren, bleibt nichts anderes übrig, als für jede andere Erweiterung von *iostream* eine Spezial-Fassung für *dstream* zu schreiben. Schließlich soll die Ausgabe einer Zahl genauso behandelt werden wie die eines einzelnen Zeichens.

Selbst wenn man alle komplexeren Ein- und Ausgabe-Operationen aus der Basisklasse herausnimmt, bleiben die weiteren Operationen, die sich nicht schlicht auf Basis anderer Operationen definieren lassen, wie z.B. Positionierung und nicht blockierende Ein- und Ausgabe. Da jede Erweiterung die Freiheit hat, neue Operationen zu definieren, benötigt man die Fähigkeit, unbekannte Operationen weiterzugeben.

Eine mögliche Lösung für C++ auf Basis von *multiple inheritance* findet sich in Abbildung 16, das durch das Entfernen der Trennung in *istream* und *ostream* vereinfacht worden ist. Statt zweier Objekte haben wir damit wieder eines, das alle Fähigkeiten von *dstream* und *XXstream* übernimmt. Es reicht jedoch nicht, einfach nur *XXdstream* als Kombination zu definieren, da die Ein- und Ausgabe-Operationen von *dstream* auf die von *XXstream* zurückgreifen müssen. So bleibt nichts anderes übrig, als *XXdstream* mit Programmtext anzufüllen, der weiß, daß die eigentliche Ein- und Ausgabe durch *XXstream* durchzuführen ist.

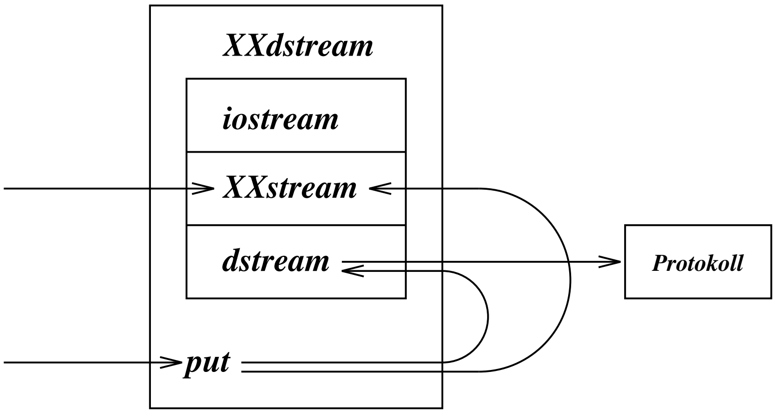

Abb. 16: Lösung des *dstream*-Problems in C++

Darüber hinaus gibt es keine Möglichkeit, einen bereits kreierten *iostream* hinter einen *dstream* zu setzen. Dies wäre jedoch naheliegend, wenn man die Ein- und Ausgabe der üblicherweise bereits geöffneten Standardkanäle mitprotokollieren möchte.

Andererseits offeriert Oberon eine sehr elegante Methode dafür, bei der mögliche Operationen durch Auftragsverbunde (*message records*) repräsentiert werden. Hierbei wird für jede unterstützte Operation eine Erweiterung eines Basistyps für Aufträge definiert, die die notwendigen Parameter enthält. Bei dieser Technik gibt es für ein Objekt eine Prozedur, die alle Aufträge entgegennimmt und dann mit Hilfe von Typentests das weitere Verfahren entscheiden kann. Da es sich beim Auftrag um einen ganz normalen Verbund (*record*) handelt, kann er problemlos an ein anderes Objekt weitergereicht werden.

Diese Technik ermöglicht auch die Weiterleitung von Aufträgen in verteilten Systemen. Für Objekte, die nicht direkt zugänglich sind, können lokale Objekte eingerichtet werden, die Aufträge verschicken und auf die Rücksendung der Antworten warten. Da auf diese Weise kein Unterschied mehr erkennbar ist zwischen Operationen auf lokalen und fernen Objekten, ist mehr Unabhängigkeit gewonnen.

Obwohl [Stein87] belegt, daß Delegationen auf Basis von Erweiterungen von Klassen gelöst werden können, reichen die klassischen Konzepte für statische Klassen nicht aus, da es darauf ankommt, daß Delegationen bzw. Erweiterungen zur Laufzeit vorgenommen werden können. Lieberman betrachtet Delegationen nicht als statisch und sieht insbesondere hier den Vorteil gegenüber dem traditionellen Ansatz. Da bei Erweiterungen zur Laufzeit Gruppierungen von untergeordnetem Interesse sind, lag es für Lieberman nahe, darauf völlig zu verzichten.

Durch die Möglichkeit, Operationen durch Auftragsverbunde zu repräsentieren, und durch die Verwendung von *Disciplines* lassen sich in Oberon sowohl Delegationen als auch Erweiterungen zur Laufzeit in einer Weise realisieren, die Abstraktionsgrenzen nicht verletzt.

1.10. Aspekte der Software-Zuverlässigkeit

Wie eingangs erwähnt wurde, sollte eine möglichst hohe Sicherheit dafür gegeben sein, daß das erstellte Software-System auch der Spezifikation entspricht. Die bislang vorgestellten Mechanismen tragen sicherlich mit dazu bei:

- Durch Modularisierung besteht die Möglichkeit, kleine überschaubare Einheiten zu erhalten, die versehentliche Zugriffe auf fremde Bereiche verhindern.
- Ein statisches Typsystem erlaubt weitreichende Überprüfungen zur Übersetzzeit.
- Das Konzept der Erweiterbarkeit bewahrt vorhandene Programmteile vor unnötigen Änderungen, die oftmals zu Fehlern führen.

Praktische Erfahrungen bestätigen, daß auf diese Weise Fehler im großen Umfang vermieden werden können [Griffiths91].

Hierbei ist insbesondere die Frage interessant, ob es zur Laufzeit mit Sicherheit bemerkt wird, wenn Anweisungen zu undefinierten Resultaten führen. Beispiele sind Überlauf bei arithmetischen Operationen oder das Verlassen der Indexgrenzen bei einem Feld. Generell kann dies kaum garantiert werden, da zumindest bei Sprachen, die Systemprogrammierung unterstützen, es auch notwendig ist, die normalen Sicherheitsgrenzen zu umgehen. Bemerkenswert ist hier der Ansatz von Modula-3 [Nelson91], der unsichere Operationen nur in explizit als unsicher deklarierten Modulen erlaubt. Dies geht nicht für jede Programmiersprache. So kann man bei C und weitgehend auch bei C++ kaum ohne unsichere Anweisungen auskommen, wie z.B. bei der Adreßarithmetik für Felder.

Es stellt sich aber natürlich die Frage, ob sich die Korrektheit beweisen läßt. Auf Basis einer formalen Definition für die Sprache (wie z.B. die von Hoare und Wirth für Pascal [Hoare73]) kann für ein Programmteil S mit gegebener Vorbedingung P und zu erfüllender Nachbedingung Q (kurz notiert: $P \{ S \} Q$) der Korrektheitsbeweis erbracht werden. Dies geschieht durch das Berechnen der *weakest precondition* $wp(S, Q)$ und dem Nachweis, daß $P \Rightarrow wp(S, Q)$ [Dijkstra75, Gries81].

Im Rahmen eines Projekts bei AEG-Telefunken in Ulm wurde für PASQUALE-L, das eine Teilmenge von Pascal um Sprachelemente zur formalen Spezifikation (auf Basis der Aussagenlogik) erweitert, ein automatisches Verifikationssystem entwickelt [Hohlfeld84]. Um den Quelltext einem normalen Pascal-Compiler zuführen zu können, werden für Spezifikationen in PASQUALE-L spezielle Kommentare verwendet.

Allerdings ist es generell nicht einfach, ein solches System auch für andere Programmiersprachen einzusetzen. Dies liegt vor allem auch daran, daß Axiomensysteme — wenn überhaupt — erst im Nachhinein für Programmiersprachen entwickelt worden sind. Zu den wenigen Ausnahmen gehört Alphard [Shaw77].

Hinzu kommt, daß die Aussagenlogik für viele Fälle nicht ausreicht oder daß es überhaupt unmöglich ist, durch eine Nachbedingung die korrekte Durchführung der Operation zu erfassen. Dieses Problem tritt vielfach im Zusammenhang mit Abstraktionen auf, deren Implementierung unbekannt ist.

Als Beispiel mag eine Prozedur *WriteInt* dienen, die als Parameter einen Kanal und eine Zahl erhält. Sicherlich hat man eine Vorstellung davon, wie eine korrekte Konvertierung von Zahlen in eine Sequenz von Zeichen aussehen könnte, und es wäre auch kein Problem, eine entsprechende Nachbedingung zu formulieren, wenn die resultierende Sequenz in Form eines Feldes vorliegen würde. Statt dessen sind jedoch die Zeichen auf dem Kanal ausgegeben worden, und es nicht einmal sicher, daß sie überhaupt noch existieren. Teilweise werden in der Literatur in solchen Fällen zusätzliche Variablen eingeführt, die nur dazu dienen, die durchgeführten Aktionen in einer nachprüfbaren Form aufzuzeichnen.

Wie De Millo, Lipton und Perlis erläutern, ist es praktisch unmöglich, formale Verifikationen für ein komplexes Software-System durchzuführen [Millo79]. Selbst wenn es dafür automatische Verifikatoren gäbe, bliebe die Frage nach dem Wert des Resultats, das von einem Vielfachen an unlesbarem generierten Beweistext belegt werden würde. Sie führen aus, daß man auch im Bereich der Software-Entwicklung die Erfahrungen aus der Mathematik übernehmen müsse, wonach Beweise auch inkorrekt sein können, und daß der Annahme der Richtigkeit ein sozialer Prozeß zugrunde liegt:

> “The point is that mathematicians’ errors are corrected, not by formal symbol logic, but by other mathematicians.”

Sie folgern daraus, daß auch unter der Annahme, daß ein Programm korrekt ist, nicht auf defensive Programmierung verzichtet werden sollte, da es sonst möglicherweise zum *Titanic effect* kommt: Bei einem dann doch auftretenden Fehler sind die Folgen massiv und unvorhersagbar.

Gries rät daher, den Beweis und das Programm Hand in Hand zu entwickeln, wobei der Beweis eine Mischung aus formalen und informalen Techniken sein sollte [Gries81]. Seine Beispiele belegen, daß mit dieser Technik auch weniger triviale Teilaufgaben in überzeugender Weise gelöst werden können.

Man kann beide zunächst konträren Standpunkte zusammenführen: Sicherlich ist es wichtig, daß alles in erster Linie für den Menschen lesbar und verstehbar ist — aber möglicherweise versteht man erst einen Algorithmus, wenn man ihn zusammen mit dem Beweis entwickelt hat.

So liefern diese Techniken eher eine Programmiermethode als Anforderungen an eine Programmiersprache und zugehörige Werkzeuge. Natürlich bleibt immer noch die Möglichkeit, Vor- und Nachbedingungen zu formulieren und zur Laufzeit überprüfen zu lassen. Wenn man sich mit der Aussagenlogik analog zu PASQUALE-L begnügt, reichen dafür normale Ausdrücke vom Typ **BOOLEAN** völlig aus. Geringe Erweiterungen sind ggf. notwendig, um zwischen dem alten und dem neuen Wert einer Variablen unterscheiden zu können.

Von der Sprache Eiffel wird dies explizit unterstützt [Meyer88]. Es lassen sich in Eiffel nicht nur Vor- und Nachbedingungen zu den exportierten Operationen angeben, sondern auch Klasseninvarianten. Eiffel berücksichtigt dabei die **is-a**-Beziehung bei Erweiterungen: Wird eine Operation überdefiniert, so darf die Vorbedingung abgeschwächt und die Nachbedingung verschärft werden. Leavens zeigt, daß auf Basis von **is-a**-Beziehungen der formale Verifikationsaufwand bei der Einführung neuer Typen reduziert werden kann [Leavens91].

Eiffel ermutigt dabei zu einem Programmierstil, der von der üblichen defensiven Technik abweicht, bei einer Operationen alle Fehlermöglichkeiten (ein Fehler liegt vor, wenn der Aufrufer die Vorbedingung verletzt) zu berücksichtigen und abzufangen. Statt dessen soll man sich darauf verlassen, daß die Vorbedingung eingehalten wird. Die Eiffel-Übersetzer erlauben es dann üblicherweise, Code für die Vor- und Nachbedingungen generieren zu lassen, um die Einhaltung zur Laufzeit zu überprüfen. Damit besteht dann auch nicht mehr die Gefahr des *Titanic effects*, da nur der normale Programmtext von den Vor- und Nachbedingungen getrennt worden ist.

Natürlich wäre es auch denkbar, daß man analog zu PASQUALE-L Vor- und Nachbedingungen innerhalb von speziellen Kommentaren angibt. Dann hätte man die Wahl, entweder das Programm normal ohne Überprüfungen zur Laufzeit übersetzen zu lassen oder mit Hilfe eines speziellen Präprozessors ein Programm generieren zu lassen, das die Überprüfungen enthält. Allerdings bedeutet dies einen erheblichen Zusatzaufwand — insbesondere würde die Unterstützung der **is-a**-Beziehung bei Erweiterungen schwierig werden. Alternativ ist es auch denkbar, daß ein Übersetzer zusätzlich zur eigentlichen Sprache auch die speziellen Kommentare mit berücksichtigt.

Oberon unterstützt entsprechende Mechanismen nicht, und bislang sind auch keine Erweiterungen bekannt, die die Spezifikation von Vor- und Nachbedingungen mit der Möglichkeit zur Laufzeitüberprüfung erlauben. Damit bleibt nur der Weg, die Überprüfungen explizit in den Programmtext aufzunehmen. Zwar fällt damit die Möglichkeit weg, sie aus Effizienzgründen auf leichte Weise auszuschalten, aber wegen des möglichen *Titanic effect* empfiehlt sich dies sowieso nicht.

Nicht zu unterschätzen ist die Bedeutung der Lesbarkeit eines Programms. Je kryptischer und schwerverständlicher ein Programmtext wird, desto größer ist die Gefahr, daß Fehler Inspektionen überleben und daß Änderungen sich fatal auswirken. Parnas und Clements beschreiben, wie wichtig es ist, parallel zur Software-Entwicklung auch die zugehörige Dokumentation zu pflegen und zwar so, daß sie sowohl dem ersten Entwurf dient als auch der späteren Wartung [Parnas86]. Insbesondere empfiehlt es sich, sie weitgehend der gleichen Modularisierung zu unterwerfen wie das zugehörige Software-System.

1.11. Zusammenfassung der Sprachkonzepte

Folgende Konzepte stellten sich beim vorangegangenen Überblick als relativ wichtig heraus:

- Abstraktionen und Implementierungen sind zu trennen. Eine Abstraktion repräsentiert einen Kontrakt zwischen möglichen Klienten und Implementierungen.
- Eine Abstraktion soll mehrere abstrakte Datentypen umfassen können. Entsprechend kann eine Abstraktion nicht einem Typ gleichgesetzt werden.
- Genauso wie eine Abstraktion viele Klienten haben kann, soll es auch möglich sein, mehrere Implementierungen zuzulassen.
- Abstraktionen müssen in aufwärtskompatibler Weise erweiterbar sein.
- Operationen müssen delegiert werden können. Dabei sollten nicht nur die bekannten Operationen Berücksichtigung finden können, sondern auch die von unbekannten Erweiterungen.
- Objekte dienen als Bezugspunkte, zu denen jedes betroffene Modul seinen eigenen privaten Status unterhalten kann. Wenn nötig, muß es möglich sein, private Informationen zur Laufzeit hinzuzufügen.

Abbildung 17 zeigt das resultierende Layout mit einer Hierarchie von Abstraktionen und aufeinander aufbauenden Implementierungen. Es gibt dabei folgende Sorten von Beziehungen:

Erweiterung:
: Eine Abstraktion kann eine Erweiterung einer anderen Abstraktion sein.

Import:
: Eine Implementierung kann auf anderen Implementierungen aufbauen oder sich auf bestimmte Abstraktionen beziehen.

Implementierung:
: Eine Implementierung kann eine oder ggf. mehrere Abstraktionen implementieren.

Wenn die Beziehungstypen dahingehend vereinfacht werden, daß man alle drei als Importbeziehung betrachtet (wobei allerdings in zwei Fällen mehr als nur eine Importanweisung notwendig ist) und die Module entsprechend anordnet, so erhalten wir den Aufbau in Abbildung 18.

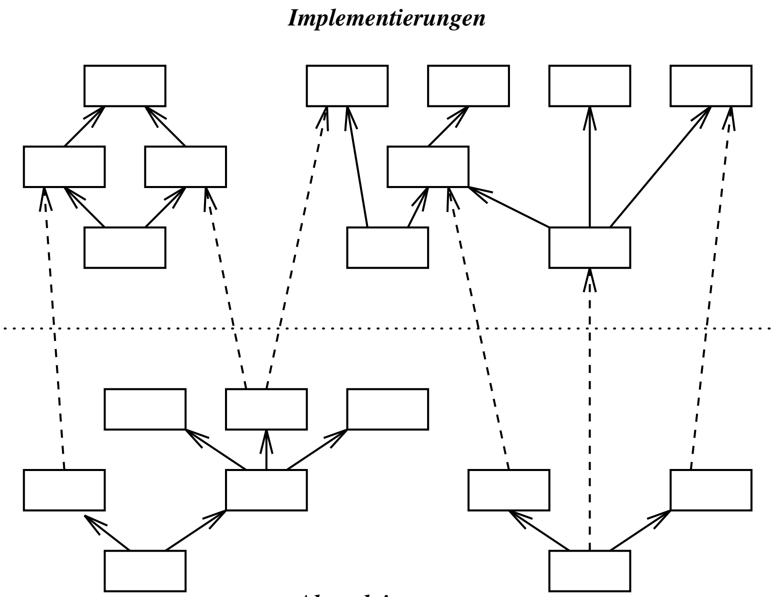

Abb. 17: Abstraktionen und Implementierungen

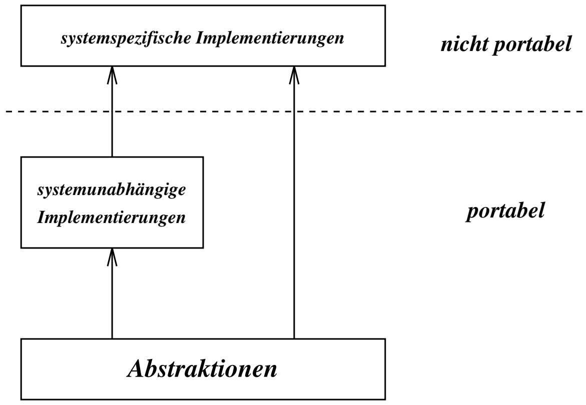

Abb. 18: Modulanordnung entsprechend der Importbeziehungen

Dieser Aufbau weicht völlig von dem traditionellen Aufbau ab, der die systemspezifischen Module der untersten Schicht zuordnet, während die systemunabhängigen Module darauf aufbauend oben thronen. Beim traditionellen Entwurf sind damit zumindest indirekte Abhängigkeiten von systemspezifischen Modulen nicht vermeidbar, die zu Portabilitätsproblemen führen können. Hingegen hat man bei dem skizzierten Aufbau die Garantie, daß es keine systemspezifischen Abhängigkeiten gibt, so daß allenfalls Portabilitätsfallen durch Varianten der Sprache selbst entstehen können (z.B. durch unterschiedliche Repräsentierung der Basistypen).

Eine Reihe von weiteren wichtigen Sprachmitteln wurde in diesem Überblick nicht berücksichtigt. Besonders wichtig ist hier die Parallelverarbeitung, die es analog zur realen Welt erlauben würde, daß Objekte auch dann aktiv sind, wenn nicht gerade von außerhalb Operationen auf ihnen durchgeführt werden. Parallelisierungen vermeiden die Einführung einer zentralen Kontrollinstanz (*main program* oder *event loop*) und erhöhen damit die Unabhängigkeit und Dezentralisierung.

Die Parallelverarbeitung wird neben anderen interessanten Techniken, wie z.B. Persistenz, Ausnahmenbehandlungen, Iteratoren und generischen Modulen in den folgenden Kapiteln zusammen mit den entsprechenden Mechanismen der Bibliothek eingeführt. Das belegt, daß auf Basis einer guten Grundlage es nicht notwendig ist, alle Techniken direkt in die Programmiersprache zu integrieren.

So bleiben als Abschluß der einführenden Diskussion zwei fundamentale und zueinander orthogonale Prinzipien, die von einer Programmiersprache für größere Software-Projekte zu fordern sind: Modularisierung und Erweiterbarkeit. Hinzu kommt die Software-Zuverlässigkeit, die durch Modularisierung und Erweiterbarkeit zwar unterstützt wird, jedoch zusätzliche Mechanismen verlangt. Alle anderen Forderungen lassen sich hiervon ableiten.

I have regarded it as the highest goal of programming language design
to enable good ideas to be elegantly expressed.
[Hoare81]

2. Objekt-orientierte Techniken in Oberon

2.1. Einführung

Knudsen und Madsen warnen davor, daß Programmiersprachen nur auf Basis technischer Unterschiede zu anderen üblichen Konstrukten eingeführt werden [Knudsen88]. Entsprechend dem "Turing Tarpit" kann — zumindest theoretisch — ohnehin jedes Problem, das in einer der vertrauten Programmiersprachen formuliert werden kann, auch in eine beliebige andere Programmiersprache übertragen werden, darunter natürlich auch Turing-Maschinen.

Somit ist es weniger interessant, welche Konstrukte in Oberon simuliert werden können, sondern vielmehr, welche Programmiermethodik tatsächlich unterstützt und angeregt wird. Allerdings darf man nicht unbedingt eine Programmiersprache für sich betrachten. Wie bereits erwähnt, kommt einer Bibliothek eine tragende Bedeutung zu, da sie weitere Sprachmittel und Methoden einbringt.

Vorweg läßt sich sagen, daß Oberon in erster Linie Modularisierung und Erweiterbarkeit zum größten Teil auf Basis konventioneller Techniken unterstützt. Oberon offeriert eine bemerkenswerte Flexibilität in Bezug auf objekt-orientierte Techniken, so daß es häufig möglich ist, spezielle Varianten für besondere Probleme zu entwickeln. Während andere Sprachen ein relativ enggefaßtes Modell anbieten, das für alle Situationen passend gemacht werden muß, bestehen solche Restriktionen in Oberon kaum. Hingegen wird die klassische objekt-orientierte Technik des inkrementellen Überdefinierens von Methoden überhaupt nicht unterstützt (natürlich wäre es möglich, sie zu simulieren). Dies ist jedoch überhaupt nicht notwendig, da, wie bereits gezeigt worden ist, diese Technik die Vermengung von Abstraktionen mit Implementierungen fördern würde.

Der Schwerpunkt im folgenden liegt bei dem Modulkonzept von Oberon und den objekt-orientierten Techniken. Eine allgemeinere Einführung zu Oberon findet sich in [Reiser92]. Alle Erläuterungen und Beispiele folgen der Ulmer Variante von Oberon, die auf [Wirth88a] basiert und im wesentlichen nur aufwärtskompatible Änderungen der Sprache aus Zürich übernommen hat. Eine detaillierte Erläuterung der Unterschiede, die nicht überbewertet werden sollten, findet sich im Anhang.

2.2. Das Modulkonzept von Oberon

Obwohl sich das Modulkonzept von Oberon von Modula-2 [Wirth82] ableitet, weist es zwei signifikante Vereinfachungen auf.

So werden lokale Module von Oberon nicht mehr unterstützt. Dies ist einerseits bedauerlich, da lokale Module helfen, auch größere Implementierungen in übersichtliche Teile zu zerlegen. Andererseits sind sie nicht ohne Probleme und Mehrdeutigkeiten, wie Cornelius schildert [Cornelius88]. Somit ist es verständlich, daß sie im Zuge der Vereinfachungen gestrichen worden sind.

Die interessantere Vereinfachung ergibt sich aus dem Verzicht auf nicht qualifizierende Importanweisungen. Modula-2 erlaubt es, fremde Module auf zwei verschiedene Weisen zu importieren. Mit **FROM**...**IMPORT** können einzelne Namen aus einem Modul ausgewählt und sichtbar gemacht werden. Bei **IMPORT** allein, ohne vorangehendes **FROM**, werden nur die Modulnamen selbst zugänglich. Namen aus diesem Modul müssen dann explizit mit einem Modulnamen qualifiziert werden:

```
MODULE SomeModule;

   FROM InOut IMPORT WriteString, WriteLn;
      (* nicht qualifizierter Import *)

   IMPORT RealInOut; (* qualifizierter Import *)

BEGIN
   WriteString("Hallo!"); WriteLn;
   RealInOut.WriteReal(1.0, 0);
END SomeModule.
```

Auf den ersten Blick sieht der qualifizierte Import umständlicher aus, da er zu längeren Namen zwingt. In der Praxis wird er in Modula-2 typischerweise nur verwendet, um Konflikte zu vermeiden, wenn zwei Module den gleichen Namen exportieren.

Trotzdem hat sich der Zwang zu Qualifikationen in Oberon bewährt. Einerseits relativiert sich das Problem der langen Namen rasch, wenn die Namen beim Entwurf einer Schnittstelle im Bewußtsein gewählt werden, daß der Modulname immer davor stehen muß. Zum anderen wird die Lesbarkeit entscheidend erhöht, da bei Verwendung von importierten Operationen der Modulname immer dabeisteht. Obiges Beispiel würde sich bei Verwendung der Ulmer Oberon-Bibliothek folgendermaßen lesen:

```
MODULE SomeModule;

   IMPORT Write;

BEGIN
   Write.String("Hallo!"); Write.Ln;
   Write.Real(1.0, 0);
END SomeModule.
```

Analog zu Modula-2 gibt es zu jedem Modul eine Schnittstelle, die gemäß dem Vertrags-Modell ein separates Dokument darstellt. Die Schnittstelle ist dabei ein Auszug des zugehörigen Moduls. Im Gegensatz zu Modula-2 müssen also in der Implementierung alle Deklarationen aus der Schnittstelle wiederholt werden. Dies ist eine bemerkenswerte Vereinfachung, da das Modul damit nicht mehr abhängig ist von der Auswahl der exportierten Deklarationen. Zudem erleichtert es das Lesen, wenn alle lokalen Namen auch im gleichen Dokument deklariert werden.

Analog wurde das Konzept des abstrakten Datentyps aus Modula-2 sowohl vereinfacht als auch verallgemeinert. In Modula-2 werden abstrakte Datentypen, deren Repräsentierung verborgen bleiben soll, gemäß folgendem Beispiel deklariert:

```
DEFINITION MODULE Streams; (* Modula-2 Version *)

   TYPE
      File; (* abstrakter Datentyp *)

   PROCEDURE Open(VAR f: File; name: ARRAY OF CHAR);
   PROCEDURE PutChar(f: File; ch: CHAR);
   PROCEDURE GetChar(f: File; VAR ch: CHAR);
   PROCEDURE Close(f: File);

END Streams.
```

In der zugehörigen Implementierung muß *File* als Zeiger auf einen beliebigen Datentyp deklariert werden [Wirth85, S. 85]. In Oberon kann ein abstrakter Datentyp teilweise offengelegt werden gemäß dem Prinzip, daß die Schnittstelle einen Auszug der Implementierung offenlegen darf:

```
DEFINITION Streams; (* Oberon Version *)

   TYPE
      File = POINTER TO FileRec;
      FileRec =
         RECORD
            eof: BOOLEAN;
            error: BOOLEAN;
         END;

   (* ... *)

END Streams.
```

Im Vergleich zu Modula-2 mag man zwar immer noch ahnen, daß sich hinter *File* ein abstrakter Datentyp mit privaten Komponenten verbirgt — jedoch läßt sich diese Tatsache nicht mehr auf syntaktische Weise aus der Schnittstelle schließen. Obwohl *FileRec* schon in der Schnittstelle beschrieben worden ist, hat die zugehörige Implementierung die Freiheit, *FileRec* um weitere Komponenten zu ergänzen:

```
MODULE Streams; (* Oberon Version *)

   TYPE
      File = POINTER TO FileRec;
      FileRec =
         RECORD
```

```
            (* oeffentliche Komponenten *)
            eof: BOOLEAN;
            error: BOOLEAN;
            (* private Komponenten *)
            fd: INTEGER;
         END;

   (* ... *)

END Streams.
```

Diese Freiheit fordert ihren Preis, da Klienten von *Streams* möglicherweise von der Speicherplatzgröße von *FileRec* abhängen. Obwohl es im obigen Beispiel nicht zweckmäßig ist, besteht die Freiheit, daß ein Klient folgendes deklariert:

```
VAR
   filerecs: ARRAY 10 OF FileRec;
```

In so einem Fall kann der Übersetzer beim Klienten kaum ohne die tatsächliche Größe von *FileRec* auskommen. Beim Ulmer Oberon-Übersetzer wird das Problem dadurch gelöst, daß auch beim Übersetzen der Implementierung die Symboldatei geändert werden darf. Um die Abhängigkeiten zu reduzieren, geschieht dies normalerweise nur, wenn es unbedingt notwendig ist, d.h. der notwendige Speicherplatz sich vergrößert. In der Praxis hat es sich gezeigt, daß Neuübersetzungen aufgrund dieser Abhängigkeit ausgesprochen rar sind.

Es stellt sich natürlich die Frage, ob eine Zuweisung erlaubt ist, wenn Teile des Verbunds privat sind. Dies kann zur Verletzung von Integritätsbedingungen führen. Wenn im obigen Beispiel ein geöffneter *FileRec* kopiert und geschlossen wird, dann ahnt das Original davon nichts. Im ersten Report über Oberon wurde diese Art der Zuweisung ausdrücklich verboten [Wirth88a, §9.1], im späteren Report verschwand jedoch diese Restriktion [Wirth89c]. Beim Ulmer Oberon-Übersetzer wurde die Restriktion beibehalten.

2.3. Typerweiterungen

Verbundtypen können als Erweiterung eines anderen Verbundtyps definiert werden. Im Beispiel sind *Rectangle* und *Circle* Erweiterungen von *Figure*:

```
TYPE
   Figure =
      RECORD
         x, y: INTEGER; (* center *)
      END;
   Rectangle =
      RECORD
         (Figure)
         w, h: INTEGER;
      END;
   Circle =
      RECORD
         (Figure)
         r: INTEGER;
      END;
```

Eine Erweiterung übernimmt dabei sämtliche Komponenten der übergeordneten Typen. Abbildung 19 zeigt dies für das obige Beispiel.

Das Wesentliche daran ist die Kompatibilität zwischen dem Basistyp (im Beispiel *Figure*) und den abgeleiteten Typen. So ist die Zuweisung eines *Rectangle* an eine *Figure* zulässig. Da der Basistyp jedoch nur einen Teil umfaßt, findet dabei implizit eine Projektion statt. Andererseits ist die Zuweisung eines Basistyps an eine Erweiterung nicht zugelassen, da dann Komponenten undefiniert bleiben würden:

```
VAR
   figure: Figure;
   rectangle: Rectangle;
   circle: Circle;

(* ... *)

figure := rectangle; (* zulaessig: nur die Komponenten x und y werden zugewiesen *)
rectangle := figure; (* nicht zulaessig *)
```

```
rectangle := circle; (* nicht zulaessig *)
```

Eine Projektion findet jedoch nicht statt, wenn ein Verbund als **VAR**-Parameter übergeben wird:

```
PROCEDURE Move(VAR figure: Figure; newx, newy: INTEGER);
BEGIN
   figure.x := newx;
   figure.y := newy;
END Move;

(* ... *)

Move(circle, 0, 0); (* ist legal *)
```

Die Kompatibilitätsregeln erstrecken sich auch auf entsprechende Zeigertypen:

```
TYPE
   FigurePtr = POINTER TO Figure;
   RectanglePtr = POINTER TO Rectangle;
   CirclePtr = POINTER TO Circle;
VAR
   figurePtr: FigurePtr;
   rectanglePtr: RectanglePtr;
   circlePtr: CirclePtr;

(* ... *)

figurePtr := rectanglePtr; (* zulaessig, jedoch keine Projektion *)
rectanglePtr := figurePtr; (* nicht zulaessig *)
rectanglePtr := circlePtr; (* nicht zulaessig *)
```

Entsprechend [Wirth88b] lassen sich folgende Definitionen geben: Gegeben sei eine Deklaration der Form

```
T0 = RECORD (T) ... END;
```

Dann ist *T0* eine (direkte) *Erweiterung* von *T* und *T* ist der (direkte) *Basistyp* von *T0*.

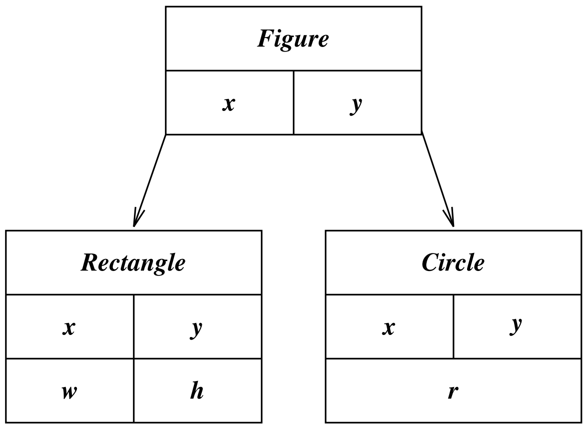

Abb. 19: Einfache Typenhierarchie mit den zugehörigen Komponenten

Erweiterte Typen können erneut erweitert werden: Ein Verbundtyp *U* ist eine Erweiterung des Verbundtyps *T* genau dann, falls

- *U* = *T* oder
- *U* eine direkte Erweiterung von *T* ist oder
- *U* eine direkte Erweiterung einer Erweiterung von *T* ist.

Entsprechend ist ein Verbundtyp *T* genau dann ein Basistyp eines Verbundtyps *U*, wenn *U* eine Erweiterung von *T* ist.

Diese Definitionen erstrecken sich auf die zugehörigen Zeigertypen. Seien folgende Definitionen gegeben:

```
TYPE P = POINTER TO T;
TYPE Q = POINTER TO U;
```

Dann ist ein Zeigertyp *Q* genau dann eine Erweiterung eines Zeigertyps *P*, wenn *U* eine Erweiterung von *T* ist. Ein Zeigertyp *P* ist genau dann ein Basistyp eines Zeigertyps *Q*, wenn *Q* eine Erweiterung von *P* ist.

Kompatibilitätsregeln sind bei Verbunden und Zeigern an zwei Stellen von Interesse: bei der Zuweisung und der Parameterübergabe. Bei Parameterübergaben gibt es Werteparameter, die analog zu Zuweisungen behandelt werden, und Variablenparameter. Generell gilt, daß eine Zuweisung oder Parameterübergabe dann zulässig ist, wenn der Typ des zuzuweisenden oder zu übergebenden Objekts eine Erweiterung des Typs des Zielobjekts bzw. des formalen Parameters ist. Hierzu gibt es einige Ausnahmen, die in diesem Zusammenhang jedoch nicht von Bedeutung sind [Wirth88a, §9.1].

Im Normalfall lassen sich alle Kompatibilitätsregeln statisch überprüfen. Eine Ausnahme ist bei Variablenparametern gegeben, wie folgendes Beispiel zeigt:

```
PROCEDURE Copy(VAR source, dest: Figure);
BEGIN
   dest := source;
END Copy;

(* ... *)

Copy(circle, rectangle);
```

Die Prozedur *Copy* ist, für sich gesehen, statisch vollkommen in Ordnung, da der formale Parametertyp von *source* und *dest* identisch ist. Andererseits ist der Aufruf ebenfalls korrekt, da sowohl *Circle* als auch *Rectangle* Erweiterungen von *Figure* sind. Zur Laufzeit stellt sich dann jedoch heraus, daß versucht wird, einer Variablen vom Typ *Circle* einen Wert vom Typ *Rectangle* zuzuweisen.

Dies passiert jedoch nur bei Zuweisungen von Variablenparametern. Wie bereits erwähnt, ist die Zuweisung von Verbunden mit unbekannten Teilen problematisch. Hier ist zwar möglicherweise *Figure* in vollem Umfang bekannt, jedoch nicht die potentiellen Erweiterungen. Ein generelles Verbot solcher Zuweisungen würde jedoch zu weit gehen, da dann auch “ganz normale” Verbunde im Rahmen konventionellen Programmierstils davon bedroht wären, die normalerweise nicht erweitert werden.

Das gleiche Problem ergibt sich auch bei Zuweisungen von Variablenparametern, die einen Zeigertyp haben. Der Ulmer Oberon-Übersetzer generiert für alle diese Fälle Laufzeitüberprüfungen und bestimmt bei Verbunden die korrekte Projektion zur Laufzeit.

2.4. Typentests

Wenn ein Zeiger oder Variablenparameter gegeben ist, dann sind zwei Typen damit assoziiert: der statisch bekannte Typ und der zur Laufzeit bestimmte Typ. Dabei gilt natürlich, daß der Typ zur Laufzeit eine Erweiterung des statischen Typs sein muß.

Oberon bietet einen Operator an, der es erlaubt, den dynamischen Typ zu überprüfen:

```
PROCEDURE Draw(VAR figure: Figure);
BEGIN
   IF figure IS Circle THEN
      (* Kreis zeichnen *)
   ELSIF figure IS Rectangle THEN
      (* Rechteck zeichnen *)
   ELSE
      (* unbekannt *)
   END;
END Draw;
```

Eine Typzusicherung ermöglicht es dann, eine Variable entsprechend der angegebenen Erweiterung zu behandeln:

```
(* Kreis zeichnen *)
DrawCircle(figure(Circle));

(* Zugriff auf den Radius *)
diameter := figure(Circle).r * 2;
```

Mit Hilfe der **WITH**-Anweisung kann sich eine Typzusicherung über eine Reihe von Anweisungen erstrecken:

```
WITH figure: Rectangle DO
   area := figure.w * figure.h;
END;
```

Die Typentests von Oberon erscheinen auf den ersten Blick im Umfeld objekt-orientierter Sprachen nicht notwendig. Gerade obiges Beispiel scheint es zu belegen, da die Methode *Draw* individuell für jede der möglichen Varianten sein sollte. Ein Test für alle möglichen Fälle sollte unbedingt vermieden werden.

So haben wohl auch die Designer von Eiffel und C++ ursprünglich gedacht, die zunächst keine entsprechenden Sprachmittel eingeführt haben. In Eiffel wurde dies später nachgeholt durch das *reverse assignment attempt*, das es erlaubt, eine Zuweisung entgegen der üblichen Kompatibilitätsregeln zu versuchen [Meyer92, §20.13]:

```
-- Eiffel's reverse assignment attempt
local
   t: T;
   u: U; -- U ist eine Erweiterung von T
-- ...
   u ?= t;
   if not u = Void then -- Test auf Erfolg
      -- Zugriff auf Komponenten der Erweiterung ueber u moeglich
   end;
```

In C++ fehlt bis heute der entsprechende Mechanismus. Dies führte dazu, daß er bei einigen Klassenbibliotheken (z.B. ET++) simuliert wird oder — noch schlimmer — durch einen ungesicherten *cast* ersetzt wird.

Das wichtigste Anwendungsfeld für Typentests und Typzusicherungen sind heterogene Datenstrukturen, bei denen nur ein gemeinsamer Basistyp festgelegt wird. Ein Beispiel dafür wäre eine Liste von *Figure*-Objekten, in der auch Kreise und Rechtecke vorkommen können. In Oberon finden Typentests und Zusicherungen auch noch Anwendung bei einer Reihe von objekt-orientierten Techniken, die im einzelnen noch vorgestellt werden.

2.5. Generische Module

Generische Module wurden zunächst bei CLU eingeführt [Liskov77, Liskov81] und etwas später in sehr ähnlicher Form in Ada [ANSI83]. Wie Meyer ausführt [Meyer86], lassen sich generische Module durch die Anwendung objekt-orientierter Techniken ersetzen, jedoch zumindest im Falle von Eiffel in Verbindung mit einigen Umständlichkeiten. Aus diesem Grund wurden in etwas limitierter Form generische Module sowohl in Eiffel, Modula-3 als auch C++ (hier nennen sie sich *templates*) eingeführt.

Natürlich handelt es sich bei den Ersatzmechanismen nicht mehr um parametrisierte Module im strengen Sinne. Interessant ist jedoch, ob die Probleme, die zur Einführung generischer Module geführt haben, sich nicht auch in anderer Form elegant lösen lassen. Das klassische Beispiel für generische Module sind Verwaltungen von Datenstrukturen, bei denen der Basistyp von untergeordneter Bedeutung ist. Folgendes einfach gehaltene Beispiel zeigt ein allgemeines Modul für Listen in Oberon. Es verwendet als Basistyp *Objects.Object*, das eine ähnliche Funktion wie *ANY* in Eiffel übernimmt,[1] jedoch nicht Bestandteil der Sprache Oberon ist.

```
DEFINITION Lists;

   IMPORT Objects;

   TYPE
      List = POINTER TO ListRec;
      ListRec = RECORD (Objects.ObjectRec) END;

   PROCEDURE Create(VAR list: List);
   PROCEDURE Append(list: List; object: Objects.Object);
```

[1] In Eiffel sind alle Klassen implizit von *ANY* abgeleitet.

```
      PROCEDURE First(list: List);
      PROCEDURE Next(list: List; VAR object: Objects.Object) : BOOLEAN;

   END Lists.
```

Create legt eine neue Liste an. Mit *Append* kann ein neues Element an das Ende der Liste angehängt werden. *First* und *Next* erlauben es, die Liste zu durchlaufen. *Next* liefert dabei **FALSE** zurück, wenn kein Objekt mehr zurückgeliefert werden kann, da das Ende der Liste erreicht worden ist. In der angebotenen Form ist dies etwas unzureichend, da nur ein Durchlauf zu einem gegebenen Zeitpunkt möglich ist. Später werden Iteratoren in allgemeinerer Form vorgestellt. Die Implementierung könnte folgendermaßen aussehen:

```
MODULE Lists;

   IMPORT Objects;

   TYPE
      Linkable = POINTER TO LinkableRec;
      LinkableRec =
         RECORD
            next: Linkable;
            object: Objects.Object;
         END;

      List = POINTER TO ListRec;
      ListRec =
         RECORD
            (Objects.ObjectRec)
            (* private Komponenten *)
            head, tail: Linkable;
            cursor: Linkable; (* fuer List und Next *)
         END;

   PROCEDURE Create(VAR list: List);
   BEGIN
      NEW(list);
      list.head := NIL; list.tail := NIL; list.cursor := NIL;
   END Create;

   PROCEDURE Append(list: List; object: Objects.Object);
      VAR
         linkable: Linkable;
   BEGIN
      NEW(linkable);
      linkable.object := object;
      linkable.next := NIL;
      IF list.head = NIL THEN
         list.head := linkable;
      ELSE
         list.tail.next := linkable;
      END;
      list.tail := linkable;
   END Append;

   PROCEDURE First(list: List);
   BEGIN
      list.cursor := list.head;
   END First;

   PROCEDURE Next(list: List; VAR object: Objects.Object) : BOOLEAN;
   BEGIN
      IF list.cursor = NIL THEN
         RETURN FALSE
      ELSE
         object := list.cursor.object;
         list.cursor := list.cursor.next;
         RETURN TRUE
      END;
   END Next;
```

```
END Lists.
```

Wie sich aus dem Beispiel ersehen läßt, erweist es sich als recht vorteilhaft, mehrere Datentypen in einem Modul vereinbaren zu können. Während bei Eiffel für *Linkable* eine eigene Klasse mit den zugehörigen Zugriffsoperationen kreiert werden müßte, erlaubt es Oberon, diesen Typ völlig privat zu halten.

Folgendes Beispiel demonstriert, wie auf der Basis von *Lists* eine Liste von Textzeilen aufgebaut und durchlaufen werden kann:

```
TYPE
   Line = ARRAY 80 OF CHAR;
   LineObject = POINTER TO LineObjectRec;
   LineObjectRec = RECORD (Objects.ObjectRec) line: Line END;

PROCEDURE ReadLines(s: Streams.Stream; VAR list: Lists.List);
   (* Lies von dem gegebenen Stream bis zum Auftreten einer
      leeren Zeile oder Eingabeende und lege alle eingelesenen
      Zeilen in der neu zu kreierenden Liste ab
   *)
   VAR
      line: Line;
      lineObject: LineObject;
BEGIN
   Lists.Create(list);
   Read.LineS(s, line);
   WHILE (s.count > 0) & (line # "") DO
      NEW(lineObject); lineObject.line := line;
      Lists.Append(list, lineObject);
      Read.LineS(s, line);
   END;
END ReadLines;

PROCEDURE WriteLines(s: Streams.Stream; list: Lists.List);
   (* Gib alle Zeilen aus, die in der Liste enthalten sind;
      andere Objekte aus der Liste werden ignoriert
   *)
   VAR
      object: Objects.Object;
BEGIN
   Lists.First(list);
   WHILE Lists.Next(list, object) DO
      IF object IS LineObject THEN
         Write.LineS(s, object(LineObject).line);
      END;
   END;
END WriteLines;
```

Dieses Beispiel verdeutlicht den Unterschied zu den generischen Modulen aus CLU und Ada. Während bei CLU und Ada der Elementtyp der Liste genau festgelegt werden würde und damit statisch überprüfbar wäre, liegt hier nur fest, daß alle Elementtypen Erweiterungen von *Objects.Object* sein müssen. Dies liefert sowohl die Freiheit, heterogene Objekte in einer Liste zu verwalten, als auch die offene Frage, welche Sorten von Objekten man später darin wiederfindet.

Eiffel, Modula-2 und C++ legen natürlich den Elementtyp nicht so fest wie CLU und Ada, da sie auch Erweiterungen akzeptieren würden. Im Gegensatz zur obigen Lösung liefern sie vollständige statische Überprüfbarkeit auf Kosten eines nicht trivialen Sprachmittels. Allerdings ist dies bei Eiffel nicht vollständig gelungen, so zeigt Cook neben einigen Problemen im Typsystem von Eiffel auch eines in Zusammenhang mit generischen Klassen auf, das — wenn überhaupt — nur zur Laufzeit entdeckt werden kann [Cook89]. [1]

Wenn im obigen Beispiel *WriteLines* absolut sicher gehen kann, daß alle Objekte der Liste Erweiterungen von *LineObject* sind, dann kann der Programmtext weiter vereinfacht werden:

[1] Teilweise wird bei Eiffel argumentiert, daß eine statische Überprüfbarkeit möglich wäre, wenn man alle Klassen, die zu einem Programm zusammengebunden werden, gemeinsam betrachten würde [Jones92]. Dies ist jedoch relativ unrealistisch sowohl vom Aufwand als auch von praktischen Gegebenheiten, da kommerzielle Bibliotheken nicht unbedingt in Quelltextform ausgeliefert werden.

```
PROCEDURE WriteLines(s: Streams.Stream; list: Lists.List);
   (* Gib alle Zeilen aus, die in der Liste enthalten sind;
      es wird davon ausgegangen, dass alle Elemente der Liste
      Erweiterungen von LineObject sind
   *)
   VAR
      lineObject: LineObject;
BEGIN
   Lists.First(list);
   WHILE Lists.Next(list, lineObject) DO
      Write.LineS(s, lineObject.line);
   END;
END WriteLines;
```

Dies ist erlaubt, da *LineObject* eine Erweiterung von *Objects.Object* ist und damit direkt an *Lists.Next* übergeben werden kann. In diesem Fall würde es innerhalb von *Lists.Next* zu einem Laufzeitfehler bei der Zuweisung führen, falls das zugewiesene Element doch keine Erweiterung von *LineObject* ist.

Ada und CLU offerieren zusätzlich die Möglichkeit, Prozeduren und Variablen als Parameter bei generischen Modulen anzugeben. Dies ist beispielsweise sinnvoll bei einem allgemeinen Sortiermodul, das eine Vergleichsprozedur benötigt. In Oberon kann hierfür der Basistyp so definiert werden, daß zu ihm eine Sortierprozedur gehört (Techniken dazu folgen), oder man kann mit konventioneller Technik analog zu Modula-2 einen Prozedurparameter explizit angeben.

2.6. Aufträge und Auftragnehmer

Wie bereits Wirth andeutet, wird *late binding* auf Basis von Verbund-Komponenten erreicht, die einen Prozedurtyp besitzen [Wirth88d]. Im einfachsten Fall gibt es genau eine solche Komponente, die alle Operationen von außerhalb entgegennimmt:

```
TYPE
   Message = RECORD END; (* Basistyp fuer alle Auftraege *)
   Object = POINTER TO ObjectRec;
   Handler = (* Auftragnehmer *)
      PROCEDURE (object: Object; VAR message: Message);
   ObjectRec =
      RECORD
         handler: Handler;
      END;
```

Um die unterschiedlichen Operationen mit ihren Parametern zu repräsentieren, können entsprechende Erweiterungen von *Message* definiert werden. Da ein Auftragnehmer den Auftrag als Variablenparameter erhält, ist es möglich, auch Werte wieder zurückzugeben.

Im Beispiel der Figuren könnte dies folgendermaßen aussehen:

```
TYPE
   MoveMessage =
      RECORD
         (Message)
         newx, newy: INTEGER;
      END;
   DrawMessage =
      RECORD
         (Message)
      END;
   Figure = POINTER TO FigureRec;
   FigureRec =
      RECORD
         (ObjectRec)
         x, y: INTEGER;
      END;
```

Ein Auftragnehmer kann dann aufgrund von Typentests die einzelnen Operationen voneinander unterscheiden. Folgendes Beispiel deutet einen Auftragnehmer für Kreise an:

```
PROCEDURE Handler(figure: Figure; VAR message: Message);
BEGIN
   WITH figure: Circle DO
      IF message IS MoveMessage THEN
         WITH message: MoveMessage DO
            (* Kreis bewegen *)
         END;
      ELSIF message IS DrawMessage THEN
         WITH message: DrawMessage DO
            (* Kreis zeichnen *)
         END;
      ELSE
         (* unbekannter Auftrag *)
      END;
   END;
END Handler;
```

Da der Prozedurtyp für Auftragnehmer nur den Basistyp *Figure* festlegt, wird eine regionale Typzusicherung benötigt, um auf die Komponenten der Erweiterung zugreifen zu können. Andererseits könnte ein Aufruf einer *Move*-Operation folgendermaßen aussehen:

```
PROCEDURE MoveFigure(figure: Figure; newx, newy: INTEGER);
   VAR
      moveMessage: MoveMessage;
BEGIN
   moveMessage.newx := newx; moveMessage.newy := newy;
   figure.handler(figure, moveMessage);
END MoveFigure;
```

Zwei Nachteile der vorgestellten Lösung ergeben sich aus dem ungehinderten Zugang zu der Komponente für den Auftragnehmer und der Umständlichkeit, einen Auftrag abzuschicken. Diese Probleme lassen sich jedoch gleich bei dem Modul, das die Aufträge und den Typ für den Auftragnehmer definiert, erledigen.

Eine weitere offene Frage ist natürlich, ob Aufträge auch bearbeitet worden sind bzw. was Auftragnehmer bei unbekannten Aufträgen tun sollen. Ein Erledigungsvermerk als zusätzliche Komponente bei den Aufträgen wäre eine mögliche Lösung. Dann können Auftragnehmer unbekannte Aufträge ignorieren, ohne daß dies unbemerkt bleibt.

Folgendes Beispiel zeigt eine vereinfachte Abstraktion für unstrukturierte Sammlungen von Objekten, die sowohl eine konventionelle Schnittstelle für Klienten anbietet als auch eine auf Auftragnehmer basierende Schnittstelle für Implementierungen. Auf Erledigungsvermerke wird in diesem Beispiel verzichtet.

```
DEFINITION Collections;

   IMPORT Disciplines, Objects;

   TYPE
      Collection = POINTER TO CollectionRec;
      CollectionRec = RECORD (Disciplines.ObjectRec) END;

   (* Schnittstelle fuer Implementierungen *)

   TYPE
      Message = (* Basistyp fuer alle Auftraege an Collections *)
         RECORD END;
      Handler = (* Auftragnehmer *)
         PROCEDURE (collection: Collection; VAR message: Message);

      AddMessage =
         RECORD
            (Message)
            object: Objects.Object;
         END;
      FirstMessage = RECORD (Message) END;
      NextMessage =
         RECORD
            (Message)
            (* Rueckgabewerte *)
```

```
            object: Objects.Object;
            endOfCollection: BOOLEAN;
         END;

   PROCEDURE SetHandler(collection: Collection; handler: Handler);

   (* Schnittstelle fuer Klienten *)

   PROCEDURE Add(collection: Collection; object: Objects.Object);

   PROCEDURE First(collection: Collection);
   PROCEDURE Next(collection: Collection;
                  VAR object: Objects.Object) : BOOLEAN;

END Collections.
```

Wie man an diesem Beispiel sieht, gibt es bei Aufträgen sowohl Parameter, die übergeben werden, als auch Werte, die man zurückerhält. Bei *Next* wird der **RETURN**-Wert explizit durch die Komponente *endOfCollection* bestimmt.

Die Komponente *handler* gehört nicht dem öffentlichen Teil an und ist nur über *SetHandler* veränderbar, die von der jeweiligen Implementierung zu Beginn aufgerufen wird. Das Modul *Collections* ist selbst keine Implementierung mehr in dem Sinne, daß es eine Sammlung von Objekten verwaltet, sondern es dient nur noch dazu, die Operationen in Aufträge zu verwandeln und an die eigentliche Implementierung zu verschicken:

```
MODULE Collections;

   IMPORT Disciplines, Objects;

   TYPE
      Message = (* Basistyp fuer alle Auftraege an Collections *)
         RECORD END;
      Collection = POINTER TO CollectionRec;
      Handler = (* Auftragnehmer *)
         PROCEDURE (collection: Collection; VAR message: Message);
      CollectionRec =
         RECORD
            (Disciplines.ObjectRec)
            (* private Komponenten *)
            handler: Handler;
         END;

   (* Schnittstelle fuer Implementierungen *)

   TYPE
      AddMessage =
         RECORD
            (Message)
            object: Objects.Object;
         END;
      FirstMessage = RECORD (Message) END;
      NextMessage =
         RECORD
            (Message)
            (* Rueckgabewerte *)
            object: Objects.Object;
            endOfCollection: BOOLEAN;
         END;

   PROCEDURE SetHandler(collection: Collection; handler: Handler);
   BEGIN
      collection.handler := handler;
   END SetHandler;

   (* Schnittstelle fuer Klienten *)

   PROCEDURE Add(collection: Collection; object: Objects.Object);
      VAR
         message: AddMessage;
```

```
   BEGIN
      message.object := object;
      collection.handler(collection, message);
   END Add;

   PROCEDURE First(collection: Collection);
      VAR
         message: FirstMessage;
   BEGIN
      collection.handler(collection, message);
   END First;

   PROCEDURE Next(collection: Collection;
                  VAR object: Objects.Object) : BOOLEAN;
      VAR
         message: NextMessage;
   BEGIN
      collection.handler(collection, message);
      IF message.endOfCollection THEN
         RETURN FALSE
      ELSE
         object := message.object;
         RETURN TRUE
      END;
   END Next;

END Collections.
```

Das bereits vorgestellte Modul für lineare Listen wäre ein geeigneter Kandidat für eine mögliche Implementierung von *Collections*. Dabei läßt sich die Schnittstelle von *Lists* radikal verkürzen, da hier nur noch *Create* benötigt wird. Alle anderen Operationen erfolgen dann über Collections:

```
DEFINITION Lists;

   IMPORT Collections;

   PROCEDURE Create(VAR collection: Collections.Collection);

END Lists.
```

Alternativ wäre es auch denkbar, den Typ der Liste zu offenbaren. Dies würde entsprechende Typentests erlauben:

```
DEFINITION Lists;

   IMPORT Collections;

   TYPE
      List = POINTER TO ListRec;
      ListRec = RECORD (Collections.CollectionRec) END;

   PROCEDURE Create(VAR collection: Collections.Collection);

END Lists.
```

Wichtig ist, daß selbst bei Offenlegung des abgeleiteten Typs *Create* den Typ der zugehörigen Abstraktion zurückliefert. Andernfalls wäre ein Anwender gezwungen, die zugehörige Variable oder Komponente mit dem Typ *Lists.List* zu deklarieren. Dies würde in folgendem Beispiel die Einführung von Hilfsvariablen erzwingen:

```
VAR
   collection: Collections.Collection;

(* ... *)

IF (* eine Liste ist vorzuziehen *) THEN
   Lists.Create(collection);
ELSIF (* ein binaerer Baum ist guenstiger *) THEN
   BinTrees.Create(collection);
```

```
(* ... *)
END;
```

Bei der Implementierung von *Lists* kommt im wesentlichen nur *Handler* hinzu, der die bereits bekannten Operationen verwendet, die jetzt privat geworden sind. *Create* hat als lokale Variable eine Liste vom erweiterten Typ, damit **NEW** die gewünschte Erweiterung anlegt. Ansonsten darf bei *Create* nicht die Verknüpfung des neu kreierten Objekts mit dem zugehörigen Auftragnehmer vergessen werden.

```
MODULE Lists;

   IMPORT Collections, Objects;

   TYPE
      Linkable = POINTER TO LinkableRec;
      LinkableRec =
         RECORD
            next: Linkable;
            object: Objects.Object;
         END;

      List = POINTER TO ListRec;
      ListRec =
         RECORD
            (Collections.CollectionRec)
            (* private Komponenten *)
            head, tail: Linkable;
            cursor: Linkable; (* fuer List und Next *)
         END;

   (* Append, First und Next bleiben unveraendert *)

   PROCEDURE Handler(collection: Collections.Collection;
                     VAR message: Collections.Message);
   BEGIN
      WITH collection: List DO
         IF message IS Collections.AddMessage THEN
            WITH message: Collections.AddMessage DO
               Append(collection, message.object);
            END;
         ELSIF message IS Collections.FirstMessage THEN
            First(collection);
         ELSIF message IS Collections.NextMessage THEN
            WITH message: Collections.NextMessage DO
               message.endOfCollection := ~Next(collection, message.object);
            END;
         END;
      END;
   END Handler;

   PROCEDURE Create(VAR collection: Collections.Collection);
      VAR
         list: List;
   BEGIN
      NEW(list);
      list.head := NIL; list.tail := NIL; list.cursor := NIL;
      Collections.SetHandler(list, Handler);
      collection := list;
   END Create;

END Lists.
```

Bemerkenswert ist bei dieser Technik, daß der Kontrollfluß anders als in klassischen objekt-orientierten Sprachen verläuft. Abbildung 20 zeigt, daß bei Oberon zunächst die Operation bei dem Modul aufgerufen wird, das die Abstraktion definiert, und erst danach (möglicherweise sogar nur optional) die zu dem Objekt gehörende Operation durchgeführt wird — eine Technik, die der von BETA ähnelt [Madsen93]. Das steht ganz im Gegensatz zu den klassischen objekt-orientierten Sprachen, bei denen zunächst die zum Objekt gehörende Operation aufgerufen wird, die dann optional die überdefinierte Operation der darüberliegenden Klasse aufrufen kann.

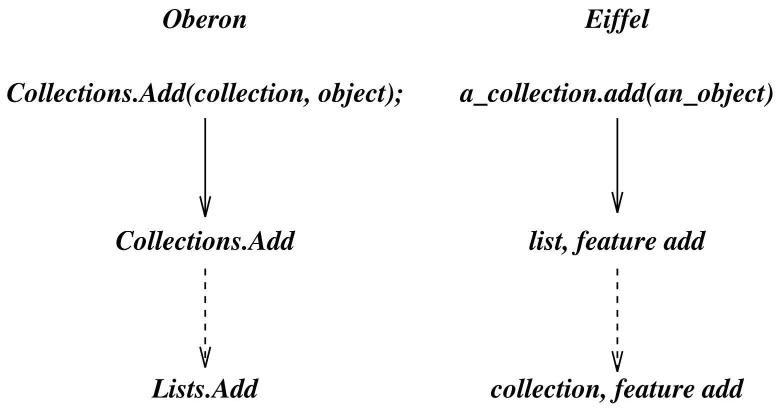

Abb. 20: Unterschiedlicher Kontrollfluß bei Oberon und klassischen OO-Sprachen

Beide Varianten haben ihre Vor- und Nachteile. Sicherlich ist die traditionelle Variante schneller, wenn die Operation der übergeordneten Klasse nicht aufgerufen wird. Andererseits ermöglicht Oberon unterschiedliche Schnittstellen für die Klienten und die Implementierungen. Wenn man Schnittstellen als Protokolle auffaßt, dann kann das Modul, das eine Abstraktion definiert, auch als Protokollkonverter dienen. In vielen Fällen ist dies außerordentlich sinnvoll. Beispiele dafür in der Ulmer Oberon-Bibliothek sind *Streams*, die bei Verwendung eines Cache Aufrufe zu der jeweiligen Implementierung minimieren, und das Modul *Conditions*, das aufgrund der zugehörigen Koordinationsaufgabe völlig unterschiedliche Schnittstellen offeriert.

Ein wichtiger Punkt ist die Erweiterbarkeit von *Collections*. Für das obige Beispiel könnte eine Erweiterung *OrderedCollections* definiert werden:

```
DEFINITION OrderedCollections;

   IMPORT Collections, Objects;

   TYPE
      OrderedCollection = POINTER TO OrderedCollectionRec;
      OrderedCollectionRec =
         RECORD
            (Collections.CollectionRec)
         END;

   (* Schnittstelle fuer Implementierungen *)

   TYPE
      Message = RECORD (Collections.Message) END;
      GetMessage =
         RECORD
            (Message)
            index: INTEGER; (* in-parameter *)
            object: Objects.Object; (* out-parameter *)
         END;

   PROCEDURE SetHandler(collection: Collections.Collection;
                        handler: Collections.Handler);

   (* Schnittstelle fuer Klienten *)

   PROCEDURE Get(collection: OrderedCollection;
                 index: INTEGER; VAR object: Objects.Object);

END OrderedCollections.
```

Bei dieser Lösung muß eine Implementierung von *OrderedCollections* sowohl für *Collections* als auch für *OrderedCollections* einen Auftragnehmer deklarieren. Dies läßt sich vermeiden, wenn man gleich im Modul von *Collections* die Möglichkeit von Erweiterungen in Betracht ziehen würde durch eine Prozedur, die das Versenden beliebiger Erweiterungen von *Message* an Implementierungen unterstützen würde:

```
DEFINITION Collections;

   (* ... analog zu oben ... *)

   (* Schnittstelle fuer Erweiterungen *)

   PROCEDURE Send(collection: Collection; message: Message);

END Collections.
```

Dann benötigt *OrderedCollections* keine Prozedur *SetHandler*, sondern könnte mit *Collections.Send* die Aufträge verschicken. Natürlich gäbe es keine Notwendigkeit für *Send*, wenn die *handler*-Komponente bei *Collections.Collection* öffentlich wäre. Allerdings würden damit die Auftragnehmer der einzelnen Implementierungen öffentlich werden, und es bestünde nicht mehr die Garantie, daß die Auftragnehmer nur eigene Objekte vorfinden würden, d.h. daß ein Laufzeitfehler bei der Typzusicherung, die am Anfang des Auftragnehmers steht, nicht mehr ausgeschlossen werden könnte.

Wenn man sich mit einem einzigen Auftragnehmer begnügt, der nur an einer Stelle angegeben werden muß, dann sind auch Delegationen denkbar, die auch unbekannte Operationen weiterreichen können. Folgendes Beispiel zeigt eine Implementierung von *Collection*, die alle Operationen an zwei andere Sammlungen weiterreicht. Die Implementierung ist nicht symmetrisch, da nur die Rückgabewerte der ersten Sammlung verwendet werden.

```
DEFINITION DribbleCollections;

   IMPORT Collections;

   PROCEDURE Create(VAR dribble: Collections.Collection;
                    primary, secondary: Collections.Collection);

END DribbleCollections.


MODULE DribbleCollections;

   IMPORT Collections, Objects;

   TYPE
      DribbleCollection = POINTER TO DribbleCollectionRec;
      DribbleCollectionRec =
         RECORD
            (Collections.CollectionRec)
            primary, secondary: Collections.Collection;
         END;

   PROCEDURE Handler(collection: Collections.Collection;
                     VAR message: Collections.Message);
   BEGIN
      WITH collection: DribbleCollection DO
         Collections.Send(collection.secondary, message);
         (* zuletzt den Auftrag an die Collection weiterreichen,
            die die endgueltigen Rueckgabewerte bestimmen soll
         *)
         Collections.Send(collection.primary, message);
      END;
   END Handler;

   PROCEDURE Create(VAR dribble: Collections.Collection;
                    primary, secondary: Collections.Collection);
      VAR
         collection: DribbleCollection;
   BEGIN
      NEW(collection);
      collection.primary := primary;
      collection.secondary := secondary;
      Collections.SetHandler(collection, Handler);
   END Create;

END DribbleCollections.
```

Ganz ohne Nebeneffekte verläuft das Verteilen des einen Auftrags an mehrere weitere Implementierungen nicht. Da Aufträge als Variablenparameter übergeben werden, könnte der erste Auftragnehmer den Auftrag in einer Weise verändern, so daß der zweite Auftragnehmer anders reagiert, als wenn er den ursprünglichen Auftrag erhalten hätte. Eine Duplikation der Aufträge ist nicht möglich, da der dynamische Typ eines Auftrags variabel und unbekannt ist.

Ein weiteres Problem ergibt sich bei *OrderedCollections.Get*. Während es auf den ersten Blick zweckmäßig erscheint, eine Erweiterung von *OrderedCollections* zu verlangen, ist dies bei möglichen Delegationen hinderlich, da die zwischengelagerten *Collections* nicht Erweiterungen beliebiger anderer Erweiterungen von *Collections.Collection* sind. Wenn andererseits *OrderedCollections.Get* nicht mehr eine *OrderedCollection* verlangt und sich auch mit einer *Collection* begnügt, so vergrößert sich natürlich die Gefahr, daß die Empfänger die speziellen Nachrichten der Erweiterungen nicht verstehen und deswegen auch nicht bearbeiten. Erhöhte Flexibilität steht hier im Gegensatz zu der Sicherheit statischer Zusicherungen.

2.7. Operationen ohne zentrale Auftragnehmer

Der vorgestellte Ansatz ist zwar recht flexibel bis zur Unterstützung der Delegation selbst unbekannter Operationen, jedoch verbleiben einige Nachteile, die auch beispielsweise von Mössenböck aufgezählt werden [Mössenböck93, S. 74]:

- Es ist unklar, welche Operationen von einem Objekt bearbeitet werden können und welche unbekannt sind.
- Zwischen dem Aufruf der Operation und der eigentlichen Durchführung liegen zwei andere Prozeduren: eine, die den Auftrag verpackt, und ein zentraler Auftragnehmer, der Aufträge auf weitere lokale Prozeduren verteilt.

Das war einer der Beweggründe für Mössenböck, typgebundene Prozeduren, die in etwa den Methoden aus den klassischen objekt-orientierten Sprachen entsprechen, in Oberon-2 einzuführen. Allerdings läßt sich dieses Problem auch in Oberon lösen durch die Einführung von Schnittstellenverbunden. Wie sich zeigt, geben sie die notwendige Sicherheit und Effizienz und weisen sogar einige Vorteile gegenüber der klassischen Lösung auf.

Als Beispiel läßt sich die Abstraktion für *Collections* nehmen:

```
DEFINITION Collections;

   IMPORT Disciplines, Objects;

   TYPE
      Collection = POINTER TO CollectionRec;
      CollectionRec = RECORD (Disciplines.ObjectRec) END;

   (* Schnittstelle fuer Implementierungen *)

   TYPE
      AddProc = PROCEDURE (collection: Collection; object: Objects.Object);
      FirstProc = PROCEDURE (collection: Collection);
      NextProc = PROCEDURE (collection: Collection;
                            VAR object: Objects.Object) : BOOLEAN;
      Interface = POINTER TO InterfaceRec;
      InterfaceRec =
         RECORD
            add: AddProc;
            first: FirstProc;
            next: NextProc;
         END;

   PROCEDURE Init(collection: Collection; if: Interface);

   (* Schnittstelle fuer Klienten *)

   PROCEDURE Add(collection: Collection; object: Objects.Object);

   PROCEDURE First(collection: Collection);
   PROCEDURE Next(collection: Collection;
                  VAR object: Objects.Object) : BOOLEAN;
```

```
END Collections.
```

Collections.Add würde dann folgendermaßen aussehen:

```
TYPE
   Collection = POINTER TO CollectionRec;
   CollectionRec =
      RECORD
         (Disciplines.ObjectRec)
         (* private Komponenten *)
         if: Interface;
      END;

(* ... *)

PROCEDURE Add(collection: Collection; object: Objects.Object);
BEGIN
   collection.if.add(collection, object);
END Add;
```

Bei einer Implementierung von *Collections* wird damit der zentrale Auftragnehmer überflüssig und die der Operation entsprechende Prozedur wird direkt aufgerufen. Zudem besteht die Sicherheit, daß alle Operationen auch bekannt sind. Nur wenn die Implementierung vergißt, *Collections.Init* beim Anlegen eines neuen Objekts aufzurufen, oder nicht alle Prozedurkomponenten initialisiert, kann es zu Laufzeitfehlern kommen. Allerdings haben solche Fehler kaum eine Chance, einen allerersten Test zu überstehen.

Wie bereits erwähnt wurde, hat das Einschieben eines Moduls zwischen dem Klienten und der eigentlichen Implementierung sowohl Vor- als auch Nachteile. Ein Vorteil besteht darin, daß beispielsweise eine Implementierung erklären kann, daß sie nicht alle Operationen unterstützt. Wie bereits im Beispiel des *iostream* bei C++ erläutert wurde, ist es nicht sinnvoll, für jede mögliche Variante eine eigene Klasse zu definieren. Statt dessen sind neben einigen Pflichtoperationen eine Reihe von optionalen Fähigkeiten denkbar, die dann auch von den Klienten überpüfbar sind, so daß Überraschungen vermieden werden können. Beim Beispiel der *Collections* könnte man beispielsweise *OrderedCollections* in die Basisabstraktion integrieren:

```
DEFINITION Collections;

   IMPORT Disciplines, Objects;

   TYPE
      Collection = POINTER TO CollectionRec;
      CollectionRec = RECORD (Disciplines.ObjectRec) END;

   (* Schnittstelle fuer Implementierungen *)

   TYPE
      AddProc = PROCEDURE (collection: Collection; object: Objects.Object);
      FirstProc = PROCEDURE (collection: Collection);
      NextProc = PROCEDURE (collection: Collection;
                             VAR object: Objects.Object) : BOOLEAN;
      GetProc = PROCEDURE (collection: Collection;
                           index: INTEGER; VAR object: Objects.Object);
      SetProc = PROCEDURE (collection: Collection;
                           index: INTEGER; object: Objects.Object);
      Interface = POINTER TO InterfaceRec;
      InterfaceRec =
         RECORD
            add: AddProc;
            first: FirstProc;
            next: NextProc;
            get: GetProc;
            set: SetProc;
         END;
   CONST
      get = 0; set = 1;
   TYPE
      Capability = SHORTINT; (* get..set *)
      CapabilitySet = SET; (* OF Capability *)
```

```
   PROCEDURE Init(collection: Collection; if: Interface; caps: CapabilitySet);

   (* Schnittstelle fuer Klienten *)

   PROCEDURE Capabilities(collection: Collection) : CapabilitySet;

   PROCEDURE Add(collection: Collection; object: Objects.Object);

   PROCEDURE First(collection: Collection);
   PROCEDURE Next(collection: Collection;
                  VAR object: Objects.Object) : BOOLEAN;

   PROCEDURE Get(collection: Collection;
                 index: INTEGER; VAR object: Objects.Object);
   PROCEDURE Set(collection: Collection;
                 index: INTEGER; object: Objects.Object);

END Collections.
```

Damit gehören *Add*, *First* und *Next* zu den Pflichtoperationen, während *Get* und *Set* optional sind. Die Prozedur *Capabilities* erlaubt es Klienten, nachzufragen, welche der optionalen Fähigkeiten vorhanden sind. *Collections.Get* und *Collections.Set* würden dann explizit das Vorhandensein der entsprechenden Fähigkeit überprüfen — die Notwendigkeit von Dummy-Prozeduren auf der Seite der Implementierung entfällt:

```
TYPE
   Collection = POINTER TO CollectionRec;
   CollectionRec =
      RECORD
         (Disciplines.ObjectRec)
         (* private Komponenten *)
         if: Interface;
         caps: CapabilitySet;
      END;

(* ... *)

PROCEDURE Get(collection: Collection;
              index: INTEGER; VAR object: Objects.Object);
BEGIN
   IF get IN collection.caps THEN
      collection.if.get(collection, index, object);
   ELSE
      (* Laufzeitfehler *)
   END;
END Get;
```

Es wäre auch denkbar, alle Operationen der Schnittstelle in die Menge der Fähigkeiten aufzunehmen, auch wenn einige davon nicht optional sind. *Collections.Init* könnte dann überprüfen, ob die Menge der Pflichtoperationen dabei ist. Diese Technik würde einer möglichen Implementierung mehr Sicherheit geben, da das Übereinstimmen der zugewiesenen Operationen beim Schnittstellenverbund mit der Menge der Fähigkeiten sich leicht lokal überprüfen läßt. Wenn später *Collections* um eine weitere Pflichtoperation erweitert wird, führt dies sofort zu einem Laufzeitfehler, selbst wenn die neue Pflichtoperation noch nicht benutzt wird.

Collections als Mittler zwischen Klienten und Implementierungen kann auch beiden Seiten individuell entgegenkommen, um einerseits Umständlichkeiten zu ersparen und andererseits besondere Effizienzsteigerungen zu ermöglichen. Ein Beispiel wäre das Hinzufügen nicht nur von einzelnen Elementen, sondern kompletten Sammlungen. Die vorgestellten Varianten würden in jedem Fall dem Aufrufer die Last aufbürden, eine Schleife zu schreiben, die die Elemente einzeln von der einen Sammlung in die andere befördert. Dabei könnten zumindest manche Implementierungen davon profitieren, wenn sie wüßten, wieviele Elemente auf einmal einzufügen sind. Andererseits wäre es natürlich sehr umständlich, wenn jede Implementierung eine Operation unterstützen müßte, die es erlaubt, vollständige Sammlungen hinzuzufügen. Folgende Änderung von *Collections* zeigt, daß man beide Seiten zufriedenstellen kann:

```
DEFINITION Collections;

   IMPORT Disciplines, Objects;

   TYPE
      Collection = POINTER TO CollectionRec;
      CollectionRec = RECORD (Disciplines.ObjectRec) END;

   (* Schnittstelle fuer Implementierungen *)

   TYPE
      AddProc = PROCEDURE (collection: Collection; object: Objects.Object);
      AddCollectionProc = PROCEDURE (collection, objects: Collection);
      (* ... weitere Prozedurtypen analog zu oben ... *)
      Interface = POINTER TO InterfaceRec;
      InterfaceRec =
         RECORD
            add: AddProc;
            addcollection: AddCollectionProc;
            first: FirstProc;
            next: NextProc;
            get: GetProc;
            set: SetProc;
         END;
   CONST
      addcollection = 0; get = 1; set = 2;
   TYPE
      Capability = SHORTINT; (* addcollection..set *)
      CapabilitySet = SET; (* OF Capability *)

   PROCEDURE Init(collection: Collection; if: Interface; caps: CapabilitySet);

   (* Schnittstelle fuer Klienten *)

   PROCEDURE Capabilities(collection: Collection) : CapabilitySet;

   PROCEDURE Add(collection: Collection; object: Objects.Object);

   PROCEDURE AddCollection(collection: Collection; objects: Collection);

   PROCEDURE First(collection: Collection);
   PROCEDURE Next(collection: Collection;
                  VAR object: Objects.Object) : BOOLEAN;

   PROCEDURE Get(collection: Collection;
                 index: INTEGER; VAR object: Objects.Object);
   PROCEDURE Set(collection: Collection;
                 index: INTEGER; object: Objects.Object);

END Collections.
```

Collections.AddCollection würde dann folgendermaßen aussehen:

```
PROCEDURE AddCollection(collection: Collection; objects: Collection);
   VAR
      object: Objects.Object;
BEGIN
   IF addcollection IN collection.caps THEN
      collection.if.addcollection(collection, objects);
   ELSE
      First(objects);
      WHILE Next(objects, object) DO
         Add(collection, object);
      END;
   END;
END AddCollection;
```

Dies entspricht effektiv der Technik des Überdefinierens von Methoden bei klassischen objekt-orientierten Sprachen. Diese lassen sich auch bei Erweiterungshierarchien sukzessiv überdefinieren. So könnte eine

Erweiterung von *Collections* bei der Initialisierung einer Sammlung die Initialisierung der Basisabstraktion übernehmen und dabei optional einzelne Operationen selbst definieren.

Zwar liefert diese Technik einen Zuwachs an Sicherheit — es bleibt jedoch die Frage, ob sich Delegationen immer noch realisieren lassen. Zunächst ist es sicherlich kein Problem, die Operationen der Basisabstraktion durchzureichen, wie die angepaßte Version der *DribbleCollections* zeigt:

```
MODULE DribbleCollections;

   IMPORT Collections, Objects;

   TYPE
      DribbleCollection = POINTER TO DribbleCollectionRec;
      DribbleCollectionRec =
         RECORD
            (Collections.CollectionRec)
            primary, secondary: Collections.Collection;
            secondaryCaps: Collections.CapabilitySet;
         END;
   VAR
      if: Collections.Interface;

   PROCEDURE Add(collection: Collections.Collection;
                 object: Objects.Object);
   BEGIN
      WITH collection: DribbleCollection DO
         Collections.Add(object, primary);
         Collections.Add(object, secondary);
      END;
   END Add;

   (* analog First, Next, Set und Get,
      wobei bei First, Next und Get darauf verzichtet werden kann,
      die Operation auch fuer secondary durchzufuehren
   *)

   PROCEDURE Init;
   BEGIN
      NEW(if);
      if.add := Add; if.first := First; if.next := Next;
      if.get := Get; if.set := Set;
   END Init;

   PROCEDURE Create(VAR dribble: Collections.Collection;
                    primary, secondary: Collections.Collection);
      VAR
         collection: DribbleCollection;
         caps: Collections.CapabilitySet;
   BEGIN
      NEW(collection);
      collection.primary := primary;
      collection.secondary := secondary;
      (* wir unterstuetzen hier die Schnittmenge der Faehigkeiten
         von beiden Sammlungen
      *)
      caps := Collections.Capabilities(primary) *
              Collections.Capabilities(secondary);
      Collections.Init(collection, if, caps);
   END Create;

BEGIN
   Init;
END DribbleCollections.
```

Bemerkenswert ist hier das Konzept der Fähigkeiten. *DribbleCollections* kann sich individuell an die gegebenen Sammlungen anpassen. Im Beispiel wird die Schnittmenge verwendet. Es wäre natürlich auch denkbar, genau die Fähigkeiten von *primary* durchzureichen und die Operationen bei *secondary* nur soweit nachzuziehen, wie sie unterstützt werden. Der Grund für die Trennung des Schnittstellenverbunds von der Menge der Fähigkeiten ergibt sich ganz deutlich: Der Schnittstellenverbund bleibt konstant und braucht nur ein einziges Mal angelegt zu werden, während die Menge der Fähigkeiten von Fall zu Fall variieren kann.

Wenn die Operationen unbekannter Erweiterungen unterstützt werden sollen, dann bleibt nichts anderes übrig, als dafür wieder das Konzept der Aufträge einzuführen. Allerdings dürften die Operationen der Erweiterungen möglicherweise nicht so häufig auftreten wie die der Basisabstraktion, so daß sich der damit verbundene Aufwand in Grenzen hält. So könnte *Collections* folgendermaßen ergänzt werden:

```
DEFINITION Collections;

   (* ... *)

      Message = RECORD END;
      HandlerProc = PROCEDURE (collection: Collection; VAR message: Message);

      Interface = POINTER TO InterfaceRec;
      InterfaceRec =
         RECORD
            add: AddProc;
            addcollection: AddCollectionProc;
            first: FirstProc;
            next: NextProc;
            get: GetProc;
            set: SetProc;
            handler: HandlerProc;
         END;
   CONST
      addcollection = 0; get = 1; set = 2; handler = 3;
   TYPE
      Capability = SHORTINT; (* addcollection..handler *)
      CapabilitySet = SET; (* OF Capability *)

   (* ... *)

   PROCEDURE Send(collection: Collection; message: Message);

END Collections.
```

Ein Auftragnehmer braucht nur dann von einer Implementierung definiert zu werden, wenn sie eine Erweiterung unterstützt, die Aufträge verschickt, oder wenn wie etwa im Beispiel von *DribbleCollections* unbekannte Operationen weitergereicht werden sollen.

2.8. Disziplinen

Das Modul *Disciplines* ermöglicht es, an Erweiterungen von *Disciplines.Object* beliebige weitere Datenstrukturen, die Erweiterungen von *Disciplines.Discipline* sind, anzuhängen und wiederzufinden:

```
DEFINITION Disciplines;

   IMPORT Objects;

   TYPE
      Identifier = LONGINT;

      Discipline = POINTER TO DisciplineRec;
      DisciplineRec =
         RECORD
            (Objects.ObjectRec)
            id: Identifier;
         END;

      Object = POINTER TO ObjectRec;
      ObjectRec =
         RECORD
            (Objects.ObjectRec)
         END;

   PROCEDURE Unique() : Identifier;
```

```
    PROCEDURE Add(object: Object; discipline: Discipline);
    PROCEDURE Remove(object: Object; id: Identifier);
    PROCEDURE Seek(object: Object; id: Identifier;
                   VAR discipline: Discipline) : BOOLEAN;

END Disciplines.
```

Für jeden Typ, der von *Disciplines.Discipline* abgeleitet wird, sollte zum Zeitpunkt der Initialisierung des zugehörigen Moduls eine eindeutige Zahl mit Hilfe von *Disciplines.Unique* bestimmt werden. Später wird dann bei dem Anlegen der entsprechenden Datenstruktur die Komponente *id* damit belegt. Nachdem die Datenstruktur mit *Add* an das Objekt angeheftet worden ist, kann sie später durch Angabe dieser Zahl mit *Seek* wiedergefunden oder mit *Remove* entfernt werden.

Als Beispiel für die Verwendung der Disziplinen eignet sich das Problem der nachträglich mit Farben zu versehenden Figuren aus [Stein89], das als offenes Problem im Bereich der objekt-orientierten Sprachen geschildert wird. Bei herkömmlichen Techniken bleibt tatsächlich nur die Erweiterung der Basisklasse um die entsprechende Farbkomponente oder die Definition einer weiteren Hierachie übrig, bei der sich zu jeder Erweiterung eine Farbvariante gesellt. Mit Hilfe der Disziplinen ist es jedoch möglich, an jede Erweiterung von *Figure* eine Farbinformation anzuheften:[1]

```
DEFINITION Colours;

   IMPORT Figures;

   TYPE
      Colour = INTEGER;

   PROCEDURE SetColour(figure: Figures.Figure; colour: Colour);
   PROCEDURE GetColour(figure: Figures.Figure; VAR colour: Colour);

END Colours.


MODULE Colours;

   IMPORT Disciplines, Figures;

   CONST
      defaultColour = 0;

   TYPE
      Colour = INTEGER;

      ColourDiscipline = POINTER TO ColourDisciplineRec;
      ColourDisciplineRec =
         RECORD
            (Disciplines.DisciplineRec)
            colour: Colour;
         END;
   VAR
      colourDiscId: Disciplines.Identifier;

   PROCEDURE SetColour(figure: Figures.Figure; colour: Colour);
      VAR
         colourDisc: ColourDiscipline;
   BEGIN
      IF Disciplines.Seek(figure, colourDiscId, colourDisc) THEN
         colourDisc.colour := colour;
      ELSE
         NEW(colourDisc); colourDisc.id := colourDiscId;
         colourDisc.colour := colour;
         Disciplines.Add(figure, colourDisc);
      END;
   END SetColour;
```

[1] Das folgende Beispiel setzt natürlich voraus, daß *Figures.Figure* eine Erweiterung von *Disciplines.Object* ist.

```
    PROCEDURE GetColour(figure: Figures.Figure; VAR colour: Colour);
       VAR
          colourDisc: ColourDiscipline;
    BEGIN
       IF Disciplines.Seek(figure, colourDiscId, colourDisc) THEN
          colour := colourDisc.colour;
       ELSE
          colour := defaultColour;
       END;
    END GetColour;

 BEGIN
    colourDiscId := Disciplines.Unique();
 END Colours.
```

Natürlich ist es auch möglich, Auftragnehmer oder Schnittstellenverbunde mit Hilfe von *Disciplines* an ein Objekt anzuhängen. Allerdings werden in dieser Form weder nicht überdefinierbare Operationen (wie die Operationen von *Colours*) noch ggf. die Operationen, die auf Basis von weiteren Schnittstellenverbunden oder Auftragnehmern definiert sind, von Delegationen erfaßt.

Es hilft jedoch nicht, die Operationen auf Basis von Aufträgen durchzuführen, die über die Operation *Send* der Basisabstraktion verschickt werden. Hierbei landet der Auftrag bei der Implementierung, die von den Disziplinen und deren Operationen nichts weiß. Eine Lösung für dieses Problem wird von dem Modul *Messages* der Ulmer Oberon-Bibliothek angeboten. Hierbei werden Aufträge an alle Interessenten weitergegeben, so daß auch Module, die nur über Disziplinen an einem Objekt beteiligt sind, ihre Aufträge über einen delegierbaren Zugang erhalten. Da die Aufträge von *Messages* persistent sind, d.h. in eine Sequenz von Bytes und wieder zurück verwandelt werden können, sind damit Aufrufe über die Grenzen eines Programms hin delegierbar (*remote procedure calls*). Bemerkenswert ist hier, daß weder der Aufrufer noch die Prozedur, die den Auftrag zusammenstellt, davon etwas bemerken.

2.9. Zusammenfassung

Es ist unzweifelhaft, daß Oberon eine problemgerechte Modularisierung unterstützt, die es erlaubt, die Sichtbarkeitsgrenzen entsprechend den vorgestellten Anforderungen zu ziehen. So können Datentypen, die zusammengehören, auch zusammen in einem Modul deklariert werden; private Datenstrukturen bleiben privat und blähen nicht die gesamte Klassenhierarchie auf. Darüber hinaus erlaubt das Konzept der Typerweiterung die Trennung eines Verbundes in einzelne private Bereiche, für die verschiedene Module verantwortlich sind.

Oberon unterstützt objekt-orientierte Konzepte nicht unmittelbar. Zunächst existiert eine feste Bindung zwischen einer Schnittstelle und der zugehörigen Implementierung. Allerdings lassen sich bei Verwendung entsprechender Konventionen alle zuvor zusammengefaßten wichtigen Konzepte realisieren. Obwohl die dynamisch zu bindenden Operationen durch die Verwendung von Prozedurvariablen realisiert werden, läßt sich das gleiche Maß an Sicherheit erreichen wie bei den Sprachen, die dies direkt unterstützen:

- Die Prozedurvariablen sind nicht öffentlich zugänglich. Damit können sie weder anderen Objekten zugeordnet werden, noch ist es möglich, sie für ein Objekt zu ändern. Entsprechend ist auch sichergestellt, daß die dynamisch gebundenen Operationen auch genau das Objekt vorfinden, das sie erwarten.
- Bei der Technik mit den Schnittstellenverbunden und dem Konzept der Fähigkeiten ist Klarheit darüber gegeben, welche Operationen zur Verfügung stehen. Bei den nicht optionalen Operationen ist dies statisch festgelegt.
- Durch Erledigungsvermerke bei den Aufträgen läßt sich sicherstellen, daß der Aufrufer nachprüfen kann, ob eine Operation bearbeitet worden ist.

Die Konventionen sind nur im Bereich der Abstraktionen und zum Teil bei den Implementierungen relevant. Für Klienten besteht praktisch kein Unterschied zwischen Oberon und klassischen objekt-orientierten Sprachen. Die einzige Ausnahme ist durch die Notation gegeben, da das Objekt nur ein normaler Parameter einer normalen Prozedur ist. Dies erhöht allerdings die Lesbarkeit durch die Angabe des Modulnamens und wurde aus ähnlichen Gründen auch in Trellis/Owl verwendet [Schaffert86].

Während sich mit den vorgestellten Konventionen auch die Techniken, die Deutsch vorgestellt hat [Deutsch89], realisieren lassen, so ist die durch die klassischen objekt-orientierten Sprachen favorisierte Technik des inkrementellen Überdefinierens nur mit unverhältnismäßig hohen Aufwand realisierbar. Hierfür ist jedoch eine zweite Hierarchie von aufeinander aufbauenden Implementierungen geeigneter, die auf normalen

Importbeziehungen beruht. Dies wird zudem von Oberon in idealer Weise unterstützt.

Bemerkenswert ist, daß die Techniken von Oberon über die klassischer objekt-orientierter Sprachen hinausgehen. So sind Delegationen realisierbar und mit der Unterstützung einiger Bibliotheksmodule auch dynamisch hinzufügbare Erweiterungen, Persistenz und die Übermittlung von Operationen an andere Prozesse.

Wäre man bestrebt, alle besprochenen Konzepte zum integralen Bestandteil einer Programmiersprache zu machen, dann würde wahrscheinlich die Tragödie, die Gutknecht beschrieben hat [Gutknecht93a], unvermeidlich sein. Statt einer ökonomischen Sprache, die sich nur aus wenigen Grundkonzepten zusammensetzt, wäre dann eine Sprache notwendig, bei der neben der Komplexität der einzelnen Konzepte noch die kombinatorische Komplexität hinzukommen würde. Dies wurde in Oberon sorgfältig vermieden. So stehen Klassen nicht in Konflikt zu Modulen und Methoden nicht in Konflikt zu Prozedurtypen.

Somit stellt Oberon einen Kompromiß dar, der einerseits einige Konventionen fordert, zum anderen jedoch damit die vorgestellten Anforderungen weit besser erfüllt als klassische objekt-orientierte Programmiersprachen. Vergleichbare Konzepte finden sich nur in hybriden Sprachen (z.B. Modula-3), die jedoch gegen das gängige Design-Konzept für Programmiersprachen verstoßen, nachdem es für ein Problem maximal ein Sprachkonstrukt geben sollte.

All the components of a library should proceed from an overall coherent design, and follow a set of systematic, explicit and uniform conventions.
[Meyer94]

3. Die Ulmer Oberon-Bibliothek

3.1. Überblick

Die Ulmer Oberon-Bibliothek dient in erster Linie als Spracherweiterung von Oberon und bietet bislang im wesentlichen nur relativ fundamentale Konzepte an. Im Gegensatz zu Klassenbibliotheken anderer objektorientierter Sprachen, die häufig nur Abstraktionen für Datenstrukturen oder Benutzerschnittstellen zur Verfügung stellen, konzentriert sich die Ulmer Oberon-Bibliothek auf folgende Bereiche:

- Unterstützung zentraler Mechanismen wie z.B. Disziplinen, Ereignisse, Ausnahmenbehandlungen, Aufträge und Persistenz.
- Definition von allgemein notwendigen Abstraktionen wie z.B. Ein- und Ausgabe.
- Definition von Kooperationsmechanismen, die alle bekannten Konfliktmöglichkeiten bezüglich Raum, Zeit und den Systemschnittstellen auflösen.
- Schnittstellen zu dem vom Übersetzer generierten Code. Auf diese Weise werden Laufzeitfehler zu von der Bibliothek erfaßten Ereignissen, **HALT** führt zur Ausführung einer entsprechenden Prozedur in der Bibliothek, und alle Module und Kommandos (parameterlose exportierte Prozeduren) können über die Bibliothek erfaßt werden.
- Schnittstellen zum UNIX-System, die die Mechanismen der Ausnahmenbehandlungen unterstützen und kooperativ verwendet werden können. Für große Teilbereiche gibt es portable Abstraktionen, so daß hier ein direkter Zugriff auf das System nicht notwendig ist.

Für Abstraktionen zur Unterstützung von Datenstrukturen gab es bislang ausgesprochen geringen Bedarf. Alle Datenstrukturen wurden bislang intern in den einzelnen Modulen individuell auf konventionelle Weise modelliert und kodiert. Eine solche Lösung ist nicht nur sehr effizient, sondern auch sehr flexibel und gut lesbar. Dies entspricht den Erfahrungen, die beim Züricher Oberon-Projekt gemacht worden sind [Wirth89e].

Natürlich wären auch Abstraktionen für Benutzerschnittstellen interessant gewesen. Jedoch empfiehlt es sich, zunächst eine Basis für alle Grundmechanismen zu entwickeln, damit nicht für jeden Bereich alle Räder neu erfunden werden müssen. Die in C geschriebenen X-Windows verdeutlichen dies ganz besonders: zahllose allgemeine Mechanismen (wie z.B. Ereignisse, das Warten auf Ereignisse, Interprozeßkommunikation, Konfiguration, Parametrisierung) wurden hier speziell für X-Windows entwickelt, so daß sie mit Sicherheit inkompatibel zu anderen parallel entwickelten Bibliotheken mit analogen Bedürfnissen sind.

3.2. Der Aufbau der Bibliothek

Bei traditionellen Bibliotheken residieren in der untersten Stufe der Importhierarchie die systemabhängigen Module, und darauf werden die systemunabhängigen Abstraktionen aufgebaut. Dieses Prinzip wurde bei der Ulmer Oberon-Bibliothek genau umgedreht (siehe Abbildung 21): ganz unten sind die allgemeinen Abstraktionen und systemunabhängige Implementierungen, die nur Module aus dem gleichen Bereich importieren. Die UNIX-Schnittstelle baut darauf auf und übernimmt dabei beispielsweise die Abstraktionen für Ausnahmenbehandlungen und Ereignisse aus der darunterliegenden Ebene. Allerdings implementiert die Schnittstelle zu UNIX noch keine der allgemeinen Abstraktionen (wie z.B. für Ein- und Ausgabe). Statt dessen wird nur auf konventionelle Weise ein für Oberon angenehmer Zugang offeriert. Erst die darüberliegende Ebene implementiert einige der Abstraktionen aus der untersten Ebene speziell für UNIX. Während die UNIX-Schnittstelle auch alle speziellen Eigenheiten der lokalen UNIX-Variante offenlegt, wurde bei der darüberliegenden Ebene sehr viel Wert darauf gelegt, daß die Schnittstellen im gesamten UNIX-Bereich weitgehend portabel sind. Die Bibliothek existiert zur Zeit auf zwei UNIX-Varianten (System V Release 3 und SunOS 4.1) und unterscheidet sich nur zu geringen Teilen in der mittleren Ebene.

Alle Abstraktionen der Basis sehen — soweit es sinnvoll ist — eine systemspezifische Implementierung vor und offerieren sie über globale Variablen. Beispiele dafür sind *Streams.stdin*, *Streams.stdout* und *Clocks.system*. Zunächst werden diese Variablen von den zugehörigen Modulen initialisiert mit Dummy-Implementierungen (bei *Streams* sind dieses *nulldev*-Implementierungen, die beliebig viele Ausgaben akzeptieren und immer Eingabeende zurückliefern; bei *Clocks* ist es eine statische Uhr, die nicht von selbst geht). Später steht es den systemspezifischen Implementierungen frei, zu ihrer Initialisierungszeit diesen globalen Variablen ihre

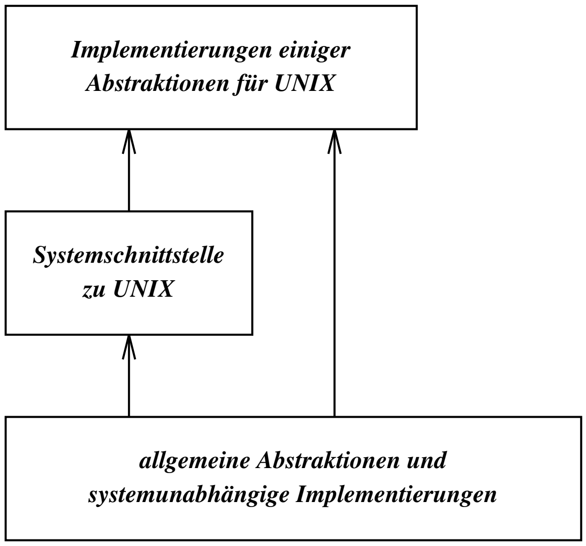

Abb. 21: Grundstruktur der Ulmer Oberon-Bibliothek

Implementierung zuzuweisen. Auf diese Weise ist es im idealen Fall nicht mehr notwendig, daß eine Applikation systemspezifische Module importieren muß, da sie über die Variablen der Basisabstraktionen auf systemunabhängige Weise auf das System zugreifen kann.

Natürlich ist es dafür notwendig, daß die systemspezifischen Implementierungen auch tatsächlich initialisiert werden. Dafür sind verschiedene Techniken denkbar. Zur Zeit wird diese Aufgabe von der Startroutine des Laufzeitsystems übernommen. Später soll dies explizit als Option beim Binden angegeben werden können.

Abbildung 22 zeigt alle Ebenen der Bibliothek. Im Vergleich zum Grundaufbau sind noch einige Ebenen hinzugekommen, um die Module der Bibliothek noch genauer entsprechend ihren Abhängigkeiten sortieren zu können:

base Allgemeine Abstraktionen und Mechanismen mitsamt den systemunabhängigen Implementierungen.

fp Konstantensammlung, die Informationen über die Implementierung der Gleitkommazahlen auf dem aktuellen Prozessor gibt. Weiter sind Schnittstellen zu einigen Gleitkommaprozessoren vorhanden, die den Zugang zu speziellen Operationen ermöglichen, die beispielsweise für mathematische Funktionen interessant sind. Jede dieser Schnittstellen exportiert eine **BOOLEAN**-Konstante *available*. Damit kann mit konstanten **BOOLEAN**-Ausdrücken entschieden werden, ob eine dieser Schnittstellen zur Verfügung steht, oder ob man einen ‘‘handkodierten’’ Algorithmus benötigt. Da konstante **BOOLEAN**-Ausdrücke bereits zur Übersetzzeit bearbeitet werden können, wird zumindest bei einem optimierenden Übersetzer nur der benötigte Teil generiert.

fpbase Hier sind systemunabhängige Implementierungen, die die Konstanten aus *fp* benötigen und ggf. auch auf die Schnittstellen der Gleitkommazahlenprozessoren zugreifen.

if Hier sind alle Module enthalten, die dem Übersetzer bekannt sind. Darunter fallen die Behandlung von Laufzeitfehlern, die Datenstrukturen mit allen Modulen und Kommandos, die Schnittstellen zur Speicherverwaltung und einige globale Variablen im Zusammenhang mit Koroutinen.

tasks Dieser Bereich enthält alle allgemeinen Abstraktionen und systemunabhängigen Implementierungen, die vom Ulmer Koroutinenschema abhängen. Da das Modul aus *if* benötigt wird, das einige globale Variablen zu Koroutinen unterhält, gibt es eine Abhängigkeit von *tasks* zu *if*.

conf Hier sind zwei Module enthalten, die die Parameter der aktuellen Installation repräsentieren. Sie werden während der Installation generiert.

sys In diesem Bereich sind alle Module enthalten, die die Schnittstellen zu dem System offerieren. Ferner befinden sich hier auch die Module für die Speicherverwaltung.

unix Hier sind alle Module, die Abstraktionen aus den unteren Ebenen auf Basis der UNIX-Schnittstelle implementieren.

Das Ulmer Oberon-System kommt mit einem vergleichsweise sehr geringem Umfang an Laufzeitunterstützung aus, die in Assembler geschrieben ist. In Assembler mußte nur der Programmstart und die UNIX-

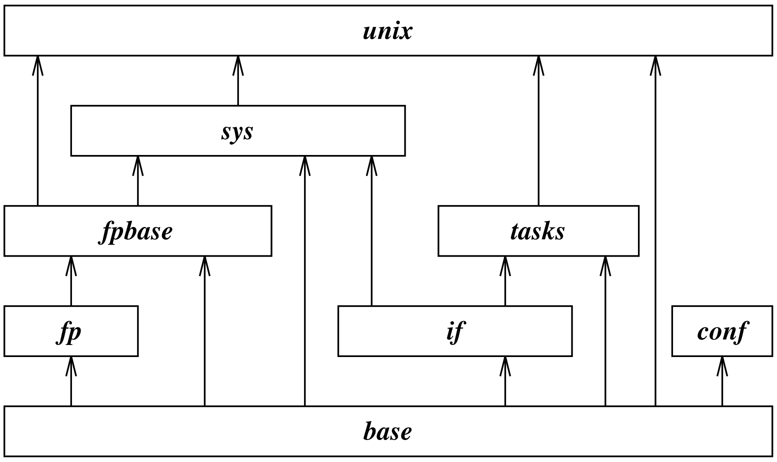

Abb. 22: Importbeziehungen zwischen den Ebenen

Signalbehandlung realisiert werden. Zusammen waren dafür ca. 400 Zeilen Assemblertext notwendig, wobei hier ein sehr hoher Anteil an Kommentaren dabei ist. Dieser Umfang liegt deutlich unter dem von C-Bibliotheken, selbst wenn man die Teile nicht mitrechnet, die nur aus Effizienzgründen in Assembler geschrieben worden sind.

Dies ist nur durch die Verlagerung vieler Aufgaben in die in Oberon geschriebene Bibliothek möglich. So werden die Behandlung von Laufzeitfehlern, die Adreßraumverwaltung, die Speicherverwaltung und die Systemaufrufe von der Bibliothek realisiert.

3.3. Stilistische Konventionen

3.3.1. Namenskonventionen

Auf den ersten Blick mögen Namenskonventionen zweitrangig erscheinen, jedoch erweisen sie sich gerade bei größeren Systemen als ausgesprochen wertvoll, da sie die Lesbarkeit deutlich erhöhen. Folgende Namenskonventionen folgen teilweise einigen Traditionen aus Modula-2 und wurden in Bezug auf Oberon ein wenig ergänzt:

- Module, Typen und Prozeduren beginnen mit einem Großbuchstaben. Konstanten, Variablen, Parameter und Komponenten eines Verbundes beginnen mit einem Kleinbuchstaben. Bei Namen, die aus mehreren Wörtern bestehen, beginnen die weiteren Wörter jeweils mit einem Großbuchstaben.
- Module erhalten Hauptwörter als Namen. Dabei sollte das Hauptwort in Plural stehen, wenn es sich um Module handelt, die Abstraktionen definieren, für die mehrere Implementierungen denkbar sind, oder die sich um einen abstrakten Datentyp ranken. Beispiele: *Streams* (allgemeine Abstraktion mit vielen Implementierungen) und *Memory* (Modul mit genau einer Implementierung, das den Adreßraum verwaltet).
- Abstrakte Datentypen erhalten typischerweise die Singularform des zugehörigen Moduls. Zeigertypen erhalten keine besondere Endung. Verbundtypen, die zu einem Zeigertyp gehören, übernehmen den Namen des Zeigertyps und erhalten die Endung *Rec*. Beispiele: *Stream* und *StreamRec*.
- Für Prozeduren werden überwiegend Verben verwendet. Im Fall von Attributabfragen sind jedoch auch Hauptwörter und Adjektive denkbar. Einige Verben haben in der Bibliothek eine durchgängige Bedeutung:

Add	Hinzufügen eines Objekts in eine Datenstruktur.
Close	Beendigung einer Beziehung.
Create	Anlegen eines Objekts.
Get	Abfragen eines Attributes.
Init	Verbindung eines existierenden Objekts mit einer Schnittstelle.
Open	Anlegen eines Objekts und Eröffnung einer Beziehung, die später explizit geschlossen werden sollte.

Remove Entfernen eines Objekts.
Set Setzen eines Attributes.

Teilweise werden die Verben durch ein nachfolgendes Hauptwort näher spezifiziert. Allerdings sollte es generell vermieden werden, das Hauptwort des Modulnamens zu wiederholen, da es ohnehin angegeben werden muß.

3.3.2. Reihenfolge der Parameter

Normalerweise werden zuerst alle Werteparameter angegeben und erst dann die Variablenparameter. Bei den Parametern sollten zunächst die abstrakten Datentypen stehen, auf sich die Operation bezieht, gefolgt von den Parametern, die die Operation näher beschreiben (Optionen). Bei Konstruktoren stehen beide Regeln zueinander in Konflikt — hier wird der Zeiger auf das zu kreierende Objekt zuerst angegeben. Bei den anderen am Schluß stehenden Variablenparametern kommen zunächst die normalen zu erwartenden Resultate und dann ggf. Parameter für Ausnahmenbehandlungen.

3.3.3. Minimierung der Anzahl der Parameter

Wie Meyer in seiner *Argument rule* vorschlägt, sollten Optionen in Parameterlisten möglichst vermieden werden und statt dessen in den Status eines Objekts aufgenommen werden, der sich durch andere Operationen verändern läßt [Meyer94, §3.7.3]. Dieses Prinzip wird in der Ulmer Oberon-Bibliothek durch Disziplinen unterstützt. So können auch Module, die Operationen auf fremden Objekte durchführen (wie z.B. *Read.Int* für ein Objekt vom Typ *Streams.Stream*) ihre Optionen beim Objekt notieren (Beispiel in §6.6).

3.3.4. Globale Variablen

Globale Variablen werden grundsätzlich nur dann exportiert, wenn es sich bei ihnen um zur Initialisierungszeit berechnete Konstanten handelt, die nur lesenderweise zu nutzen sind. Eine Ausnahme sind Zeiger auf besondere Implementierungen, die von einem anderen Modul ebenfalls zur Initialisierungszeit initialisiert werden und zur Vermeidung von Importbeziehungen dienen.

3.4. Kooperativität

In einem dezentral organisierten Software-System nehmen viele Module aufeinander Einfluß, obwohl sie einander fremd sind. Zwei Module A und B (mit $A \neq B$) sind dann einander fremd, wenn weder $A \to^{+} B$ noch $B \to^{+} A$ gilt, wobei man bei den ggf. zwischenliegenden Modulen $M_1 \cdots M_n$ nur die öffentlichen Importbeziehungen betrachtet (d.h. die in Oberon im Definitionsteil sind).

Konfliktsituationen durch einander fremde Module entstehen durch gemeinsam zugängliche Objekte und durch globale Ressourcen. Zu den globalen Ressourcen gehören Raum, Zeit, globale Variablen und das umgebende System.

Unter Raum wird hier der logische Adreßraum verstanden, in dem das Programm und seine Daten repräsentiert werden. Während es bei älteren Betriebssystemen (wie z.B. bei UNIX Edition VII) wenig Einflußmöglichkeiten auf die Aufteilung dieses Raumes gab, erlauben neuere Systeme innerhalb einiger Restriktionen die freie Verwendung des gesamten Raums. So gibt es unter SunOS 4.1 neben dem traditionellen *brk(2)* die Systemaufrufe *mmap(2)* und *shmat(2)*, die es erlauben, an (fast) beliebigen Stellen im Adreßraum Speicher zu allokieren, Dateienbereiche hereinzuholen oder gemeinsame Speicherbereiche mit anderen Prozessen zu unterhalten. Die Ulmer Oberon-Bibliothek trennt die Verwaltung des Adreßraumes von der Speicherverwaltung, die sich um **NEW** und die automatische Freigabe von Speicher kümmert.

Die Adreßraumverwaltung wird durch das Modul *Memory* durchgeführt, das abgesehen von der Kenntnis eines **INTEGER**-Typs, der ausreichend für den logischen Adreßraum sein muß, vollkommen systemunabhängig ist. Es unterstützt auch vorwärts oder rückwärts wachsende Bereiche, die insbesondere für Keller geeignet sind. *Memory* bemüht sich dabei um eine größtmögliche Fairneß bei der Allokation wachsender Bereiche. Dies gelingt recht gut, wenn der logische Adreßraum bedeutend größer ist als der maximale physisch zur Verfügung stehende Raum.

Konflikte bezüglich der Zeit ergeben sich durch die begrenzte Parallelisierbarkeit. Oberon sieht in der Sprachdefinition nur einen einzigen Kontrollfluß vor. Dies ist eine Restriktion, die im Ulmer Oberon-System durch die Einführung von Koroutinen gelockert worden ist. Da jedoch immer nur eine Koroutine aktiv sein kann, müssen auch Koroutinen untereinander kooperieren, indem sie die Ausführung der anderen Koroutinen nicht unnötig blockieren.

Globale Variablen können Konfliktauslöser sein, wenn sie exportiert sind und von jedem Modul importiert werden können. Das Problem läßt sich weitgehend eliminieren, wenn Zugriffsoperationen eingeführt werden oder wenn man globale Variablen nur noch in Form von Konstanten verwendet, die nach der Initialisierung des zugehörigen Moduls nicht mehr modifiziert werden dürfen.

Auf das umgebende System kann von jedem Modul aus zugegriffen werden. Grundsätzlich kann man davon ausgehen, daß das umgebende System Lokalität nur auf Basis von Prozessen unterstützt, jedoch nicht auf Basis der internen Sichtbereiche eines Software-Systems. Somit können alle Ressourcen des umgebenden Systems zu potentiellen Konflikten führen. Wie schon Hansen in Concurrent Pascal zeigt, lassen sich solche Konflikte auf Basis von virtuellen Ressourcen lösen [Hansen75]. Bei dieser Technik gibt es zu jeder externen Ressource ein Modul, das als einzige Partei darauf zugreift. Gegenüber seinen Klienten stellt es virtuelle Ressourcen zur Verfügung, die dann konfliktfrei verwendet werden können. Konkurrierende Verwendungen virtueller Ressourcen werden dann von dem Zugriffsmodul in kooperativer Weise gehandhabt.

In der Ulmer Oberon-Bibliothek gibt es zahlreiche Beispiele für diese Technik. So werden z.B. UNIX-Signale in Ereignisse abgebildet, wobei es für Ereignisse beliebig viele Interessenten geben kann. Während UNIX nur einen einzigen Weckauftrag für jeden Prozeß unterstützt, können bei *Timers* beliebig viele Aufträge eingetragen werden.

All den Konflikten bezüglich Raum, Zeit und den Ressourcen der Außenwelt ist gemeinsam, daß sie nur unter Verzicht auf die Kombination mehrerer Programmiersprachen gelöst werden können. Viele Sprachsysteme unter UNIX (wie z.B. ISE Eiffel oder das Mjølner BETA-System) unterstützen die Möglichkeit, in C geschriebene Bibliotheken aufzurufen, und machen davon auch regen Gebrauch, insbesondere im Bereich der Schnittstelle zum Betriebssystem und dem Anschluß an X-Windows. Da weder die Standardbibliothek für C noch die X-Windows-Bibliotheken kooperative Mechanismen kennen, ist es nachher auf dieser Basis nicht möglich, dezentrale und damit erweiterbare Software-Systeme aufzubauen.

Aus diesem Grund wurde die Ulmer Oberon-Bibliothek vollständig in Oberon geschrieben. Beim Ulmer Oberon-Übersetzer wurde entsprechend auch auf die Unterstützung der Anbindung an andere Programmiersprachen verzichtet. Sie wäre mittlerweile auch kaum durchzuführen, da die Bibliothek, die automatische Speicherverwaltung und die Anordnung der Module im Speicher[1] völlig inkompatibel zu Bibliotheken anderer Programmiersprachen sind.

[1] In Oberon gibt es kein Hauptprogramm. Daher wird zu Beginn der Ausführung eines Programms festgestellt, welche Module vorhanden sind. Dieses Verfahren setzt voraus, daß die Objekte (übersetzte Module) alle hintereinander im Speicher liegen und durch ein spezielles Objekt beendet werden. Bei der Verwendung fremder Objekte bestünde die Gefahr, daß sie zwischen den Oberon-Modulen angeordnet würden. Dies würde zum sofortigen Absturz führen.

4. Ereignisse

4.1. Einführung

Ereignisse repräsentieren Informationen, die jederzeit erzeugt werden können und an eine Reihe von Interessenten weiterzugeben sind. Im Unterschied zu Aufträgen gibt es normalerweise keine direkte Rückkopplung vom Interessenten zum Informationsgeber, und es kann Zeit verstreichen zwischen dem Zeitpunkt der Erzeugung und des Lesens der Information. Ereignisse werden Ereignistypen zugeordnet und jedes Modul kann Interesse an bestimmten Ereignistypen bekunden.

Hier ein Auszug der wichtigsten Deklarationen aus *Events*:

```
TYPE
   EventType = POINTER TO EventTypeRec;
   EventTypeRec = RECORD (Disciplines.ObjectRec) END;

   Message = ARRAY 80 OF CHAR;
   Event = POINTER TO EventRec;
   EventRec =
      RECORD
         (Objects.ObjectRec)
         type: EventType;
         message: Message;
      END;
   EventHandler = PROCEDURE (event: Event);

PROCEDURE Define(VAR type: EventType);
PROCEDURE Handler(type: EventType; handler: EventHandler);
PROCEDURE Raise(event: Event);
```

Zu einem Ereignis gehört nicht nur der Ereignistyp, sondern auch noch ein Text, der z.B. als Fehlermeldung dienen kann. Im Einzelfall kann ein Ereignis um geeignete Komponenten erweitert werden, die es weiter beschreiben und spezifizieren. Mit *Define* kann ein Ereignistyp kreiert werden, und durch *Handler* wird ein Interesse an einem Ereignistyp zum Ausdruck gebracht. Mit *Raise* wird ein neu erzeugtes Ereignis an *Events* zur Weiterleitung übergeben.

Folgendes Beispiel sieht ein Ende-Ereignis vor, das erzeugt wird, wenn der Benutzer eine Sitzung beendet:

```
DEFINITION UserInterface;

   IMPORT Events;

   VAR
      endOfSession: Events.EventType;

   (* ... *)

END UserInterface.
```

Innerhalb der Implementierung von *UserInterface* wird der Ereignistyp *endOfSession* während der Initialisierung definiert:

```
Events.Define(endOfSession);
```

In der Implementierung von *UserInterface* kann dann beim Ende der Sitzung *SignalEndOfSession* aufgerufen werden:

```
PROCEDURE SignalEndOfSession;
   VAR
      event: Events.Event;
BEGIN
   NEW(event);
   event.type := endOfSession;
   event.message := "end of session";
   Events.Raise(event);
END SignalEndOfSession;
```

Generell wird zwischen drei Sorten von Reaktionen auf Ereignisse unterschieden:

default Voreinstellungsgemäß führen Ereignisse, für die es keine Interessenten gibt, zur Beendigung des Programms. Dies ist primär motiviert durch Ereignisse, die aufgrund von Fehlern ausgelöst werden. Hier soll es vermieden werden, daß Fehler unbemerkt bleiben.

ignore Es ist möglich, Ereignisse eines Typs vollständig zu ignorieren. In diesem Fall führen Ereignisse nicht zum Programmende.

funcs Wenn ein oder mehrere Interessenten für einen Ereignistyp eingetragen sind, dann werden alle nacheinander mit dem Ereignis als Parameter aufgerufen.

4.2. Ereignisse im Vergleich zu UNIX-Signalen

Diese Aufteilung ähnelt den Reaktionstypen bei UNIX-Signalen. Allerdings gibt es zwei wichtige Unterschiede: Es kann eine beliebige Anzahl von Interessenten für einen Ereignistyp definiert werden, und ein Interessent bleibt eingetragen, bis die Reaktion explizit auf *ignore* oder *default* gesetzt wird.

Aufgrund dieser Unterschiede sind sie viel leichter zu benutzen als die vergleichbaren Techniken in C. In dem Augenblick, wenn in C ein Interesse an einem Signal deklariert wird, ist man unkooperativ gegenüber anderen Bibliotheken, die möglicherweise ebenfalls Interesse an dem gleichen Signal haben. Zwar liefert die entsprechende C-Funktion den vorherigen Interessenten zurück, und es wäre natürlich theoretisch möglich, sich den zu merken und zu berücksichtigen. Dies ist jedoch relativ umständlich und kommt in der Praxis eher selten vor.

Zum anderen sind Interessenten in C gezwungen, sofort nach Eintreffen eines Signals das Interesse neu zum Ausdruck zu bringen. Hier bleibt ein Fenster offen zwischen dem Eintreffen des Signals und der neuen Bestätigung des Interesses, so daß nicht garantiert ist, daß alle Signale auch tatsächlich bearbeitet werden können. Für dieses Teilproblem wurde allerdings eine alternative Signalbehandlung in BSD-UNIX entwickelt [Leffler89, §4.7].

4.3. Abbildung von UNIX-Signalen auf Ereignisse

Events offeriert eine Schnittstelle für Implementierungen von Ereignistypen, die dann optional genutzt werden kann, wenn es von Bedeutung ist, daß Änderungen der Reaktion der zugehörigen Implementierung mitgeteilt werden:

```
TYPE
   EventManager = PROCEDURE (type: EventType; reaction: Reaction);

PROCEDURE Manager(type: EventType; manager: EventManager);
```

Das Modul *SysSignals* exportiert alle Signale der jeweiligen UNIX-Implementierung und trägt sich bei *Events.Manager* ein. Auf diese Weise können Reaktionsänderungen bei *Events* auf der Systemebene nachgezogen werden. Im weiteren sorgt es in der System-V-Implementierung dafür, daß die Systemreaktion nicht auf *default* zurückgesetzt wird, wenn ein Signal eingetroffen ist. Folgendes Beispiel demonstriert einen Interessenten für **SIGINT**. Dieses Signal erreicht dann den Prozeß, wenn auf dem zugehörigen kontrollierenden Terminal eine bestimmte Taste gedrückt wird (typischerweise Control-C):

```
PROCEDURE CatchSIGINT(event: Events.Event);
BEGIN
   Write.Line("Interrupted...");
END CatchSIGINT;

(* ... *)

Events.Handler(SysSignals.INT, CatchSIGINT);
```

Ein besonderes Problem stellen die UNIX-Signale dar, die normalerweise auf einen ernsthaften Fehler hindeuten. Beispiele dafür sind **SIGSEGV**, das bei Zugriffen auf unzulässige Adressen generiert wird, und **SIGFPE**, das bei Division durch 0 oder bei Ausnahmenbehandlungen des Gleitkommazahlenprozessors entsteht. Hier würde normalerweise die Fortsetzung des Programms zu einer Wiederholung des Signals und damit zu einer Endlosschleife führen. Allerdings gibt es auch Fälle, bei denen das Problem behebbar ist. So kann **SIGSEGV** darauf hindeuten, daß der Keller einer Koroutine übergelaufen ist. Dies ist ein Problem, das durch Anfordern weiteren Speichers beim System gelöst werden kann. Bei **SIGFPE** könnte man beispielsweise den Fehler notieren und dann — nach Abschaltung der Ausnahmenbehandlungen — die Operation erneut versuchen.

Damit *UnixSignals* bei diesen Signalen unterscheiden kann, ob entsprechende Maßnahmen stattgefunden haben oder nicht, sind die entsprechenden Ereignisse um eine Komponente *fixed* erweitert. Jeder der Interessenten ist dann frei, *fixed* auf **TRUE** zu setzen, wenn das Problem behoben worden ist. *UnixSignals* überprüft dann nach *Events.Raise*, ob *fixed* auf **TRUE** gesetzt worden ist, und leitet, falls dies nicht der Fall sein sollte, den Abbruch des Programms ein.

4.4. Prioritäten

Im Gegensatz zu den Ereignissen, die innerhalb des Programms generiert werden, treffen UNIX-Signale asynchron ein. Dies eröffnet eine Reihe von Problemen, da nicht vorhersagbar ist, in welchem Zustand sich das Programm dann befindet. So ist im obigen Beispiel nicht ersichtlich, womit zu rechnen ist, wenn **SIGINT** eine Operation auf *Streams.stdout* unterbricht, da *Write.Line* ebenfalls auf *Streams.stdout* operiert. Außerdem können asynchrone Ereignisse in rascher Folge eintreten und damit eine Bedrohung der bereits arbeitenden Interessenten darstellen.

Ein einfaches und effizientes Konzept für dieses Problem gibt es bereits sehr lange auf der Hardware-Ebene (z.B. PDP11). Hier ist der laufende Prozeß und jedes asynchrone Ereignis mit einer Priorität verbunden, die durch eine natürliche Zahl repräsentiert wird. Wenn ein Ereignis eintritt, wird überprüft, ob die Priorität des Ereignisses über der des laufenden Prozesses liegt. Ist dies der Fall, wird sofort der zugehörige Interessent aufgerufen, wobei während der Zeit der Abarbeitung die Priorität des Prozesses auf die des asynchronen Ereignisses angehoben wird. Liegt die Priorität des Ereignisses unter der des Prozesses oder ist zu ihr gleich, dann wird das Ereignis an eine Warteschlange angehängt, die dann abgearbeitet wird, wenn die Priorität wieder sinkt.

Das Prioritätensystem der PDP11 wird auch von Modula-2 unterstützt [Wirth82, §30]. Hier kann bei Modulen optional eine Priorität angegeben werden, die für alle enthaltenen Prozeduren gilt. Entsprechend gibt es eine Garantie, daß nur Ereignisse mit höherer Priorität die Bearbeitung einer Prozedur unterbrechen können. Problematisch ist in Modula-2 natürlich der Aufruf von anderen Modulen, die ebenfalls eine Priorität besitzen. Hier dürfen nur Module aufgerufen werden, die entweder überhaupt keine Prioritätsangabe besitzen und daher die aktuelle übernehmen, oder solche, die eine mindestens genauso hohe Priorität haben. Dies läßt sich wegen der möglichen Verwendung von Prozedurvariablen nicht völlig statisch überprüfen. Die Ulmer Modula-2-Implementierung hat dieses Konzept übernommen und auf UNIX-Signale übertragen, nachdem den einzelnen Signalen Prioritäten zugeordnet worden sind.

In der Ulmer Oberon-Bibliothek wurde das Prioritätenkonzept ebenfalls übernommen und für alle Ereignisse verallgemeinert. Durch die Verallgemeinerung läßt sich die Abhängigkeit von UNIX völlig vermeiden. So kann beispielsweise mit

```
Events.EnterPriority(Priorities.interrupts);
(* Operationen, die nicht von normalen asynchronen Ereignissen
   unterbrochen werden sollten
*)
Events.ExitPriority;
```

ein Bereich eingeklammert werden, der vor Überraschungen gesichert ist. Allen Ereignistypen können dabei beliebig Prioritäten zugeordnet werden. Allerdings gibt es einige Konventionen, um einen sinnvollen und portablen Umgang mit Prioritäten zu ermöglichen.

Prioritäten müssen natürlich lokal zu Koroutinen sein, da sonst im obigen Beispiel inmitten des kritischen Bereichs durch einen Koroutinenwechsel die Priorität global heruntergesetzt werden könnte. *Events* selbst ist nicht von Koroutinen abhängig, sondern kennt nur Prioritätensysteme, die gewechselt werden können. Bei dem Wechsel einer Task (ein den Koroutinen übergeordnetes Konzept, das noch vorgestellt wird) wird dann das Prioritätensystem gewechselt.

4.5. Ereignisse bei der Terminierung eines Programms

Oberon unterstützt analog zu Modula-2 die Möglichkeit, für ein Modul einen Initialisierungsteil anzugeben. Genauso wäre es natürlich attraktiv, bei der Beendigung eines Programms Aufräumarbeiten in Bezug auf Beziehungen zum Benutzer oder System durchführen zu können. Dies ist nicht nur bei einem regulären Programmende interessant, sondern auch bei einem Programmabsturz z.B. aufgrund eines Laufzeitfehlers.

Hierfür gibt es die Ereignisse *Process.termination* und *Process.abort*. Da Interessenten grundsätzlich in der umgekehrten Reihenfolge aufgerufen werden, in der sie deklariert worden sind, können Module bei ihren Aufräumarbeiten davon ausgehen, daß die Module, die sie importieren, noch nicht aufgeräumt haben und somit uneingeschränkt benutzbar sind.

4.6. Ereignisse aufgrund von Fehlern

Das Auftreten von Laufzeitfehlern oder anderen Fehlern kann natürlich auch durch ein Ereignis repräsentiert werden. So generiert der Übersetzer Laufzeittests, die im Fehlerfall den möglichen Fehlern zugeordnete Prozeduren aus dem Modul *RTErrors* aufrufen. Diese generieren jeweils ein entsprechendes Ereignis und rufen dafür *Events.Raise* auf. Anschließend wird *Process.Abort* aufgerufen, das zunächst ein Ereignis vom Typ *Process.abort* generiert und anschließend den Prozeß nach Möglichkeit mit einem Speicherauszug (*core dump*) abbricht.

Analog können natürlich auch andere Fehler auf diese Weise angezeigt und bearbeitet werden. Da jedoch Ereignistypen typischerweise global sind und Fehler häufig in einem lokalen Kontext betrachtet werden müssen, ist dies nicht immer praktikabel. Das nächste Kapitel zeigt hierfür eine Technik, die auf *Events* aufbaut.

4.7. Zusammenfassung

Die Abbildung 23 zeigt die vorgestellten Module mit ihren Importbeziehungen und ihrer Verteilung auf die einzelnen Ebenen.

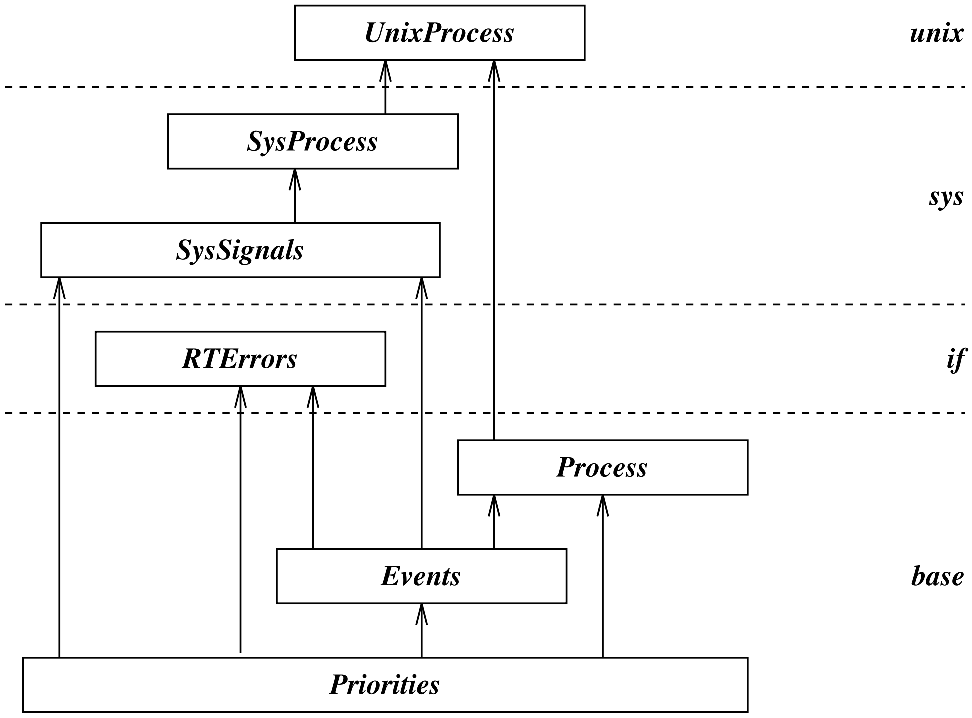

Abb. 23: Modulhierarchie für Ereignisse

Bemerkenswert ist die Aufteilung von *Process* in drei Ebenen: *Process* selbst gehört *base* an und offeriert die portablen Operationen für das Terminieren und Abbrechen. Die Implementierung der Operationen ist offen gelassen und wird auf Basis einer Schnittstelle realisiert, die von einem anderen Modul während der Initialisierungszeit zu übergeben ist. Dies geschieht bei der aktuellen Implementierung durch *UnixProcess*, das wiederum auf *SysProcess* beruht. *SysProcess* offeriert alle prozeßbezogenen Operationen unter UNIX, die umfangmäßig weit über *Process* hinausgehen, und hängt deswegen auch von *SysSignals* ab. Da es nur eine Implementierung von *Process* bei einem konkreten Programm gibt, wäre es auch denkbar gewesen, *UnixProcess* direkt im Modul von *Process* zu integrieren. Zur Aufrechterhaltung der Abhängigkeiten müßte dann die Definition von *Process* der Ebene *base* zugeordnet werden, während das zugehörige Modul bei *unix* liegen würde. Dies ist eine Technik, die nur beim Vertrags-Modell denkbar ist und somit bei der Züricher Oberon-Version nicht umsetzbar wäre.

Entwurfsziel bei *Events* war es, eine relativ allgemeine Abstraktion zu finden, die für alle Arten von Ereignissen praktisch zu verwenden ist. Neben asynchronen Signalen des Systems lassen sich auch Fehlerindikationen und Terminierungsereignisse gut auf dieser Basis realisieren. Wenn die Priorität des Ereignisses zu einer sofortigen

Ausführung der Interessenten führt, ist eine Rückmeldung über den Ereignisverbund möglich. In jedem Fall können sich die Interessenten untereinander über den Ereignisverbund koordinieren, wenn dies notwendig sein sollte.

Gelegentlich erweisen sich globale Ereignistypen als Nachteil, da es vielfach wichtig ist, Ereignisse in Abhängigkeit eines lokalen Kontextes auszuwerten. Hierfür wurde das Modul *RelatedEvents* entwickelt, das es erlaubt, Ereignisse in Zusammenhang mit einem Objekt zu bearbeiten. Die Ausnahmenbehandlungen der Ulmer Oberon-Bibliothek, die im nächsten Kapitel vorgestellt werden, basieren hauptsächlich darauf.

5. Ausnahmenbehandlungen

5.1. Einführung

Eine Reihe von Programmiersprachen wie z.B. CLU [Liskov79], Ada [Ichbiah79, ANSI83], Eiffel [Meyer92], BETA [Knudsen87, Madsen93], Modula-3 [Nelson91], C++ [Koenig90, Stroustrup91] und Exceptional C [Gehani92] offerieren Ausnahmenbehandlungen (*exception handling*) als Kontrollstruktur für die Behandlung von Laufzeitfehlern, Betriebssystemsfehlern und Fehlern, die von der Bibliothek selbst entdeckt werden. Einige der Sprachen verwenden Ausnahmenbehandlungen auch für asynchrone Ereignisse. Traditionelle Techniken sehen die explizite Rückgabe von Fehlerindikationen bei Prozeduraufrufen vor (z.B. C [Kernighan78]) oder die Abbildung von Fehlerentdeckungen auf Ereignisse, für die es einen Interessenten gibt, der typischerweise nach der Ausgabe der Fehlermeldung das Programm abbricht oder wieder in einen konsistenten Zustand überführt (z.B. POSTSCRIPT [Adobe Systems Inc.86]).

Folgt man Meyer [Meyer88], so kann das Aufrufen einer importierten Prozedur unter dem Gesichtspunkt eines Vertrags betrachtet werden, bei dem

(V1) der aufrufende Klient Parameter und einen Zustand vorzuliefern hat, der die Vorbedingung der Prozedur erfüllt, worauf

(V2) die aufgerufene Prozedur sich verpflichtet, einen Zustand und ggf. Rückgabewerte zurückzuliefern, die der Nachbedingung entsprechen.

Obwohl es sich um den Aufruf von Prozeduren handelt, sind die Parteien eines Kontrakts Module, wobei die Schnittstellen als Verträge dienen, die durch den Import akzeptiert werden (siehe Vertrags-Modell in §1.3.1).

Es gibt eine Reihe von typischen Fällen, in denen eine der beiden Parteien den Vertrag nicht hält bzw. nicht halten kann:

- Die aufrufende Partei übergibt Parameter, die die Vorbedingung nicht erfüllen. Dies läßt sich nicht immer vermeiden, da es Fälle gibt, in denen erst die aufgerufene Prozedur dies mit vertretbaren Aufwand feststellen kann. Beispiele dafür sind Matrixinvertierungen und Übersetzer. Die Singularität einer Matrix läßt sich beim Invertieren problemlos feststellen, jedoch in vielen Fällen nicht ohne fast den gleichen Aufwand im voraus überprüfen. Analog kann erst der Übersetzer feststellen, ob Syntaxfehler in einer Quelle vorhanden sind.
- Laufzeitfehler, die durch Programmierfehler entstehen, wie z.B. das Verlassen des Indexbereiches eines Feldes.
- Ausfälle der Systemumgebung, z.B. wenn es keinen weiteren Speicherplatz mehr gibt oder Netzverbindungen zusammenbrechen.
- Transitive Ausfälle (siehe Abbildung 24): *B* ist auf die Dienstleistung von *C* angewiesen und kann seinen Vertrag mit *A* nicht erfüllen, da *C* seinen Vertrag mit *B* nicht erfüllte.

Zusammengefaßt läßt sich entsprechend der Verletzung von (V1) oder (V2) zwischen Ausfällen aufgrund einer nicht erfüllten Vorbedingung (*domain failure*) und Ausfällen aufgrund der Unmöglichkeit, die Nachbedingung zu gewährleisten (*range failure*), unterscheiden [Goodenough75].

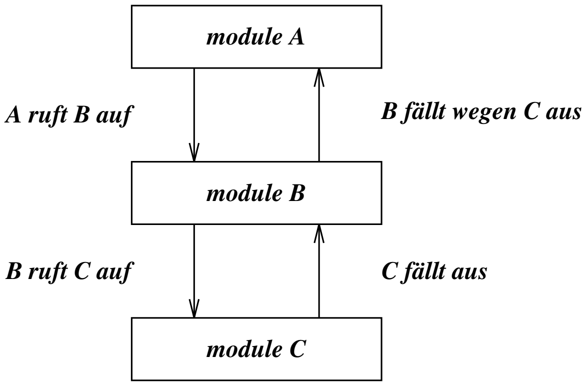

Abb. 24: Transitive Ausfälle

Während eines Programmlaufs führen Fehler zu Vertragsverletzungen und damit zu Ausfällen. Ausfälle ergeben Ausnahmesituationen, die bei Entdeckung einer Ausnahmenbehandlung unterzogen werden. Fehler sind zum einen statische Programmierfehler (z.B. falls "i := i + j" getippt worden ist, wenn "i := i - j" korrekt gewesen wäre — dies entspricht den *algorithmic faults* von [Melliar-Smith77]). Zum anderen ist es durchaus üblich, mit Fehlern zu leben, wenn Vorbedingungen nicht im voraus garantiert werden können. Auch wenn die Vorbedingung eines Übersetzers festlegt, daß die Eingabedatei der Syntax und Semantik einer bestimmten Programmiersprache genügen muß, so wird trotzdem erwartet, daß der Übersetzer robust ist und Fehlermeldungen ausgeben kann. Diese Terminologie leitet sich von Cristian ab, der hierfür präzise mathematische Definitionen gegeben hat [Cristian85, Cristian89].

Einige Autoren beziehen den Begriff Ausnahmesituation auch auf andere seltene Vorkommnisse. So unterscheidet Gehani zwischen gewöhnlichen Ausnahmesituationen (*ordinary exceptions*) und Ausnahmesituationen aufgrund von Signalen (*signal exceptions*). Wie bereits vorgestellt worden ist, werden in der Ulmer Oberon-Bibliothek Signale als asynchrone Ereignisse betrachtet. Allerdings ist der Übergang fließend: so dürfte zwar **SIGINT** (*keyboard interrupt*) nicht auf einen Fehler zurückzuführen sein, sondern auf den Benutzer, der eine entsprechende Taste gedrückt hat. Jedoch führt andererseits ein Zugriff durch **NIL** zu dem Signal **SIGSEGV** (*segmentation violation*). Somit können einzelne Signale zwar aufgrund von Fehlern entstehen, trotzdem ist für jedes Signal auch eine Ursache denkbar, die nichts mit Fehlern zu tun hat. So kann **SIGSEGV** auch durch den wachsenden Keller einer Koroutine ausgelöst werden — ein Vorfall, der nach Behebung zur Fortsetzung der Koroutine führt.

Liskov gibt ein Beispiel an, bei dem das Ende der Eingabe zu einer Ausnahmenbehandlung führt, die den einzigen Weg darstellt, die Einleseschleife zu terminieren [Liskov79]. Nach der obigen Terminologie dürfte das Ende einer Eingabe nur dann zu einer Ausnahmenbehandlung führen, wenn die Vorbedingung angibt, daß damit nicht zu rechnen ist.

Goodenough gibt noch weitere Einsatzmöglichkeiten für Ausnahmenbehandlungen an [Goodenough75]. So schlägt er vor, daß sie zur Klassifizierung eines Resultats dienen können. Ferner könnten Ausnahmenbehandlungen als Kommunikationsmedium zwischen den beiden Parteien dienen, um den Fortschritt der Durchführung zu beobachten und ggf. weitere Informationen zu geben. Für Klassifizierungen sind jedoch entsprechende Rückgabewerte oder Variablenparameter geeigneter, und Ereignisse oder Iteratoren dürften Kommunikationsprobleme besser lösen als Ausnahmenbehandlungen [Cristian82].

Um die Korrektheit eines Software-Systems zu gewährleisten, ist es notwendig, daß Operationen entweder ihren Vertrag erfüllen oder unüberhörbar einen Ausfall signalisieren. Statt einen Ausfall als Verletzung von (V2) zu betrachten, könnte man (V2) zu (V2') erweitern:

(V2') Die aufgerufene Prozedur verpflichtet sich, einen Zustand und ggf. Rückgabewerte zurückzuliefern, die der Nachbedingung entsprechen. Bei Verletzung der Vorbedingung ist ein Ausfall zu signalisieren.

Eine Prozedur, die (V2') für alle möglichen Anfangszustände erfüllt, ist *robust*. Ihr Verhalten ist immer vorhersehbar, selbst wenn der Aufrufer seinen Teil am Vertrag nicht erfüllt [Cristian84].

5.2. Traditionelle Techniken

Es gibt sehr unterschiedliche Techniken zur Unterstützung von Ausnahmenbehandlungen. In Sprachen, die Ausnahmenbehandlungen nicht durch besondere Sprachkonstrukte unterstützen, wird typischerweise eine Technik verwendet, die auf besondere Rückgabewerte oder Statusvariablen beruht [Gehani92]. Als Beispiel dafür kann die C-Bibliothek verwendet werden:

```
/* Fehlerbehandlung in C */
if (write(fd, buf, nbytes) < 0)
        /* Ausfall von write() -- errno enthaelt die Fehlernummer */
else
        /* write() hat geklappt */
```

Hier liefert *write* entweder eine positive Anzahl zurück, die für die Anzahl der gelesenen Bytes steht, oder **-1**, das eine Fehlerindikation darstellt. In der globalen Variablen *errno* ist dann die zugehörige Fehlernummer zu finden. Obwohl es naturgemäß erscheint, Untersuchungen auf mögliche Ausnahmesituationen direkt hinter einem Prozeduraufruf vorzunehmen, gibt es bei dieser Technik einige Probleme:

- Wie kann die Fehlerindikation sinnvoll weitergegeben werden, wenn die Prozedur, die *write* aufgerufen hat, aufgeben möchte? Da es möglicherweise auch andere Fehlervarianten gibt (z.B. Parameter, die der Vorbedingung nicht genügen), reicht es nicht, *errno* zurückzuliefern. Ferner wäre möglicherweise für die aufrufende Partei auch der exakte Kontext des Ausfalls von Interesse, wie z.B. die Parameter von *write*.

- Der Programmtext droht umständlich zu werden, wenn zahlreiche Aufrufe, die von einem möglichen Ausfall betroffen sein können, hintereinander aufgeführt sind.
- Die Ausnahmenbehandlung hängt nicht vom jeweiligen Kontext ab (z.B. von dem Objekt, auf dem eine Prozedur operiert). Manchmal ist es angemessen (z.B. für Fehleruntersuchungen), den Klienten sofort zu unterrichten, selbst wenn es alternative Versuchsmöglichkeiten gibt [Avižienis85].
- Der Fehlerindikationstest könnte vergessen werden. Selbst bei Rückgabewerten ist dies in C ohne Fehlermeldung möglich, und im Falle von globalen Statusvariablen (z.B. *InOut.Done* in Modula-2) fällt dies aus dem Rahmen der überprüfbaren Möglichkeiten des Übersetzers. Darüber hinaus werden globale Variablen durch parallele Kontrollflüsse bedroht.

5.3. Mechanismen für Ausnahmenbehandlungen

Allen Mechanismen für Ausnahmenbehandlungen ist gemein, daß eine Prozedur auf verschiedenen Wegen beendet werden kann. Um die Diskussion zu vereinfachen, kann man jede Anweisung als einen Prozeduraufruf betrachten, selbst wenn der Übersetzer dafür keine Prozeduraufrufe generiert. Entweder wird eine Prozedur normal beendet oder sie initiiert eine Ausnahmenbehandlung aufgrund eines entdeckten Ausfalls. Eine Ausnahmenbehandlung führt dazu, daß die Kontrolle an einen *Ausnahmenbearbeiter* weitergeleitet wird. Die Prozedur, die die Ausnahmenbehandlung initiiert hat, wird als *Signalgeber* bezeichnet. Die Prozedur, die den Ausnahmenbearbeiter enthält, der aufgrund der Ausnahmenbehandlung aktiviert wird, heißt *Signalempfänger*.

Es gibt eine Reihe von Techniken, einen Signalempfänger zu lokalisieren. Denkbare Kandidaten sind aktivierte Prozeduren (also der Klient und der Klient des Klienten usw.), statisch übergeordnete Prozeduren, globale und damit kontextunabhängige Ausnahmenbearbeiter und objektbezogene Bearbeiter, die von einem der Parameter des Signalgebers abhängen. Der Signalgeber selbst wird als Signalempfänger nicht in Betracht gezogen, da er per Definition dazu nicht in der Lage ist.

Wenn der Signalempfänger eine der aktivierten Prozeduren sein soll, so kann dies der direkte Aufrufer sein (*Ebenenmodell* oder *single level mechanism*) oder einer der übergeordneten Klienten (*distanziertes Ebenenmodell* oder *multilevel mechanism*), d.h. im Beispiel könnte *A* der Signalempfänger für den Signalgeber *C* sein, ohne daß *B* daran beteiligt ist. Da dies zum Bruch von Abstraktionsgrenzen führt, wird dies von kaum einer Sprache unterstützt (PL/1 ist eine Ausnahme [Noble69, MacLaren77, Horowitz83, §9.2]). BETA unterstützt das distanzierte Ebenenmodell nur in Bezug auf statisch übergeordnete Klienten und vermeidet damit auf elegante Weise den Bruch von Abstraktionsgrenzen.

Globale Ausnahmenbearbeiter sind einfach zu implementieren und benötigen keine zusätzliche Sprachunterstützung, wenn es Prozedurtypen gibt. Allerdings können sie im Bereich von Bibliotheken oder größeren Software-Systemen nicht die einzige Lösung sein.

Einige objekt-orientierte Sprachen (z.B. Eiffel oder Lore [Dony88]) unterstützen objektbezogene Ausnahmenbearbeiter.

Ausnahmenbearbeitern stehen — nach einigen normalen Anweisungen — folgende Optionen offen (siehe Abbildung 25):

Weiterleitung
: In diesem Fall wird eine neue Ausnahmenbehandlung initiiert, d.h. der Signalempfänger wird selbst zum Signalgeber. Bei dem Ebenenmodell handelt es sich dabei um den jeweils nächsten Klienten. Normalerweise ist diese Option die Voreinstellung, wenn kein Ausnahmenbearbeiter für einen Signalempfänger explizit gegeben ist.

Wiederaufnahme
: Diese Option erlaubt dem Ausnahmenbearbeiter, die Ausführung am Punkt des Ausfalls fortzusetzen. Dies ist eine denkbare Reaktion, wenn das zugrundeliegende Problem gelöst worden ist. Die Kombination des Ebenenmodells mit einer möglichen Wiederaufnahme ist nicht einfach zu implementieren, da beide Prozedurkontexte analog zu Koroutinen parallel benutzt werden. Berücksichtigt man, daß Ausnahmenbearbeiter andere Prozeduren aufrufen können, die wiederum zu Ausnahmenbehandlungen führen können, dann ist klar, daß man eine Reihe von Kellern mit gemeinsamen Teilen benötigt (sogenannte *cactus stacks*) [Gehani92]. Im Falle von globalen oder objektbezogenen Ausnahmenbearbeitern ist Wiederaufnahme kein Problem. Allerdings muß dann der Signalgeber damit rechnen, daß das Auslösen einer Ausnahmenbehandlung nicht notwendigerweise ein Punkt ohne Rückkehr ist.

Neustart
: Dies ist eine Technik, die von einigen Mechanismen, die auf dem Ebenenmodell basieren, unterstützt wird. Hierbei wird der Keller bis zu den Daten des Signalempfängers abgebaut und anschließend die Prozedur bzw. der aktuelle Block neu

gestartet. Wenn der Ausnahmenbearbeiter die Anzahl der Versuche in einer lokalen Variablen notiert, dann können auf diese Weise auch Alternativen durchprobiert werden (analog zu [Avižienis85]).

Beendigung Einige Programmiersprachen erlauben dem Ausnahmenbearbeiter, beim Ebenenmodell den Signalempfänger mit **RETURN** zu beenden, ohne daß die Ausnahmenbehandlung weitergeleitet wird. Auf den ersten Blick verletzt dies das Prinzip, daß Ausfälle nicht unerkannt bleiben sollen: Wenn ein einfaches **RETURN** ausreicht, um die Nachbedingung zu erfüllen, warum überhaupt etwas anderes unternehmen? Allerdings gibt es Fälle, in denen es sinnvoll ist, mögliche Ausfälle in normale Rückgabewerte zu konvertieren. Als Beispiel kann eine Prozedur genommen werden, die überprüft, ob eine gegebene Datei existiert. Hierbei ist sie möglicherweise auf eine Operation zum Eröffnen angewiesen, die in ihrer Vorbedingung die Existenz der Datei voraussetzt.

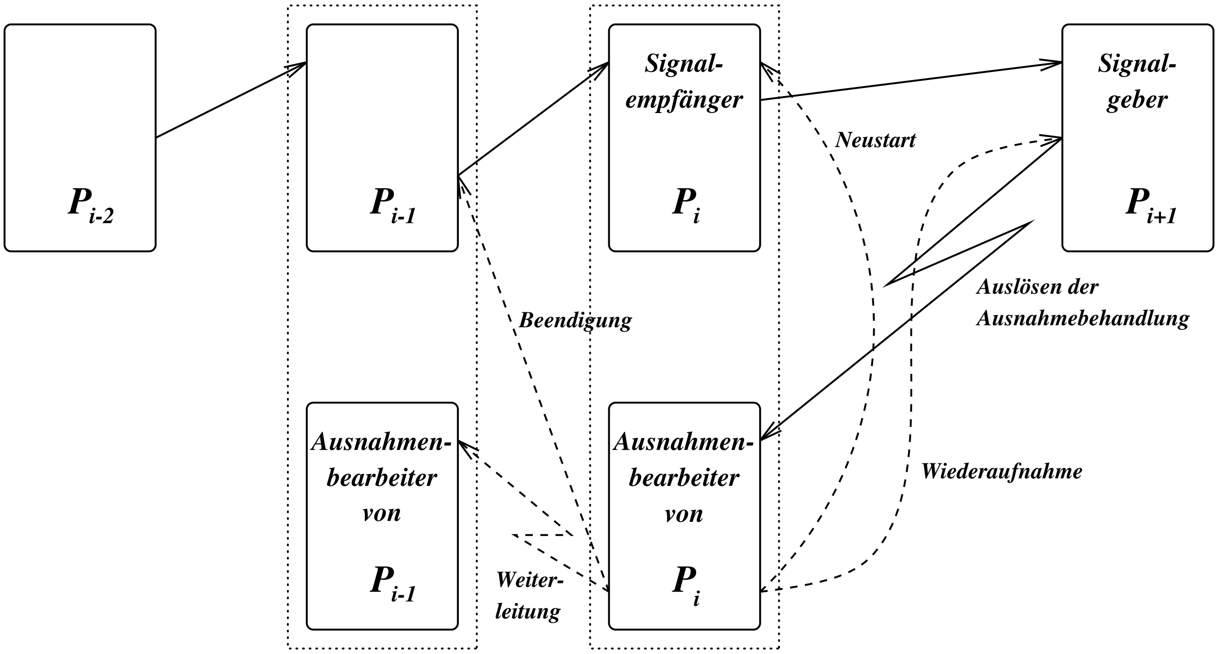

Abb. 25: Optionen eines Ausnahmenbearbeiters

Von der Sprache unterstützte Ausnahmenbehandlungen können einige der genannten Probleme der traditionellen Technik lösen. Als ein Beispiel mag in Modula-3[1] die Prozedur *PutChar* aus dem Modul *Wr* dienen (in abgekürzter Form aus [Harbison92] entnommen):

```
(* excerpt of the interface of module Wr *)
TYPE
   Code = {Closed, Unseekable};
EXCEPTION
   Failure(REFANY);
   Error(Code);

PROCEDURE PutChar(wr: T; ch: CHAR) RAISES {Failure, Error};
```

Die **RAISES**-Deklaration gibt an, daß *PutChar* entweder erfolgreich beendet wird oder *PutChar* eine der angegebenen Ausnahmen (*Failure* oder *Error*) initiiert. Beide Ausnahmen sind parametrisiert: *Failure* durch **REFANY**, einen Typ, der zu allen Zeigertypen kompatibel ist, und *Error* durch *Code*. In dem Beispiel initiiert *PutChar* selbst nur *Error*, während *Failure* von den zugrundeliegenden Implementierungen ausgelöst werden kann.

Ein Klient von *Wr* könnte folgendermaßen aussehen:

[1] Die vorgestellten Techniken aus Modula-3 leiten sich weitgehend von Modula-2+ ab [Rovner86].

```
PROCEDURE SomeOperation(...) RAISES {Wr.Failure, Wr.Error, ...};
   VAR
      failure: REFANY;
      code: Wr.Code;
BEGIN
   (* ... *)
   TRY
      PutChar(wr, ch);
   EXCEPT
   | Wr.Failure(failure) =>
      TYPECASE failure OF
      | FileStream.Failure => (* ... *)
      (* ... *)
      END;
   | Wr.Error(code) =>
      (* ... *)
   END;
END SomeOperation;
```

Hierbei wird mit **TRY**...**EXCEPT**...**END** eine Folge von Anweisungen mit einem Ausnahmenbearbeiter verknüpft. Der Ausnahmenbearbeiter hat die Möglichkeit, zwischen den einzelnen Ausnahmen zu unterscheiden und auf die zugehörigen Parameter zuzugreifen. Wenn notwendig, kann der Ausnahmenbearbeiter auch alle weiteren und möglicherweise unbekannten Ausnahmen abfangen.

Ada unterstützt keine Parametrisierungen von Ausnahmen, und Eiffel läßt nur einen *STRING*-Parameter zu. In Eiffel bezieht sich ein Ausnahmenbearbeiter immer auf eine vollständige Prozedur und nicht auf eine beliebige Anweisungsfolge innerhalb der Prozedur. Darüber hinaus erlaubt Eiffel den Ausnahmenbearbeitern nur Neustarts oder eine Weiterleitung der Ausnahme. Dies soll zwar ein stilles **RETURN** verhindern, allerdings ist es möglich, dies über einen Neustart zu simulieren.

Während diese eingebauten Mechanismen eine mächtige Kontrollstruktur für Ausnahmenbehandlungen sind, können sie genauso für große Sprünge über Prozedurgrenzen hinweg mißbraucht werden (analog zu *longjmp* in C). Im Falle von tief verschachtelten Ausfällen ist weder die detaillierte Fehlermeldung der unten liegenden Prozedur noch die Fehlerindikation auf der höheren Abstraktionsebene allein ausreichend, um eine aussagefähige Fehlermeldung zusammenzustellen. Für lesbare Fehlermeldungen wäre eine Beschreibung des Verlaufs der Ausfälle interessant. Im Beispiel: *A* fiel aus, da *B* wegen des Ausfalls von *C* ausfiel.

Modula-3, Eiffel und Exceptional C versuchen, auch asynchrone Ereignisse (wie z.B. **SIGINT** unter UNIX) als Ausnahmen zu behandeln. Hier erweist sich jedoch das vorgestellte Modell auf Basis von Ereignissen und Interessenten als günstiger, da es wichtig ist, daß asynchrone Ereignisse, die nicht aufgrund von Fehlern entstehen, unabhängig vom lokalen Kontext bearbeitet werden. Sonst kann es im Ebenenmodell passieren, daß keine der aktivierten Prozeduren darauf vorbereitet ist, oder daß eine ungestörte Fortsetzung nach der Bearbeitung unmöglich ist.

5.4. Charakterisierung von Ausnahmen

Es gibt verschiedene Ansätze, Ausnahmen zu benennen und zu parametrisieren. Ada unterstützt nur einfache Namen analog zu Variablennamen:

```
my_exception: exception;
-- ...
raise my_exception;
```

Bei CLU und Modula-3 kommen nur noch Parameter hinzu, die die Ausnahme näher beschreiben.

Dies ist jedoch unzureichend, da damit ein flacher Namensraum gegeben ist, der nicht weiter strukturiert werden kann. So ist es nicht möglich, alle Ausnahmen abzufangen, die zu einem bestimmten Bereich gehören (z.B. Ausnahmen im Ein- und Ausgabebereich), es sei denn durch das explizite Aufzählen aller Möglichkeiten. Dies führt dazu, daß bei Erweiterungen der Liste der Ausnahmen zahlreiche Listen ergänzt werden müssen. Natürlich ist es möglich, sich auf ganz wenige Typen von Ausnahmen zu beschränken, die dann nur noch über Parametrisierungen weiter spezifiziert werden.

Im Bereich objekt-orientierter Sprachen liegt es jedoch nahe, eine Typenhierarchie für Ausnahmen aufzubauen [Koenig90, Dony90]. Entsprechend kann auf der einen Seite eine sehr detaillierte Information über die Ausnahme generiert werden, die an anderer Stelle gemäß der dortigen Sicht und Interessenslage projiziert wird. Abbildung 26 zeigt, wie eine solche Hierarchie aussehen könnte. Im Beispiel könnte *RuntimeException*

Angaben über das Modul und die Zeile in der Quelle enthalten, bei der der Fehler aufgetreten ist. Bei der Erweiterung *RangeCheckException* könnten dann weitere Informationen über den zulässigen Bereich und den Wert des aus dem Bereich fallenden Index angegeben werden. Beim Basistyp *Exception* könnten Informationen darüber aufgenommen werden, ob eine Fortsetzung (*resumption*) möglich ist, oder ob eine Terminierung des Signalgebers zwingend ist.

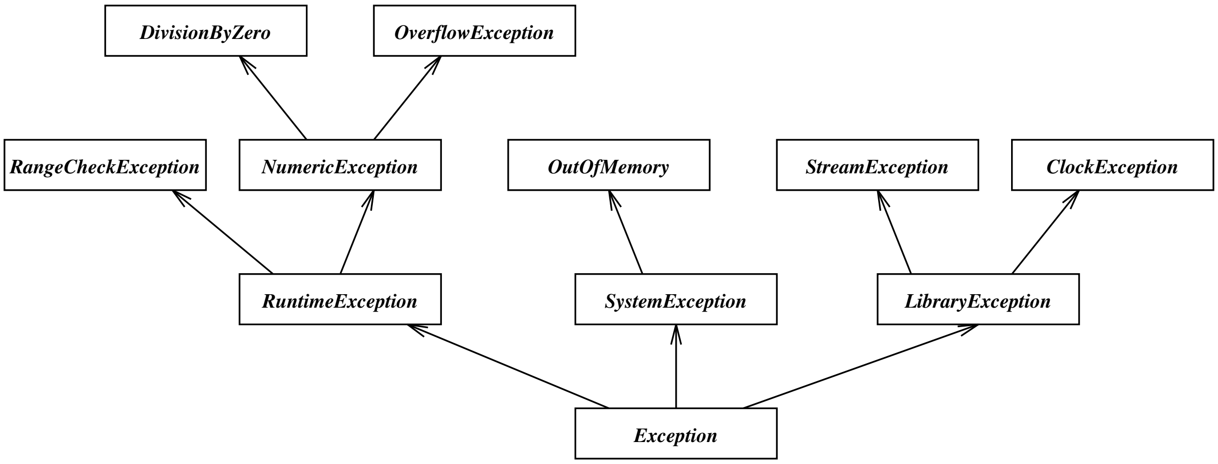

Abb. 26: Beispiel für eine Hierarchie von Ausnahmen

Während Goodenough noch zwischen drei unterschiedlichen signalisierenden Anweisungen unterschied (*escape*: keine Fortsetzung erlaubt, *notify*: eine Terminierung wird nicht erwünscht, *signal*: die Entscheidung über Terminierung oder Fortsetzung wird dem Ausnahmenbearbeiter überlassen), lassen sich diese Hinweise nun über den Typ oder Komponenten im Basistyp *Exception* charakterisieren.

5.5. Ausnahmen als Ereignisse

Da Oberon keine Sprachkonstrukte für Ausnahmenbehandlungen offeriert, liegt es nahe, Ausnahmen als Ereignisse zu betrachten. Ereignisse können hierarchisch organisiert werden und jede Information tragen, die von Interesse sein könnte. Hinzu kommt, daß Ereignisse bereits eine Textkomponente besitzen, die für Fehlermeldungen benutzt werden kann.

Ereignisse unterstützen in der bislang vorgestellten Form im wesentlichen nur globale Ausnahmenbearbeiter. Folgendes Beispiel zeigt dies:

```
DEFINITION SomeModule;

   IMPORT Events;

   CONST
      xxxFailed = 0; yyyFailed = 1; (* ... *)
   TYPE
      ErrorEvent = POINTER TO ErrorEventRec;
      ErrorEventRec =
         RECORD
            (Events.EventRec)
            code: INTEGER; (* xxxFailed etc. *)
            (* Komponenten mit weiteren Informationen *)
         END;
   VAR
      error: Events.EventType;

   PROCEDURE SomeOperation(....) : BOOLEAN;
      (* initiiere ‘exception’ und liefere FALSE zurueck bei Ausfaellen *)

END SomeModule.
```

Ereignisse werden vielfach ignoriert und es ist häufig umständlich, Fehlersituationen auf diese Weise zu notieren. Daher liegt es nahe, daß *SomeOperation* nach der Auslösung der Ausnahmenbehandlung mit **FALSE** als Rückgabewert beendet wird. **BOOLEAN**-Rückgabewerte sind besonders praktisch, da sie einerseits die

Verknüpfung einer Reihe von Operationen mit dem Und-Operator (in Oberon **&**) erlauben und zum anderen nicht versehentlich ignoriert werden können. Die Implementierung würde dann folgendermaßen aussehen:

```
MODULE SomeModule;

   (* ... *)

   PROCEDURE Error(code: INTEGER; ...);
      VAR
         event: ErrorEvent;
   BEGIN
      NEW(event); event.type := error;
      event.message := (* etwas in Abhaengigkeit von 'code' *);
      event.code := code;
      (* Ausfuellen weiterer beschreibender Komponenten *)
      Events.Raise(event);
   END Error;

   PROCEDURE SomeOperation(...) : BOOLEAN;
   BEGIN
      (* ... *)
      IF (* Ausfall entdeckt *) THEN
         Error(...); RETURN FALSE
      END;
      (* ... *)
      RETURN TRUE
   END SomeOperation;

BEGIN
   Events.Define(error);
END SomeModule.
```

Diese Lösung ist jedoch normalerweise unzureichend, da es schwierig ist, die Ausnahme zu dem Klienten und den Parametern von *SomeOperation* in Bezug zu setzen. Zwar erfährt der Klient über den Rückgabewert, daß *SomeOperation* fehlschlug, doch es erscheint sehr umständlich, dem Klienten oder dem zugehörigen Objekt mehr Informationen zugänglich zu machen. Hierfür wäre es notwendig, temporäre Interessenten für die potentiellen Ausnahmeereignisse anzugeben, die dann trotzdem immer noch Schwierigkeiten hätten, die Zuordnung durchzuführen. Dies würde wohl entsprechende globale Variablen unvermeidbar machen.

5.6. Objekt-orientierte Ausnahmenbehandlung

Cui und Gannon haben am Beispiel eines größeren Ada-Software-Systems und einiger empirischer Studien belegt, daß das Ebenenmodell in vielen Fällen unzureichend ist, und eine Ausnahmenbehandlung, die sich an den Objekten orientiert, der Vereinfachung dienen kann [Cui92]. In ihrer Spracherweiterung von Ada ermöglichen sie es, parametrisierte Ausnahmen direkt bei einem Datentyp zu definieren. Einer Variablen des entsprechenden Datentyps können dann direkt Ausnahmenbearbeiter zugeordnet werden. Folgendes Beispiel demonstriert dies:

```
generic

   type elem_type is private;
   tentative_size_limit: positive;

package stack_pkg is

   type stack is limited private
      #exception overflow(s: stack),
                 no_space(place: string) => put_line(place);

   procedure create(s: out stack);
   procedure push(s: in out stack; e: elem_type);
   -- ...
   procedure expand(s: in out stack; amount: in positive);

private
   -- ...
end stack_pkg
```

Die **#exception**-Klausel ermöglicht die Definition von objektbezogenen Ausnahmen. Die Ausnahme *overflow* wird initiiert, wenn der mit *tentative_size_limit* gegebene Rahmen überschritten wird, und *no_space* signalisiert, daß kein Speicherplatz mehr zur Verfügung steht, falls mit *expand* versucht worden ist, den Keller zu vergrößern.

Deklarationen könnten dann entsprechend dem folgenden Beispiel mit Ausnahmenbearbeitern versehen werden:

```
package int_stack is new stack_pkg(integer, 20);

s1: integer_stack.stack
       #when overflow(s: stack) => expand(s, growth_rate);
s2: integer_stack.stack
       #when overflow(s: stack) => retain(s, 90);
```

Hier soll *s1* in bestimmten Schritten erweitert werden, wenn die bisherige Größe überschritten wird, während bei *s2* alte Elemente des Kellers entfernt werden.

In Eiffel wird ein ähnliches Konzept unterstützt, wenngleich dort das Ebenenmodell Vorrang besitzt. Wenn eine Prozedur keinen Ausnahmenbearbeiter spezifiziert, so wird per Voreinstellung *default_rescue* aufgerufen. Die Prozedur *default_rescue* ist Teil der universellen Klasse *ANY* und kann von jeder Erweiterung überdefiniert werden. Da *default_rescue* bei *ANY* leer ist, wird die Ausnahme voreinstellungsgemäß weitergeleitet.

Im Unterschied zu der vorgestellten Ada-Erweiterung ist man in Eiffel gezwungen, jeweils eine Erweiterung der Klasse zu definieren, um *default_rescue* zu definieren. Zum anderen ist es nicht garantiert, daß *default_rescue* aufgerufen wird. Beide Ansätze verlangen eine statische Festlegung des Ausnahmenbearbeiters.

5.7. Objektbezogene Ereignisse

Angesichts der Defizite bei globalen Ereignistypen für Ausnahmenbehandlungen und der Attraktivität von Ausnahmenbearbeitern, die direkt mit Objekten verknüpft werden, liegt es in Oberon nahe, Ereignisse bei Bedarf in Bezug zu einem Objekt zu setzen.

Eine einfache Technik wäre die Definition eines individuellen Ereignistyps für jedes Objekt:

```
TYPE
   Object = POINTER TO ObjectRec;
   ObjectRec =
      RECORD
         eventType: Events.EventType;
      END;
```

Dies würde es erlauben, einen Interessenten für Ereignisse, die ein bestimmtes Objekt betreffen, zu definieren. Bei Ausnahmen, die in Zusammenhang mit einem Objekt entstehen, würde dann der objekt-spezifische Ereignistyp verwendet werden. Diese Technik hat allerdings folgende Nachteile:

- Einem Interessenten an einem objektbezogenen Ereignis liegt das Objekt selbst nicht vor. Dies könnte allerdings gelöst werden durch die Erweiterung des Ausfallereignisses um eine Komponente, die auf das Objekt verweist.
- Nach wie vor bleibt es schwierig, nach einer fehlgeschlagenen Operation an die Informationen über den Ausfall zu gelangen.
- Alle Objekte, die in Frage kommen, müssen eine Erweiterung von *ObjectRec* sein, und alle Konstruktoren müssen *Events.Define* für den Ereignistyp aufrufen.

All diese Nachteile werden in der Ulmer Oberon-Bibliothek durch das Modul *RelatedEvents* behoben. Dieses Modul erwartet von den Objekten nur, daß sie eine Erweiterung von *Disciplines.Object* sind. Damit ist *RelatedEvents* in der Lage, seine eigene Datenstruktur zu einem Objekt zu unterhalten, und es gibt keinen erhöhten Initialisierungsaufwand. Nur wenn es zu objektbezogenen Ereignissen kommt oder wenn besondere Maßnahmen zu treffen sind, ist *RelatedEvents* gezwungen, private Informationen zu einem Objekt zu verwalten.

Auf der Seite derjenigen, die Ereignisse generieren, ist der Unterschied minimal. Statt *Events.Raise* wird *RelatedEvents.Raise* aufgerufen, das als zusätzlichen Parameter ein Objekt akzeptiert:

```
DEFINITION RelatedEvents;

   IMPORT Disciplines, Events;

   (* ... *)
```

```
      PROCEDURE Raise(object: Disciplines.Object; event: Events.Event);

   END RelatedEvents.
```

Bemerkenswert ist, daß diese Technik keine zusätzlichen Parameter bei Operationen benötigt, die möglicherweise eine Ausnahmenbehandlung initiieren. Statt dessen wird einfach der Parameter genommen, auf den sich die Operation bezieht:

```
MODULE SomeModule; (* revidierte Version auf Basis von RelatedEvents *)

   (* ... *)

   PROCEDURE Error(object: Disciplines.Object; code: INTEGER; ...);
      VAR
         event: ErrorEvent;
   BEGIN
      NEW(event); event.type := error;
      event.message := (* etwas in Abhaengigkeit von 'code' *)
      event.code := code;
      (* Ausfuellen weiterer beschreibender Komponenten *)
      RelatedEvents.Raise(object, event);
   END Error;

   PROCEDURE SomeOperation(object: ...; ...) : BOOLEAN;
   BEGIN
      (* ... *)
      IF (* Ausfall entdeckt *) THEN
         Error(object, ...); RETURN FALSE
      END;
      (* ... *)
      RETURN TRUE
   END SomeOperation;

BEGIN
   Events.Define(error);
END SomeModule.
```

Außerdem offeriert *RelatedEvents* vier Reaktionsmöglichkeiten auf objektbezogene Ereignisse, die auch kombiniert benutzt werden können:

- Ereignisse können beim Objekt zur späteren Untersuchung an eine Ereignisliste angehängt werden.
- Ereignisse können an andere Objekte weitergeleitet werden.
- Das Eintreten eines Ereignisses kann ein objektspezifisches Sonderereignis hervorrufen, das auf das Objekt und das objektbezogene Ereignis verweist.

Wird keine der angegebenen Möglichkeiten gewählt, so werden Ereignisse an *Events.Raise* weitergeleitet. Wenn also keine Vorkehrungen getroffen worden sind, um objektbezogene Ereignisse zu bearbeiten, dann führen sie per Voreinstellung zum Abbruch des Programms. Natürlich ist es möglich, pauschal *Events.Ignore* für den zugehörigen Ereignistyp aufzurufen oder globale Interessenten anzugeben. Dies ist jedoch eher die Ausnahme.

Folgendes Beispiel zeigt, wie auf dieser Basis Ausnahmenbehandlungstechniken realisiert werden können, die dem Ebenenmodell entsprechen:

```
PROCEDURE ClientOfSomeOperation(object: ...; ...) : BOOLEAN;
   VAR
      queue, error: RelatedEvents.Queue;
BEGIN
   RelatedEvents.QueueEvents(object);
   (* ... *)
   IF ~SomeOperation(object, ....) OR
         ~AnotherOperation(object, ...) THEN
      RelatedEvents.GetQueue(object, queue);
      error := queue;
      WHILE error # NIL DO
         IF error.event IS SomeModule.ErrorEvent THEN
            (* ... *)
         ELSIF error.event IS AnotherModule.ErrorEvent THEN
            (* ... *)
```

```
         END;
         error := error.next;
      END;
      (* Entscheidung darueber, ob man es erneut probiert oder
         einen Ausfall signalisiert
      *)
      IF (* Ausfall *) THEN
         RelatedEvents.AppendQueue(object, queue);
         Error(object, ...); RETURN FALSE
      END;
   END;
   (* ... *)
END ClientOfSomeOperation;
```

Mit *RelatedEvents.QueueEvents* wird sichergestellt, daß alle Ereignisse in einer objektbezogenen Liste gesammelt werden. Sollte *QueueEvents* bereits vorher aufgerufen worden sein, so bewirkt dies keine Änderung. Mit *RelatedEvents.GetQueue* kann diese Liste von dem Objekt entfernt und untersucht werden. Möchte man später die Liste wieder an das Objekt hängen, um sie auch übergeordneten Klienten zur Verfügung zu stellen, so kann dies durch *RelatedEvents.AppendQueue* geschehen.

Bei traditionellen Techniken kann entweder die ursprüngliche Ausfallbeschreibung oder eine neue Beschreibung weitergeleitet werden — jedoch nicht beides. Mit Hilfe von *RelatedEvents.QueueEvents* können alle Ausfallbeschreibungen erhalten und später untersucht werden. Dabei handelt es sich in der Regel nicht um Verletzung von Abstraktionsgrenzen. Im Gegenteil, durch die Verwendung von *up calls* können Ausnahmen des Moduls *C* von größerem Interesse für *A* sein als für *B*. Während beispielsweise für *Write.Int* und *Streams* der exakte Grund für einen Schreibfehler uninteressant ist, so ist auf der höheren Ebene durchaus bekannt, daß es sich um eine Datei auf einem Dateisystem des Betriebssystems handelt, die z.B. von einer vollen Platte bedroht sein kann. Statt einem pauschalen "write error" kann dann dem Benutzer eine aussagefähige Fehlermeldung gegeben werden.

Auch objekt-orientierte Ausnahmenbearbeiter können mit Hilfe von *RelatedEvents* definiert werden. Das vorgestellte Beispiel für die Erweiterung von Ada könnte etwa folgendermaßen in Oberon aussehen:

```
DEFINITION Stacks;

   IMPORT Disciplines, Events, Objects;

   CONST
      overflow = 0; noSpace = 1;
   TYPE
      ErrorEvent = POINTER TO ErrorEventRec;
      ErrorEventRec =
         RECORD
            (Events.EventRec)
            code: INTEGER; (* overflow oder noSpace *)
            fixed: BOOLEAN; (* Problem durch Ausnahmenbehandlung behoben? *)
         END;
   VAR
      error: Events.EventType;

   TYPE
      Stack = POINTER TO StackRec;
      StackRec = RECORD (Disciplines.ObjectRec) END;

   PROCEDURE Create(VAR stack: Stack; tentativeSizeLimit: INTEGER) : BOOLEAN;

   PROCEDURE Push(stack: Stack; element: Objects.Object) : BOOLEAN;

   (* ... *)

   PROCEDURE Expand(stack: Stack; amount: INTEGER) : BOOLEAN;

END Stacks.
```

Das Beispiel verlangt die Möglichkeit, die Ausführung einer Operation fortzusetzen, nachdem eine Ausnahmenbehandlung initiiert worden ist. Aus diesem Grund wurde *Stacks.EventRec* um die Komponente *fixed* erweitert. Entsprechend kann *Stacks* nach *Events.Raise* überprüfen, ob *fixed* auf **TRUE** gesetzt worden ist, und dann einen erneuten Versuch unternehmen.

```
PROCEDURE ExpandingHandler(event: Events.Event);
   VAR
      stackevent: Stacks.ErrorEvent;
BEGIN
   WITH event: RelatedEvents.Event DO
      stackevent := event.event(Stacks.ErrorEvent);
      IF stackevent.code = Stacks.overflow THEN
         stackevent.fixed := Stacks.Expand(event.object, growthRate);
      END;
   END;
END ExpandingHandler;

PROCEDURE CreateExpandingStack(VAR stack: Stacks.Stack) : BOOLEAN;
   VAR
      stack: Stacks.Stack;
      eventType: Events.EventType;
BEGIN
   IF Stacks.Create(stack, 20) THEN
      RelatedEvents.GetEventType(stack, eventType);
      Events.Handler(eventType, ExpandingHandler);
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END CreateExpandingStack;
```

ExpandingHandler findet hier zunächst das Ereignis von *RelatedEvents* vor, das nur die Verweise auf das eigentliche Ereignis und das Objekt enthält. Wenn *ExpandingHandler* dann feststellt, daß es sich um ein ‘‘overflow’’-Ereignis handelt, wird versucht, den Keller zu expandieren. Nur wenn dies gut geht, wird *fixed* auf **TRUE** gesetzt und ein erneuter Versuch ermöglicht.

CreateExpandingStack versucht zunächst, einen Keller anzulegen, und erfährt dann durch *RelatedEvents.GetEventType* den zum Objekt gehörenden Ereignistyp, für den *ExpandingHandler* als Interessent angegeben wird.

RelatedEvents.GetEventType und *RelatedEvents.QueueEvents* sind kombinierbar. Wenn *Stacks.Push* vor einem erneuten Versuch das Ereignis nicht mit *RelatedEvents.GetQueue* entfernt, steht es nachher auch dann zur Verfügung, wenn der zweite Versuch erfolgreich ist. Bemerkenswert ist hier, daß beliebig viele Module ihr Interesse an Ereignissen an einem bestimmten Objekt bekunden können.

Die gezeigten Techniken sind in der Praxis jedoch relativ selten notwendig. Häufig ist es nur nötig, die Historie über die Ausfälle beim Objekt aufzubewahren, um eine sinnvolle Fehlermeldung zusammenzustellen, die die komplette Geschichte erzählt. Da die *message*-Komponente mit ihrer Begrenzung auf 80 Zeichen manchmal unzureichend für eine textuelle Problembeschreibung ist, gibt es eine Möglichkeit, Ausgabeprozeduren mit Ereignistypen zu assoziieren:

```
PROCEDURE PrintError(s: Streams.Stream; event: Events.Event);
BEGIN
   WITH event: ErrorEvent DO
      (* Ausgabe einer detaillierten Fehlermeldung auf 's' *)
   END;
END PrintError;

PROCEDURE SetupErrorHandling;
   (* wird waehrend der Initialisierung des Moduls aufgerufen *)
BEGIN
   Events.Define(error);
   Errors.AssignWriteProcedure(error, PrintError);
END SetupErrorHandling;
```

Errors.AssignWriteProcedure nutzt hierbei aus, daß *Events.EventType* eine Erweiterung von *Disciplines.Object* ist, und deponiert die Prozedur mit Hilfe einer Disziplin. Später kann dann mit *Errors.Write* eine Ausgabe eines Ereignisses erfolgen. Wurde für das Ereignis *Errors.AssignWriteProcedure* vorher nicht aufgerufen, so wird statt dessen die *message*-Komponente ausgegeben.

Ein einfaches Kopierkommando für UNIX-Dateien, das die Fehlermeldungen an den Benutzer selbst ausgeben soll, könnte dann folgendermaßen aussehen:

```
PROCEDURE Copy(infile, outfile: ARRAY OF CHAR);
   VAR
      errors: Disciplines.Object; (* Dummy-Objekt fuer die Ausnahmenbehandlung *)
      infile, outfile: NameSpaces.Name;
      in, out: Streams.Stream;
BEGIN
   NEW(errors); RelatedEvents.QueueEvents(errors);
   IF ~UnixFiles.Open(in, infile, UnixFiles.read, errors) OR
         ~UnixFiles.Open(out, outfile,
                         UnixFiles.write + UnixFiles.create, errors) THEN
      Conclusions.Conclude(errors, Errors.error, "Copy");
      RETURN
   END;
   RelatedEvents.Forward(in, errors);
   RelatedEvents.Forward(out, errors);
   IF ~Streams.Copy(in, out) OR
         ~Streams.Close(in) OR ~Streams.Close(out) THEN
      Conclusions.Conclude(errors, Errors.error, "Copy");
   END;
END Copy;
```

Da bei Konstruktoren ein Bezugsobjekt noch nicht zur Verfügung steht, ist hier ein zusätzlicher Parameter notwendig, um eine Ausnahmenbehandlung zu ermöglichen. Hierfür verwendet man ein ‘‘leeres’’ Objekt vom Typ *Disciplines.Object*, das nur dazu dient, die Fehler aufzufangen. Da bei *Streams.Copy* Ausfälle sich sowohl auf *in* als auch *out* beziehen können, erweist es sich hier als günstig, die Ereignisse bei beiden Objekten an *errors* weiterzuleiten. *Conclusions.Conclude* gibt dann die Liste der Ereignisse in zusammengestellter Form auf *Streams.stderr* aus.

Im obigen Beispiel könnte eingewendet werden, daß bei einem Ausfall von *Streams.Copy* weder *in* noch *out* geschlossen werden. Dies läßt sich dadurch beheben, daß man alle drei Operationen hintereinander ausführt und erst anschließend überprüft, ob es zu Ausfällen gekommen ist:

```
IF ~Streams.Copy(in, out) THEN (* Ausfall zunaechst ignorieren *) END;
Streams.Release(in); (* aequivalent zu Streams.Close ohne Rueckgabewert *)
Streams.Release(out);
IF RelatedEvents.EventsPending(errors) THEN
   Conclusions.Conclude(errors, Errors.error, "Copy");
END;
```

RelatedEvents.EventsPending liefert **TRUE** zurück, wenn die Liste der zum Objekt gehörenden Ereignisse nicht leer ist. Und genau in diesem Fall wird eine Fehlermeldung ausgegeben.

5.8. Klassenbezogene Ereignisse

Einige objekt-orientierte Sprachen wie z.B. Smalltalk und Lore unterstützen klassenbezogene Ausnahmenbehandlungen. Teilweise kann diese Technik dazu benutzt werden, objektbezogene Ausnahmenbehandlungen zu simulieren, wenn statische Erweiterungen einer Klasse definiert werden, um einzelne Varianten mit unterschiedlichen Ausnahmenbehandlungen zu erhalten. Generell ist allerdings eine klassenbezogene Ausnahmenbehandlung relativ weit entfernt vom aktuellen Geschehen und zu vergleichen mit globalen zugeordneten Ausnahmenbearbeitern.

Auf der Basis von objektbezogenen Ausnahmenbehandlungen können natürlich relativ leicht auch klassenbezogene Ausnahmenbehandlungen definiert werden. So kann ein Konstruktor für das neue Objekt sogleich einen Interessenten eintragen. Da beliebig viele Interessenten für objektbezogene Ereignisse angegeben werden können, gibt es keine Konflikte, wenn sich die einzelnen Interessenten kooperativ verhalten.

5.9. Fehlertoleranz

Normalerweise führen Ausfälle, die sich nicht sofort beheben lassen, zum endgültigen Abbruch einer Operation und durch transitive Ausfälle bis hin zum Ausfall einer höheren Einheit (z.B. Programm, Kommando oder Koroutine). In einigen Bereichen, wie z.B. in Software-Systemen, von derem korrekten Funktionieren Menschenleben abhängen, reichen jedoch Fehlermeldungen und einige Aufräumaktionen nicht aus. In so einer Situation ergibt sich dann die Frage, ob nicht ein erneuter Versuch oder eine alternative Vorgehensweise erfolgreich sein könnte. Mehrfache Versuche bzw. Berechnungen können in drei Dimensionen stattfinden: Zeit (einfache Wiederholung), Raum (durch die Verwendung von mehreren Prozessoren oder Rechnern) und Software (mehrere Varianten einer Operation) [Avižienis85, Kelly89].

Ein bekanntes Beispiel ist die Redundanz von Hardware und Software beim Computer-System des Space Shuttle [Spector84]. Während bei der Hardware fünf gleiche Rechner zur Verfügung stehen, wurden zwei Varianten für die Software weitgehend unabhängig von zwei Firmen entwickelt. Zwar laufen beide Programme gleichzeitig (das primäre Programm PASS auf 4 Rechnern und das sekundäre Programm BFS nur auf einem Rechner), jedoch ist immer nur ein Programm operativ. Nur durch eine manuelle Aktion eines Astronauten kann im Notfall von PASS auf BFS umgeschaltet werden.

Betrachtungen über Fehlertoleranz sind in erster Linie sinnvoll für Operationen, deren Vorbedingungen erfüllt wurden, die aber trotzdem nicht in der Lage sind, die Nachbedingung zu erfüllen. Wurde die Vorbedingung nicht erfüllt, so liegt entweder ein Ausfall des Aufrufers oder eine Testsituation vor. Wenn es für eine Operation Anfangszustände gibt, die die Vorbedingung erfüllen, und die trotzdem zu einer Verletzung der Nachbedingung bei der Operation führen, dann sind in der Operation latente Programmierfehler vorhanden.

Selbst wenn alle denkbaren Ausnahmefälle abgefangen werden und alternative Varianten ohne latente Programmierfehler zur Verfügung stehen, ist die Korrektheit des gesamten Software-Systems noch nicht gewährleistet, da zwischen dem Auftreten des latenten Programmierfehlers und seiner nicht notwendigerweise garantierten Entdeckung einige Zeit vergehen kann [Cristian82]. Während dieser Zeit ist das Verhalten des Systems undefiniert. Deswegen muß gerade auch bei fehlertoleranten Systemen bei von Fehlern bedrohten Einzelkomponenten darauf geachtet werden, daß alle kritischen Teilfunktionen mit Überprüfungen aller denkbaren Ausfallmöglichkeiten umgeben sind.

Üblicherweise strukturiert man fehlertolerante Software-Systeme auf Basis von *recovery blocks*. Hierbei handelt es sich um Prozeduren oder Programmkonstrukte, die Varianten starten, wiederholen oder durchprobieren und auf Korrektheit untersuchen. Es gibt kompliziertere Formen, bei denen die einzelnen Varianten einem Akzeptanztest unterzogen werden, oder bei denen mehrere Varianten auf jeden Fall durchprobiert und deren Ergebnisse anschließend miteinander verglichen werden.

Um die Diskussion zu vereinfachen, kann man jedoch davon ausgehen, daß eine Variante einen Ausfall sofort selbst bemerkt und eine entsprechende Ausnahmenbehandlung initiiert. In diesem Fall braucht der *recovery block* nur dann eine weitere Variante zu probieren, wenn die vorherige mit einem Ausfall beendet wurde. Fallen alle Varianten aus, dann fällt der *recovery block* selbst aus.

Eiffel unterstützt diese Technik mit der **retry**-Anweisung [Meyer88, §7.10.4], die es in ähnlicher Form schon in einer Pascal-Erweiterung von Wasserman gab [Melliar-Smith77]:

```
do_task is
   local
      times_tried: INTEGER
   do
      if times_tried = 0 then
         implementation_1
      elsif times_tried = 1 then
         implementation_2
      end
   rescue
      times_tried := times_tried + 1;
      if times_tried < 2 then
         -- moeglicherweise Instruktionen, die zu einem konsistenten
         -- Zustand fuehren
         retry
      end
   end -- do_task
```

In dem Beispiel dient *times_tried* dazu, die Anzahl der probierten Varianten durchzuzählen. Da in Eiffel lokale Variablen vom Typ *INTEGER* automatisch zu Beginn auf 0 gesetzt werden, ist für *times_tried* keine besondere Initialisierung notwendig. Die **rescue**-Klausel enthält den Ausnahmenbearbeiter. Solange noch weitere Varianten zur Auswahl stehen, wird mit **retry** ein neuer Versuch gestartet. Wenn die **rescue**-Klausel nicht mit einem **retry** beendet wird, kommt es zur Weiterleitung der Ausnahme.

Auf Basis von *RelatedEvents* könnte in Oberon eine Lösung folgendermaßen aussehen:

```
PROCEDURE DoTask(object: SomeObject; ...) : BOOLEAN;
   CONST
      numberOfVariants = 2;
   VAR
      timesTried: INTEGER;
      succeeded: BOOLEAN;
BEGIN
   timesTried := 0; succeeded := FALSE;
   WHILE ~succeeded & (timesTried < numberOfVariants) DO
      CASE timesTried OF
      | 0: succeeded := Implementation1(object, ...);
      | 1: succeeded := Implementation2(object, ...);
      END;
      INC(timesTried);
   END;
   RETURN succeeded
END DoTask;
```

Entsprechend der Konventionen muß *SomeObject* eine Erweiterung von *Disciplines.Object* sein. Die einzelnen Varianten können dann Meldungen über ihren Ausfall dem Objekt zuordnen. Natürlich könnte man *DoTask* noch dahingehend erweitern, daß es selbst ein Ausnahmeereignis initiiert, wenn keine der Varianten erfolgreich gewesen ist.

5.10. Zusammenfassung

Die vorgestellten Techniken in Oberon können als weiteren Beleg dafür gesehen werden, daß spezielle Sprachmittel häufig durch Bibliotheken und einige einfache Konventionen ersetzt werden können, die mindestens genauso gut sind.

Die vorgestellte Technik unterstützt sowohl Ausnahmenbehandlungen entlang der Aufrufkette (analog zu der auf Sprüngen basierenden Technik) als auch objekt- und klassenbezogene Ausnahmenbehandlungen. Durch die vielfältigen Weiterleitungen, sowohl auf Objektebene (entweder individuell durch ein erneutes *RelatedEvents.Raise* oder generell durch *RelatedEvents.Forward*) als auch auf Prozedurebene, verbunden mit der Möglichkeit, sowohl alle interessanten Informationen bezüglich einer Ausnahme als auch die Historie aufzuzeichnen, wird ein Ausnahmenbehandlungssystem erreicht, das in seiner Mächtigkeit alle gängigen Varianten übertrifft.

Bei alternativen Techniken in objekt-orientierten Sprachen kann normalerweise eine Ausnahme entweder nur objektbezogen oder entlang der Aufrufkette bearbeitet werden. *RelatedEvents* ermöglicht statt dessen die Kooperation von vielen Interessenten an einer Ausnahme. Dies setzt allerdings ggf. kooperative Techniken der einzelnen Interessenten voraus, damit es nicht zu mehrfachen Reparaturversuchen kommt. Eine mögliche Technik hierfür sind Erweiterungen des Ereignisverbunds mit Erledigungsvermerken.

Keiner der Nachteile der traditionellen Technik gilt für die vorgestellte Lösung, denn:

- Durch die Konvention, daß Operationen, die fehlschlagen können, einen **BOOLEAN**-Rückgabewert besitzen, ist es notwendig, für jeden Aufruf einer Operation einen Test zu formulieren.
- Normalerweise sind keine zusätzlichen Parameter notwendig, da das zu bearbeitende Objekt als Aufhänger dient.
- Ausfälle werden nicht stillschweigend ignoriert. Selbst wenn der **BOOLEAN**-Rückgabewert ignoriert wird, wird der Klient bzw. das Objekt benachrichtigt, wenn dies zuvor gewünscht wurde.
- Informationen über Ausfälle gehen nicht verloren. Mit Hilfe von *RelatedEvents.QueueEvents* kann die vollständige Ausfallhistorie aufgezeichnet werden.

Es verbleibt jedoch ein Nachteil. Fehler, die von Überprüfungen entdeckt werden, die der Übersetzer generiert hat (wie z.B. für Indexgrenzen bei Feldern), führen zu Ereignissen, die direkt *Events.Raise* zugeleitet werden. Damit können sie nicht in Abhängigkeit des aktuellen Kontexts bearbeitet werden, und eine Fortsetzung der Operation ist natürlich nicht möglich. Dies ist der einzige Fall, bei dem Sprünge tatsächlich vorteilhaft sein könnten. Da das Ulmer Oberon-System Koroutinen unterstützt, können sie als Begrenzung bei einem solchen Ausfall verwendet werden, da Koroutinen von Ausnahmenbearbeitern terminiert werden können (Einzelheiten hierzu in §7.5).

Durch den Verzicht auf Sprünge über Prozedurgrenzen hinweg (*long jumps*) ergibt sich einige Vorteile:

- Die mit diesen Sprüngen verbundene Mißbrauchsmöglichkeit fällt weg.
- Es ist bei normalem Ablauf kein zusätzlicher Aufwand gegeben, wenn man von der Rückgabe eines **BOOLEAN**-Wertes absieht. Zahlreiche Implementierungen wie z.B. Exceptional C [Gehani92] und verschiedene Lösungen für Modula-2 [Padiou90, Egorov92, Egorov93] sind gezwungen, im voraus den Kontext in aufwendiger Weise zu sichern, so daß hier ein erheblicher Zusatzaufwand einzuberechnen ist. Zwar gibt es für *long jumps* bei Ausnahmenbehandlungen spezielle Techniken, die nach einigem Voraufwand beim Start des Programms keinen Zusatzaufwand bei normalem Programmverlauf benötigen [Koenig90]. Doch ist dann der Aufwand im Falle einer Ausnahmenbearbeitung extrem hoch, da für die Lokalisierung des Ausnahmenbearbeiters zahlreiche Tabellen zu konsultieren sind.

6. Abstraktionen für Ein- und Ausgabe

6.1. Einführung

Sowohl auf der Ebene von Betriebssystemen als auch von Programmiersprachen und zugehörigen Bibliotheken gehören Ein- und Ausgabe zu den wichtigsten Abstraktionen. Selbst bei sehr frühen Computer-Anlagen und Programmiersprachen gibt es Abstraktionen, die es ermöglichen, auf die Unterscheidung zwischen Ausgaben auf einen Drucker oder in eine Datei zu verzichten.

Ältere Programmiersprachen wie Fortran, Algol, Cobol und Pascal bieten diese Abstraktionen als Bestandteil der Sprache an. Obwohl es zugehörige Standards gibt, erweisen sich jedoch die fest integrierten Abstraktionen als wenig portabel, da wegen der fehlenden Erweiterbarkeit häufig lokale Besonderheiten ihren Eingang in einzelne Implementierungen der Sprache finden. Somit sind besonders die Abstraktionen für Ein- und Ausgabe sowohl ein Beispiel für die Notwendigkeit objekt-orientierter Techniken als auch ein Prüfstein für die Bibliothek und die dafür notwendigen Sprachmittel.

Umso überraschender ist es, daß viele Bibliotheken objekt-orientierter Programmiersprachen außerordentlich eng mit dem jeweiligen Betriebssystem in diesem Bereich verknüpft sind. So akzeptiert beispielsweise einer der Konstruktoren der Basisklasse von *ostream* in der in [Stroustrup86] veröffentlichten Form der C++ Streams-Bibliothek einen UNIX-Dateideskriptor. Erst später wurde dies besser herausgetrennt [Stroustrup91]. Ähnlich gestaltet sich die in [Meyer88] veröffentlichte Eiffel-Bibliothek. Anstatt Abstraktionen von Implementierungen zu trennen, ranken sich in beiden Fällen die Basisklassen um die Schnittstellen zum Betriebssystem. Zwar sind Überdefinitionen möglich — jedoch müssen diese mit viel Sorgfalt erfolgen, damit nicht versehentlich die falsche Implementierung zum Vorschein kommt.

Wenn Ein- und Ausgabe unabhängig von irgendeiner konkreten Systemumgebung betrachtet werden, dann stellt sich die Frage nach geeigneten Abstraktionen völlig neu:

- Betrachtet man bei Ein- und Ausgabe komplexe Objekte oder nur Bytes?
- Sollen Ein- und Ausgabe auch im Sinne von komplexen Benutzerschnittstellen gesehen werden, oder ist das Bild einer Sequenz von Objekten oder Bytes, auf die man ggf. positionieren kann, ausreichend?

Die zu Smalltalk gehörende Bibliothek betrachtet Ein- und Ausgabe zunächst allgemein als Zugriff auf Kollektionen von Objekten und byte-orientierte Ein- und Ausgabe als Spezialfall davon (*ExternalStream*). Damit wird jedoch die **is-a**-Beziehung verletzt, da nicht mehr beliebige Objekte an einen externen Strom angefügt werden können. Wegen dieser Problematik erweist es sich als günstig, den Zugriff auf Kollektionen von byte-orientierter Ein- und Ausgabe zu trennen. Beide Bereiche sind so unterschiedlich, daß sie kaum unter eine gemeinsame Oberklasse zu integrieren sind, wenn man **is-a**-Beziehungen nicht verletzen möchte. Aus dem gleichen Grund ist es ratsam, Benutzerschnittstellen aus der Betrachtung herauszunehmen.

Somit verbleiben nur Folgen von Bytes in der engeren Betrachtung. Obwohl dies im Umfeld objekt-orientierter Systeme ein Rückfall zu sein scheint, sind Abstraktionen für Bytefolgen von großer Bedeutung. Immer noch wird alles in Computern in Form von Bits und Bytes repräsentiert. Sowohl Speicherung von Daten als auch Kommunikation erfolgt letztlich immer in Form von Bytes, die auf Systemebene nicht mehr weiter interpretiert werden. Erst darauf aufbauend können komplexe Objekte auf Basis von Bytes repräsentiert werden. Wenn die Abbildung von Objekten in Bytefolgen aus gutem Grund nicht in der Programmiersprache oder der Systemumgebung realisiert wird, muß dies in erweiterbarer Form in der Bibliothek geschehen.

Die Ulmer Oberon-Bibliothek deckt alle Bereiche ab: Es gibt sowohl Abstraktionen für Bytefolgen als auch darauf aufbauend Abstraktionen für persistente Objekte. Ein persistentes Objekt kann in eine Bytefolge und umgekehrt verwandelt werden. Da die Bytefolge selbst eine Abstraktion ist, wird sie somit zum Transportmedium für Objekte. Im Gegensatz zu einer Reihe anderer objekt-orientierter Systeme, die Persistenz auf Systemebene in Form von Spracherweiterungen oder vergleichbarem anbieten (z.B. Eiffel, Smalltalk und Modula-3), können bei der Ulmer Oberon-Bibliothek beispielsweise persistente Objekte via Interprozeßkommunikation über das Netz geschickt werden.

6.2. Abstraktionen für Bytefolgen und Kommunikation

Zwei Abstraktionen sind im Bereich der byteweisen Ein- und Ausgabe üblich:

- die der Datei, die einem Feld mit Bytes entspricht (siehe Abbildung 27), und
- die der Kommunikation zwischen einem Produzenten, der Bytes generiert, und einem Konsumenten, der sie liest (siehe Abbildung 28).

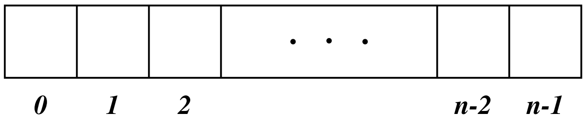

Abb. 27: Abstraktion einer Datei der Länge *n*

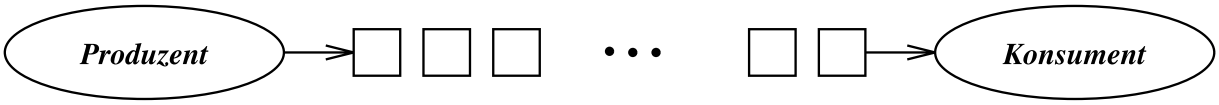

Abb. 28: Abstraktion der Kommunikation zwischen zwei Partnern

Viele Betriebssysteme und Bibliotheken haben dafür getrennte Operationen entwickelt und die Sache noch weiter verkompliziert durch die Strukturierung von Dateien und Kommunikationskanälen. Es ist das Verdienst von UNIX, hier zur Vereinheitlichung und Vereinfachung beigetragen zu haben. Seit dem UNIX-System auf der PDP7 im Jahr 1969 bis zu den heutigen UNIX-Versionen ist man mit den Zugriffsoperationen *read*, *write*, *seek* und *close* (siehe Tabelle 2) sowohl für Dateizugriffe als auch Interprozeßkommunikation ausgekommen [Ritchie84].

read(fd, buf, nbytes)	Einlesen von bis zu *nbytes* Bytes
write(fd, buf, nbytes)	Ausgabe von genau *nbytes* Bytes
seek(fd, offset, whence)	Positionierung (sowohl absolut als auch relativ)
close(fd)	Freigabe des Kanals

Tab. 2: UNIX-Zugriffsfunktionen für Ein- und Ausgabe

Bei *read* und *write* werden jeweils nur eine Referenz (Dateideskriptor), eine Speicherfläche und die Anzahl der Bytes angegeben. Sowohl *read* als auch *write* liefern die Anzahl der umgesetzten Bytes zurück und setzen implizit die Position entsprechend weiter. Handelt es sich um eine Datei, so sind Positionierungen (sowohl absolut als auch relativ) mit *seek* möglich. Mit *close* wird die Verbindung geschlossen.

Die Operation *read* ist in dieser Form besonders für Kommunikation geeignet: Wenn für den Konsumenten schon eine positive Anzahl von Bytes vorliegt, werden sie sofort geliefert, auch wenn die Quantität unter der geforderten liegt. Auf diese Weise sind keine zusätzlichen Abstraktionen für Pakete erforderlich.

UNIX verzichtet auch auf die Strukturierung von Dateien in Zeilen, Verbunde oder vergleichbares. Dafür erlaubt es beliebige Positionierungen, und auf jeder Position kann jede gewünschte Quantität von Bytes gelesen oder geschrieben werden.

Die Schnittstelle der Ulmer Oberon-Bibliothek für *Streams* orientiert sich an den UNIX-Operationen für Ein- und Ausgabe. Dies geschah weniger aus dem Grund, daß die Bibliothek unter UNIX entwickelt wurde, sondern weil die Ein- und Ausgabe-Operationen von UNIX in ihrer Einfachheit ohne Konkurrenz sind.

6.3. Design-Prinzipien für *Streams*

Nach dem Vorbild der UNIX-Schnittstelle wurde das Modul *Streams* auf der Grundlage folgender Prinzipien entworfen:

- Offeriert werden nur Zugriffsoperationen für byteweise Ein- und Ausgabe. Formatierte Ein- und Ausgabe von Basistypen und die Persistenz bleiben anderen Modulen überlassen.
- Es erfolgt keine Interpretation der Bytes. Es gibt keine Zeilen, keine Verbunde und keine Blöcke. Die einzige Ausnahme hierfür ergibt sich aus der optionalen Pufferung: Bei zeilenweiser Pufferung kann ein Zeilentrenner angegeben werden.
- *Streams* selbst unterstützt dementsprechend auch keine Parameter wie Angaben über Füllzeichen (*white space*), Formatierstile und Feldtrenner. Dies bleibt entsprechenden Disziplinen überlassen, die von auf *Streams* aufbauenden Modulen unterhalten werden.

Neben der Einfachheit ist die Laufzeit-Effizienz ein sehr wichtiges Kriterium. Wenn man von sehr rechenintensiven Programmen absieht, wird im Normalfall die Effizienz im wesentlichen durch den Ein- und

Ausgabebereich bestimmt. Ein- und Ausgabe verursachen fast bei allen Programmen die Mehrzahl der Systemaufrufe, die vergleichsweise teuer sind gegenüber normalen Prozeduraufrufen.

Aus diesem Grund verwendet *Streams* einen Schnittstellenverbund und unterstützt das Versenden von Aufträgen nur für Operationen von möglichen Erweiterungen. Den größten Effizienzgewinn bringen jedoch die optionalen Pufferungsmodi. Natürlich wäre es denkbar gewesen, die Pufferung in einem separaten Modul abzuhandeln. Dies wäre jedoch eine künstliche Trennung, die keine wesentlichen Vorteile gebracht hätte. Würde *Streams* keine Pufferung anbieten, so müßte es nur die Operationen einfach durchreichen und würde entsprechend keine "Intelligenz" besitzen.

Die optionale Pufferung trägt dazu bei, daß die Aufrufe der zugehörigen Implementierung minimiert werden. Auf diese Weise liegt der zusätzliche Aufwand durch objekt-orientierte Techniken nahe bei Null, da im Normalfall eine Ein- oder Ausgabe-Operation nur zum statischen Aufruf der *Streams*-Operation führt — jedoch nicht zum Aufruf der entsprechenden Operation der zugehörigen Implementierung.

Für die Ein- und Ausgabe bietet *Streams* folgende Operationen an:

```
TYPE Count = LONGINT;

PROCEDURE ReadByte(s: Stream; VAR byte: BYTE) : BOOLEAN;
PROCEDURE ReadPart(s: Stream; VAR buf: ARRAY OF BYTE;
                   off, cnt: Count) : BOOLEAN;
PROCEDURE Read(s: Stream; VAR buf: ARRAY OF BYTE) : BOOLEAN;

PROCEDURE WriteByte(s: Stream; byte: BYTE) : BOOLEAN;
PROCEDURE WritePart(s: Stream; buf: ARRAY OF BYTE;
                    off, cnt: Count) : BOOLEAN;
PROCEDURE Write(s: Stream; buf: ARRAY OF BYTE) : BOOLEAN;
```

ReadPart und *WritePart* entsprechen in etwa den Operationen von UNIX. Allerdings sind aus der Sicht von Oberon Felder angenehmer als Zeiger, so daß hier die Aufrufkonventionen geändert worden sind. *Read* und *Write* sind Kurzformen für den Normalfall. Entsprechend den Kompatibilitätsregeln von Oberon wäre es nicht notwendig, noch *ReadByte* und *WriteByte* hinzuzufügen, da auch *Read* bzw. *Write* ein einzelnes Byte akzeptieren würden. Dennoch wurden sie eingeführt, da die Ein- und Ausgabe einzelner Bytes ein sehr häufiger Fall ist, für den es sich gelohnt hat, eine spezielle optimierte Form zu entwickeln. Somit sind *ReadByte* bzw. *WriteByte* wesentlich schneller als *Read* bzw. *Write*. Nur bei größeren Quantitäten sind *Read* bzw. *Write* günstiger.

Ein *Stream* hat einige öffentliche Komponenten, die nach jeder Operation gesetzt werden und einige gängige Tests erlauben:

```
Stream = POINTER TO StreamRec;
StreamRec =
   RECORD
      (Disciplines.ObjectRec)
      (* following components are set after i/o-operations *)
      count: Count; (* resulting count of last operation *)
      errors: INTEGER; (* incremented for each error; may be set to 0 *)
      error: BOOLEAN; (* last operation successful? *)
      lasterror: ErrorCode; (* error code of last error *)
      eof: BOOLEAN; (* last read-operation returned with count=0 *)
   END;
```

Die öffentlichen Komponenten sind insbesondere nach Lese-Operationen wichtig, um die Anzahl der zurückgelieferten Bytes zu erfahren und ggf. zwischen Eingabeende und Fehlern zu unterscheiden.

Entsprechend den im zweiten Kapitel vorgestellten Techniken sind die Fähigkeiten eines Kanals (*stream*) nicht statisch festgelegt. So gibt es Kanäle, die nur Eingabe bzw. Ausgabe zulassen, und Kanäle, die beides ermöglichen. Im Gegensatz zu bekannten Varianten anderer Bibliotheken (z.B. *stdio* für C) gibt es keine Notwendigkeit, zwischen diesen Möglichkeiten "hin- und herzuschalten". Alle Operationen, die der jeweilige Kanal zur Verfügung stellt, können in beliebiger Reihenfolge benutzt werden. Somit existiert auch kein "verdeckter" Status, der zu beachten ist.

Für Kanäle, die eine Positionierung unterstützen, stehen folgende Operationen zur Verfügung:

```
CONST
   fromStart = 0;  (* absolute Positionierung *)
   fromPos = 1;    (* relative Positionierung *)
   fromEnd = 2;    (* Positionierung relativ zum Ende der Datei *)
TYPE Whence = SHORTINT; (* fromStart..fromEnd *)

PROCEDURE Seek(s: Stream; offset: Count; whence: Whence) : BOOLEAN;
PROCEDURE Tell(s: Stream; VAR offset: Count) : BOOLEAN;
PROCEDURE GetPos(s: Stream; VAR offset: Count);
PROCEDURE SetPos(s: Stream; offset: Count);
```

Die redundanten Operationen *GetPos* und *SetPos* wurden nachträglich eingeführt, um die zahlreichen Fälle, in denen man sich des Erfolges sicher ist, zu vereinfachen. Auf den ersten Blick ließe sich auch auf die relativen Positionierungen verzichten. Jedoch ersparen sie unnötige Abfragen, und die Positionierung relativ zum Ende ist die einzige Möglichkeit, die Länge einer Datei zu bestimmen:

```
PROCEDURE Length(s: Streams.Stream; VAR length: Streams.Count) : BOOLEAN;
   (* ermittelt die Laenge einer Datei, wobei die
      alte Position anschliessend wieder aufgesucht wird
   *)
   VAR
      oldpos: Streams.Count;
BEGIN
   RETURN Streams.Tell(s, oldpos) &
          Streams.Seek(s, 0, Streams.fromEnd) &
          Streams.Tell(s, length) &
          Streams.Seek(s, oldpos, Streams.fromStart)
END Length;
```

Diese Operation wurde nicht in *Streams* aufgenommen, da sie einerseits wie gezeigt relativ einfach auf vorhandene Operationen zurückgeführt werden kann. Andererseits ist die Länge einer Datei nicht notwendigerweise definiert. *Streams* ist relativ freizügig gegenüber den Implementierungen und verlangt nicht, daß sie einen zusammenhängenden Positionierungsbereich unterstützen. So wäre es beispielsweise denkbar, daß ein Kanal einen Zugriff auf den Bereich [3, 8] und [17, 28] zuläßt und alle anderen Positionen für unzulässig erklärt. Die Definition der Länge wäre für eine solche Datei offen: Soll die Anzahl der vorhandenen Bytes herangezogen werden oder die höchste zulässige Position?

Außerdem ist es denkbar, daß ein Kanal kein definiertes Ende besitzt. Man könnte sich beispielsweise einen Kanal mit allen Primzahlen in aufsteigender Reihenfolge vorstellen. Die Primzahlen werden dann jeweils nur bis zur aktuellen Position berechnet. Ein solcher Kanal würde eine Positionierung relativ zum Ende nicht zulassen.

In Übereinstimmung mit der UNIX-Schnittstelle und im Gegensatz zu vielen bekannten Bibliotheken (z.B. *iostream* in C++) gibt es auch bei bidirektionalen Kanälen nur eine Position, die sowohl bei Schreib- als auch bei Lesezugriffen entsprechend weitergesetzt wird. Dies führt zu einer einfacheren und verständlicheren Schnittstelle. Allerdings trägt dies auch zur Verkomplizierung der Pufferungs-Algorithmen in *Streams* bei. Dafür erhält man jedoch die Sicherheit, daß es nicht zu Problemen durch getrennte Puffer für Lese- und Schreibzugriffe kommt.

6.4. Die Schnittstelle von *Streams* für Implementierungen

6.4.1. Die Definition der Schnittstelle

Ziel beim Entwurf der Schnittstelle für Implementierungen war es, den Aufwand für eine einzelne Implementierung zu minimalisieren. So kann eine Schnittstellenprozedur (*read* oder *write*) mit zwei Parametern (für den Kanal und ein Byte) schon ausreichen, um einen funktionsfähigen *Stream* zu erhalten. Für die Ein- und Ausgabe wurden insgesamt drei Operations-Paare so eingerichtet, daß jede Implementierung die für sie bequemste Form auswählen kann. Über die Menge der Fähigkeiten läßt sich dann genau spezifizieren, welche Operationen unterstützt werden.

Folgende Fähigkeiten können angegeben werden:

read	Der Kanal unterstützt Lese-Operationen.
write	Der Kanal unterstützt Schreib-Operationen.
addrio	Der Kanal unterstützt die Schnittstellenoperationen zur Ein- und Ausgabe, bei denen eine Adresse und eine Quantität angegeben wird. Dies kommt Implementierungen entgegen, die auf ihre Speicherflächen direkt über die Adresse zugreifen oder die UNIX-Systemaufrufe verwenden.
bufio	Der Kanal unterstützt die Schnittstellenoperationen zur Ein- und Ausgabe, bei denen ein **ARRAY OF BYTE** zusammen mit einem Anfangsindex und einer Quantität angegeben wird. Gegenüber den Ein-Byte-Operationen wird damit ein besserer Durchsatz ermöglicht.
seek	Der Kanal unterstützt Positionierungen.
tell	Operationen, die die Position abfragen, sind legal.
trunc	Die hinter dem Kanal liegende Datei kann gekürzt werden.
flush	Bei *Flush*-Operationen (die die Ausgabe geänderter Puffer erzwingen) ist die Implementierung zu verständigen, um ggf. die *Flush*-Operation weiterzuleiten. Dies ist sehr günstig für zwischengelegte Kanäle, die einen Kanal auf Basis eines oder mehrerer anderer Kanäle definieren.
close	Wenn der Kanal geschlossen wird, ist die zugehörige Implementierung davon zu benachrichtigen.
holes	Diese Fähigkeit muß angegeben werden, wenn der adressierbare Bereich Löcher enthalten kann.
handler	Ein Bearbeiter erlaubt die Durchführung außerhalb von *Streams* definierter Operationen, die auf Basis von Aufträgen aufgerufen werden.

Eine Implementierung muß zumindest *read* oder *write* angeben und die entsprechende einfache byteweise Operation offerieren. Alle anderen Fähigkeiten und damit verbundenen Operationen sind optional. Die Schnittstellenprozeduren orientieren sich an den Operationen von *Streams*, die nach außen zur Verfügung stehen:

```
Message = RECORD (Objects.ObjectRec) END;

AddrIOProc = PROCEDURE (s: Stream; ptr: Address; cnt: Count) : Count;
BufIOProc = PROCEDURE (s: Stream; VAR buf: ARRAY OF BYTE;
                       off, cnt: Count) : Count;
SeekProc = PROCEDURE (s: Stream; cnt: Count; whence: Whence) : BOOLEAN;
TellProc = PROCEDURE (s: Stream; VAR cnt: Count) : BOOLEAN;
ReadProc = PROCEDURE (s: Stream; VAR byte: BYTE) : BOOLEAN;
WriteProc = PROCEDURE (s: Stream; byte: BYTE) : BOOLEAN;
TruncProc = PROCEDURE (s: Stream; cnt: Count) : BOOLEAN;
FlushProc = PROCEDURE (s: Stream) : BOOLEAN;
CloseProc = PROCEDURE (s: Stream) : BOOLEAN;
HandlerProc = PROCEDURE (s: Stream; VAR msg: Message);

Interface = POINTER TO InterfaceRec;
InterfaceRec =                  (* zugehoerige Faehigkeiten *)
   RECORD
      (Objects.ObjectRec)
      addrread: AddrIOProc;     (* read, addrio *)
      addrwrite: AddrIOProc;    (* write, addrio *)
      bufread: BufIOProc;       (* read, bufio *)
      bufwrite: BufIOProc;      (* write, bufio *)
      read: ReadProc;           (* read *)
      write: WriteProc;         (* write *)
      seek: SeekProc;           (* seek *)
      tell: TellProc;           (* tell *)
      trunc: TruncProc;         (* trunc *)
      flush: FlushProc;         (* flush *)
      close: CloseProc;         (* close *)
      handler: HandlerProc;     (* handler *)
   END;
```

Eine Implementierung besteht aus zwei Teilen: Eine oder mehrere Eröffnungsprozeduren, die *Streams.Init* aufrufen und damit einen Kanal mit einer Schnittstelle und den zugehörigen Fähigkeiten verknüpfen; dabei wird auch der Pufferungsmodus festgelegt, der sich später nicht mehr ändern läßt. Der andere Teil besteht aus den angegebenen Schnittstellenprozeduren.

6.4.2. *String80* — ein Beispiel für eine Implementierung

Als übersichtliches Beispiel für eine Implementierung bietet sich ein Kanal an, der den Zugang zu einem festdimensionierten Zeichenfeld der Länge 80 offeriert. Eine Schnittstellendefinition für *String80* könnte folgendermaßen aussehen:

```
DEFINITION String80;
   (* Kanaele, die ARRAY 80 OF CHAR repraesentieren *)
   IMPORT Streams;

   TYPE
      Stream = POINTER TO StreamRec;
      StreamRec = RECORD (Streams.StreamRec) END;

   PROCEDURE Open(VAR s: Streams.Stream);

END String80.
```

Vorzugsweise wird *Streams.Stream* und nicht *Stream* als Ergebnistyp von *Open* verwendet. Wenn *Stream* angegeben würde, wäre ein benutzendes Modul gezwungen, die entsprechende Kanal-Variable vom Typ *String80.Stream* zu deklarieren. Trotzdem kann es sinnvoll sein, die Deklaration von *String80.Stream* zu exportieren, obwohl sie sonst nicht in der Schnittstelle verwendet wird, da der nach außen sichtbare Typ *String80.Stream* Typentests ermöglicht.

Auf der privaten Seite von *String80.Stream* werden die Daten, die die zum Kanal gehörige Datei repräsentieren sollen, oder zumindest ein Verweis darauf benötigt. Darüber hinaus muß eine aktuelle Position verwaltet werden, wenn *Seek* und *Tell* unterstützt werden sollen. Ferner muß die aktuelle Länge der Zeichenkette notiert werden. Ein Nullbyte wie bei normalen Zeichenketten ist hier nicht möglich, da schließlich auch Nullbytes geschrieben werden können.

```
CONST
   buflen = 80;
TYPE
   StreamRec =
      RECORD
         (Streams.StreamRec)
         (* private Komponenten *)
         data: ARRAY buflen OF BYTE;
         len: Streams.Count; (* Laenge der Zeichenkette *)
         pos: Streams.Count; (* aktuelle Position *)
      END;
```

Zu Beginn müßten *len* und *pos* auf **0** gesetzt werden. Die aktuelle Position bewegt sich anschließend in dem Bereich **[0..*buflen*]**. Wenn die aktuelle Position bei *buflen* ist, sind Lese- und Schreib-Operationen nicht zugelassen, bis die Position geändert wird.

Die Schnittstellenprozedur *Write* könnte dann folgendermaßen aussehen:

```
PROCEDURE Write(s: Streams.Stream; byte: BYTE) : BOOLEAN;
BEGIN
   WITH s: Stream DO
      IF s.pos >= buflen THEN
         RETURN FALSE
      ELSE
         s.data[s.pos] := byte;
         INC(s.pos);
         (* Laenge der Zeichenkette aktualisieren *)
         IF s.pos > s.len THEN
            s.len := s.pos;
         END;
         RETURN TRUE
      END;
   END;
END Write;
```

Entsprechend der Schnittstellendefinition muß *Streams.Stream* anstatt *Stream* als Typ für den Parameter *s* angegeben werden. Andererseits kann man sicher sein, daß *s* vom Typ *Stream* ist, da *Write* nicht exportiert ist und nur dem Modul *Streams* über den Schnittstellenverbund bekannt gemacht wird.

Die Aufgabe von *Write* ist es, *byte* an der aktuellen Position der Datei abzulegen. Die aktuelle Position wird dabei implizit weitergesetzt. Dabei darf die aktuelle Position nicht nur den Bereich der Zeichenkette umfassen, sondern auch einen Schritt weitergehen, obwohl bei dieser Position alle Schreib- und Lese-Operationen nicht erfolgreich verlaufen. Wenn jedoch die aktuelle Position auf den Bereich der Zeichenkette beschränkt wäre, würde eine Schreib- oder Lese-Operation niemals fehlschlagen, da *Write* oder *Read* an der letzten zulässigen Position die Position nicht verändern würden.

Entsprechend ergibt sich die *Read*-Operation:

```
PROCEDURE Read(s: Streams.Stream; VAR byte: BYTE) : BOOLEAN;
BEGIN
   WITH s: Stream DO
      IF s.pos < s.len THEN
         byte := s.data[s.pos];
         INC(s.pos);
         RETURN TRUE
      ELSE
         RETURN FALSE
      END;
   END;
END Read;
```

Positionierungen sind ebenfalls leicht zu realisieren:

```
PROCEDURE Seek(s: Streams.Stream;
               offset: Streams.Count;
               whence: Streams.Whence) : BOOLEAN;
   VAR
      newpos: Streams.Count;
BEGIN
   WITH s: Stream DO
      CASE whence OF
      | Streams.fromStart: newpos := offset;
      | Streams.fromPos:   newpos := offset + s.pos;
      | Streams.fromEnd:   newpos := offset + s.len;
      END;
      IF (newpos >= 0) & (newpos <= s.len) THEN
         s.pos := newpos; RETURN TRUE
      ELSE
         RETURN FALSE
      END;
   END;
END Seek;

PROCEDURE Tell(s: Streams.Stream;
               VAR pos: Streams.Count) : BOOLEAN;
BEGIN
   WITH s: Stream DO
      pos := s.pos;
      RETURN TRUE
   END;
END Tell;
```

Abschließend muß *Open* realisiert werden, das die Aufgabe hat, obige Schnittstellenprozeduren mit einem neu zu kreierenden Kanal zu verbinden. Da der Schnittstellenverbund für alle Kanäle von *String80* konstant ist, wird er global angelegt:

```
MODULE String80;

   IMPORT Streams;

   (* ... Typdefinitionen (siehe oben) ... *)

   VAR
      (* globale Variablen *)
      caps: Streams.CapabilitySet;
      if: Streams.Interface;

   (* ... Schnittstellenprozeduren (siehe oben) ... *)

   PROCEDURE Open(VAR s: Streams.Stream);
      VAR
         stream: Stream;
```

```
      BEGIN
         NEW(stream);
         stream.pos := 0; stream.len := 0;
         Streams.Init(stream, if, caps, Streams.nobuf);
         s := stream;
      END Open;

   BEGIN (* Initialisierung von String80 *)
      NEW(if);
      if.read := Read; if.write := Write;
      if.seek := Seek; if.tell := Tell;
      caps := {Streams.read, Streams.write, Streams.seek, Streams.tell};
   END String80.
```

In diesem Beispiel wird die Pufferung von *Streams* generell ausgeschaltet (*Streams.nobuf*), da es nicht sinnvoll ist, den Inhalt der Zeichenkette doppelt im Speicher zu halten.

Dieses Beispiel ist noch unvollständig in Bezug auf Fehlerbehandlung, da bei Problemen nur **FALSE** zurückgeliefert wird, ohne entsprechende Ereignisse zu generieren. Eine vollständige Version davon findet sich in [Borchert92].

6.5. Die Pufferungs-Mechanismen von *Streams*

6.5.1. Die exportierte Schnittstelle

Streams offeriert vier Pufferungsmodi, die allen üblichen Anforderungen gerecht werden:

Streams.nobuf — Überhaupt keine Pufferung. Alle Operationen werden direkt auf die Operationen der Schnittstelle abgebildet. Dies ist einerseits sinnvoll für Dateirepräsentierungen, die sich bereits im Speicher befinden (wie z.B. *String80*) als auch für ungewöhnliche Kanäle, die die Forderungen an Implementierungen bezüglich der Pufferung nicht erfüllen (siehe unten).

Streams.onebuf — Hier wird genau ein Puffer für den Kanal gehalten. Dies ist ideal, wenn es kaum zu Positionierungen kommt. Wenn der entsprechende Kanal keine Positionierung kennt und bidirektional ist, dann wird sowohl ein Puffer für die Eingabe als auch ein Puffer für die Ausgabe angelegt.

Streams.linebuf — Hier wird analog zu *Streams.onebuf* auch nur ein Puffer gehalten (bzw. zwei bei bidirektionalen Kanälen), der allerdings bei Ausgabe eines definierbaren Zeilentrenners automatisch geleert wird. Dies ist günstig für traditionelle zeilenorientierte Ein- und Ausgabe.

Streams.bufpool — Hier wird eine vom Aufrufer bestimmbare Anzahl von Puffern zu einem Kanal unterhalten. Dies ist ideal, wenn es häufig zu Positionierungen kommt (wie z.B. bei Datenbanken).

Der Pufferungsmodus wird von der jeweiligen Implementierung innerhalb der *Open*-Prozedur festgelegt. Wenn es sinnvoll ist, offerieren die *Open*-Prozeduren den Pufferungsmodus als Parameter. Später kann der Pufferungsmodus nicht mehr geändert werden. Nur der Zeilentrenner bei *Streams.linebuf* und die Anzahl der zu verwendenden Puffer bei *Streams.bufpool* kann danach variiert werden. Die Größe eines Puffers ist grundsätzlich nach außen unbekannt.

Folgende Forderungen müssen von Implementierungen erfüllt werden, damit es keine Unterschiede im Ergebnis in Abhängigkeit des Pufferungsmodus gibt. Treffen diese Forderungen für eine Implementierung nicht zu, so kommt nur *Streams.nobuf* als Pufferungsmodus in Frage.

(1) Positionen sind nicht-negative Offsets relativ zum Anfang der Datei, wobei mit 0 das erste und mit 1 das zweite Byte adressiert wird.

(2) Lesen und Schreiben erfolgt immer in Abhängigkeit der aktuellen Position. Das Lesen oder Schreiben von n Bytes inkrementiert die Position um n.

(3) Lesen und Schreiben sind unabhängig von der Partitionierung, d.h. Lesen bzw. Schreiben von n Bytes durch n Aufrufe mit jeweils einem Byte ist äquivalent zu einem Aufruf mit einer Quantität von n Bytes. Allerdings steht es den Lese-Operationen immer frei, weniger Bytes als verlangt zurückzuliefern.

Ein Beispiel für die Verletzung von (3) sind Bandgeräte, bei denen die Quantitäten Teil der Information sind. (1) und (2) müssen natürlich nur von Kanälen unterstützt werden, die die Fähigkeiten *seek* und *tell* haben und damit Positionen definieren.

Folgende Operationen offeriert *Streams* in Zusammenhang mit der Pufferung:

```
PROCEDURE GetBufMode(s: Stream) : BufMode;
PROCEDURE LineTerm(s: Stream; termch: Byte);
PROCEDURE Tie(in, out: Stream);
PROCEDURE SetBufferPoolSize(s: Stream; nbuf: INTEGER);
PROCEDURE GetBufferPoolSize(s: Stream; VAR nbuf: INTEGER);

PROCEDURE Back(s: Stream) : BOOLEAN;
PROCEDURE Insert(s: Stream; byte: Byte) : BOOLEAN;

PROCEDURE InputInBuffer(s: Stream) : BOOLEAN;
PROCEDURE OutputInBuffer(s: Stream) : BOOLEAN;
PROCEDURE OutputWillBeBuffered(s: Stream) : BOOLEAN;
PROCEDURE Flush(s: Stream) : BOOLEAN;
PROCEDURE Touch(s: Stream);

PROCEDURE GetFlushEvent(s: Stream; VAR type: Events.EventType);
```

Mit *LineTerm* kann der das automatische Leeren des Puffers auslösende Zeilentrenner geändert werden.

Häufig sind ein Eingabe- und ein Ausgabekanal so miteinander verbunden, daß das Einlesen von dem Eingabekanal zu dem Leeren des Puffers auf dem Ausgabekanal führen soll. Diese Verbindung läßt sich mit *Tie* angeben.

Back und *Insert* erlauben es, die letzte Lese-Operation rückgängig zu machen, wobei *Insert* sogar eine Änderung des nächsten Eingabezeichens erlaubt. Beide Operationen werden nur durch die Pufferung realisiert und verlangen somit keine Unterstützung durch die jeweilige Implementierung.

InputInBuffer, *OutputInBuffer* und *OutputWillBeBuffered* erlauben es, den Zustand des aktuellen Puffers zu klären in Hinblick darauf, ob die nächste Schreib- oder Lese-Operation (für ein Byte) noch gepuffert wird. *Flush* führt dazu, daß alle geänderten Puffer geleert werden. *Touch* entspricht *Flush* und führt anschließend zum Vergessen der Pufferinhalte und ggf. der aktuellen Position. Diese Operation ist nützlich, wenn das zugrundeliegende Medium oder der Kanal von einer anderen Seite manipuliert worden sind.

GetFlushEvent erlaubt es, durch ein Ereignis über eine bevorstehende *Flush*-Operation informiert zu werden. Insbesondere bei bidirektionalen Kommunikationskanälen kann auf diese Weise durch eine vorherige Leerung des Eingabepuffers sichergestellt werden, daß die *Flush*-Operation nicht zur gegenseitigen Blockade führt.

6.5.2. Die Pufferungs-Techniken

6.5.2.1. Der aktuelle Puffer

Die Größe eines Puffers ist eine private Konstante von *Streams*. Abgesehen davon, daß es sich dabei natürlich um eine Zweierpotenz handeln sollte, ist es sinnvoll (jedoch nicht zwingend notwendig), die Blockgröße des Betriebssystems zu verwenden. Bei der aktuellen Implementierung liegt die Puffergröße bei 2048 Bytes.

Wenn eine Datei als Folge von Blöcken betrachtet wird und davon auszugehen ist, daß die Blockgröße der Puffergröße entspricht, dann ist es nicht unwichtig, von welchen Positionen aus Puffer eingelesen werden. Traditionelle Bibliotheken (wie z.B. die *stdio*-Bibliothek für C) lesen einen Puffer ab der aktuellen logischen Position[1] ein. In der Abbildung 29 läßt sich sehen, wie dies zu versetzten Puffern führen kann, bei der das Betriebssystem möglicherweise gezwungen ist, zwei Blöcke statt nur einem einzulesen. Wenn statt dessen (siehe Abbildung 30) der Puffer immer auf eine Puffergrößenkante gelegt wird, kann dies vermieden werden.

Das führt dazu, daß die aktuelle physische Position bekannt sein muß, da sie bei der Eröffnung eines Kanals nicht notwendigerweise bei 0 liegt. So wird bei der ersten Lese-Operation bei einem gepufferten Kanal zuerst die aktuelle physische Position bestimmt, ggf. rückwärts positioniert und dann von der nächstgelegenen Kante aus ein vollständiger Puffer gelesen. Später ist es nicht mehr notwendig, die aktuelle physische Position zu bestimmen, da sie immer mitberechnet wird. Nur durch eine *Touch*-Operation ist *Streams* wieder gezwungen, bei der nächsten Lese- oder Schreib-Operation die aktuelle physische Position in Erfahrung zu bringen.

Da die zugrundeliegende Datei Löcher haben kann, ist es natürlich möglich, daß der Anfang eines Blocks nicht lesbar ist, obwohl von der aktuellen logischen Position aus eine Lese-Operation erfolgreich sein würde. In

[1] Die logische Position ist die Position aus der Sicht des Anwenders, die von der Position des Kanals (der physischen Position) aufgrund der Pufferung abweichen kann.

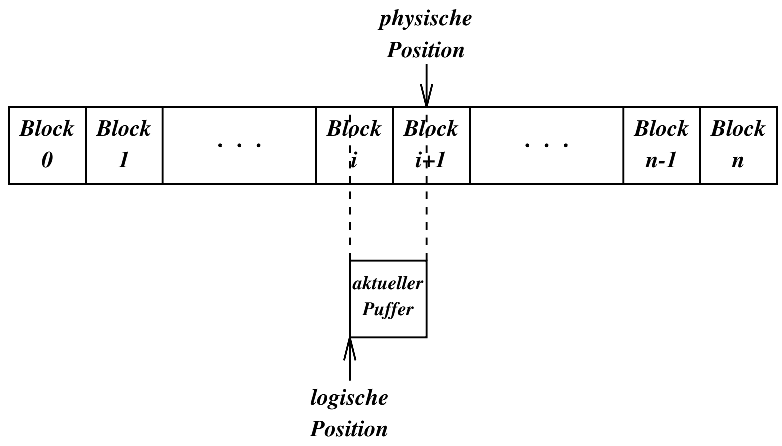

Abb. 29: Versetzter Puffer

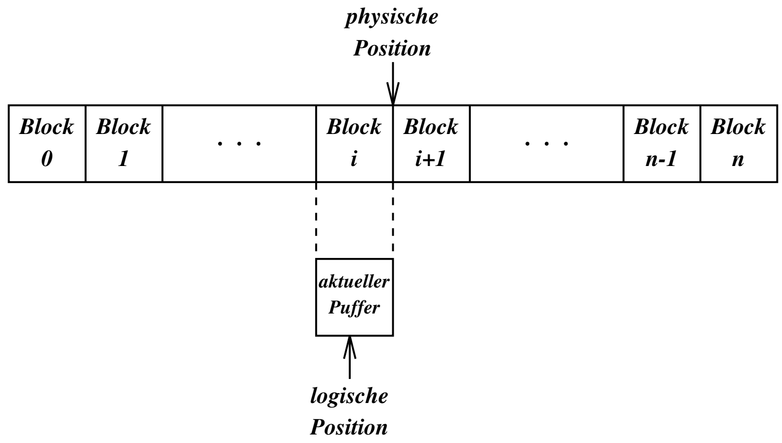

Abb. 30: Ausgerichteter Puffer

diesem Fall gibt es zwar einen Fehlversuch beim Füllen des Puffers, aber anschließend wird versucht, den Puffer ab der logischen Position aus teilweise zu füllen.

Es wird der gleiche Puffer für Ein- und für Ausgaben verwendet. Aus diesem Grund wird sowohl ein Leseintervall als auch ein Schreibintervall für einen Puffer verwaltet. Das Leseintervall enthält alle Daten, die aus dem Puffer zum Lesen entnommen werden können. Das Schreibintervall markiert den Bereich, der beim Leeren des Puffers an die Implementierung weitergegeben werden muß.

Anstatt von Intervallen wäre es auch denkbar gewesen, Mengen zu unterhalten. Dies würde aber alle Operationen aufwendiger gestalten (insbesondere *Flush*) und nur bei sehr ungewöhnlichen und daher seltenen Fällen Verbesserungen bringen. Entsprechend ergibt sich die Datenstruktur für einen Puffer:

```
Buffer = POINTER TO BufferRec;
BufferRec =
   RECORD
      ok: BOOLEAN; (* TRUE if other components are valid *)
      pos: Count;  (* file position which corresponds to cont[0] *)
      rbegin: Count;  (* read-region: starting index *)
      rend: Count;  (* read-region: ending index *)
      wbegin: Count; (* write-region: starting index of dirty region *)
      wend: Count; (* write-region: ending index *)
      cont: ARRAY bufsize OF BYTE; (* buffer contents *)
      nextfree: Buffer; (* needed for released buffers only *)
      (* components for buffers which are members of a buffer pool *)
      prevh, nexth: Buffer; (* next buffer with the same hash value *)
      preva, nexta: Buffer; (* sorted list of buffers (access time) *)
   END;
```

Zu Beginn und nach *Touch*-Operationen wird *ok* auf **FALSE** gesetzt. Erst wenn es zu einer Lese- oder Schreib-Operation kommt, wird ein Puffer gefüllt. Die Komponente *pos* entspricht immer der Kantenposition und ist deswegen immer durch die Puffergröße *bufsize* teilbar. Das Leseintervall ist durch **[***rbegin***..***rend***)** und das Schreibintervall durch **[***wbegin***..***wend***)** gegeben.

Wenn keine automatische Speicherverwaltung aktiv ist, unterhält *Streams* selbst eine Liste nicht mehr benötigter Puffer — daher die Komponente *nextfree*. Die weiteren Komponenten werden nur bei dem Pufferungsmodus *Streams.bufpool* benötigt.

Zu jedem gepufferten Kanal gibt es genau einen aktuellen Puffer. Um den Testaufwand bei einfachen Lese- und Schreib-Operationen zu reduzieren, gibt es beim Kanal die Komponenten *left* und *write*, die angeben, wieviel Bytes noch beim aktuellen Puffer gelesen bzw. geschrieben werden können. Zu Beginn sind *left* und *write* auf **0** und die Komponente *ok* des aktuellen Puffers auf **FALSE** gesetzt.

Für ein Beispiel liege zu Beginn die aktuelle Position bei **2500** und es erfolge ein Aufruf von *ReadByte*. *ReadByte* findet *left* **= 0** vor und läßt daher den Puffer füllen. Unter der Annahme, daß es gelingt, den Bereich **[2048..3999]** einzulesen, würde eine Situation gemäß der Abbildung 31 vorliegen. Die aktuelle logische Position ist durch *pos* gegeben; *rpos* markiert die aktuelle physische Position. Bei frischgefüllten Puffern werden *write*, *wbegin* und *wend* auf **0** gesetzt. *ReadByte* kopiert anschließend das Byte aus dem Puffer bei *pos* **MOD** *bufsize*. Danach wird *pos* inkrementiert und *left* dekrementiert.

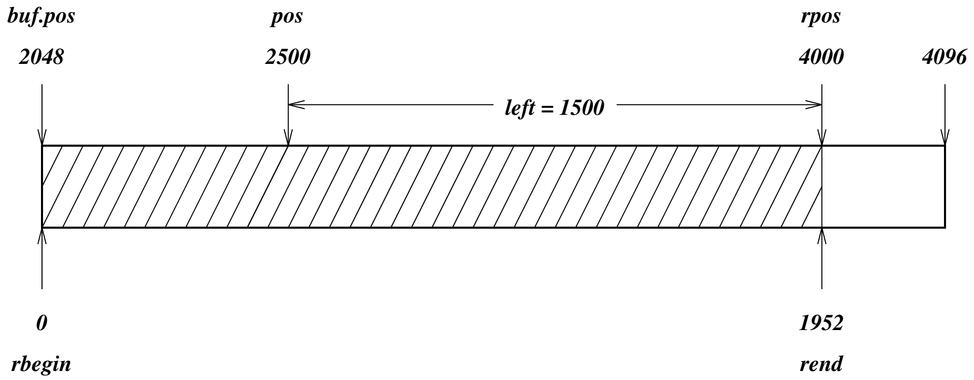

Abb. 31: Frisch gefüllter Puffer im Beispiel

Abbildung 32 zeigt die Situation, wenn anschließend 100 Bytes geschrieben wurden. Dies führte zur Einrichtung eines Schreibintervalls von 453 bis 553. *write* wurde auf 1495 gesetzt, da auch über die Grenze des eingelesenen Bereichs weiter geschrieben werden kann bis zum Ende des Puffers. Nach dem Schreiben weiterer 1450 Bytes ergibt sich die Situation in Abbildung 33. Inzwischen erreichte das Schreibintervall die Grenze des Leseintervalls. Da gerade geschriebene Daten gelesen werden könnten, wird das Leseintervall mit expandiert. Eine Positionierung innerhalb des Schreibintervalls würde *left* und *write* wieder erhöhen. Sollte jedoch eine Positionierung außerhalb des Schreibintervalls erfolgen (z.B. auf 2200), so würde bei Dateien mit potentiellen Löchern ggf. eine verzögerte *Flush*-Operation durchgeführt werden,[1] während bei Dateien ohne Löcher einfach das Schreibintervall nach hinten erweitert wird. Letzteres läßt sich durchführen, da der erweiterte Bereich vom Lesebereich überdeckt ist.

6.5.2.2. Puffer-Pools

Die Datenstruktur eines Puffer-Pools ähnelt sehr dem Puffer-Cache von UNIX [Bach86, §3.2]. Wie Abbildung 34 zeigt, sind alle Puffer eines Kanals über eine Hash-Tabelle zugänglich und zusätzlich in einer doppelt verketteten Liste eingeordnet, die nach der letzten Zugriffszeit sortiert ist. Für die Hash-Funktion wird die Anfangsposition des jeweiligen Puffers durch die Puffergröße dividiert und modulo der Hash-Tabellengröße genommen. Die Hash-Tabellengröße der aktuellen Implementierung liegt bei 128.

Der aktuelle Puffer ist immer direkt zugänglich, und somit werden alle Lese- und Schreib-Operationen nicht komplizierter, solange der aktuelle Puffer nicht verlassen wird. Wird der aktuelle Puffer verlassen und eine Lese- oder Schreib-Operation durchgeführt, wird zunächst über die Hash-Tabelle nach einem der aktuellen

[1] Die *Flush*-Operation würde nicht sofort stattfinden. Statt dessen wird zunächst nur *write* auf **0** gesetzt und erst bei einer Schreib-Operation außerhalb des Schreibintervalls würde es zu einer *Flush*-Operation kommen.

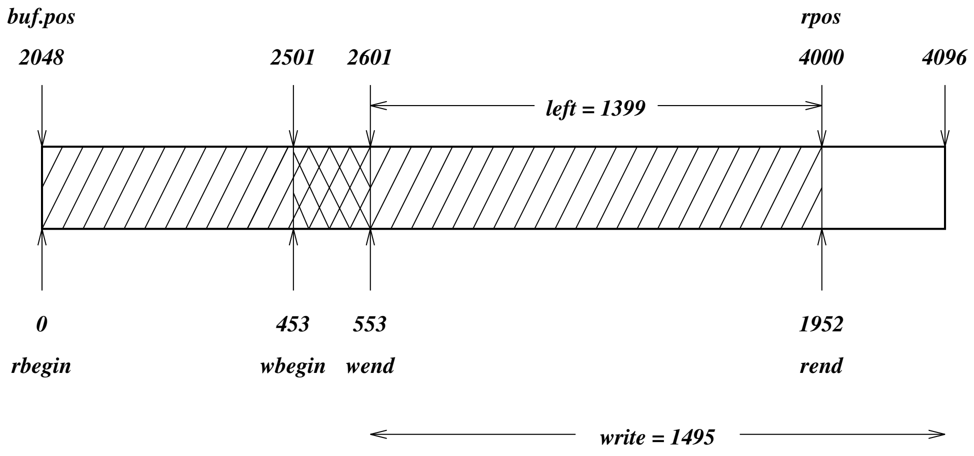

Abb. 32: Zustand des Puffers nach der 1. Schreib-Operation

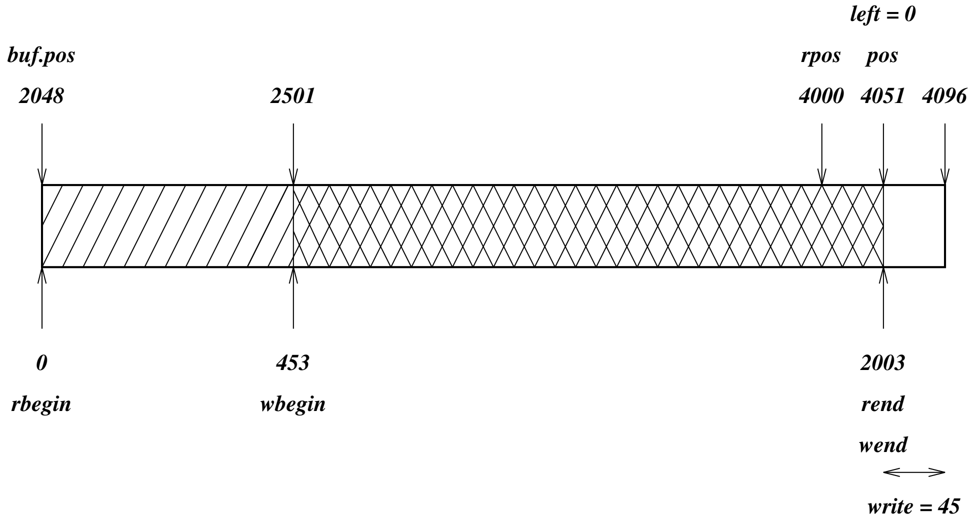

Abb. 33: Zustand des Puffers nach der 2. Schreib-Operation

Position entsprechenden Puffer gesucht. Wenn ein Puffer gefunden wird, dann wird er aus der nach Zugriffszeiten sortierten Liste entfernt und wieder ganz vorne eingefügt. Gibt es für den gewünschten Block noch keinen Puffer, so wird ein neuer Puffer angelegt, sofern noch nicht die maximale Anzahl von Puffern erreicht worden ist (*maxbuf*). Andernfalls, falls die maximale Anzahl von Puffern bereits erreicht worden ist, wird der älteste Puffer in Hinblick auf fällige Schreib-Operationen überprüft und dann für den gewünschten Bereich wiederverwendet.

Jeder Wechsel des aktuellen Puffers führt zur Neubestimmung von *read* und *write*. Hingegen behält jeder Puffer seine individuellen Lese- und Schreibintervalle.

6.6. Formatierte Ein- und Ausgabe

Mit formatierter Ein- und Ausgabe wird der Bereich des Einlesens und Schreibens von Texten umrissen, die typischerweise von einem menschlichen Benutzer direkt oder indirekt eingegeben worden sind oder von ihm zu lesen sind. Häufig ist jedoch der menschliche Benutzer völlig fiktional und es werden "lesbarere" Formen nur verwendet, da sie unabhängiger von bestimmten maschinenabhängigen Binärformaten sind und auch leichter im Fehlerfall überprüft werden können.

Besonders im Bereich von UNIX hat sich der Trend zu an lesbarem Text orientierten Schnittstellen zwischen Programmen bewährt. So werden viele Daten in Form von "kleinen" in Textform gehaltenen Datenbanken aufbewahrt. Dabei entsprechen eine Zeile einem Datensatz und durch Feldtrenner separierte Teile einer Zeile den einzelnen Attributen eines Datensatzes. Abbildung 35 zeigt einen Auszug aus der Datei mit den Benutzern eines UNIX-Systems. Als Feldtrenner dient hier der Doppelpunkt. Im ersten Feld steht der Benutzername — es

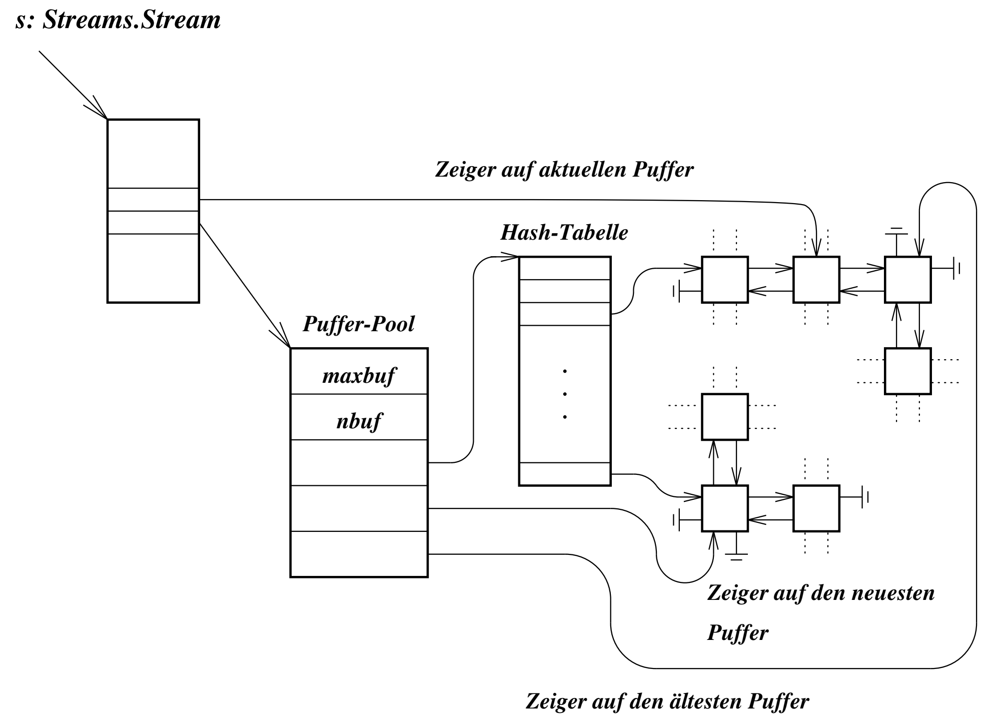

Abb. 34: Datenstruktur eines Puffer-Pools

folgen das kodierte Passwort (aus Sicherheitsgründen ist dies bei neueren UNIX-Systemen in einer anderen Datei untergebracht), die Benutzernummer, die Gruppennummer, der ausgeschriebene Name, das Verzeichnis mit den Benutzerdaten und optional die Angabe eines Kommandointerpreters (*shell*).

```
root:x:0:1:0000-Admin(0000):/:
daemon:x:1:1:0000-Admin(0000):/:
bin:x:2:2:0000-Admin(0000):/bin:
sys:x:3:3:0000-Admin(0000):/usr:
adm:x:4:4:0000-Admin(0000):/usr/adm:
borchert:x:120:1:Andreas Borchert:/usr/borchert:
```

Abb. 35: /etc/passwd als Beispiel einer ‘‘kleinen’’ textuellen Datenbank

Dadurch, daß Informationen in dieser Form gehalten werden, sind sie außerordentlich leicht zugänglich und manipulierbar. Viele Aufgaben, die ansonsten komplizierte Programme erfordern würden, können unter UNIX mit Hilfe von wenigen Kommandos gelöst werden. So ist es z.B. leicht möglich, alle Benutzer, die einer bestimmten Gruppe angehören, einem anderen Plattenbereich zuzuordnen.

Obwohl sehr viele Daten unter UNIX in dieser Form unterhalten werden, gibt es davon unzählige Variationen. Es kommen nicht nur unterschiedliche Feldtrenner vor, sondern manchmal werden auch Folgen von Füllzeichen (z.B. Leerzeichen und Tabs) jeweils als ein Feldtrenner betrachtet. Hinzu kommen möglicherweise Kommentare, und leere Zeilen sind ggf. auch zu ignorieren.

So einfach sich der Zugriff über den UNIX-Werkzeugkasten gestaltet, so fehleranfällig oder empfindlich sind häufig die Programme, die auch darauf zugreifen. Aus diesem Grund lag es nahe, einige Unterstützungen hierfür in der Bibliothek anzubieten, die insbesondere das Einlesen erleichtern.

Anstatt alle Operationen für formatierte Ein- und Ausgabe mit vielen Parametern auszustatten, werden alle für einen Kanal typischerweise konstanten Parameter über eine Disziplin verwaltet. Das Modul *StreamDisciplines* offeriert folgende Parameter, die von jedem Modul abgefragt und verändert werden können:

Zeilentrenner

Ein Zeilentrenner besteht aus ein bis drei Zeichen, die paarweise unterschiedlich sein müssen. Voreinstellung ist das Zeichen *ASCII.EOL*, das unter UNIX dem *line feed* (0AX) entspricht. Eine leere Zeichenkette steht für 0X als Zeilentrenner. Die Einschränkung auf paarweise verschiedene Zeichen erfolgte, um den Erkennungsalgorithmus zu vereinfachen.

Füllzeichen

Die Füllzeichen sind eine Menge von Zeichen, die bei Lese-Operationen zu Beginn überlesen werden und die auch als Trenner dienen können. Voreinstellungsgemäß gehören *ASCII.tab*, *ASCII.lf*, *ASCII.ff* und *ASCII.sp* dazu. Somit ist es durchaus möglich, den Zeilentrenner (bzw. Zeichen davon) auch als Füllzeichen zu betrachten.

Feldtrenner

Die Feldtrenner sind ebenfalls eine Menge von Zeichen, wobei eines davon als Feldtrenner für die Ausgabe hervorgehoben werden kann.

Normalerweise zählt jedes Zeichen aus der Menge der Feldtrenner als Separator, so daß direkt aufeinander folgende Feldtrenner leere Felder begrenzen. Wenn jedoch die Feldtrenner eine Teilmenge der Füllzeichen sind, so wird jede Folge von Feldtrennern als ein einziger Feldtrenner betrachtet. Entsprechend gibt es dann nur ein leeres Feld, wenn die Zeile selbst leer ist. Wenn die Feldtrenner keine Teilmenge der Füllzeichen sind, so ergänzen sich beide Mengen bei der Begrenzung der Felder: Die Feldtrenner separieren die Felder, wobei bei der Extraktion eines Feldes die Füllzeichen zu Beginn und am Ende nicht zum Feldinhalt zählen. Wünscht man diesen Effekt nicht, so kann man die Menge der Füllzeichen als die leere Menge definieren.

Die Wirkungsweise dieser Parameter kann an den Beispielen der Tabelle 3 verdeutlicht werden (zur besseren Lesbarkeit wurden Leerzeichen durch ∪ ersetzt). Die erste Zeile entspricht den Parametern, die man z.B. für die Benutzerdatei (/etc/passwd) nehmen würde. Wie die zweite Zeile zeigt, würden umgebende Leerzeichen bei einer leeren Menge von Füllzeichen den einzelnen Feldern zugerechnet werden. Bei der dritten Zeile wurde das Leerzeichen als Füllzeichen definiert. Das führt dazu, daß umgebende Leerzeichen verschwinden, jedoch innenliegende Leerzeichen (wie z.B. zwischen ‘feld’ und ‘1’) beibehalten werden. In der vierten Zeile ist die Menge der Feldtrenner eine Teilmenge der Menge der Füllzeichen. Dadurch werden die beiden Leerzeichen zwischen “feld1” und “feld2” als ein Feldtrenner gezählt. Bei der fünften Zeile wird ein Leerzeichen nur als Feldtrenner und nicht als Füllzeichen betrachtet. Das führt dazu, daß es insgesamt auf der Zeile fünf Felder gibt.

Inhalt einer Zeile	Menge der Feldtrenner	Menge der Füllzeichen	Felder		
			1	2	3
"feld∪1:feld∪2:feld∪3"	{":"}	{}	"feld∪1"	"feld∪2"	"feld∪3"
"feld∪1∪:∪feld∪2∪:∪feld∪3"	{":"}	{}	"feld∪1∪"	"∪feld∪2∪"	"∪feld∪3"
"feld∪1∪:∪feld∪2∪:∪feld∪3"	{":"}	{"∪"}	"feld∪1"	"feld∪2"	"feld∪3"
"feld1∪∪feld2∪∪feld3"	{"∪"}	{"∪"}	"feld1"	"feld2"	"feld3"
"feld1∪∪feld2∪∪feld3"	{"∪"}	{}	"feld1"	""	"feld2"

Tab. 3: Beispiele für die Wirkungsweise von Feldtrennern und Füllzeichen

StreamDisciplines verwaltet nur diese Parameter und weist ihnen die erläuterte Semantik zu. Die Erkennung und Beachtung dieser Parameter bleibt jedoch anderen Modulen überlassen. Dies führt zu der Modulhierarchie von Abbildung 36. Natürlich sind diese Parameter nicht ausschließlich. Wenn es Bedarf für weitere Formatierstile und zugehörige Parametrisierungen gibt, so können weitere Disziplinen definiert werden und Module, die sie beachten.

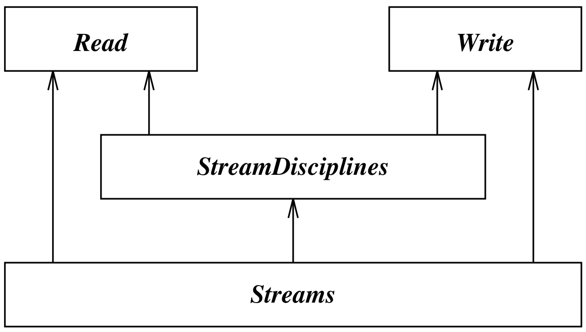

Abb. 36: Modulhierarchie für formatierte Ein- und Ausgabe

Die formatierte Ein- und Ausgabe wurde nur deswegen in die beiden Module *Read* und *Write* geteilt, um längere Schreibweisen zu vermeiden. Statt *Inout.WriteInt* analog zu Modula-2 reicht *Write.Int*. *Read* ebenso wie *Write* offeriert sowohl Prozeduren, die die Spezifikation eines Kanals erlauben, als auch Prozeduren, die den zugehörigen Standardkanal (*Streams.stdin* bzw. *Streams.stdout*) verwenden. Die Prozeduren, die einen Kanal als zusätzlichen Parameter haben, werden durch ein *S* am Ende des Namens gekennzeichnet (also beispielsweise *Write.Int* vs. *Write.IntS*).

Die Tabelle 4 ist eine Aufstellung der Operationen von *Read*. *Write* offeriert analog die gleichen Operationen mit Ausnahme der kürzeren numerischen Typen (durch die Kompatibilitätsregeln von Oberon können diese wegfallen) und der feldbezogenen Operation, da hierfür einfach *Write.String* (ggf. gefolgt von *Write.Char* mit dem Feldtrenner) verwendet werden kann.

Read.ShortInt	Einlesen von **SHORTINT**
Read.Int	Einlesen von **INTEGER**
Read.LongInt	Einlesen von **LONGINT**
Read.Real	Einlesen von **REAL**
Read.LongReal	Einlesen von **LONGREAL**
Read.Char	Einlesen von **CHAR**
Read.Byte	Einlesen von **BYTE**
Read.Line	Einlesen einer Zeile
Read.Ln	Einlesen eines Zeilentrenners
Read.String	Einlesen einer durch Füllzeichen getrennten Zeichenkette
Read.Field	Einlesen eines Feldes

Tab. 4: Formatierte Einlese-Operationen von *Read*

Alle Einlese-Operationen konsumieren exakt die Zeichen, die zu dem eingelesenen Objekt gehören. Das trennende Zeichen wird — sofern möglich — wieder an den Kanal zurückgegeben, so daß die nächste Lese-Operation es wieder vorfindet. Ausnahmen sind *Read.Line*, *Read.Ln* und *Read.Field*. *Read.Line* liest genau eine Zeile ein,[1] legt aber nicht den Zeilentrenner im Zeichenfeld ab. *Read.Ln* arbeitet analog zu *Read.Line*, ohne jedoch etwas zurückzuliefern. *Read.Field* darf den terminierenden Feldtrenner nicht zurückschieben, da es selbst immer davon ausgeht, am Anfang und nicht am Ende eines Feldes zu stehen.

Alle Operationen, die ihr Ergebnis in einem Zeichenfeld ablegen (*Read.String*, *Read.Line* und *Read.Field*), konsumieren immer das vollständige Objekt von der Eingabe — unabhängig davon, ob es sich in dem Zeichenfeld unterbringen läßt oder nicht. Mit Ausnahme von *Read.Field* hat keine der Operationen einen Rückgabewert. Statt dessen läßt sich über die öffentlichen Komponenten des Kanals überprüfen, ob die Operation erfolgreich war.

Read.Field liefert ein **BOOLEAN**-Resultat zurück. Wenn ein Feld gefunden wurde, erhält man **TRUE**. Am Ende der Zeile liefert *Read.Field* **FALSE**, so daß man gezwungen ist, *Read.Ln* explizit aufzurufen, um den Zeilentrenner zu überspringen. Folgendes Programmfragment zeigt, wie eine “kleine” Datenbank eingelesen werden kann:

```
(* Einlesen einer kleinen Datenbank vom Kanal ‘s’ *)
lineNumber := 1;
WHILE ˜s.eof & ˜s.error DO
   fieldNumber := 0;
   WHILE (fieldNumber < maxField) & Read.FieldS(s, field[fieldNumber]) DO
      INC(fieldNumber);
   END;
   (* bearbeite die Felder des aktuellen Datensatzes *)
   IF ˜s.eof & ˜s.error THEN
      Read.LnS(s); INC(lineNumber);
   END;
END;
```

Die Definition von Füllzeichen und Feldtrennern ist nicht nur für “kleine” Datenbanken nützlich, sondern führt auch zur Vereinfachung traditioneller zeilenweiser Interaktion mit dem Benutzer. Als Beispiel kann man das Einlesen folgenden Verbundes betrachten:

[1] Genauer gesagt, den Rest einer Zeile.

```
TYPE
   Book =
      RECORD
         title: ARRAY 80 OF CHAR;
         author: ARRAY 20 OF CHAR;
         year: INTEGER;
      END;
```

Eine erste Lösung könnte folgendermaßen aussehen:

```
PROCEDURE ReadBook(VAR book: Book);
BEGIN
   Write.String("Title:  "); Read.Line(book.title);
   Write.String("Author: "); Read.Line(book.author);
   Write.String("Year:   "); Read.Int(book.year); Read.Ln;
END ReadBook;
```

Hierfür wäre es sinnvoll, bei den Füllzeichen den Zeilentrenner zu entfernen, der voreinstellungsgemäß dabei ist. Damit wird sichergestellt, daß *Read.Int* bei der Eingabe einer leeren Zeile sofort (mit Fehler) aufhört. Folgende Version von *ReadBook* stellt sicher, daß alle drei Komponenten auch tatsächlich eingelesen werden, und wiederholt ggf. die Eingabe:

```
PROCEDURE ReadBook(VAR book: Book);

   PROCEDURE ReadString(prompt: ARRAY OF CHAR;
                        VAR string: ARRAY OF CHAR);
      VAR
         ok: BOOLEAN;
   BEGIN
      REPEAT
         Write.String(prompt);
         ok := Read.Field(string) & (Streams.stdin.count > 0);
         Read.Ln;
      UNTIL ok;
   END ReadString;

   PROCEDURE ReadInt(prompt: ARRAY OF CHAR;
                     VAR int: INTEGER);
      VAR
         ok: BOOLEAN;
   BEGIN
      REPEAT
         Write.String(prompt);
         Read.Int(int);
         ok := Streams.stdin.count > 0;
         Read.Ln;
      UNTIL ok;
   END ReadInt;

BEGIN
   ReadString("Title:  ", book.title);
   ReadString("Author: ", book.author);
   ReadInt("Year:   ", book.year);
END ReadBook;
```

Hierfür ist es sinnvoll, für Feldtrenner die leere Menge zu nehmen (d.h. pro Zeile genau ein Feld) und als Füllzeichen das Leerzeichen und ggf. den Tabulator, jedoch nicht den Zeilentrenner. Das führt dazu, daß es nichts ausmacht, wenn vor oder nach dem Titel oder dem Autor überflüssige Leerzeichen eingetippt worden sind. Trotzdem bleiben die Leerzeichen innerhalb des Titels oder des Autors erhalten.

Viele der ‘‘kleinen’’ Datenbanken sind mit Kommentaren versehen. Hierfür gibt es einen Filter *Comments*, mit dem alle gängigen Kommentartypen aus der Eingabe herausgefiltert werden können. Die Syntax des Kommentars wird dabei durch folgenden Verbund spezifiziert:

```
TYPE
   Delimiter = ARRAY 4 OF CHAR;
   Syntax = POINTER TO SyntaxRec;
   SyntaxRec =
      RECORD
         (Objects.ObjectRec)
         start: Delimiter;
         inclStart: BOOLEAN;
         end: Delimiter;
         inclEnd: BOOLEAN;
         nested: BOOLEAN;
      END;
```

Mit *start* wird der Anfang und mit *end* das Ende eines Kommentars beschrieben. *inclStart* und *inclEnd* geben an, ob *start* bzw. *end* selbst zum Kommentar gehören oder nicht. Mit *nested* kann angegeben werden, ob die Kommentare schachtelbar sind. Die Tabelle 5 gibt zwei Beispiele für die Spezifikation von Kommentaren. Bei einigen Programmiersprachen wie z.B. ISO-Pascal und C++ reicht eine Beschreibung nicht aus. In diesem Fall könnten mehrere Filter hintereinander gesetzt werden.

	Oberon	Bourne Shell
start	(*	#
inclStart	**TRUE**	**TRUE**
end	*)	*ASCII.nl*
inclEnd	**TRUE**	**FALSE**
nested	**TRUE**	**FALSE**

Tab. 5: Zwei Beispiele für die Spezifikation von Kommentaren

Folgende Einlese-Prozedur demonstriert diese Techniken am Beispiel der Datei */etc/hosts* auf UNIX-Systemen. Sie enthält IP-Adressen gefolgt von zugehörigen Rechnernamen. Als Feldtrenner dienen beliebige Folgen von Füllzeichen. Leere Zeilen und Kommentare sind analog zur Bourne-Shell zulässig.

```
TYPE
   Address = ARRAY 16 OF CHAR;
   Hostname = ARRAY 64 OF CHAR;
   Entry = POINTER TO EntryRec;
   EntryRec =
      RECORD
         (Objects.ObjectRec)
         address: Address;
         hostname: Hostname;
      END;

PROCEDURE OpenHostsFile(VAR s: Streams.Stream;
                        errors: Disciplines.Object) : BOOLEAN;
   VAR
      file: Streams.Stream;
      etcHostsCmt: Comments.Syntax;
      whitespace: Sets.CharSet;
BEGIN
   IF ~UnixFiles.Open(file, "/etc/hosts", UnixFiles.read,
                      Streams.onebuf, errors) THEN
      RETURN FALSE
   END;

   NEW(etcHostsCmt);
   etcHostsCmt.start := "#"; etcHostsCmt.inclStart := TRUE;
   etcHostsCmt.end := ASCII.nl; etcHostsCmt.inclEnd := FALSE;
   etcHostsCmt.nested := FALSE;
   Comments.Open(s, file, etcHostsCmt);

   Sets.InitSet(whitespace);
   Sets.InclChar(whitespace, " ");
   Sets.InclChar(whitespace, ASCII.tab);
   StreamDisciplines.SetWhiteSpace(s, whitespace);
```

```
      StreamDisciplines.SetFieldSepSet(s, whitespace);
      RETURN TRUE
   END OpenHostsFile;

   PROCEDURE ScanHostsFile(s: Streams.Stream;
                           collection: Collections.Collection);
      VAR
         address: Address;
         hostname: Hostname;
         entry: Entry;
   BEGIN
      REPEAT
         IF Read.FieldS(s, address) THEN
            WHILE Read.FieldS(s, hostname) DO
               NEW(entry);
               entry.address := address;
               entry.hostname := hostname;
               Collections.Add(collection, entry);
            END;
         END;
         Read.LnS(s);
      UNTIL s.eof OR s.error;
   END ScanHostsFile;
```

Die Aufgabe, die Datei */etc/hosts* einzulesen, wurde in zwei Teile zerlegt: *OpenHostsFile* eröffnet die Datei, setzt den Kommentarfilter auf und richtet die zugehörigen Disziplinen ein. *ScanHostsFile* ist davon getrennt und geht nur von der Grundstruktur aus, daß in jeder Zeile eine Adresse steht, der die zugehörigen Rechnernamen folgen. Auf diese Weise wurde die präzise Information über die Form von der Analyse des Inhalts getrennt. Der Prozedur *ScanHostsFile* muß nicht notwendigerweise ein Kanal übergeben werden, der von *OpenHostsFile* eröffnet worden ist. Alternativ wäre es auch beispielsweise denkbar, daß die Daten von einem Server aus dem Netzwerk bezogen werden und in einer etwas geänderten Form vorliegen.

Die Trennung von Form und Analyse des Inhalts ist nicht nur auf Basis der vorgegebenen Parameter von *StreamDisciplines* möglich. Bei gegebenen Bedarf können problemlos weitere Disziplinen definiert werden, die von speziellen Ein- oder Ausgabeprozeduren zu beachten sind.

Das obige Beispiel läßt noch ein Problem offen: Wie kann die Schließung des Kanals *file* veranlaßt werden, wenn *s* geschlossen wird? *Comments* selbst führt dieses nicht durch, so daß die Verantwortung hierfür bei der Prozedur *OpenHostsFile* verbleibt. *Streams* offeriert eine Prozedur *GetCloseEvent*, die es ermöglicht, Aktionen mit dem Ende eines Kanals zu verbinden:

```
   PROCEDURE GetCloseEvent(s: Stream; VAR type: Events.EventType);
```

Allerdings trägt das Ereignis selbst natürlich nicht die Information, welcher andere Kanal auch noch zu schließen ist, so daß hierfür eine spezielle Disziplin definiert werden muß. Dieser Aufwand ist jedoch nur notwendig, wenn das Eröffnen und Schließen in völlig verschiedenen Kontexten durchgeführt wird.

6.7. Zusammenfassung

Das Entwicklungsziel war eine erweiterbare Abstraktion für byte-orientierte Ein- und Ausgabe, die sich auf effiziente Weise implementieren läßt. Das Basismodul *Streams* definiert dabei nur die grundlegenden Operationen und überläßt darüber hinausgehende Parametrisierungen, Strukturierungen und Operationen anderen Modulen.

Die Abbildung 37 gibt einen Überblick der wichtigsten auf *Streams* basierenden Module mit ihrer Verteilung auf die einzelnen Ebenen. Man sieht daran genau die Trennung zwischen der Abstraktion *Streams*, den darauf aufbauenden Operationen und weiteren Abstraktionen (*Read* und *Write*), die Systemschnittstellen für Ein- und Ausgabe (*SysIO*) und Implementierungen sowohl allgemeiner Art (z.B. *Texts* und *ConstStrings*) als auch spezieller Art für UNIX (z.B. *UnixFiles* und *Internet*).

Selbst nachdem die Schnittstelle von *Streams* feststand, konnten nachträglich auf Basis von erweiternden Modulen — ohne die Änderung von *Streams* — Abstraktionen für asynchrone Ein- und Ausgabe (siehe nächstes Kapitel) oder Terminalschnittstellen eingeführt werden. Außerdem wurde *Streams* zum Medium für persistente Objekte und Netzwerke, die im Kapitel über Persistenz vorgestellt werden.

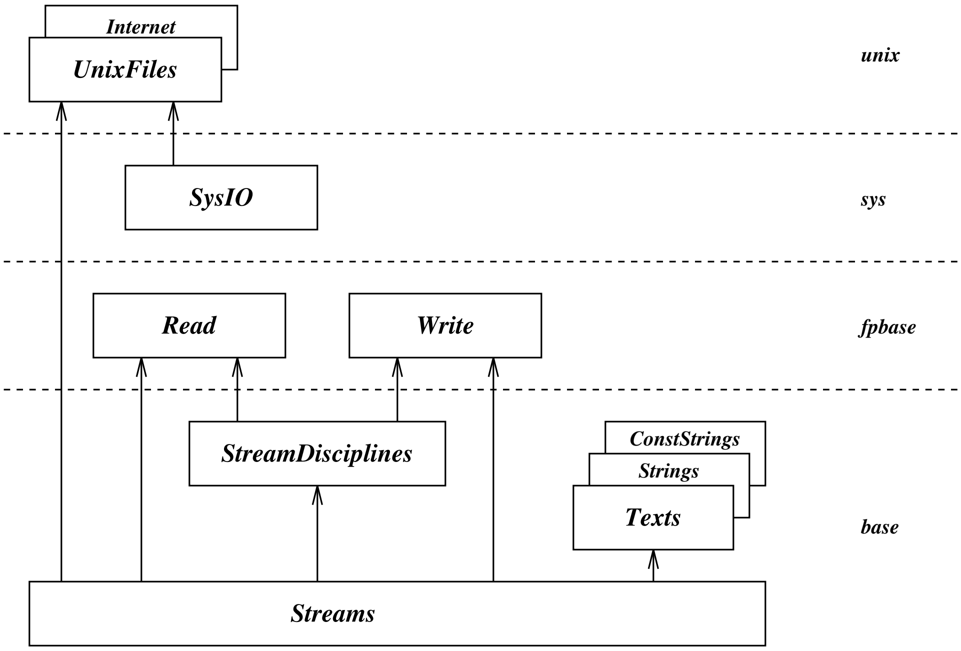

Abb. 37: Modulhierarchie für *Streams*

The more complex an organism is,
the greater is the need to think of it as a network of agents.
[Milner89]

7. Parallelität

7.1. Einführung

Parallelität im Bereich von Programmiersprachen ist in zwei Gesichtspunkten interessant, die ziemlich voneinander abweichen:

- Parallelität als Mittel zur Beschleunigung eines Programms auf Rechnern mit mehr als einem Prozessor. Hier liegt häufig der Algorithmus zuerst in sequentieller Form vor, und kann teilweise nur mit erheblichem Aufwand parallelisiert werden kann. Aus diesem Grund wurden parallelisierende Übersetzer entwickelt, die es erlauben, die Programme in der lesbareren sequentiellen Form zu belassen [Allen87].
- Parallelität als Mittel zur Modellierung. Die reale Welt besteht aus einer Vielzahl parallel agierender Objekte. Ein Abbild davon in Form von Programm-Elementen sollte nicht auf unnatürliche Weise sequentialisiert werden. Ein globaler Kontrollfluß macht dies jedoch fast unmöglich. Vom Standpunkt der Modellierung ist es dabei zunächst nicht relevant, ob nur ein Prozessor oder eine Multiprozessormaschine vorliegt. Einige objekt-orientierte Programmiersprachen, wie z.B. Simula [Birtwistle73], Actor [Tomlinson89] und ABCL/1 [Yonezawa86], haben aus diesem Grund Parallelität in die Sprache integriert.

Es gibt eine Reihe unterschiedlicher Ansätze, um Parallelität zu erreichen. Sofern nicht Parallelität im Sinne von gleichzeitiger Verarbeitung auf mehreren Prozessoren erreicht wird, spricht man auch häufig von Quasi-Parallelität. Allerdings muß sich der Unterschied nicht notwendigerweise auf die Gestaltung des Programms auswirken.

- Koroutinen erlauben es auf sehr einfache und kostengünstige Weise, mehrere Kontrollflüsse in einem Programm zu unterhalten. Zu jeder Koroutine gehört ein Keller und eine aktuelle Position im Programm. Es ist immer nur eine Koroutine aktiv, und der Wechsel der Koroutine erfolgt typischerweise explizit. Viele Programmiersprachen offerieren Koroutinen oder bieten Sprachmittel auf deren Basis an, wie z.B. Modula-2 [Wirth82]. Der besondere Vorteil von Koroutinen ist, daß keine Unterstützung von Seiten des Betriebssystems erforderlich ist.
- Verschiedene Betriebssysteme (insbesondere neuere UNIX-Varianten wie Mach und Solaris 2) erlauben die Ausführung mehrerer “leichtgewichtiger” Prozesse (*threads*) im gleichen Adreßraum mit den gleichen Rechten. Auf Multiprozessormaschinen ist eine echte Parallelität möglich, während auf Einprozessor-Maschinen nur auf Basis von Zeitscheiben eine Quasi-Parallelität hergestellt wird. Der gemeinsame Adreßraum erleichtert die gemeinsame Datenhaltung — Objekte müssen nicht auf aufwendige Weise über Interprozeßkommunikation ausgetauscht werden. Andererseits wird die Sicherstellung der Korrektheit deutlich erschwert, worauf z.B. Hoare in [Hoare85, §7.2.1] hinweist:

  “In its full generality, multithreading is an incredibly complex and error-prone technique, not to be recommended in any but the smallest programs.”
- Mehrbenutzersysteme wie UNIX erlauben die Ausführung mehrerer Prozesse, die jeweils einen individuellen Adreßraum besitzen. Analog zu den leichtgewichtigen Prozessen kann entweder echte Parallelität oder nur eine auf Basis von Zeitscheiben vorliegen. Die Kommunikation zwischen mehreren Prozessen kann auf Basis von konventionellen Dateien, Kommunikationskanälen (wie z.B. Pipelines unter UNIX) oder gemeinsam zugänglichen Speicherbereichen (*shared memory*) erfolgen.
- Verteilte Systeme erlauben die Parallelisierung auf Basis von Rechnern, die über Netze miteinander verbunden sind. Hier entstehen zusätzliche Probleme durch die Vielfalt existierender Netze und Rechner.

Koroutinen sind das einzige Mittel zur Parallelisierung, das sich auf allen Rechnern und bei allen Umgebungen auf portable Weise realisieren läßt. Da jedoch eine aktive Koroutine auf "freiwillige" Weise auch andere Koroutinen zur Ausführung kommen lassen sollte, ergeben sich besondere Anforderungen an Fairneß und Kooperationsmechanismen. Hierauf wurde in der Ulmer Oberon-Bibliothek besonderer Wert gelegt, wobei teilweise die hierfür entwickelten Mechanismen eine weitgehende Unabhängigkeit von der Art der Parallelisierung gewährleisten.

Verteilte Systeme basieren in der Ulmer Oberon-Bibliothek auf persistenten Objekten und werden daher im Rahmen des Kapitels über Persistenz vorgestellt.

Auf die Unterstützung der Ausführung mehrerer leichtgewichtiger Prozesse innerhalb eines Adreßraumes wurde verzichtet. Neben den Schwierigkeiten, die Korrektheit solcher Systeme sicherzustellen, kommt erschwerend hinzu, daß hierbei die Synchronisierungsmechanismen auf nicht portablen Operationen des jeweils zur Verfügung stehenden Betriebssystems basieren müßten. Das steht ganz im Gegensatz zu den auf Koroutinen und verteilten Systemen entwickelten Mechanismen, die Bestandteil der Bibliothek sind.

7.2. Koroutinen

7.2.1. Einführung

Koroutinen erlauben die Nebenläufigkeit von mehreren Programmteilen auf sehr einfache Weise und wurden als Sprachmittel zuerst von Conway und Erdwinn eingeführt [Conway63]. Conway entwickelte damals einen Cobol-Übersetzer, dessen einzelne Module als Koroutinen liefen, die als Produzenten und Konsumenten über insgesamt neun Kanäle miteinander verbunden waren. Je nach der Größe des vorhandenen Speichers konnte der Übersetzer vollständig auf einem Rechner laufen oder in mehrere Teile zerlegt werden.

Jede Koroutine hat ihren eigenen Keller und eine eigene aktuelle Position. Zu einem bestimmten Zeitpunkt ist immer nur eine Koroutine aktiv. Eine Koroutine kann jederzeit die Kontrolle einer anderen Koroutine übergeben. Hierbei wird zunächst der eigene Stand (aktuelle Position und die Register) gesichert und dann der Stand der anderen benannten Koroutine wieder restauriert. Der Wechsel von einer Koroutine zu einer anderen benötigt nur sehr wenige Maschineninstruktionen und ist damit nicht teurer als ein durchschnittlicher Prozeduraufruf.

7.2.2. Koroutinen in Modula-2

Modula-2 offeriert Koroutinen auf Basis von zwei Operationen aus dem Modul **SYSTEM** [Wirth82, §30]:

```
PROCEDURE NEWPROCESS(proc: PROC;
                     addr: ADDRESS; size: CARDINAL;
                     VAR new: ADDRESS);
PROCEDURE TRANSFER(VAR source, destination: ADDRESS);
```

NEWPROCESS kreiert eine neue Koroutine mit einem Keller, dessen Anfangsadresse und Größe mit *addr* und *size* gegeben wird. Die Koroutine beginnt dann später die Ausführung mit dem Aufruf der parameterlosen Prozedur *proc*. Ein Verweis auf die neue Koroutine wird mit *new* zurückgeliefert.

Einer der Nachteile von **NEWPROCESS** ist der Zwang, globale Variablen zur Parametrisierung der Koroutine verwenden zu müssen. Dies führt zu folgendem typischen Aufsetzen einer Koroutine:

```
VAR (* globale Variablen *)
   crparams: CoroutineParameters;
      (* die Parameter der Koroutine *)
   source: ADDRESS;
      (* die Koroutine, die zuletzt aktiv war *)
   newcr: ADDRESS;
      (* die Koroutine, die von NEWPROCESS zurueckgeliefert wird *)

PROCEDURE Coroutine;
   VAR
      myparams: CoroutineParameters;
BEGIN
   (* kopiere die Parameter und kehre sofort wieder zurueck *)
   myparams := crparams;
   TRANSFER(newcr, source);

   (* Rest der Koroutine *)
END Coroutine;

PROCEDURE SetupCoroutine(params: CoroutineParameters;
                         proc: PROC;
                         stackaddr: ADDRESS;
                         stacksize: CARDINAL);
   (* kreiere eine Koroutine und uebergib die Parameter *)
BEGIN
```

```
      NEWPROCESS(proc, stackaddr, stacksize, newcr);
      crparams := params;
      TRANSFER(source, newcr);
   END SetupCoroutine;
```

Hierbei wird jede neue Koroutine sofort nach dem Kreieren gestartet, damit sie die Parameter zu sich kopieren kann. Nachdem dies geschehen ist, kehrt sie sofort wieder zurück, da üblicherweise der Rest der Koroutine die Existenz weiterer Koroutinen voraussetzt.

NEWPROCESS und **TRANSFER** werden von nahezu jeder Modula-2-Implementierung unterstützt und erscheinen damit auf den ersten Blick sehr portabel. Allerdings ist die anzugebene Kellergröße sehr systemabhängig. Da der Keller mit Anfangsadresse und Größenangabe fest vom Aufrufer vorgegeben werden muß, ist es konzeptionell unmöglich, Koroutinen mit wachsendem Keller zu unterstützen.

TRANSFER hat zwei **VAR**-Parameter: *source* und *destination*. Dies ermöglicht zwei Techniken zur Allokation der Verwaltungsinformation für Koroutinen:

- Die Adresse zu der Verwaltungsinformation bleibt konstant. So könnte die Verwaltungsinformation am Anfang des Kellers abgelegt werden.
- Die Verwaltungsinformation wird allokiert zum Zeitpunkt des Aufrufs von **TRANSFER**. Geeignet wäre hierfür die Spitze des Kellers.

Die erste Koroutine spielt eine Sonderrolle, da sie implizit während des Programmstarts kreiert wird. Der Zeiger auf die zugehörige Verwaltungsinformation wird durch den ersten Aufruf von **TRANSFER** bekannt. Damit ist klar, daß zumindest hierfür *source* als **VAR**-Parameter von **TRANSFER** benötigt wird. Dazu ist entweder die zweite Technik oder ein vorher nicht sichtbarer Zeiger auf die Verwaltungsinformation notwendig. Entsprechend könnte man auf den *source*-Parameter verzichten, wenn man die erste Technik verwendet und den Zeiger auf die erste Koroutine von Anfang an zugänglich macht.

Die zweite Technik führt zudem zu veralteten Koroutinenzeigern, wie folgendes Beispiel zeigt:

```
VAR
   main, newcr: ADDRESS;

PROCEDURE Coroutine;
   VAR
      cr: ADDRESS;
BEGIN
   TRANSFER(cr, main);
   (* ... *)
END Coroutine;

PROCEDURE Setup;
BEGIN
   NEWPROCESS(Coroutine, addr, size, newcr);
   TRANSFER(main, newcr);
   (* newcr ist inzwischen ein veralteter Zeiger *)
END Setup;
```

7.2.3. Das Ulmer Koroutinenkonzept

Im Gegensatz zu Modula-2 sieht der Oberon-Report keine Koroutinen vor. Dies mag im Rahmen des übersichtlichen Züricher Oberon-Systems noch vertretbar sein — in einem allgemeinen Umfeld würde dies jedoch die Erweiterbarkeit behindern.[1] Aus diesem Grund wird analog zu Modula-2 ein Koroutinenkonzept von **SYSTEM** exportiert, das allerdings deutlich vereinfacht wurde und somit leichter anzuwenden ist. **NEWPROCESS** und **TRANSFER** wurden durch **CRSPAWN**, **CRSWITCH** und dem Modul *Coroutines* (siehe unten) ersetzt:

[1] Seit kurzem gibt es deswegen auch eine Variante des Züricher Oberon-Systems mit Koroutinen [Lalis94].

```
PROCEDURE CRSPAWN(VAR newcr: COROUTINE);
PROCEDURE CRSWITCH(dest: COROUTINE);
```

Im Gegensatz zu Modula-2 können Prozeduren, die eine Koroutine repräsentieren, sich selbst zu Koroutinen deklarieren, wie folgendes Beispiel zeigt:

```
PROCEDURE Coroutine(myparams: CoroutineParameters;
                    VAR newcr: SYSTEM.COROUTINE);
BEGIN
   (* Start als normale Prozedur *)
   SYSTEM.CRSPAWN(newcr); (* Ruecksprung zu Setup *)
   (* hier wird die Koroutine nach CRSWITCH(newcr) fortgesetzt *)
END Coroutine;

PROCEDURE Setup;
   VAR
      cr: SYSTEM.COROUTINE;
BEGIN
   Coroutine(crparams, cr);
   (* ... *)
END Setup;
```

Im Gegensatz zu **NEWPROCESS** in Modula-2 dürfen Koroutinen Parameter besitzen, da sie selbst nicht als Parameter übergeben werden und damit einem vorgegebenen Prozedurtyp entsprechen müssen. Darüber hinaus muß weder ein Keller übergeben noch die Größe des Kellers festgelegt werden.

CRSPAWN ist dadurch natürlich etwas aufwendiger als **NEWPROCESS** und führt folgende Schritte durch:

- Kreieren eines neuen Kellers. Die Anfangsgröße des Kellers wird von dem Umfang der lokalen Daten der Koroutinenprozedur und einem vorgegebenen Wert aus *Coroutines.defaultsize* bestimmt. *Coroutines.defaultsize* ist eine unter normalen Umständen ausreichende Größe, die natürlich leicht erhöht werden kann. In diesem Fall bezieht sich die Vergrößerung gleichmäßig auf alle Koroutinen. Der Keller muß nicht vom Aufrufer angelegt werden und unterliegt analog zur Allokation mit **NEW** der Kontrolle systemspezifischer Module. Nur durch diese geänderte Schnittstelle ist es bei der Ulmer Bibliothek möglich, daß die Keller von Koroutinen wachsen können.
- Die Verwaltungsinformation für die Koroutine wird am Anfang des Kellers angelegt und initialisiert. Die Verwaltungsinformation enthält die Kellerverwaltungsregister und die aktuelle Position der Koroutine, die auf die nächste Instruktion nach dem Aufruf von **CRSPAWN** gesetzt wird.
- Die Parameter und die lokalen Variablen der Prozedur, die **CRSPAWN** aufgerufen hat, werden zu dem neuen Keller kopiert.
- Anschließend erfolgt ein sofortiger Rücksprung zum Aufrufer der zur Koroutine gewordenen Prozedur. Aus diesem Grund dürfen Koroutinenprozeduren keine Funktionsprozeduren sein.

Auch **CRSWITCH** ist in der Anwendung gegenüber **TRANSFER** vereinfacht worden. Da die Koroutinenzeiger konstant bleiben, ist es nur noch notwendig, die Zielkoroutine anzugeben. Der Zeiger auf die erste Koroutine wird beim Programmstart gesetzt und ist über das Modul *Coroutines* zugänglich:

```
DEFINITION Coroutines;

   IMPORT SYS := SYSTEM, Types;

   TYPE
      Coroutine = SYS.COROUTINE;
   VAR
      defaultsize: Types.Size;
         (* die Voreinstellung fuer die Anfangsgroesse
            einer Koroutine
         *)
      main: Coroutine;
         (* der Zeiger auf die erste Koroutine *)
      source: Coroutine;
         (* die Koroutine, die zuletzt aktiv war *)
      current: Coroutine;
         (* die Koroutine, die zur Zeit aktiv ist *)
```

```
END Coroutines.
```

Die Zeiger *source* und *current* werden dabei von **CRSWITCH** gesetzt. Auf diese Weise ist es recht einfach, zur zuletzt aktiven Koroutine wieder zurückzuwechseln. In Modula-2 wurde hierfür der *source*-Parameter von **TRANSFER** verwendet.

Auf dieser Basis läßt sich leicht das Produzenten/Konsumenten-Problem lösen, wie folgendes Beispiel zeigt:

```
PROCEDURE Produce(VAR newtoken: Token;
                  VAR producer, consumer: Coroutines.Coroutine);
   (* produziere jeweils ein Token, lege es in 'newtoken' ab
      und wechsle dann zu 'consumer'
   *)
   VAR
      token: Token;
BEGIN
   SYSTEM.CRSPAWN(producer);
   LOOP
      (* produziere 'token' *)
      newtoken := token;
      SYSTEM.CRSWITCH(consumer);
   END;
END Produce;

PROCEDURE Consume(VAR newtoken: Token;
                  VAR producer, consumer: Coroutines.Coroutine);
   (* hole von 'newtoken' jeweils ein neues Token,
      konsumiere es und wechsle dann zu 'producer'
   *)
   VAR
      token: Token;
BEGIN
   SYSTEM.CRSPAWN(consumer);
   LOOP
      token := newtoken;
      (* konsumiere 'token' *)
      SYSTEM.CRSWITCH(producer);
   END;
END Consume;

PROCEDURE Setup;
   VAR
      token: Token;
      producer, consumer: Coroutines.Coroutine;
BEGIN
   Produce(token, producer, consumer);
   Consume(token, producer, consumer);
   SYSTEM.CRSWITCH(producer);
END Setup;
```

7.2.4. Iteratoren

Im obigen Beispiel erfolgt der Wechsel in der Kontrolle explizit. Damit wird der Programmtext schwerer lesbar und auch abhängig von der verwendeten Technik der Parallelisierung. Das Modul *Iterators* in der Ulmer Oberon-Bibliothek erlaubt die Realisierung von Produzenten-Konsumenten-Problemen unabhängig von der Art des darunterliegenden Koroutinenschemas.

Bei Iteratoren wird nur eine Koroutine zusätzlich kreiert, die die Rolle des Produzenten übernimmt. Danach übernimmt die aufrufende Koroutine den Konsumenten. Verbunden sind beide über die Abstraktion eines Iterators: Der Produzent liefert ein produziertes Objekt mit *Yield*, während der Konsument sie mit *Get* abholt:

```
TYPE Reference = Objects.Object; Mode = SHORTINT;
TYPE Iterator = POINTER TO IteratorRec;
TYPE IteratorRec = RECORD (Disciplines.ObjectRec) END;
TYPE IteratorProc = PROCEDURE (it: Iterator; ref: Reference; mode: Mode);

PROCEDURE Create(VAR it: Iterator; itproc: IteratorProc;
                 ref: Reference; mode: Mode);
```

```
PROCEDURE Yield(it: Iterator; object: Objects.Object);
PROCEDURE Get(it: Iterator; VAR object: Objects.Object) : BOOLEAN;
```

Natürlich muß der Prozedurtyp einer Koroutinenprozedur festgelegt werden, wenn man wie bei *Iterators* eine Abstraktion zur Parallelisierung unabhängig von der direkten Verwendung von **CRSPAWN** und **CRSWITCH** gestalten möchte. Allerdings ist das nicht so unangenehm wie in Modula-2, da über Typerweiterungen letztendlich beliebig parametrisiert werden kann. Der Prozedurtyp *IteratorProc* sieht natürlich den Verweis auf den Iterator selbst vor, dann eine beliebige Erweiterung von *Objects.Object* und einen weiteren kleinen Parameter, der zur Auswahl verschiedener Iterationsmodi dient (z.B. aufsteigend oder absteigend bei sortierten Listen).

Im Gegensatz zu direkten Koroutinenprozeduren darf die bei *Create* übergebene Prozedur die Iteration mit **RETURN** beenden. Das führt dann dazu, daß auf der Konsumentenseite *Get* **FALSE** zurückliefert.

7.3. Sprachmittel zur Synchronisierung

7.3.1. Einführung

Viele bekannte Sprachmittel zur Synchronisierung, seien es Dijkstras Semaphore [Dijkstra68], Hoares kritische Bereiche mit Bedingungen, Monitore [Hoare74], *path expressions* [Campbell73] oder Kapseln [Gehani93], gehen davon aus, daß alle Parteien (d.h. die konkurrierenden Einheiten) zu *einem* Programm gehören. Somit lassen sich keine Synchronisierungen auf dieser Basis mit externen Einheiten, wie z.B. dem Betriebssystem oder anderen Prozessen, auf dem gleichen oder anderen Rechnern modellieren. Dies ist der Preis, der häufig dafür gezahlt werden muß, wenn das Sprachmittel zur Synchronisierung in die Programmiersprache integriert wird. Bei einer Bibliothek besteht jedoch die Chance, erweiterbare Abstraktionen zu entwickeln, die es auch erlauben, externe Einheiten mit einzubinden.

7.3.2. Ereignisse in CSP

Hoare bietet mit seiner Modellierung von Prozessen in CSP[1] viel grundlegendere Begriffe und Mechanismen an, die es insbesondere erlauben, interne mit externen Einheiten zu integrieren [Hoare85]. Hoare betrachtet Prozesse nur aus der Sicht, an welchen Ereignissen sie in welcher Folge teilnehmen. Für jeden Prozeß ist die Menge der interessanten Ereignisse definiert, die als Alphabet bezeichnet wird. Ein Ereignis findet genau dann statt, wenn alle daran interessierten Prozesse in synchroner Weise dazu bereit sind.

Als Beispiel läßt sich ein Getränkeautomat und ein Benutzer modellieren:

$$Automat = (muenze \rightarrow (taste\_kaffee \rightarrow kaffee \rightarrow Automat \mid$$
$$taste\_tee \rightarrow tee \rightarrow Automat))$$

Das Alphabet des Getränkeautomaten besteht aus den Ereignissen *muenze*, *taste_kaffee*, *kaffee*, *taste_tee* und *tee*. Wenn es zu dem Ereignis *muenze* kommt, kann er wahlweise das Ereignis *taste_kaffee* oder *taste_tee* akzeptieren, worauf es entsprechend zu *kaffee* oder *tee* kommt. Durch die Rekursion wiederholt sich der Prozeß.

Passend zu dem Automaten kann ein Benutzer modelliert werden, der beispielsweise nur Tee bevorzugt:

$$Benutzer = (muenze \rightarrow taste\_tee \rightarrow tee \rightarrow trinken \rightarrow Benutzer)$$

Das Alphabet des Benutzers besteht somit aus den Ereignissen *muenze*, *taste_tee*, *tee* und *trinken*. Der Benutzer wirft eine Münze in den Automaten ein, entnimmt einen Tee und trinkt ihn anschließend, worauf der Prozeß per Rekursion wiederholt wird. Werden beide Prozesse in paralleler Ausführung zusammengebracht, können Ereignisse, die in der Schnittmenge der Alphabete enthalten sind, nur simultan bei beiden Prozessen bearbeitet werden:

$$Automat \parallel Benutzer = \mu X.\,(muenze \rightarrow taste\_tee \rightarrow tee \rightarrow trinken \rightarrow X)$$

$\mu X.\,(F(X))$ steht dabei für die eindeutige Lösung von $X = F(X)$.[2]

Während Ereignisse ideal zur Modellierung von Synchronisierungen sowohl mit internen als auch externen Parteien geeignet sind, wäre es zu aufwendig, auf das simultane Bearbeiten eines Ereignisses zu bestehen.

[1] CSP steht für *Communicating Sequential Processes*

[2] Wie Hoare zeigt, läßt sich die Eindeutigkeit auf Basis der Fixpunkt-Theorie beweisen, wenn auf der rechten Seite vor dem X mindestens ein Ereignis steht.

Insbesondere bei verteilten Systemen kann einige Zeit vergehen, bis eine Nachricht von einem Prozeß zu einem anderen gelangt ist. Natürlich läßt CSP auch eine Modellierung von asynchronen Ereignissen auf Basis von sogenannten Puffern zu. Hoare entschied sich wohl für die Synchronität von Ereignissen, um zu mehr und einfacheren Beweisregeln zu gelangen.

Wie könnte die Implementierung eines Automaten aussehen? Ein Automat würde zunächst darauf warten, daß eine Münze eingeworfen wird, dann auf die Wahl zwischen Kaffee und Tee warten, um anschließend das gewünschte Getränk auszuliefern:

```
LOOP
   (* Warte darauf, dass eine Muenze eingeworfen wird *)
   (* Warte darauf, dass entweder die Taste fuer Kaffee
      oder Tee gedrueckt wird
   *)
   IF (* Kaffee ausgewaehlt *) THEN
      (* Liefere Kaffee aus *)
   ELSE (* Tee ausgewaehlt *)
      (* Liefere Tee aus *)
   END;
END;
```

Hier wird eine Unterscheidung getroffen, ob ein Ereignis von außen (Münze einwerfen, Wahl festlegen) oder von innen (Kaffee bzw. Tee ausliefern) generiert wird. Hoare unterstützt die Unterscheidung mit der Notation von Kommunikationskanälen:

$$\begin{aligned} Automat = (in?muenze \rightarrow (in?taste\_kaffee \rightarrow out!kaffee \rightarrow Automat \mid \\ in?taste\_tee \rightarrow out!tee \rightarrow Automat)) \end{aligned}$$

7.4. Bedingungen

7.4.1. Einführung

Eine Bedingung ist eine Funktion[1] mit dem Resultatstyp **BOOLEAN**, die angibt, ob ein bestimmtes Ereignis im Sinne von CSP[2] (ggf. nach einem bestimmten Zeitpunkt) eingetreten ist. *Conditions* offeriert in der Ulmer Oberon-Bibliothek die Abstraktion für Bedingungen, die frei definiert werden können. Minimale Vorgabe ist nur die Zuordnung einer **BOOLEAN**-Funktion, die die Feststellung trifft, ob das Ereignis eingetreten ist. Die Ausführung dieser Funktion darf nur vergleichsweise wenig Zeit benötigen. Analog zu dem $\Box$-Operator in CSP ist es möglich, auf das Eintreten verschiedener Ereignisse gleichzeitig zu warten.

Um eine größtmögliche Flexibilität und Erweiterbarkeit zu erreichen, wurde *Conditions* von der Art der Parallelisierung unabhängig gehalten. Der Warte-Algorithmus von *Conditions* bezieht sich somit nur auf den Fall, daß alle aktiven Einheiten eines Programms auf Ereignisse von außen warten. Darauf können Abstraktionen aufgebaut werden, die auf Koroutinen basieren, wie z.B. *Tasks* und *Schedulers*, die später vorgestellt werden.

7.4.2. Der Warte-Algorithmus von *Conditions*

Die gewonnene Freiheit bei der Modellierung hat ihren Preis. Das Hauptproblem ist das effiziente Warten darauf, daß aus einer gegebenen Menge von Bedingungen eine den Wert **TRUE** zurückliefert. Im einfachsten Fall ließe sich das durch eine *busy loop* erledigen:[3]

[1] *Funktion* ist hier nicht im Sinne einer mathematischen Funktion zu verstehen, sondern als Funktionsprozedur, die Seiteneffekte haben darf (und teilweise sogar haben muß).

[2] Wenn nichts anderes angegeben ist, ist *Ereignis* im ganzen Abschnitt im Sinne von CSP zu verstehen und nicht im Sinne des Moduls *Events* der Ulmer Oberon-Bibliothek. Allerdings lassen sich die Ereignisse aus *Events* im Sinne von CSP verwenden.

[3] Hier und im folgenden wird die Behandlung von Bedingungsmengen im Oberon-Programmtext zur besseren Lesbarkeit vereinfacht.

```
PROCEDURE Wait(cset: ConditionSet);
BEGIN
   LOOP
      FOR ALL c ∈ cset DO
         IF Test(c) THEN
            RETURN
         END;
      END;
   END;
END Wait;
```

Dies wäre jedoch ausgesprochen unfair, wenn es so auf einem Mehrbenutzersystem wie UNIX zur Ausführung kommen würde. Das Einführen einer Verzögerung in der Schleife bringt ebenfalls Nachteile: Ist die Verzögerung zu klein, so bleibt der Charakter der *busy loop* erhalten, ist sie zu groß, besteht die Gefahr, daß viel zu spät auf das Eintreffen von Ereignissen reagiert wird.

Glücklicherweise sind bei vielen konkreten Bedingungen Mechanismen bekannt, wie man gezielt warten könnte. Wenn man beispielsweise nur auf Eingabe aus einem Kanal des Betriebssystems warten würde, könnte man statt dessen eine blockierende Lese-Operation aufrufen. Problematisch ist hier jedoch die potentielle Kombination einer Vielzahl unterschiedlicher Bedingungen. Um diese Schwierigkeit zu überwinden, wurde eine Reihe zusätzlicher Mechanismen eingeführt.

Einige Begriffe: Mit System oder Betriebssystem werden im folgenden Komponenten beschrieben, die *außerhalb* des Programms sind. Der Begriff Prozeß bezeichnet (im Gegensatz zu CSP) eine vom Betriebssystem aus identifizierbare ausführbare Einheit.

7.4.2.1. Unterbrechungen

Viele Betriebssysteme unterstützen asynchrone Unterbrechungen, die das Eintreffen eines Ereignisses signalisieren können. Unter UNIX gibt es hierfür den *signal*-Mechanismus. Er läßt sich beispielsweise dafür verwenden, das Erreichen eines bestimmten Zeitpunktes oder die Terminierung eines Prozesses zu signalisieren. Solange keine Schutzmaßnahmen getroffen worden sind (wie z.B. implizit durch das Erhöhen der Priorität in der Ulmer Oberon-Bibliothek), kann das Signal jederzeit eintreffen und sogar längerdauernde Systemaufrufe (wie das Einlesen von einem Terminal) unterbrechen.

Process.Pause dient dabei als Schnittstelle zu einem Systemaufruf, der den Prozeß bis zum Eintreffen einer Unterbrechung suspendiert. Wenn sich z.B. alle Bedingungen nur durch das Eintreffen einer Unterbrechung ändern können, würde es völlig ausreichen, *Process.Pause* aufzurufen, um auf die nächste Unterbrechung zu warten.

7.4.2.2. Interne und externe Bedingungen

Conditions unterscheidet zwischen Bedingungen, die sich auf interne und solche, die sich auf externe Ereignisse beziehen. Ein internes Ereignis kann nur dadurch eintreten, daß der Prozeß weiterläuft (sei es auf normale Weise oder durch eine Unterbrechung). Somit können aus der obigen Warteschleife alle internen Bedingungen ohne Verlust herausgenommen werden, nachdem in einem Eingangstest festgestellt worden ist, daß keine von ihnen zu Beginn wahr ist.

7.4.2.3. Zusammengehörende Bedingungen

Insbesondere bei externen Ereignissen ist es typisch, daß eine Reihe von verschiedenen Bedingungen zusammengefaßt werden kann in der Form, daß es für sie allein einen günstigen Warte-Algorithmus gibt. Im idealen Fall könnte man alle externen Ereignisse auf diese Weise zusammenfassen, da sie alle auf Interaktionen mit dem Betriebssystem beruhen. Leider gibt es bei den bekannten Betriebssystemen wie auch bei UNIX keine Unterstützung in dem Sinne, daß es einen Systemaufruf gibt, der es erlaubt, auf eine beliebige Menge vom Betriebssystem definierter Ereignisse zu warten. UNIX unterstützt dies nur in Teilbereichen: So kann mit *select* unter BSD oder *poll* unter System V auf eine beliebige Menge von Ereignissen im Bereich der Ein- und Ausgabe gewartet werden.

Das Dilemma läßt sich glücklicherweise auf der Basis von Unterbrechungen teilweise lösen. Wenn beispielsweise als Ereignisse insgesamt nur Prozeßterminierungen (von aus dem eigenen Prozeß hervorgegangen Prozessen) und Ein- und Ausgabe-Ereignisse vorliegen, so wäre es denkbar, via *select* bzw. *poll* zu blockieren und sich bei Prozeßterminierungen von UNIX-Signalen unterbrechen zu lassen. Bei neueren UNIX-Versionen (wie z.B. UNIX-System V 4.0) würde es nur Schwierigkeiten geben, wenn man Ein- und Ausgabe-Ereignisse mit den Ereignissen aus den inzwischen obsoleten Interprozeßkommunikationsmechanismen *messages* und *semaphores* kombinieren würde.

7.4.2.4. Vorbedingungen

Viele Ereignisse lassen sich nur durch kompliziertere Bedingungen beschreiben. So kommt es beispielsweise vor, daß nicht allgemein auf Eingabe aus einem bestimmten Kanal gewartet werden soll, sondern auf das Eintreffen einer bestimmten Eingabe (das ist typischerweise der Fall, wenn mehrere logische Verbindungen auf Basis eines einzelnen Kanals existieren). Eine solche Bedingung läßt sich natürlich nicht einfach mit anderen externen Bedingungen in einer Weise zusammenfassen, die ein vereinfachtes Warten erlauben würde.

Um dieses Problem zu umgehen, kann eine Bedingung eine Menge von Vorbedingungen angeben, wovon mindestens eine erfüllt sein muß, bevor die eigentliche Bedingung wahr werden kann. Vorbedingungen können natürlich selbst wieder Vorbedingungen enthalten. Auf diese Weise können aus einer Reihe von zusammengesetzten Bedingungen die zunächst interessanten Bedingungen extrahiert werden, die in dieser Form direkt an das System (über eine der Zusammenfassungen) übergeben werden können.

7.4.2.5. Algorithmus

In der folgenden Form des Algorithmus ist *WaitForAndSelect* eine Prozedur, die eine Menge von Bedingungen erhält, auf das Wahrwerden mindestens einer der Bedingungen wartet und dann die Menge der wahren Bedingungen zurückliefert:

```
PROCEDURE WaitForAndSelect(cset: ConditionSet; VAR trueconds: ConditionSet);
```

Dies stellt eine Vereinfachung gegenüber der Version aus *Conditions* dar, bei der noch ein optionales Zeitlimit und die Fehlerbehandlung hinzukommen. Ferner gibt es noch einige weitere Mechanismen, die den unangenehmen Fall einer *busy loop* abmildern sollen oder einige kleinere Optimierungen in Spezialfällen erlauben. Auf die Darstellung davon wurde hier aus Gründen der Übersichtlichkeit verzichtet.

Zunächst wird die rekursive Ersetzung von Bedingungen durch ihre Vorbedingungen notwendig. Der Algorithmus hierfür ähnelt sehr dem der Bestimmung einer transitiven Hülle. Der einzige Unterschied besteht darin, daß hier eine Bedingung durch ihre Vorbedingungen ersetzt wird (sofern sie welche hat), während bei der transitiven Hülle beide Seiten einer Beziehung in der Ergebnismenge verbleiben. Die folgende Funktion *PrecondClosure* verwendet die Funktion *Preconds*, die für eine gegebene Bedingung die Menge der Vorbedingungen zurückliefert, die auch leer sein kann.

```
PROCEDURE PrecondClosure(cset: ConditionSet) : ConditionSet;
   VAR
      closure: ConditionSet;
      preconds: ConditionSet;
BEGIN
   closure := ∅;
   REPEAT
      preconds := ∅;
      FOR ALL c ∈ cset DO
         IF Preconds(c) = ∅ THEN
            closure := closure ∪ { c };
         ELSE
            preconds := preconds ∪ Preconds(c);
         END;
      END;
      cset := preconds;
   UNTIL preconds = ∅;
   RETURN closure
END PrecondClosure;
```

Weiter wird eine Prozedur benötigt, die eine Menge von Bedingungen *cset* in folgende disjunktive Zerlegungen überführt:

$$cset = external \bigcup internal \quad mit \quad external \bigcap internal = \emptyset$$

$$external = asyncconds \bigcup other \quad mit \quad asyncconds \bigcap other = \emptyset$$

$$other = \bigcup_{i=1}^{n} cset_i \quad mit \quad \forall i \neq j: cset_i \bigcap cset_j = \emptyset$$

wobei

external	Menge der externen Bedingungen
internal	Menge der internen Bedingungen
asyncconds	Menge der Bedingungen, deren zugehöriges Ereignis durch eine Unterbrechung signalisiert werden kann. Es gilt $asyncconds \subset external$.
$cset_i$	Menge aller übrigen Bedingungen, die jeweils genau einer Zusammenfassung $domain_i$ angehören, wobei $domain_i \neq domain_j \;\; \forall\, i \neq j$. Zu jeder Zusammenfassung $domain_i$ gibt es eine Operation $select_i$, die ein effizientes Warten für diesen Bereich ermöglicht.
n	Anzahl der verschiedenen Zusammenfassungen in *other*.

Ferner wird *TrueConds* benötigt, die die Menge der erfüllten Bedingungen zurückliefert:

```
PROCEDURE TrueConds(cset: ConditionSet) : ConditionSet;
   VAR trueconds: ConditionSet;
BEGIN
   trueconds := ∅;
   FOR ALL c ∈ cset DO
      IF Test(c) THEN
         trueconds := trueconds ∪ { c };
      END;
   END;
END TrueConds;
```

Auf dieser Basis kann *WaitForAndSelect* folgendermaßen implementiert werden:

```
PROCEDURE WaitForAndSelect(cset: ConditionSet; VAR trueconds: ConditionSet);
   VAR
      conds: ConditionSet;
BEGIN
   trueconds := TrueConds(cset);
   WHILE trueconds = ∅ DO
      conds := PrecondClosure(cset);
      (* disjunktive Zerlegung wie oben spezifiziert durchfuehren *)
      (* fuer alle Bedingungen, die zu asyncconds gehoeren, ist
         die Signalisierung durch Unterbrechungen anzufordern
      *)
      WHILE trueconds = ∅ DO
         IF n = 1 THEN
            trueconds := select1(cset1);
         ELSE
            (* busy loop:
               fuer ein beliebiges i aus 1..n
               trueconds := selecti(cseti);
               mit Zeitlimit ausfuehren
            *)
         END;
         (* hierhin gelangen wir, wenn entweder ein selecti
            terminierte oder unterbrochen worden ist
         *)
         IF trueconds = ∅ THEN
            (* im Falle einer Unterbrechung (oder bei der busy loop)
               muss nachgesehen werden, ob sich etwas geaendert hat
            *)
            trueconds := TrueConds(conds);
         END;
      END;
      trueconds := TrueConds(cset);
   END;
END WaitForAndSelect;
```

7.4.3. Die Semantik von Bedingungen

7.4.3.1. Das Automatenbeispiel

Auf der Basis von Bedingungen läßt sich das Beispiel für den Getränkeautomaten leicht formulieren. Im folgenden wird davon ausgegangen, daß es sich bei dem Alphabet des Automaten um externe Ereignisse handelt und keine Kooperation mit anderen Koroutinen erforderlich ist und somit *Conditions.WaitForAndSelect* direkt verwendet werden kann. Im Normalfall würde hier die entsprechende Operation aus *Tasks* verwendet werden.

Die Bedingungen *muenze*, *taste_kaffee*, und *taste_tee* werden als definiert und gegeben angenommen. Sie beziehen sich alle darauf, ob das entsprechende Ereignis seit dem letzten Aufruf von *Accept* für das jeweilige Ereignis stattgefunden hat oder nicht. Die Prozeduren *Kaffee* und *Tee* generieren die Ereignisse *kaffee* bzw. *tee* in einer Weise, daß sie von externen Prozessen wahrnehmbar sind.

```
PROCEDURE Automat;
   VAR
      set: ConditionSet;
BEGIN
   LOOP
      Conditions.WaitForAndSelect({muenze}, set); Accept(muenze);
      Conditions.WaitForAndSelect({taste_kaffee, taste_tee}, set);
      IF taste_kaffee ∈ set THEN
         Accept(taste_kaffee);
         Kaffee;
      ELSE (* taste_tee ∈ set *)
         Accept(taste_tee);
         Tee;
      END;
   END;
END Automat;
```

7.4.3.2. Ereignisse und Bedingungen

Die Einführung einer Prozedur *Accept* wirkt auf den ersten Blick befremdlich, wird aber im Rahmen der Definition von Bedingungen klar. Eine Bedingung ist genau dann wahr, wenn sich das Ereignis ereignet hat, auf das sie Bezug nimmt. Als Ereignis sollte dabei aber sinnvollerweise nicht das Einwerfen einer Münze zu irgendeinem Zeitpunkt betrachtet werden, sondern der Eingang einer *neuen* Münze.

Dies entspricht der asynchronen Natur der Bedingungen, die sich in CSP am besten durch die Einführung eines Puffers modellieren läßt:

$$Buffer(in, out) = \mu X.\,(in?x \rightarrow out!x \rightarrow X)$$

Das Ereignis tritt genau dann ein, wenn es mit $in?x$ von *Buffer* akzeptiert wird. Der Puffer hat in diesem Beispiel nur die Kapazität von 1 — anderswo kann es sinnvoller sein, dem Puffer eine größere Kapazität (vielleicht sogar unbeschränkt) zu geben, damit es nicht zum Verzögern von Ereignissen kommt. Dann sollte man nur davon ausgehen, daß *Buffer* folgende Spezifikation erfüllt:

$$Buffer \textbf{ sat } tr{\downarrow}out \leq tr{\downarrow}in$$

Dies steht dafür, daß die Folge der Ereignisse von *out* eine anfängliche Teilfolge von *in* ist.

Nach der ersten zeitlichen Verzögerung kann das Ereignis in den Sichtbereich einer Bedingung gelangen, die hier mit vier Kanälen modelliert ist: einem Eingabekanal für das Ergebnis und dann noch Ein- und Ausgabekanälen für die Operationen *test* und *accept*:[1]

$$\begin{aligned} Condition(in, test, testout, accept, out) = \mu X.\,(&\mu Y.\,(in?x \rightarrow SKIP \mid test \rightarrow testout!FALSE \rightarrow Y); \\ &\mu Z.\,(accept \rightarrow out!x \rightarrow SKIP \mid test \rightarrow testout!TRUE \rightarrow Z); X) \end{aligned}$$

Condition wartet zunächst auf die Mitteilung, daß das Ereignis eingetreten ist. Während dieser Zeit liefert die Testoperation von *Condition* **FALSE** zurück. Sobald *Condition* erfährt, daß das Ereignis eingetreten ist, wird in einen anderen Zustand übergewechselt, bei dem die Testoperation **TRUE** liefert. Dies bleibt solange der Fall, bis *Condition* nach *accept* wieder bereit ist, auf das erneute Eintreten des Ereignisses zu warten.

Ein möglicher Verlauf von

$$Buffer(in, bc) \parallel Condition(bc, test, testout, accept, out)$$

ist

$$< in?x,\, test,\, testout!FALSE,\, bc.\,x,\, test,\, testout!TRUE >$$

Man sieht hieran deutlich, daß es auch noch nach dem Eintreten des Ereignisses ($in?x$) durch die zeitliche

[1] *SKIP* steht in CSP für einen Prozeß, der nichts tut, aber erfolgreich terminiert.

Verzögerung vorkommen kann, daß die Testoperation **FALSE** zurückliefert.

7.4.3.3. Das Warten

Das Warten auf das Wahrwerden von einer oder zwei Bedingungen (und analog damit für mehr Bedingungen) auf der Basis von *test* läßt sich unter Verwendung einer *busy loop* folgendermaßen in CSP beschreiben:

$$\begin{aligned} Wait1(cond) = \mu X.\,(&cond.\,test \rightarrow cond.\,testout?boolval \rightarrow \\ &\textbf{if } boolval \textbf{ then } SKIP \\ &\textbf{else} \rightarrow X) \end{aligned}$$

$$\begin{aligned} Wait2(cond1, cond2) = \mu X.\,(&cond1.\,test \rightarrow cond1.\,testout?boolval1 \rightarrow \\ &cond2.\,test \rightarrow cond2.\,testout?boolval2 \rightarrow \\ &\textbf{if } boolval1 \textbf{ or } boolval2 \textbf{ then } SKIP \\ &\textbf{else} \rightarrow X) \end{aligned}$$

Man sieht hier den von CSP formal erfaßten Nachteil einer *busy loop*. So ist

$$(Condition(in, cond.\,test, cond.\,testout, accept, out) \parallel Wait(cond)) \setminus \{cond.\,test, \alpha cond.\,testout(Condition)\}$$

divergent, d.h. der Umfang der Kommunikation zwischen *Condition* und *Wait*, der stattfindet, ohne daß es zu einer anderen Interaktion kommt, ist nicht nach oben beschränkt. Glücklicherweise kann dies im Normalfall durch den verbesserten Warte-Algorithmus verhindert werden.

7.4.3.4. Das Beispiel mit der gleichen Semantik in CSP

Auf dieser Basis kann der Automat in CSP mit der Semantik der Oberon-Prozedur *Automat* angegeben werden:[1]

$$\begin{aligned} &Automat(muenze, taste\_kaffee, taste\_tee\colon Condition; kaffee, tee\colon Event) = \\ &\quad \mu X.\,(Wait1(muenze); Accept(muenze); \\ &\qquad Wait2(taste\_kaffee, taste\_tee); \\ &\qquad (taste\_kaffee.\,test \rightarrow taste\_kaffee.\,testout?boolval \rightarrow \\ &\qquad \textbf{if } boolval \textbf{ then } Accept(taste\_kaffee) \rightarrow kaffee \rightarrow SKIP \\ &\qquad \textbf{else } Accept(taste\_tee) \rightarrow tee \rightarrow SKIP); \\ &\qquad X) \end{aligned}$$

wobei *Accept* als Kurzform zum Akzeptieren und Ignorieren dient:

$$Accept(cond\colon Condition) = cond.\,accept \rightarrow cond.\,out?x$$

Natürlich unterscheidet sich diese Version des Automaten von der zuerst in CSP vorgestellten Version. Durch die Pufferung, die die Asynchronität und damit den Zeitabstand zwischen einem Ereignis und dem Wahrwerden einer Bedingung simuliert, kann aus der Sicht des Automaten nicht mehr verfolgt werden, in welcher ursprünglichen Reihenfolge die Ereignisse (z.B. Münze einwerfen und Taste für Tee drücken) eingetreten sind. Wenn dies von Interesse wäre, müßten Zeitstempel hinzugefügt werden. In diesem Beispiel ist das jedoch nicht der Fall, hier interessiert nur, *ob* ein Ereignis eingetreten ist.

7.4.3.5. Anforderungen an das Speichermodell

Conditions ist zwar unabhängig von der Art der Parallelisierung, es sind allerdings einige zusätzliche Anforderungen zu beachten im Falle von Multiprozessormaschinen mit gemeinsamen Datenbereichen. Wenn mehrere leichtgewichtige Prozesse gleichzeitig auf einen gemeinsamen Datenbereich zugreifen, ist das zugrundeliegende Speichermodell von entscheidender Bedeutung.

Im Fall von *Conditions* ist es denkbar, daß die Testprozedur für eine Bedingung auf einen gemeinsamen Datenbereich zugreift, um zu überprüfen, ob ein von einem anderen leichtgewichtigen Prozeß erwartetes Ereignis eingetreten ist. Entsprechend muß das Speichermodell die Antwort darauf geben, wovon ein leichtgewichtiger Prozeß P_1 ausgehen kann, wenn ein anderer leichtgewichtiger Prozeß P_2 eine Reihe von Zuweisungen durchgeführt hat.

[1] Der einzige Unterschied besteht darin, daß die obige Variante in Oberon *Conditions.Test* nicht mehr aufruft, sondern die von *Conditions.WaitForAndSelect* gelieferte Menge überprüft.

Die Betrachtung sollte dabei natürlich auf die Basistypen beschränkt werden, die im Rahmen eines Buszyklus mit dem Speicher ausgetauscht werden können. Da Bussysteme normalerweise nur auf bestimmten Speicherkanten arbeiten (z.B. 4-Byte-Kanten bei 32-Bit-Bussystemen), sollte weiter angenommen werden, daß Variablen, die größer als ein Byte sind, vom Übersetzer auf Wortkanten allokiert werden. In Oberon kann man **INTEGER**, **SHORTINT**, **BOOLEAN**, **CHAR**, **BYTE**, **SET** und Zeiger mit Sicherheit zu den Typen zählen, die sich im Rahmen eines Buszyklus übertragen lassen.

Unter diesen Voraussetzungen ist sichergestellt, daß wir beim Laden einer Variablen immer einen konsistenten Wert vorfinden, den die Variable entweder gerade hat oder kurz zuvor gehabt hat. Allerdings ist damit noch nicht klargelegt, in welcher Reihenfolge Speicheränderungen für andere leichtgewichtige Prozesse sichtbar werden.

Viele Prozessoren verwalten einen lokalen Cache, um die Anzahl der Zugriffe über das Bussystem zum Speicher zu verringern. Bei Multiprozessorsystemen würde man natürlich nur ungern auf diesen Vorteil verzichten. Ein Cache lebt insbesondere von dem Vorteil, daß Speicheroperationen nicht notwendigerweise in der Reihenfolge weitergereicht werden, wie sie ursprünglich ausgeführt worden sind. Diese Technik ist bei einem Einprozessor-System nicht zu bemerken, wohl aber bei einem Multiprozessorsystem, wie folgendes Szenario zeigt:

Schritt	Speicher		P_1	*Cache*$_1$		P_2	*Cache*$_2$	
	a	b		a	b		a	b
1	0	0						
2	0	0	STORE a := 1					
3	0	0		1				
4	0	0	STORE b := 1	1				
5	0	0		1	1			
6	0	0		1	STORE			
7	0	1		1	1			
8	0	1		1	1	LOAD b		
9	0	1		1	1			LOAD
10	0	1		1	1			1
11	0	1		1	1	LOAD a		1
12	0	1		1	1		LOAD	1
13	0	1		1	1		0	1

In den Schritten 2 und 4 wird von P_1 zunächst die Variable a und danach b auf 1 gesetzt. Der Cache von P_1 nimmt sich jedoch die Freiheit, zuerst b in Schritt 6 abzuspeichern. Dies führt dazu, daß P_2 in den Schritten 10 und 13 zu $a = 0$ und $b = 1$ kommt, und dies, obwohl in der umgekehrten Reihenfolge geladen wird.

Natürlich bieten entsprechende Multiprozessorsysteme Instruktionen an, die eine bestimmte Reihenfolge erzwingen können. Dennoch wird dabei offenbar, wie schwierig es ist, bei solchen Systemen die Korrektheit nachzuweisen. Sollten Bedingungen im Rahmen eines Multiprozessorsystems bewertet werden, die sich auf gemeinsame Speicherflächen beziehen, dann ist es zwingend notwendig, daß Zuweisungen in der Ausführungsreihenfolge von anderen Prozessoren zu beobachten sind. Die beiden Modelle nennen sich TSO (*total store ordering*) und PSO (*partial store ordering*). Eine formale Definition für beide Modelle findet sich in [SPARC International, Inc.92].

7.4.3.6. Die Klassifizierung von Bedingungen und Ereignissen

Während die Verzögerung und das Warten generell in der obigen Weise modelliert werden können, gibt es bei Bedingungen weitere Varianten. Bei dem Beispiel mit dem Automaten sind alle vorkommenden Ereignisse in Zusammenhang mit dem Automaten definiert und nur für ihn und einen einzelnen Benutzer sinnvoll. Man kann entsprechend von lokalen Ereignissen reden, wenn nie mehr als zwei Prozesse daran beteiligt sind.

Darüber hinaus gibt es auch Ereignisse, die für eine Gruppe oder global von Interesse sind. Bei Gruppen ist es notwendig, daß vorher eine Registrierung stattfindet. Ohne Registrierung kann eine Partei später nicht feststellen, ob ein Ereignis geschehen ist oder nicht. Bei globalen Ereignissen ist das jederzeit ohne Registrierung möglich. Ein Beispiel für gruppenbezogene Ereignisse sind die Ereignisse im Sinne der Ulmer Oberon-Bibliothek (*Events*). Zu den globalen Ereignissen gehört beispielsweise das Erreichen einer bestimmten Uhrzeit.

Es gibt gruppenbezogene Ereignisse, die in konkurrierender Weise analog zum $|||$-Operator in CSP behandelt werden, d.h. wenn ein Ereignis eintritt, dann wird es innerhalb der Gruppe nur einem der Interessenten zuteil. Beispiele hierfür sind Semaphore (nur einer erhält die freiwerdende Ressource) oder eingehende Aufträge mit einer Vielzahl von Auftragnehmern, wobei jeder Auftrag natürlich genau einen Auftragnehmer erhalten soll.

Überraschenderweise bleiben von all diesen Varianten der Warte-Algorithmus und alle darauf aufbauenden Mechanismen unberührt. Nur die Benutzung von Bedingungen auf der anwendenden Seite kann teilweise unterschiedlich aussehen. So ist es bei einigen gruppenbezogenen Ereignissen (z.B. bei *EventConditions*) notwendig, explizit zu erklären, daß man nicht länger daran interessiert ist.

Es ist hier wichtig anzumerken, daß Bedingungen als Datenstruktur immer lokal benutzt und kreiert werden. Nur die Ereignisse, die dahinter stehen, sind ggf. mehreren Parteien zugänglich.

7.4.3.7. Globale Ereignisse

Im Gegensatz zu der vorher gezeigten Bedingung, die zwischen dem alten und dem erneuten Auftreten eines Ereignisses unterscheiden muß, interessiert bei globalen Ereignissen nur, ob sie überhaupt einmal stattgefunden haben. Somit ändert sich der Funktionswert der Bedingung nicht mehr, nachdem er einmal wahr geworden ist. Entsprechend wird auch *accept* nicht mehr benötigt:

$$\begin{aligned} GlobalCondition(in, test, testout) = (&\mu X.\,(in?x \rightarrow SKIP \mid test \rightarrow testout!FALSE \rightarrow X); \\ &\mu Y.\,(test \rightarrow testout!TRUE \rightarrow Y)) \end{aligned}$$

Ein Beispiel dafür in der Bibliothek ist das Modul *TimeConditions*. Es erlaubt, auf das Eintreten einer bestimmten Uhrzeit zu warten.

7.4.3.8. Kooperative gruppenbezogene Ereignisse

Bei kooperativen gruppenbezogenen Ereignissen bleibt *Condition* unverändert im Vergleich zu lokalen Ereignissen. Es ist nur notwendig, entsprechend der Registrierungen einen Verteiler davor einzubauen:

$$Broadcast(in, out_1, \ldots, out_n) = \mu X.\,(in?x \rightarrow out_1!x \rightarrow \cdots \rightarrow out_n!x \rightarrow X)$$

Ein Beispiel hierfür ist *EventConditions*.

7.4.3.9. Konkurrierende Bedingungen

Das klassische Beispiel hierfür sind Semaphore, die sich in CSP leicht formulieren lassen:

$$Resource(sema) = \mu X.\,(sema.P \rightarrow sema.V \rightarrow X)$$

wobei in CSP die konkurrierenden Prozesse mit dem $|||$-Operator und nicht dem $\|$-Operator zu verknüpfen sind. Auf dieser Basis läßt sich dann die Bedingung formulieren:

$$\begin{aligned} &ResourceCondition(sema, test, testout, release) = \\ &\quad \mu X.\,(sema.P \rightarrow SKIP \mid test \rightarrow testout!FALSE \rightarrow X); \\ &\quad \mu Y.\,(release \rightarrow sema.V \rightarrow SKIP \mid test \rightarrow testout!TRUE \rightarrow Y) \end{aligned}$$

Im Unterschied zu den anderen Klassen von Bedingungen nimmt diese sofort nach ihrer Kreierung auf das Ereignis bzw. die Ressource selbst Einfluß, indem sie versucht, die Ressource zu ergattern. Die Bedingung hält dann die Ressource fest und muß nachher explizit aufgefordert werden, die Ressource wieder freizugeben. Darüber hinaus läßt sich diese Bedingung nicht wiederholt verwenden, da sie ansonsten nach dem Freigeben sofort wieder versuchen würde, auf die Ressource zuzugreifen.

Bei n registrierten Interessenten ergibt dies dann:

$$\begin{gathered} Resource(sema)\| \\ (ResourceCondition_1(sema, test_1, testout_1, release_1) \;|||\cdots|||\; ResourceCondition_n(sema, test_n, testout_n, release_n)) \end{gathered}$$

Ein Beispiel hierfür ist *LocalSemaphores*.

7.5. Tasks und Schedulers

Conditions bietet nur die Abstraktion für Bedingungen an und einen Warte-Algorithmus, der den gesamten Prozeß blockiert, bis eines der externen Ereignisse eintrifft und die zugehörige Bedingung wahr wird. Damit fehlt noch ein Kooperations- und Synchronisierungsmechanismus für die konkurrierenden Einheiten innerhalb eines Prozesses, den Koroutinen.

Tasks definiert die von der Verwaltung erfaßten konkurrierenden Einheiten — die Tasks — und eine generelle Schnittstelle für verwaltende Module — die Scheduler. Zu jeder Task gehört eine Koroutine, die mehrere andere Koroutinen kreieren und/oder benutzen kann, die nicht der Verwaltung von *Tasks* unterstehen. So können einfachere Mechanismen wie z.B. *Iterators* ohne Verwendung dieser Abstraktionen direkt unter der Verwendung von **CRSPAWN** und **CRSWITCH** geschrieben werden.

Tasks werden zu Taskgruppen zusammengefaßt. Zu jeder Taskgruppe gibt es genau einen Scheduler, der selbst nicht zu der Taskgruppe gehört. Dies führt zu einer Hierarchie, an deren Spitze eine Taskgruppe steht, die nur aus der Task der ersten Koroutine (*Coroutines.main*) besteht (siehe Abbildung 38). Der Scheduler dieser Taskgruppe ist der einzige, der von *Tasks* selbst implementiert ist.

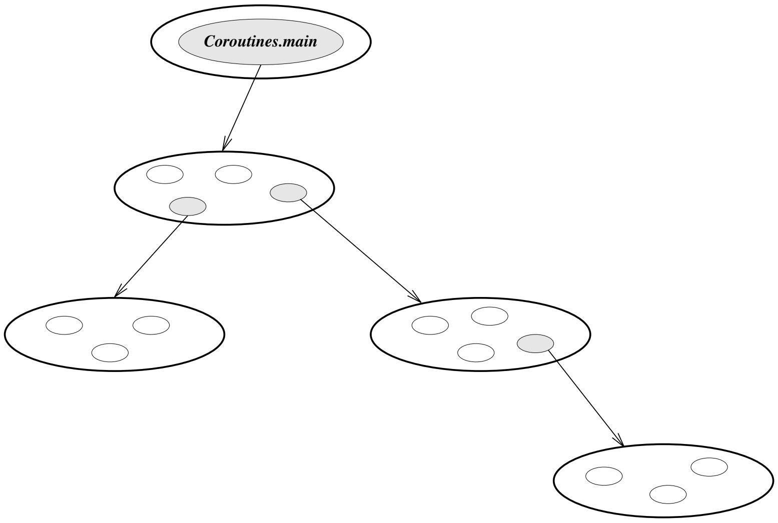

Abb. 38: Beispiel für eine Taskhierarchie

Tasks offeriert folgende Operationen für eine Task, die sich alle auf die aufrufende Task beziehen:

```
PROCEDURE WaitForOneOf(conditionSet: Conditions.ConditionSet);
PROCEDURE WaitFor(condition: Condition.Condition);
PROCEDURE Suspend;
PROCEDURE Terminate;
```

Statt *Conditions.WaitFor* ist aus der Sicht einer Task entweder *Tasks.WaitForOneOf* oder *Tasks.WaitFor* aufzurufen. Dies ermöglicht das Blockieren einer Task, während andere Tasks weiter ausgeführt werden können. *Tasks.Suspend* suspendiert den Aufrufer, ohne ihn zu blockieren. Da sich Tasks nur freiwillig suspendieren können, ist damit eine Möglichkeit gegeben, auch andere Tasks zur Ausführung gelangen zu lassen, selbst wenn auf kein Ereignis gewartet wird.

Aus der Sicht eines Schedulers gibt es folgende Operationen:

```
CONST
   (* Werte von code, die Dispatch zurueckliefert: *)
   terminate = 0; suspend = 1; waitfor = 2;

PROCEDURE Init(task: Task; cr: Coroutines.Coroutine; tg: TaskGroup);
PROCEDURE InitGroup(tg: TaskGroup);
PROCEDURE Dispatch(tg: TaskGroup; task: Task;
                   VAR code: SHORTINT;
                   VAR conditionSet: Conditions.ConditionSet);
```

Neue Taskgruppen und Tasks werden vom jeweiligen Scheduler erzeugt und dann mit *Init* und *InitGroup* dem Modul *Tasks* bekannt gegeben. Jeder Scheduler muß selbst eine Datenstruktur unterhalten, die ihm einen

Überblick darüber verschafft, welche Tasks unter seiner Verwaltung stehen. Es liegt dann in der Freiheit des Schedulers, eine der rechenbereiten Tasks mit *Dispatch* zur Ausführung zu bringen.

Unterscheidet man die drei möglichen Werte, die *Dispatch* in *code* zurückliefert, und zählt noch den Fall hinzu, daß eine Task gerade läuft, so erhält man vier mögliche Zustände, die in Abbildung 39 dargestellt sind. Durch eine vom Scheduler abhängige Operation können neue Tasks zu einer Taskgruppe hinzukommen und erwarten dann ihre Ausführung (*ready*). Ein Scheduler wählt aus allen Tasks, die zur weiteren Ausführung bereit sind, eine aus und bringt sie per *Tasks.Dispatch* zur Ausführung. Eine Task kann mit einem Aufruf von *Tasks.WaitFor*, *Tasks.Suspend* oder *Tasks.Terminate* die Kontrolle wieder an den Scheduler zurückgeben.

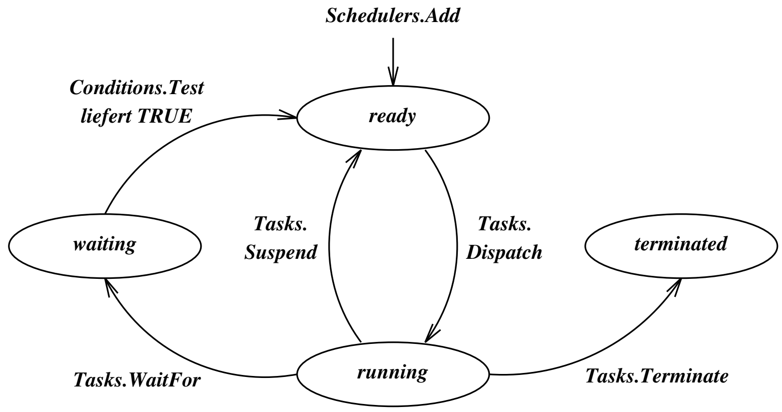

Abb. 39: Zustandsübergänge für Tasks

Auf dieser Basis kann ein Rahmen für einen Scheduler geschaffen werden:

```
PROCEDURE Scheduler(tg: Tasks.TaskGroup;
                    tasks: SET OF Tasks.Task);
   VAR
      ready, waiting: SET OF Tasks.Task;
      t: Tasks.Task;
      cset: Conditions.ConditionSet;
      code: SHORTINT;
BEGIN
   ready := tasks; waiting := ∅;
   WHILE tasks ≠ ∅ DO
      (* Invariante: tasks = ready ∪ waiting
            ready ∩ waiting = ∅
            ∀t∈waiting ist cset_t
            die Menge der zugehoerigen Wartebedingungen
      *)
      IF waiting ≠ ∅ THEN
         FOR ALL t ∈ waiting DO
            IF Conditions.TrueConds(cset_t) ≠ ∅ THEN
               waiting := waiting \ { t };
               ready := ready ∪ { t };
            END;
         END;
      END;
      IF ready = ∅ THEN
         cset :=    ⋃        cset_t;
                 t∈waiting
         Tasks.WaitForOneOf(cset);
      ELSE
         (* ggf. Tasks.Suspend aus Fairness *)
         (* Auswahl von t ∈ ready *)
         ready := ready \ { t };
         Tasks.Dispatch(tg, t, code, cset_t);
         CASE code OF
         | Tasks.terminate:   tasks := tasks \ { t };
         | Tasks.suspend:     ready := ready ∪ { t };
         | Tasks.waitfor:     waiting := waiting ∪ { t };
         END;
      END;
```

```
      END;
   END Scheduler;
```

Schedulers bietet genau diesen Rahmen an, der noch um die Möglichkeit ergänzt worden ist, Tasks jederzeit in eine Taskgruppe aufzunehmen. Einzig frei bleibt die Auswahl einer der rechenbereiten Tasks aus *ready*. Denkbare Implementierungen hierfür wären *round robin* oder Algorithmen, die auf Prioritäten basieren. Ersteres wird von *RoundRobin* zur Verfügung gestellt — für auf Prioritäten basierende Scheduler gab es (im Rahmen dieses Projekts) bislang noch keinen Bedarf.

7.6. Beispiele und Vergleiche

7.6.1. Die Kriterien von Bloom

Bloom veröffentlichte einen Kriterienkatalog für Synchronisierungsmechanismen, der es erlaubt, Mechanismen zu beurteilen und miteinander zu vergleichen [Bloom79]. Bloom betrachtet dabei drei Bereiche:

- Modularität: Ressourcen sollten abstrakte Datentypen sein. Entsprechend sollte die Implementierung des Synchronisierungsschemas zur Implementierung der Abstraktion gehören, und es sollte nicht notwendig sein, Zugriffsoperationen auf eine Ressource explizit in Synchronisierungsanweisungen einzuklammern.
- Ausdruckskraft: Es sollte problemlos sein, Prioritäten und Ausschlußvorschriften in Abhängigkeit von der Zugriffsoperation und der zugehörigen Parameter, dem Zeitpunkt bzw. der Reihenfolge der Aufrufe, dem Synchronisierungszustand der Ressource, dem lokalen Zustand der Ressource und der Vorgeschichte zu geben.
- Leichte Benutzbarkeit: Es sollte leicht möglich sein, individuelle Ausschlußvorschriften miteinander zu kombinieren.

Im folgenden werden einige Beispiele vorgestellt, die zeigen, daß Bedingungen allen Anforderungen von Bloom gerecht werden können, wenngleich teilweise ein gewisser Programmieraufwand erforderlich ist. Allerdings besteht immer die Möglichkeit, für häufig auftretende Varianten einen Rahmen zu entwickeln und als Modul zur Verfügung zu stellen.

7.6.2. Ringpuffer

Die Forderung der Modularität läßt sich natürlich von Oberon aus relativ leicht erfüllen. Die Schnittstelle für einen Ringpuffer könnte folgendermaßen aussehen:

```
DEFINITION RingBuffers;

   IMPORT Disciplines, Objects;

   TYPE
      RingBuffer = POINTER TO RingBufferRec;
      RingBufferRec = RECORD (Disciplines.ObjectRec) END;

   PROCEDURE Create(VAR ring: RingBuffer);

   PROCEDURE Put(ring: RingBuffer; object: Objects.Object);
   PROCEDURE Get(ring: RingBuffer; VAR object: Objects.Object);

END RingBuffers.
```

Natürlich wäre es möglich, zunächst eine allgemeine Abstraktion für Ringpuffer zu definieren (z.B. auf Basis von *Collections*). Hierauf und auf die Möglichkeit zur Spezifikation der Größe des Puffers wurde aus Gründen der Einfachheit verzichtet.

Ein Ringpuffer ist ein Beispiel dafür, daß ein paralleler Zugriff von *Put* und *Get* möglich ist, wenn der Puffer momentan weder ganz leer noch ganz gefüllt ist. Als Basis zur Synchronisierung empfehlen sich die Semaphore — sowohl ein Semaphor für *Put* als auch eines für *Get*. *Semaphores* offeriert eine allgemeine Schnittstelle für Semaphore, und *LocalSemaphores* stellt eine spezielle Variante für lokale Semaphore zur Verfügung. Abgesehen von der Schnittstelle für Implementierungen stellt *Semaphores* folgende Operationen zur Verfügung:

```
PROCEDURE P(sema: Semaphore);
PROCEDURE V(sema: Semaphore);
PROCEDURE GetCondition(sema: Semaphore; VAR condition: Conditions.Condition);
PROCEDURE Status(sema: Semaphore) : INTEGER;
```

Dabei wird *P* auf Basis von *GetCondition* realisiert:

```
PROCEDURE P(sema: Semaphore);
   VAR condition: Conditions.Condition;
BEGIN
   GetCondition(sema, condition);
   Tasks.WaitFor(condition);
END P;
```

Auf dieser Basis läßt sich die interne Datenstruktur für den Ringpuffer definieren:

```
CONST
   bufsize = 32;
TYPE
   RingBuffer = POINTER TO RingBufferRec;
   RingBufferRec =
      RECORD
         (Disciplines.ObjectRec)
         buffer: ARRAY bufsize OF Objects.Object;
         readIndex, writeIndex: INTEGER;
         put, get: Semaphores.Semaphore;
         putCond, getCond: Condition;
      END;
```

Die Semaphore *put* und *get* sorgen dafür, daß immer nur maximal ein *Put* und nur maximal ein *Get* aktiv ist. Dies reicht allerdings nicht aus. Wenn beispielsweise der Puffer voll ist, dann muß *Put* solange warten, bis mindestens ein *Get* ausgeführt wird. Hierfür werden die lokalen Bedingungen *putCond* und *getCond* benötigt:

```
CONST
   put = 0; get = 1;
TYPE
   Condition = POINTER TO ConditionRec;
   ConditionRec =
      RECORD
         (Conditions.ConditionRec)
         op: SHORTINT; (* put oder get *)
         ring: RingBuffer;
      END;
VAR
   condDomain: Conditions.Domain;

PROCEDURE TestCondition(domain: Conditions.Domain;
                        condition: Conditions.Condition;
                        errors: RelatedEvents.Object) : BOOLEAN;
   VAR
      ring: RingBuffer;
BEGIN
   WITH condition: Condition DO
      ring := condition.ring;
      RETURN (condition.op = put) &
                ((ring.writeIndex + 1) MOD bufsize # ring.readIndex) OR
             (condition.op = get) &
                (ring.readIndex # ring.writeIndex)
   END;
END TestCondition;

PROCEDURE InitDomain;
   VAR
      if: Conditions.Interface;
      desc: Conditions.Description;
BEGIN
   NEW(if); if.test := TestCondition;
   NEW(desc); desc.caps := {}; desc.internal := TRUE;
   NEW(condDomain);
```

```
      Conditions.InitDomain(condDomain, if, desc);
   END InitDomain;

   PROCEDURE Create(VAR ring: RingBuffer);

      PROCEDURE InitCondition(VAR condition: Condition; op: SHORTINT);
      BEGIN
         NEW(condition);
         condition.op := op; condition.ring := ring;
         Conditions.Init(condition, condDomain);
      END InitCondition;

   BEGIN
      NEW(ring);
      ring.readIndex := 0; ring.writeIndex := 0;
      LocalSemaphores.Create(ring.put);
      LocalSemaphores.Create(ring.get);
      InitCondition(ring.putCond, put);
      InitCondition(ring.getCond, get);
   END Create;

   PROCEDURE Put(ring: RingBuffer; object: Objects.Object);
   BEGIN
      Semaphores.P(ring.put);
      Tasks.WaitFor(ring.putCond);
      ring.buffer[ring.writeIndex] := object;
      ring.writeIndex := (ring.writeIndex + 1) MOD bufsize;
      Semaphores.V(ring.put);
   END Put;

   PROCEDURE Get(ring: RingBuffer; VAR object: Objects.Object);
   BEGIN
      Semaphores.P(ring.get);
      Tasks.WaitFor(ring.getCond);
      object := ring.buffer[ring.readIndex];
      ring.readIndex := (ring.readIndex + 1) MOD bufsize;
      Semaphores.V(ring.get);
   END Get;
```

Obwohl es auf den ersten Blick riskant erscheinen mag, kann man *Put* und *Get* durchaus parallel agieren lassen. Nur *Put* modifiziert *writeIndex* und nur *Get* modifiziert *readIndex*. Beide ziehen die Indizes nur nach, nachdem die entsprechende Aktion am Ringpuffer durchgeführt worden ist. Wenn der Index der anderen Partei mit dem eigenen Index verglichen wird, ist es denkbar, daß der andere Index mittlerweile weiter gesetzt worden ist. Somit kann der entsprechende Test eventuell zu hart ausfallen, d.h. es kommt zur unnötigen Blockade. Es kann jedoch nicht passieren, daß der Test versehentlich zu günstig ausfällt.

Natürlich wird hier im Falle von Multiprozessorsystemen das TSO-Modell vorausgesetzt, ansonsten würde folgende Gefahr bestehen:

Task T_1	Task T_2
führt *Put* aus:	führt *Get* aus:
(* writeIndex = 0; readIndex = 0 *)	
	Tasks.WaitFor(ring.getCond);

ring.buffer[writeIndex] := object;
writeIndex := (ring.writeIndex + 1) MOD bufsize;
(* der Cache von T_1 sichert *writeIndex*,
jedoch nicht *ring.buffer*[*writeIndex*]
*)	

	(* writeIndex = 1; readIndex = 0 *) object := ring.buffer[readIndex]; (* *object* ist hier undefiniert *)

Das Beispiel entspricht dem parallelen Puffer aus [Gehani93, §VI.C]. Die Synchronisierungsbedingungen

```
accept put(char c) suchthat ((in + 1) % max != out);
accept get(void) suchthat (in != out);
```

finden sich analog in *TestCondition* wieder.[1] Die Parallelisierung, die in Concurrent C++ mit

```
{put(char), get(void)};
```

angegeben worden ist, wurde hier über Semaphore geregelt.

Die Verwendung der Semaphore ist hier relativ ungefährlich, da sie zu der privaten Datenstruktur des Moduls gehören und nicht von außen zugänglich sind. Somit läßt sich leicht nachprüfen, ob die *P*- und *V*-Operationen richtig balanciert sind.

7.6.3. Die speisenden Philosophen

Dijkstras Problem der speisende Philosophen gehört zu den klassischen Beispielen [Dijkstra71]. Fünf Philosophen verbringen ihr Leben damit, Spaghetti zu essen und nachzudenken. Gegessen wird an einem runden Tisch mit fünf Stühlen. Der Stuhl mit der Nummer i ist dem Philosophen i fest zugeordnet ($0 \le i \le 4$). Die Philosophen sind sehr sparsam und haben sich nur fünf Gabeln angeschafft. Die Gabel i liegt auf der linken Seite des Stuhls i. Betritt ein Philosoph den Speiseraum und setzt sich an seinen Stuhl, dann benötigt er die beiden Gabeln zu seiner linken und rechten Seite, um essen zu können. Wenn die entsprechenden Gabeln teilweise oder ganz im Besitz seiner Nachbarn sind, muß er warten, bis sie wieder frei werden. Wenn der Philosoph die Gabeln erhalten hat, ißt er eine Weile, legt anschließend die Gabeln wieder an die ursprüngliche Position und verläßt den Speiseraum, um weiter nachzudenken.

Als Ressource sind hier die Gabeln zu verwalten:

```
DEFINITION Forks;

   IMPORT Disciplines;

   CONST
      phils = 5; (* die Anzahl der Gabeln und der Philosophen *)
   TYPE
      Philosopher = SHORTINT; (* 0..phils-1 *)

   TYPE
      Forks = POINTER TO ForksRec;
      ForksRec = RECORD (Disciplines.ObjectRec) END;

   PROCEDURE Create(VAR forks: Forks);
   PROCEDURE Pickup(forks: Forks; phil: Philosopher);
   PROCEDURE Putdown(forks: Forks; phil: Philosopher);

END Forks.
```

Die Datenstruktur ist relativ einfach: für jede Gabel gibt es ein Semaphor. Wer den Zugriff auf das Semaphor hat, hat in dem Augenblick auch die Gabel in Händen:

```
CONST
   used = 0; free = 1;
TYPE
   Forks = POINTER TO ForksRec;
   ForksRec =
      RECORD
         (Disciplines.ObjectRec)
         fork: ARRAY phils OF Semaphores.Semaphore;
         cond: ARRAY phils OF Condition;
      END;
```

Zur leichteren Arbeitsweise ist es recht praktisch, Felder mit den Indizes für die benachbarten Gabeln zu unterhalten:

[1] Statt *in* und *out* wurden hier *writeIndex* und *readIndex* verwendet. *max* entspricht *bufsize*.

```
VAR
   left, right: ARRAY phils OF Philosopher;

PROCEDURE Init;
   VAR
      index: Philosopher;
BEGIN
   index := 0;
   WHILE index < phils DO
      left[index] := index;
      right[index] := (index + 1) MOD phils;
      INC(index);
   END;
END Init;
```

Möchte ein Philosoph speisen, so versucht er die linke und die rechte Gabel zu erhalten. Gelingt dies nicht sofort, werden alle Semaphore sofort wieder freigegeben, damit nicht nur ein Deadlock verhindert wird, sondern auch möglichst hohe Parallelisierung erreicht wird. Wichtig ist, daß die Testprozedur nicht blockiert. Somit darf nicht etwa *Semaphores.P* aufgerufen werden. Statt dessen wird die entsprechende Bedingung verwendet und getestet.

```
TYPE
   Condition = POINTER TO ConditionRec;
   ConditionRec =
      RECORD
         (Conditions.ConditionRec)
         forks: Forks;
         phil: Philosopher;
      END;
VAR
   condDomain: Conditions.Domain;

PROCEDURE Release(forks: Forks; phil: Philosopher);
   (* gib alle zugehoerigen Semaphore wieder frei *)
BEGIN
   Semaphores.V(forks.fork[left[phil]]);
   Semaphores.V(forks.fork[right[phil]]);
END Release;

PROCEDURE TestCondition(domain: Conditions.Domain;
                        condition: Conditions.Condition;
                        errors: RelatedEvents.Object) : BOOLEAN;
   VAR
      semaCond: Conditions.Condition;
      forks: Forks;
      phil: Philosopher;
BEGIN
   WITH condition: Condition DO
      forks := condition.forks; phil := condition.phil;
      Semaphores.GetCondition(forks.fork[left[phil]], semaCond);
      IF ~Conditions.Test(semaCond, errors) THEN
         Semaphores.V(forks.fork[left[phil]]); RETURN FALSE
      END;
      Semaphores.GetCondition(forks.fork[right[phil]], semaCond);
      IF ~Conditions.Test(semaCond, errors) THEN
         Release(forks, phil);
         RETURN FALSE
      END;
      RETURN TRUE
   END;
END TestCondition;

PROCEDURE InitDomain;
   VAR
      if: Conditions.Interface;
      desc: Conditions.Description;
BEGIN
   NEW(if); if.test := TestCondition;
```

```
      NEW(desc); desc.caps := {}; desc.internal := TRUE;
      NEW(condDomain);
      Conditions.InitDomain(condDomain, if, desc);
   END InitDomain;

   PROCEDURE Create(VAR forks: Forks);
      VAR
         index: Philosopher;
         condition: Condition;
   BEGIN
      NEW(forks);
      index := 0;
      WHILE index < phils DO
         LocalSemaphores.Create(forks.fork[index]);
         NEW(condition);
         condition.forks := forks; condition.phil := index;
         Conditions.Init(condition, condDomain);
         forks.cond[index] := condition;
         INC(index);
      END;
   END Create;

   PROCEDURE Pickup(forks: Forks; phil: Philosopher);
   BEGIN
      Tasks.WaitFor(forks.cond[phil]);
   END Pickup;

   PROCEDURE Putdown(forks: Forks; phil: Philosopher);
   BEGIN
      Release(forks, phil);
   END Putdown;
```

Dieses Beispiel erlaubt eine weitrechende Parallelisierung. So können beispielsweise die Philosophen 0 und 2 gleichzeitig loslegen, ohne sich irgendwie zu behindern, da Philosoph 0 nur auf fork[0] und fork[1] zugreift, während Philosoph 2 auf fork[2] und fork[3] angewiesen ist.

Im Gegensatz zu Concurrent C++ erfolgt die Festlegung über die Art der Parallelisierung nicht in der nach außen sichtbaren Spezifikation, sondern innerhalb der Implementierung. Dies empfiehlt sich sowohl entsprechend der Forderung Blooms nach der Modularität und der Kapselung des Synchronisierungsmechanismus als auch unter dem Gesichtspunkt, daß die Art der Parallelisierung sehr eng mit der Implementierung der Operationen verknüpft ist.

In Concurrent C++ läßt sich die Art der Parallelisierung bei einer abgeleiteten Kapsel verändern, wodurch die Gefahr entsteht, daß die Art der Parallelisierung nicht mehr mit der privaten Implementierung der abgeleiteten Kapsel übereinstimmt. Genau dieses ist bei Gehanis Beispiel in [Gehani93, §VI.D] passiert. Zuerst gibt er die Schnittstelle in ähnlicher Form wie die obige Variante in Oberon an. Da hier zunächst noch keine Parallelisierung zugelassen ist, kann analog zu Hoares Monitoren nur eine Operation zugleich aktiv sein. Eine danach von ihm vorgestellte Ableitung dieser Kapsel gibt dann die intuitiv richtige Parallelisierung an, die auch das Beispiel in Oberon verwendet:

```
capsule pforks: public forks {
par:
   {pickup(int i)*, pickup(int j)}
   suchthat (((i + 1) % 5 != j) && ((j + 1) % 5 != i));
};
```

Hiermit wird ausgeschlossen, daß zwei benachbarte Philosophen gleichzeitig aktiv sind. Allerdings erlaubt die Datenstruktur der Implementierung der Basiskapsel nicht diese Parallelisierung. Gehanis Datenstruktur gibt für jeden Philosophen die erreichbaren Gabeln an. Deswegen muß ein Philosoph nicht nur zuerst überprüfen, ob er selbst zwei Gabeln hat, sondern sie auch bei seinen beiden Nachbarn abziehen:

```
void forks::pickup(int phil)
{
   fork[left[phil]]--;
   fork[right[phil]]--;
}
```

Entsprechend greift beispielsweise der Philosoph 0 auf *fork*[4] und *fork*[1] zu. Die Erweiterung erlaubt hier den gleichzeitigen Zugriff des Philosophen 2, der dann auf *fork*[1] und *fork*[3] zugreift. Damit ist ein potentieller Konflikt gegeben, der dazu führen kann, daß *fork*[1] auf 1 statt auf 0 gesetzt wird.

Hieran wird deutlich, daß es besser ist, die Art der Parallelisierung nicht in der Spezifikation, sondern in der Implementierung unterzubringen, obwohl Gehani die Offenlegung als einen der Hauptvorteile von Concurrent C++ betrachtet.

7.6.4. Paralleles Lesen und Schreiben

Ein weiteres klassisches Problem ist das der Parallelisierung von Lese- und Schreibzugriffen auf eine Ressource. Hierbei sind entweder mehrere lesende Zugriffe oder ein schreibender Zugriff zugelassen. Allerdings reichen diese Ausschlußvorschriften nicht aus. Wenn beispielsweise ständig Lesezugriffe erfolgen, so daß fast immer einige aktiv sind, könnte es passieren, daß Schreibzugriffe beliebig lange suspendiert werden. Von daher ist es notwendig, Prioritäten zu verwenden, um das ‘‘Verhungern’’ von Tasks zu verhindern.

Die Schnittstelle für dieses Problem könnte folgendermaßen aussehen:

```
DEFINITION RWResources;

   IMPORT Disciplines;

   TYPE
      RWResource = POINTER TO RWResourceRec;
      RWResourceRec = RECORD (Disciplines.ObjectRec) END;

   PROCEDURE Create(VAR rw: RWResource);

   PROCEDURE ReadP(rw: RWResource);
   PROCEDURE ReadV(rw: RWResource);

   PROCEDURE WriteP(rw: RWResource);
   PROCEDURE WriteV(rw: RWResource);

END RWResources.
```

Das würde erlauben, Lese- und Schreibzugriffe in anderen Modulen entsprechend einzuklammern. Wenn dies nur für private Daten durchgeführt wird, ist es wiederum auch kein Problem, festzustellen, ob *ReadP* und *ReadV* bzw. *WriteP* und *WriteV* richtig balanciert sind.

Die Implementierung von *RWResources* muß für die wartenden Lese- und Schreibzugriffe Warteschlangen unterhalten. Dies läßt sich am leichtesten über verkettete Bedingungen erreichen, wobei jede Bedingung eine wartende Partei repräsentiert:

```
TYPE
   Condition = POINTER TO ConditionRec;
   ConditionRec =
      RECORD
         (Conditions.ConditionRec)
         access: BOOLEAN; (* Zugriff freigegeben? *)
         next: Condition;
      END;
VAR
   condDomain: Conditions.Domain;
```

Im Gegensatz zum letzten Beispiel ist die genaue Bedingung für die Freigabe nicht in der Testprozedur für *Conditions* enthalten. Statt dessen wird *access* zunächst auf **FALSE** gesetzt und später auf **TRUE**, wenn die zugehörige Partei ihren Zugriff ausführen darf:

```
PROCEDURE TestCondition(domain: Conditions.Domain;
                        condition: Conditions.Condition;
                        errors: RelatedEvents.Object) : BOOLEAN;
BEGIN
   WITH condition: Condition DO
      RETURN condition.access
   END;
END TestCondition;
```

```
PROCEDURE InitDomain;
   VAR
      if: Conditions.Interface;
      desc: Conditions.Description;
BEGIN
   NEW(if); if.test := TestCondition;
   NEW(desc); desc.caps := {}; desc.internal := TRUE;
   NEW(condDomain);
   Conditions.InitDomain(condDomain, if, desc);
END InitDomain;
```

Bei der privaten Datenstruktur für *RWResource* werden neben der Anzahl der zur Zeit stattfindenen Lese- und Schreibzugriffe (*readers* und *writers*) die Warteschlangen unterhalten. Hinzu kommt ein Semaphor, das den Zugang zu der Datenstruktur regelt.

```
TYPE
   Queue =
      RECORD
         head, tail: Condition;
      END;
   RWResource = POINTER TO RWResourceRec;
   RWResourceRec =
      RECORD
         (Disciplines.ObjectRec)
         readers, writers: INTEGER; (* Anzahl *)
            (* Invariante:
                  ((readers = 0) OR (writers = 0)) & (writers <= 1)
            *)
         waitingReaders, waitingWriters: Queue;
         sema: Semaphores.Semaphore;
      END;
```

Auf dieser Basis können dann die Operationen erfolgen:

```
PROCEDURE CreateCondition(VAR condition: Condition;
                          VAR queue: Queue);
BEGIN
   NEW(condition); condition.access := FALSE; condition.next := NIL;
   Conditions.Init(condition, condDomain);
   IF queue.head = NIL THEN
      queue.head := condition;
   ELSE
      queue.tail.next := condition;
   END;
   queue.tail := condition;
END CreateCondition;

PROCEDURE CheckQueue(VAR queue: Queue; VAR counter: INTEGER);
   (* wecke den ersten aus der Warteschlange auf,
      sofern dort jemand wartet;
      das Aufwecken wird mit dem Erhoehen von
      readers bzw. writers verbunden
   *)
BEGIN
   IF queue.head # NIL THEN
      queue.head.access := TRUE;
      queue.head := queue.head.next;
      IF queue.head = NIL THEN
         queue.tail := NIL;
      END;
      INC(counter);
   END;
END CheckQueue;

PROCEDURE Create(VAR rw: RWResource);

   PROCEDURE InitQueue(VAR queue: Queue);
   BEGIN
      queue.head := NIL; queue.tail := NIL;
```

```
      END InitQueue;

   BEGIN
      NEW(rw);
      rw.readers := 0; rw.writers := 0;
      InitQueue(rw.waitingReaders); InitQueue(rw.waitingWriters);
      LocalSemaphores.Create(rw.sema);
   END Create;

   PROCEDURE ReadP(rw: RWResource);
      VAR
         condition: Condition;
   BEGIN
      Semaphores.P(rw.sema);
      IF (rw.writers > 0) OR (rw.waitingWriters.head # NIL) THEN
         CreateCondition(condition, rw.waitingReaders);
         Semaphores.V(rw.sema);
         Tasks.WaitFor(condition); (* setzt rw.readers hoch *)
      ELSE
         INC(rw.readers);
         Semaphores.V(rw.sema);
      END;
   END ReadP;

   PROCEDURE ReadV(rw: RWResource);
   BEGIN
      Semaphores.P(rw.sema);
      DEC(rw.readers);
      IF rw.readers = 0 THEN
         CheckQueue(rw.waitingWriters, rw.writers);
      END;
      Semaphores.V(rw.sema);
   END ReadV;

   PROCEDURE WriteP(rw: RWResource);
      VAR
         condition: Condition;
   BEGIN
      Semaphores.P(rw.sema);
      IF (rw.readers > 0) OR (rw.writers > 0) THEN
         CreateCondition(condition, rw.waitingWriters);
         Semaphores.V(rw.sema);
         Tasks.WaitFor(condition); (* setzt rw.writers hoch *)
      ELSE
         rw.writers := 1;
         Semaphores.V(rw.sema);
      END;
   END WriteP;

   PROCEDURE WriteV(rw: RWResource);
   BEGIN
      Semaphores.P(rw.sema);
      rw.writers := 0;
      IF rw.waitingReaders.head # NIL THEN
         CheckQueue(rw.waitingReaders, rw.readers);
      ELSE
         CheckQueue(rw.waitingWriters, rw.writers);
      END;
      Semaphores.V(rw.sema);
   END WriteV;
```

Die vorgestellte Lösungsvariante verhindert nicht nur ‘‘verhungernde’’ Schreiber, sondern auch ‘‘verhungernde’’ Leser, da *WriteV* beim Aufwecken den lesenden Parteien den Vorzug gibt. Wenn unter großer Last es dazu kommen sollte, daß es immer sowohl Parteien gibt, die lesen möchten, als auch Parteien, die schreiben wollen, dann kommt es zu einer Abwechslung zwischen Lese- und Schreibzugriffen, wobei parallele Lesezugriffe nicht mehr zugelassen werden.

Die Lösung zeigt, wie eine Reihe von verschiedenen Parametertypen in die Synchronisierungsbedingungen einfließen kann. Die Synchronisierung hängt hier ab von der Art des Zugriffs (*ReadP* oder *WriteP*), der Reihenfolge, in der die Zugriffswünsche geschehen, und dem Synchronisierungsstatus der Ressource. Analog ließen sich auch ggf. Parameter und Informationen aus der Vorgeschichte in die Bedingungen integrieren.

Das Beispiel zeigt ebenfalls, wie leicht die Zugriffsprioritäten geändert werden können: Wenn lesende Prozesse immer Vorrang haben sollen, dann bräuchte nur die Eingangsbedingung von *ReadP* vereinfacht werden:

```
(* Statt ... *)

IF (rw.writers > 0) OR (rw.waitingWriters.head # NIL) THEN

(* ... dann: *)

IF rw.writers > 0 THEN
```

Wenn schreibenden Prozessen immer Vorrang einzuräumen wäre, könnte *WriteV* folgendermaßen geändert werden:

```
(* Statt ... *)

IF rw.waitingReaders.head # NIL THEN
   CheckQueue(rw.waitingReaders, rw.readers);
ELSE
   CheckQueue(rw.waitingWriters, rw.writers);
END;

(* ... dann: *)

IF rw.waitingWriters.head # NIL THEN
   CheckQueue(rw.waitingWriters, rw.writers);
ELSE
   CheckQueue(rw.waitingReaders, rw.readers);
END;
```

Somit erfüllen Bedingungen alle Anforderungen von Bloom.

7.6.5. Rendezvous

7.6.5.1. Rendezvous in Ada

Rendezvous basieren auf der Prämisse, daß Synchronisierungen und Datenübertragungen nicht voneinander zu trennen sind. Die Kommunikation in CSP beruht darauf, und CSP hat seinerseits sehr das Rendezvous-Konzept in Ada beeinflußt [Ichbiah79, §11.4.2]. Der Sinn dieser Prämisse liegt in der leichteren Beweisführung. Wenn der Datenlieferant T_1 eine Vorbedingung P bezüglich seines Zustands zum Zeitpunkt der Datenübertragung erfüllt, dann kann der Datenempfänger T_2 zum Zeitpunkt des Empfangs immer noch von P ausgehen. Diese Art der Beweistechnik wurde schon beim Monitorkonzept von Hoare vorgeschlagen [Hoare74].

Als Beispiel für diese Problematik kann der Ringpuffer aus §7.6.2 herangezogen werden. So geht *Get* von der Vorbedingung $ring.readIndex \neq ring.writeIndex$ aus. Die Operation $Tasks.WaitFor(ring.getCond)$ stellt jedoch nur sicher, daß diese Vorbedingung kurz zuvor bestanden hat. Nur aus der genauen Kenntnis der anderen Operationen (im Beispiel nur *Put*) läßt sich schließen, daß diese Vorbedingung sich nicht unerwartet ändert. Würde eine Operation *Clear* hinzugefügt werden, die den Inhalt des Puffers eliminieren soll, dann könnte *Get* nicht mehr davon ausgehen, daß die Vorbedingung erhalten bleibt.

Wie der Name bereits nahelegt, findet ein Rendezvous immer zwischen genau zwei Tasks statt. Während in CSP beide Tasks über den Namen des Kanals zueinander fanden, sind Rendezvous in Ada asymmetrisch. Eine Task bietet hier einen Service (*entry*) an, der von anderen in Anspruch genommen werden kann. Der Service-Anbieter erfährt dabei nicht die Identität des Interessenten, während auf der anderen Seite die den Service ausnutzende Task den Service-Anbieter kennen muß. Ein weiterer Unterschied zwischen Ada und CSP besteht darin, daß beim Rendezvous ein Datenaustausch in beide Richtungen erfolgen kann, während Kanäle nur unidirektional sind.

Dies läßt sich an dem Automatenbeispiel demonstrieren. Während in CSP das Drücken einer Taste und das Erhalten des entsprechendes Getränks zwei Synchronisierungen hintereinander erforderlich machen, läßt sich dies in Ada in einem Zug erledigen:

```
task AUTOMAT is

   type SORTE is (KAFFEE, TEE);
   type GETRAENK is ...;

   entry MUENZE;
   entry ENTNAHME(WUNSCH : SORTE; AUSGABE : out GETRAENK);

end;

task body AUTOMAT is
begin
   loop
      accept MUENZE;
      accept ENTNAHME(WUNSCH : SORTE; AUSGABE : out GETRAENK) do
         case WUNSCH is
            when KAFFEE => AUSGABE := ...;
            when TEE    => AUSGABE := ...;
         end case;
      end ENTNAHME;
   end loop;
end;
```

Analog zu CSP ist es möglich, auf verschiedene Ereignisse gleichzeitig zu reagieren. Dabei kann die Akzeptanz von dem lokalen Zustand der Tasks abhängig gemacht werden und u.U. von der Anzahl der Tasks, die auf einen bestimmten Service warten. Das läßt sich an einem Beispiel für einen Ringpuffer aus [Ichbiah79, §11.2.5] demonstrieren:

```
task BUFFERING is
   entry READ (V : out ITEM);
   entry WRITE(E : in  ITEM);
end;

task body BUFFERING is
   SIZE      : constant INTEGER := 10;
   BUFFER    : array (1 .. SIZE) of ITEM;
   INX, OUTX : INTEGER range 1 .. SIZE := 1;
   COUNT     : INTEGER range 0 .. SIZE := 0;
begin
   loop
      select
         when COUNT < SIZE =>
            accept WRITE(E : in  ITEM) do
               BUFFER(INX) := E;
            end;
            INX := INX mod SIZE + 1;
            COUNT := COUNT + 1;
      or
         when COUNT > 0 =>
            accept READ (V : out ITEM) do
               V := BUFFER(OUTX);
            end;
            OUTX := OUTX mod SIZE + 1;
            COUNT := COUNT - 1;
      end select;
   end loop;
end BUFFERING;
```

Die Bedingungen vor den **accept**-Anweisungen (*guards*) werden zu Beginn der **select**-Anweisung alle bewertet. Nur die Bedingungen, die mit **TRUE** bewertet worden sind, kommen dann in die Auswahl einer der **accept**-Anweisungen.

7.6.5.2. Eine Schnittstelle für Rendezvous in Oberon

Wenn das Rendezvous-Konzept in Oberon realisiert werden soll, empfiehlt es sich, die Ein- und Ausgabeparameter in Aufträge zu verpacken. Verschiedene Sorten von Aufträgen werden in Oberon normalerweise mit Hilfe von Typentests unterschieden. Da eine Menge verschiedener Typen in Oberon nicht als

Wert repräsentierbar ist, empfiehlt es sich, ein zusätzliches Unterscheidungsmerkmal einzuführen. Normalerweise offeriert eine Task keine sehr große Anzahl verschiedener Dienstleistungen, so daß es möglich ist, den unterschiedlichen Angeboten Nummern zwischen **0** und **MAX(SET)** zuzuordnen. Auf diese Weise kann eine Menge von Auftragstypen als **SET** angegeben werden:

```
DEFINITION Rendezvous;

   IMPORT Objects, Tasks;

   TYPE
      Message = POINTER TO MessageRec;
      MessageRec = RECORD (Objects.ObjectRec) END;
      Entry = SHORTINT; (* 0..MAX(SET) *)
      EntrySet = SET;
      TaskBody = PROCEDURE;

   PROCEDURE Accept(entry: Entry; VAR message: Message);
   PROCEDURE AcceptEnd;
   PROCEDURE Select(entries: EntrySet; VAR entry: Entry);

   PROCEDURE Connect(task: Tasks.Task; entry: Entry; message: Message);
   PROCEDURE Initiate(body: TaskBody; VAR task: Tasks.Task);

END Rendezvous.
```

Das Entgegennehmen eines Auftrages erfolgt mit *Accept*, und mit *AcceptEnd* endet das Rendezvous. *Select* blockiert solange, bis ein Auftrag aus der gegebenen Menge vorliegt, und liefert einen dieser Aufträge zurück, der angenommen werden kann, ohne blockiert zu werden. Auf der anderen Seite kann ein Kunde mit *Connect* einen Auftrag aufgeben und wird dabei solange blockiert, bis der Partner des Rendezvous *AcceptEnd* aufruft.[1]

Auf dieser Basis ließe sich das Beispiel des Ringpuffers in Ada direkt in Oberon übertragen:

```
DEFINITION RingBuffers;

   IMPORT Disciplines, Objects;

   TYPE
      Item = Objects.Object;
      RingBuffer = POINTER TO RingBufferRec;
      RingBufferRec = RECORD (Disciplines.ObjectRec) END;

   PROCEDURE Create(VAR ring: RingBuffer);
   PROCEDURE Read(ring: RingBuffer; VAR v: Item);
   PROCEDURE Write(ring: RingBuffer; e: Item);

END RingBuffers.
```

Hierbei wird das Umsetzen von Prozeduraufrufen in Aufträge sinnvollerweise der Implementierung überlassen:

```
TYPE
   Item = Objects.Object;
   RingBuffer = POINTER TO RingBufferRec;
   RingBufferRec =
      RECORD
         (Disciplines.ObjectRec)
         task: Tasks.Task;
      END;

CONST
   read = 0; write = 1; (* entries *)
TYPE
   Message = POINTER TO MessageRec;
   MessageRec =
```

[1] Der Name *Connect* entspricht dem Schlüsselwort **connect**, das in einer Vorversion von Green zum Aufruf eines Services verwendet wurde. Dies stellte analog zu CSP syntaktisch heraus, daß eine Interprozeßkommunikation stattfindet. Später gab es in Ada keinen syntaktischen Unterschied mehr zwischen einem normalen Prozeduraufruf und der Benutzung eines Services.

```
        RECORD
           (Rendezvous.MessageRec)
           item: Item;
        END;

  PROCEDURE Read(ring: RingBuffer; VAR v: Item);
     VAR
        message: Message;
  BEGIN
     NEW(message);
     Rendezvous.Connect(ring.task, read, message);
     v := message.item;
  END Read;

  PROCEDURE Write(ring: RingBuffer; e: Item);
     VAR
        message: Message;
  BEGIN
     NEW(message);
     message.item := e;
     Rendezvous.Connect(ring.task, write, message);
  END Write;
```

Die Task in Ada wird durch eine normale Prozedur repräsentiert. Als einziges fällt hier auf, daß die Entscheidung über die Akzeptanz der beiden Service-Angebote *read* und *write* vor dem Aufruf von *Rendezvous.Select* erfolgt. Dies entspricht aber genau der Ausführungsreihenfolge in Ada.

```
  PROCEDURE Buffering;
     CONST
        size = 10;
     VAR
        buffer: ARRAY size OF Item;
        inx, outx: INTEGER;
        count: INTEGER;
        entries: Rendezvous.EntrySet;
        entry: Rendezvous.Entry;
        message: Message;
  BEGIN
     count := 0; inx := 0; outx := 0;
     LOOP
        entries := {};
        IF count < size THEN
           INCL(entries, write);
        END;
        IF count > 0 THEN
           INCL(entries, read);
        END;
        Rendezvous.Select(entries, entry);
        CASE entry OF
        | write:
              Rendezvous.Accept(write, message);
                 buffer[outx] := message.item;
              Rendezvous.AcceptEnd;
              outx := (outx + 1) MOD size;
              INC(count);
        | read:
              Rendezvous.Accept(read, message);
                 message.item := buffer[inx];
              Rendezvous.AcceptEnd;
              inx := (inx + 1) MOD size;
              DEC(count);
        END;
     END;
  END Buffering;
```

Im Gegensatz zu Ada können neue Tasks in Oberon jederzeit in jedem beliebigen Umfang (soweit der Speicher reicht) dynamisch erzeugt werden:[1]

[1] Ein Ausweg ergibt sich in Ada, wenn man auf in rekursive Prozeduren eingebettete Tasks zurückgreift.

```
PROCEDURE Create(VAR ring: RingBuffer);
BEGIN
   NEW(ring); Rendezvous.Initiate(Buffering, ring.task);
END Create;
```

7.6.5.3. Die Implementierung von Rendezvous in Oberon

Die Schnittstelle für *Rendezvous* sollte möglichst einfach und nahe an Ada angelehnt sein. Entsprechend wurde darauf verzichtet, daß die als Task agierende Prozedur bei *Select*, *Accept* und *AcceptEnd* einen sich selbst adressierenden Parameter angeben muß. Weiter liefert *Initiate* ein Ergebnis vom Typ *Tasks.Task* zurück, jedoch keinen Verweis auf eigene Datenstrukturen. Das daraus resultierende Problem bei der Implementierung läßt sich nur auf Basis von *Disciplines* lösen. Da *Tasks.Task* eine Erweiterung von *Disciplines.Object* ist, kann *Rendezvous* seine Datenstruktur als Disziplin bei der Task unterbringen und dort auch wiederfinden:

```
CONST
   maxentries = MAX(SET) + 1;
TYPE
   TaskDiscipline = POINTER TO TaskDisciplineRec;
   TaskDisciplineRec =
      RECORD
         (Disciplines.DisciplineRec)
         awaited: EntrySet;
         queues: ARRAY maxentries OF Queue;
         mutex: Semaphores.Semaphore;
         entry: Entry;
      END;
VAR
   taskdID: Disciplines.Identifier;
```

Die Komponente *queues* unterhält für alle möglichen Service-Angebote (*entries*) eine Warteschlange. Die Menge der Warteschlangen, die nicht leer sind, wird in *awaited* verzeichnet. Innerhalb eines *Accept*/*AcceptEnd*-Bereiches gibt *entry* den aktiven Service an. Da die Datenstrukturen sowohl von der aufrufenden Task als auch von der bedienenden Task gelesen und modifiziert werden, ist es notwendig, auf Basis eines Semaphors sich kurzfristig gegenseitig auszuschließen. Die Warteschlangen sollten genauso wie in Ada nach der FIFO-Strategie verwaltet werden:

```
TYPE
   Queue = POINTER TO QueueRec;
   QueueRec =
      RECORD
         head, tail: ConnectCondition;
      END;
```

Die Bedingung, die zum Warten für die Aufrufer von *Connect* dient, trägt auch den Auftrag. Die Komponente *done* wird zunächst von *Connect* auf **FALSE** gesetzt. Später wird *AcceptEnd* das Ende des Aufrufs dadurch signalisieren, daß *done* auf **TRUE** gesetzt wird.

```
TYPE
   ConnectCondition = POINTER TO ConnectConditionRec;
   ConnectConditionRec =
      RECORD
         (Conditions.ConditionRec)
         message: Message;
         done: BOOLEAN;
         next: ConnectCondition;
      END;
VAR
   ccDomain: Conditions.Domain;

PROCEDURE TestConnectCondition(domain: Conditions.Domain;
                               condition: Conditions.Condition;
                               errors: RelatedEvents.Object) : BOOLEAN;
BEGIN
   RETURN condition(ConnectCondition).done
END TestConnectCondition;
```

Auf der anderen Seite müssen *Select* und *Accept* solange blockiert werden, bis entsprechende Aufträge vorliegen:

```
TYPE
   AcceptCondition = POINTER TO AcceptConditionRec;
   AcceptConditionRec =
      RECORD
         (Conditions.ConditionRec)
         taskd: TaskDiscipline;
         entries: EntrySet;
      END;
VAR
   acDomain: Conditions.Domain;

PROCEDURE TestAcceptCondition(domain: Conditions.Domain;
                              condition: Conditions.Condition;
                              errors: RelatedEvents.Object) : BOOLEAN;
BEGIN
   WITH condition: AcceptCondition DO
      RETURN condition.taskd.awaited * condition.entries # {}
   END;
END TestAcceptCondition;
```

Auf dieser Basis können die Operationen von *Rendezvous* realisiert werden:

```
PROCEDURE GetTaskDiscipline(VAR taskd: TaskDiscipline);
BEGIN
   IF ~Disciplines.Seek(Tasks.Current(), taskdID, taskd) THEN
      (* Aufruf erfolgte ausserhalb eines TaskBody *)
      (* hier sollte mit Events.Raise ein entsprechendes
         Ereignis generiert werden
      *)
      Tasks.Terminate;
   END;
END GetTaskDiscipline;

PROCEDURE Accept(entry: Entry; VAR message: Message);
   VAR
      taskd: TaskDiscipline;
      condition: AcceptCondition;
BEGIN
   GetTaskDiscipline(taskd);
   IF ~(entry IN taskd.awaited) THEN
      NEW(condition); Conditions.Init(condition, acDomain);
      condition.taskd := taskd; condition.entries := {entry};
      Tasks.WaitFor(condition);
   END;
   Semaphores.P(taskd.mutex);
   message := taskd.queues[entry].head.message;
   taskd.entry := entry;
   Semaphores.V(taskd.mutex);
END Accept;

PROCEDURE AcceptEnd;
   VAR
      taskd: TaskDiscipline;
      queue: Queue;
BEGIN
   GetTaskDiscipline(taskd);
   Semaphores.P(taskd.mutex);
   queue := taskd.queues[taskd.entry];
   queue.head.done := TRUE;
   queue.head := queue.head.next;
   IF queue.head = NIL THEN
      queue.tail := NIL;
      EXCL(taskd.awaited, taskd.entry);
   END;
   Semaphores.V(taskd.mutex);
END AcceptEnd;
```

```
PROCEDURE Select(entries: EntrySet; VAR entry: Entry);
   VAR
      taskd: TaskDiscipline;
      condition: AcceptCondition;
BEGIN
   GetTaskDiscipline(taskd);
   IF entries * taskd.awaited = {} THEN
      NEW(condition); Conditions.Init(condition, acDomain);
      condition.taskd := taskd; condition.entries := entries;
      Tasks.WaitFor(condition);
   END;
   (* select-marker method *)
   entry := taskd.entry;
   WHILE ~(entry IN entries * taskd.awaited) DO
      entry := (entry + 1) MOD maxentries;
   END;
END Select;

PROCEDURE Connect(task: Tasks.Task; entry: Entry; message: Message);
   VAR
      taskd: TaskDiscipline;
      condition: ConnectCondition;
      queue: Queue;
BEGIN
   IF ~Disciplines.Seek(task, taskdID, taskd) THEN
      (* keine von Initiate gestartete Task *)
      (* entsprechende Fehlerbehandlung einleiten *)
      RETURN
   END;
   NEW(condition); Conditions.Init(condition, ccDomain);
   condition.message := message; condition.done := FALSE;
   condition.next := NIL;
   Semaphores.P(taskd.mutex);
   queue := taskd.queues[entry];
   IF queue.head = NIL THEN
      queue.head := condition;
   ELSE
      queue.tail.next := condition;
   END;
   queue.tail := condition;
   INCL(taskd.awaited, entry);
   Semaphores.V(taskd.mutex);
   Tasks.WaitFor(condition);
END Connect;
```

Die folgende Variante von *Initiate* basiert auf dem Ulmer Koroutinenkonzept und geht davon aus, daß gerade ein Scheduler läuft, der neue Tasks akzeptiert:

```
PROCEDURE CallTaskBody(VAR cr: Coroutines.Coroutine;
                       taskbody: TaskBody);
BEGIN
   SYSTEM.CRSPAWN(cr);
   taskbody;
   Tasks.Terminate;
END CallTaskBody;

PROCEDURE Initiate(body: TaskBody; VAR task: Tasks.Task);
   VAR
      cr: Coroutines.Coroutine;
      tg: Tasks.TaskGroup;
      taskd: TaskDiscipline;
      entry: Entry;
BEGIN
   tg := Tasks.CurrentGroup();
   IF ~(tg IS Schedulers.TaskGroup) THEN
      (* neue Tasks werden nicht akzeptiert *)
      (* entsprechende Fehlerbehandlung einleiten *)
      RETURN
   END;
```

```
      CallTaskBody(cr, body);
      Schedulers.AddTask(tg(Schedulers.TaskGroup), cr, task);
      NEW(taskd); taskd.id := taskdID;
      taskd.awaited := {}; taskd.entry := 0;
      entry := 0;
      WHILE entry < maxentries DO
         NEW(taskd.queues[entry]);
         taskd.queues[entry].head := NIL; taskd.queues[entry].tail := NIL;
         INC(entry);
      END;
      LocalSemaphores.Create(taskd.mutex);
      Disciplines.Add(task, taskd);
   END Initiate;
```

Zum Schluß noch die Prozedur, die bei der Initialisierung des Moduls aufgerufen wird:

```
   PROCEDURE Init;

      PROCEDURE InitDomain(VAR domain: Conditions.Domain;
                           test: Conditions.TestProc);
         VAR
            if: Conditions.Interface;
            desc: Conditions.Description;
      BEGIN
         NEW(if); if.test := test;
         NEW(desc); desc.caps := {}; desc.internal := TRUE;
         NEW(domain);
         Conditions.InitDomain(domain, if, desc);
      END InitDomain;

   BEGIN
      taskdID := Disciplines.Unique();
      InitDomain(ccDomain, TestConnectCondition);
      InitDomain(acDomain, TestAcceptCondition);
   END Init;
```

7.7. Kooperativität

Wenn Parallelität auf der Basis von Koroutinen erreicht wird, ist es notwendig, daß sich die einzelnen Koroutinen kooperativ zueinander verhalten. Da Koroutinen sich selbst nur freiwillig suspendieren können, würde eine Dauerschleife (oder auch eine sehr lang anhaltende Schleife) oder ein blockierender Systemaufruf alle anderen Koroutinen benachteiligen.

Natürlich wäre es möglich, ein Zeitscheibensystem einzuführen. In bestimmten Zeitabständen würde dann ein asynchrones Signal eintreffen und zum Wechsel der Koroutine führen. Dies ist jedoch vergleichsweise kostspielig und sorgt dafür, daß fast alle Koroutinen sich im unterbrochenen Zustand befinden. Dies verträgt sich leider nicht mit einer automatischen Speicherverwaltung, die die lebenden Objekte kopiert, da eine unterbrochene Koroutine ungleich schwerer zu analysieren ist als eine Koroutine, die normal suspendiert worden ist oder die Speicherverwaltung über **NEW** aufgerufen hat. Ein *copying garbage collector* muß alle existierenden Verweise genau kennen. Bei unterbrochenen Koroutinen wäre also nur ein konservativer Ansatz bei der Speicherverwaltung denkbar.[1]

Bei länger dauernden Berechnungen, wie z.B. bei diversen numerischen Algorithmen, würde es sich empfehlen, an geeigneten Stellen ein *Tasks.Suspend* einzufügen. Wenn keine anderen Tasks aktiv sind, ist der dadurch verursachte Zusatzaufwand außerordentlich gering. Im Rahmen der Ulmer Oberon-Bibliothek gibt es bislang keine Stelle, wo dies notwendig wäre.

Anders sieht es aus bei den Systemaufrufen, die blockieren könnten. Unter UNIX lassen sich viele blockierende Systemaufrufe entweder unterbrechen oder in nicht blockierender Form verwenden.[2] In der Ulmer Oberon-

[1] Beim konservativen Ansatz werden die Objekte nicht kopiert, und ein Objekt wird als lebend betrachtet, solange ein potentieller Zeiger darauf oder hinein zeigt. Die potentiellen Zeiger bilden dann eine Obermenge der tatsächlichen Zeiger. Der konservative Ansatz ist sowohl in Bezug auf Laufzeit- als auch Speichereffizienz der kopierenden Speicherverwaltung unterlegen (solange das Kopieren nicht zu teuer ist).

[2] Leider nicht alle. So sind Ein- und Ausgabe-Operationen auf Dateien nicht unterbrechbar, selbst wenn sie über das Netz via NFS bezogen werden und deswegen längere Zeit in Anspruch nehmen können. Im Extremfall (*hard mount* bei NFS) kann dies zum "Aufhängen" des Prozesses führen.

Bibliothek bieten die *Sys*-Module Systemaufrufe in der Originalform an, wobei — soweit notwendig — die Option zur Verfügung steht, den Systemaufruf bei Unterbrechungen automatisch wiederholen zu lassen.

Die Module, die die Schnittstellen zum Betriebssystem in Form allgemeiner Abstraktionen zur Verfügung stellen, sind dann dafür verantwortlich, daß es zu keinen unnötigen Blockierungen kommt. Zwar wäre es möglich, diese Aufgabe in die Anwendungsprogramme zu verlagern. Dies wäre allerdings weniger effizient und führt zu relativ umständlichem Programmtext. Bei der Ulmer Oberon-Bibliothek braucht man also nur auf *Tasks.WaitFor* zurückzugreifen, wenn man auf mehrere Ereignisse gleichzeitig oder auf ein internes Ereignis warten möchte.

Folgender Ausschnitt aus *UnixFiles* zeigt, wie dies in sehr effizienter Weise für Ein- und Ausgabe-Operationen geschieht:

```
PROCEDURE ReadBuf(s: Streams.Stream; buf: SysTypes.Address;
                  cnt: SysTypes.Count) : SysTypes.Count;
   VAR
      count: SysTypes.Count;
BEGIN
   WITH s: Stream DO
      IF s.blocking THEN
         Tasks.WaitFor(s.readCond);
      END;
      LOOP
         count := SysIO.Read(s.file, buf, cnt, s, s.retry, s.interrupted);
         IF (count >= 0) OR ~WouldBlock(s) THEN
            RETURN count
         END;
         Tasks.WaitFor(s.readCond);
      END;
   END;
END ReadBuf;
```

ReadBuf ist die Schnittstellenprozedur für *addrread* von *Streams*. Dateideskriptoren, die *UnixFiles* selbst eröffnet, werden in nicht-blockierenden Modus versetzt. Bei anderen Dateideskriptoren wird dies unterlassen, da es bei vererbten Dateideskriptoren zu Problemen beim vererbenden Prozeß führen kann.[1] Die Komponente *s.blocking* gibt somit an, ob der zugehörige Dateideskriptor sich im blockierenden Modus befindet. Im nicht-blockierenden Modus reagiert der Systemaufruf *read* mit einer Fehlerindikation, wenn die Eingabe noch nicht zur Verfügung steht.

Entsprechend wird beim nicht-blockierenden Modus sofort *SysIO.Read* abgesetzt. Wenn die Lese-Operation blockieren würde, kommt es zu einem Fehler. *WouldBlock* stellt fest, ob dies der Fall ist, und eliminiert ggf. das entsprechende Fehlerereignis (das aufrufende Module nur verwirren würde). Danach wird mit *Tasks.WaitFor* die Koroutine auf kooperative Weise blockiert, bis die Eingabe soweit ist.

Die beiden letzten Parameter von *SysIO.Read* beziehen sich auf Unterbrechungen. Im Normalfall ist *s.retry* wahr, und *SysIO.Read* startet dann die Operation sofort erneut, falls es zu einer Unterbrechung kam. In der Komponente *s.interrupted* wird notiert, ob eine Unterbrechung vorlag oder nicht.

Das Beispiel zeigt, daß für die Kooperativität hier kein Effizienzverlust hingenommen werden muß. Bei Lese-Operationen, die sich auf Dateien beziehen (und die weder unterbrechbar sind noch lange dauern), wird genau ein Systemaufruf abgesetzt.

7.8. Anwendungsbeispiele

7.8.1. Ein Konferenz-Dämon

7.8.1.1. Einführung

Als Dämonen werden Programme bezeichnet, die langfristig laufen und bestimmte Dienstleistungen anbieten oder übernehmen. Im folgenden wird ein Konferenz-Dämon vorgestellt, der seine Dienste auf einem Netzwerk zur Verfügung stellt. Einmal gestartet, kann jeder auf dem Netzwerk den Kontakt zu dem Dämon aufnehmen und nach einer Registrierung sich mit allen anderen Kontaktaufnehmern unterhalten.

[1] So würden alle gängigen UNIX-Shells sofort auf Ende der Eingabe erkennen und die Sitzung beenden.

Auf dem Internet könnte eine Sitzung dann etwa so ablaufen:

```
laborix$ telnet titania 4711                      Kontaktaufnahme
Trying 134.60.66.21...
Connected to titania.
Escape character is '^]'.                         Kontakt ist hergestellt
Hallo, hier ist der Konferenz-Daemon.             Begrüßung durch den Dämon
Gib bitte Deinen Namen ein.                       Aufforderung zur Identifizierung
Andreas                                           Eingabe des eigenen Namens
Hallo Andreas, Du bist herzlich willkommen!
Hallo zusammen, ist jemand da?                    Eingabe einer Textzeile
Andreas: Hallo zusammen, ist jemand da?           Echo der eigenen Zeile
Robert: Hallo Andreas!                            Eine Zeile eines anderen Teilnehmers
Wolfgang: Nett, Dich zu treffen, Andreas!         Noch ein weiterer Teilnehmer
quit                                              Verlassen des Konferenz-Dämons
Connection closed by foreign host.
laborix$
```

Die Schnittstelle eines entsprechenden Moduls könnte folgendermaßen aussehen:

```
DEFINITION ChatDaemons;

   IMPORT Disciplines, Networks;

   TYPE
      ChatDaemon = POINTER TO ChatDaemonRec;
      ChatDaemonRec = RECORD (Disciplines.ObjectRec) END;

   PROCEDURE Create(VAR daemon: ChatDaemon; port: Networks.Address);
   PROCEDURE Terminate(daemon: ChatDaemon);

END ChatDaemons.
```

Das Modul *Networks* stellt die allgemeine Abstraktion für Netzwerke und Netzwerkverbindungen zur Verfügung. Eine mögliche Implementierung hierfür ist in der obigen Beispielsitzung *Internet*. Der Parameter *port* ist eine Adresse, bei der der neu zu startende Dämon sich registrieren soll, so daß spätere Teilnehmer ihn anwählen können.

Tasks läßt es nur zu, daß eine Task sich selbst terminiert. Trotzdem lassen sich indirekt Tasks auch von außen terminieren, wenn die Task bereit ist, einen entsprechenden Wunsch entgegenzunehmen. *Terminate* dient hier als Beispiel für diese Technik.

7.8.1.2. Die Implementierung von *ChatDaemons*

Die Arbeit eines Konferenz-Dämons wird von einer Reihe von Tasks übernommen: Eine Task kümmert sich um neue Kontaktaufnahmen, und für jede bestehende Verbindung gibt es eine weitere Task. Alle Tasks gehören einer Taskgruppe an, die von *Create* kreiert wird. Alle Tasks dieser Gruppe interessieren sich für den Terminierungswunsch (*termination*), und alle Tasks, die mit einer Verbindung zu tun haben, interessieren sich für die Textzeilen anderer Benutzer (*broadcast*):

```
TYPE
   ChatDaemon = POINTER TO ChatDaemonRec;
   ChatDaemonRec =
      RECORD
         (Disciplines.ObjectRec)
         tg: Schedulers.TaskGroup;
         broadcast: Events.EventType;
         termination: Events.EventType;
      END;

   Name = ARRAY 32 OF CHAR;
   Message = Events.Message;
   BroadcastEvent = POINTER TO BroadcastEventRec;
   BroadcastEventRec =
      RECORD
         (Events.EventRec)
         sender: Name;
      END;
```

Das Verteilen eingegebener Textzeilen an andere Benutzer erfolgt über *Broadcast*:

```
PROCEDURE Broadcast(daemon: ChatDaemon;
                    sender: Name; line: Message);
   VAR
      event: BroadcastEvent;
BEGIN
   NEW(event); event.type := daemon.broadcast;
   event.message := line;
   event.sender := sender;
   Events.Raise(event);
END Broadcast;
```

Jede Task, die sich um eine Verbindung zu einem Benutzer kümmert, wartet während des Dialogs auf eines von drei Ereignissen: die Eingabe einer Zeile vom Benutzer, das Eingeben einer Zeile von einem anderen Benutzer oder die Terminierung. Das Warten auf Eingabe erfolgt mit Hilfe von *StreamConditions*, das die Abstraktion von *Streams* in Richtung asynchroner Ein- und Ausgabe erweitert. Dies geschieht auf Basis von Aufträgen, die Erweiterungen von *Streams.Message* sind und mit Hilfe von *Streams.Send* an die Implementierung verschickt werden können.

Dialog ist eine Prozedur, die sich selbst zur Koroutine erklärt und dann den Dialog mit einem Benutzer aufnimmt. Der Parameter *s* steht für den bidirektionalen Kommunikationskanal zum Benutzer:

```
PROCEDURE Dialog(VAR cr: Coroutines.Coroutine;
                 daemon: ChatDaemon;
                 s: Streams.Stream);
   CONST
      hello1 = "Hallo, hier ist der Konferenz-Daemon.";
      hello2 = "Gib bitte Deinen Namen ein.";
      welcome1 = "Hallo ";
      welcome2 = ", Du bist herzlich willkommen!";
      greeting = "ist mit von der Partie.";
      leaves = "verabschiedet sich.";
   VAR
      name: Name;    (* the name of our user *)
      line: Message; (* last line typed in by our user *)
      (* conditions *)
      readyForReading: Conditions.Condition;
         (* yields TRUE if our user has sth typed in *)
      broadcastMsgAvailable: Conditions.Condition;
         (* yields TRUE if anybody has sth typed in *)
      terminationMsgAvailable: Conditions.Condition;
         (* yields TRUE on termination *)
      conditions: Conditions.ConditionSet;
         (* the set of all conditions above *)
      errors: RelatedEvents.Object;
      event: Events.Event; (* broadcast or termination event *)

   PROCEDURE Terminate;
      (* clean up and terminate *)
   BEGIN
      EventConditions.Drop(broadcastMsgAvailable);
      EventConditions.Drop(terminationMsgAvailable);
      Streams.Release(s);
      Tasks.Terminate;
   END Terminate;

BEGIN
   SYSTEM.CRSPAWN(cr);

   NEW(errors); RelatedEvents.QueueEvents(errors);

   (* create set of conditions *)
   StreamConditions.Create(readyForReading, s, StreamConditions.read);
   EventConditions.Create(broadcastMsgAvailable, daemon.broadcast);
   EventConditions.Create(terminationMsgAvailable, daemon.termination);
   Conditions.CreateSet(conditions);
   Conditions.Incl(conditions, readyForReading);
```

```
      Conditions.Incl(conditions, broadcastMsgAvailable);
      Conditions.Incl(conditions, terminationMsgAvailable);

      (* first hello and identification *)
      Write.LineS(s, hello1); Write.LineS(s, hello2);
      Read.LineS(s, name);
      IF name = "" THEN
         Terminate;
      END;
      (* welcome and first broadcast *)
      Write.StringS(s, welcome1);
      Write.StringS(s, name);
      Write.LineS(s, welcome2);
      Broadcast(daemon, name, greeting);

      LOOP (* until termination *)
         Tasks.WaitForOneOf(conditions);
         WHILE Conditions.Test(readyForReading, errors) &
               ~RelatedEvents.EventsPending(errors) DO
            Read.LineS(s, line);
            IF s.eof OR (line = "quit") THEN
               EXIT
            END;
            Broadcast(daemon, name, line);
         END;
         IF RelatedEvents.EventsPending(errors) THEN EXIT END;

         WHILE EventConditions.TestAndGet(broadcastMsgAvailable, event) DO
            WITH event: BroadcastEvent DO
               Write.StringS(s, event.sender);
               Write.StringS(s, ": ");
               Write.LineS(s, event.message);
            END;
         END;
         IF EventConditions.TestAndGet(terminationMsgAvailable, event) THEN
            EXIT
         END;
      END;
      Broadcast(daemon, name, leaves);
      Terminate;
   END Dialog;
```

Die Task, die sich um das Aufnehmen neuer Kontakte kümmert, macht von *Networks.AcceptCondition* Gebrauch, das das Warten auf neue Kontaktaufnahmen ermöglicht. Jede neue Kontaktaufnahme führt zum Erzeugen einer neuen Koroutine für den Dialog, die dann dem Scheduler der Taskgruppe übergeben wird:

```
   PROCEDURE ConnectionManager(VAR cr: Coroutines.Coroutine;
                               daemon: ChatDaemon; port: Networks.Address);
      VAR
         newConnection: Conditions.Condition;
         terminationMsgAvailable: Conditions.Condition;
         conditions: Conditions.ConditionSet;
         socket: Networks.Socket; (* registered port *)
         s: Streams.Stream; (* new communication channel *)
         event: Events.Event;
         errors: RelatedEvents.Object;
         dialogCR: Coroutines.Coroutine;
         task: Tasks.Task;
   BEGIN
      SYSTEM.CRSPAWN(cr);

      NEW(errors); RelatedEvents.QueueEvents(errors);

      (* register port address so that clients may connect to *)
      IF ~Networks.Listen(socket, port, errors) THEN Tasks.Terminate END;

      (* create conditions *)
      Networks.CreateAcceptCondition(newConnection, socket);
      EventConditions.Create(terminationMsgAvailable, daemon.termination);
```

```
   Conditions.CreateSet(conditions);
   Conditions.Incl(conditions, newConnection);
   Conditions.Incl(conditions, terminationMsgAvailable);

   LOOP
      Tasks.WaitForOneOf(conditions);
      IF EventConditions.TestAndGet(terminationMsgAvailable, event) THEN
         EXIT
      END;
      WHILE Conditions.Test(newConnection, errors) &
            ~RelatedEvents.EventsPending(errors) &
            Networks.Accept(socket, s, Streams.linebuf) DO
         Dialog(dialogCR, daemon, s);
         Schedulers.AddTask(daemon.tg, dialogCR, task);
      END;
   END;
   EventConditions.Drop(terminationMsgAvailable);
   Networks.Release(socket);
   Tasks.Terminate;
END ConnectionManager;
```

Create braucht dann nur noch eine neue Taskgruppe zu kreieren und *ConnectionManager* zu starten:

```
PROCEDURE SpawnScheduler(VAR cr: Coroutines.Coroutine;
                         tg: Schedulers.TaskGroup);
   (* spawn scheduler for a new task group *)
BEGIN
   SYSTEM.CRSPAWN(cr);
   Schedulers.StartScheduler(tg);
   Tasks.Terminate;
END SpawnScheduler;

PROCEDURE Create(VAR daemon: ChatDaemon; port: Networks.Address);
   VAR
      cm, sched: Coroutines.Coroutine;
      tg: Tasks.TaskGroup;
      task: Tasks.Task;
BEGIN
   tg := Tasks.CurrentGroup();
   IF ~(tg IS Schedulers.TaskGroup) THEN
      (* new tasks are not accepted *)
      (* initiate exception handling *)
      RETURN
   END;

   NEW(daemon);
   Events.Define(daemon.broadcast); Events.Ignore(daemon.broadcast);
   Events.Define(daemon.termination); Events.Ignore(daemon.termination);
   RoundRobin.CreateTaskGroup(daemon.tg);
   ConnectionManager(cm, daemon, port);
   Schedulers.AddTask(daemon.tg, cm, task);

   SpawnScheduler(sched, daemon.tg);
   Schedulers.AddTask(tg(Schedulers.TaskGroup), sched, task);
END Create;
```

Terminate kreiert für den gegebenen Dämon ein Terminierungsereignis. Da alle zugehörigen Tasks auf dieses Ereignis warten und bei Eintreffen sofort *Tasks.Terminate* ausführen, kommt es zur Terminierung aller Tasks dieser Taskgruppe und anschließend zur Rückkehr von *StartScheduler* in *SpawnScheduler* und damit auch zur Terminierung des Schedulers:

```
PROCEDURE Terminate(daemon: ChatDaemon);
   VAR
      event: Events.Event;
BEGIN
   NEW(event); event.type := daemon.terminate;
   Events.Raise(event);
END Terminate;
```

7.8.2. Eine Schnittstelle zu NeWS

7.8.2.1. Einführung

NeWS steht für *Network/extensible Window System* und wurde von der Firma Sun Microsystems entwickelt [Gosling89]. Analog zum X-Window-System erlaubt NeWS die räumliche Trennung zwischen dem Server-Prozeß, der für den Bildschirm verantwortlich ist, und den Klienten, die auf möglicherweise entfernten Rechnern laufen.

Beim X-Window-System gibt es eine statische Festlegung aller Operationen und der zugehörigen Parameter. Im Rahmen des X-Protokolls sind diese Operationen und ihre zugehörigen Kodierungen festgelegt. Da es sich um sehr einfache Operationen handelt, müssen komplexere Operationen in eine Reihe von einfachen Operationen umgesetzt und über das Netz zum Server geschickt werden. Selbst im Falle einfacher Wiederholungen und Änderungen führt dies zu einem sehr umfangreichen Datenaustausch. Hinzu kommt, daß typischerweise zu viele Ereignisse (z.B. Mausbewegungen) zum Klienten weitergereicht werden müssen, da nur im Klienten der Programmtext existiert, der auf Ereignisse reagieren kann.

Darüber hinaus basieren X-Windows und die meisten der anderen Fenstersysteme auf Bitmap-Operationen. Auf Basis von Bitmap-Operationen ist es jedoch außerordentlich schwierig, auf portable Weise Grafiken darzustellen. Nicht nur, daß die Punktdichten bezüglich der x- und y-Achse unterschiedlich sein können (Kreise werden so zu Ellipsen), sondern auch unterschiedliche Formen der Punkte führen zu einem großen Variantenreichtum an optischen Effekten. Außerordentlich problematisch ist die Skalierbarkeit, wenn der optische Eindruck erhalten bleiben soll, und endgültig unübersichtlich wird es bei Farben. Von daher sind Bitmap-Operationen keine solide Grundlage für ein verteiltes Fenstersystem.

In den Siebziger Jahren wurde bei Evans & Sutherland, danach bei Xerox PARC und schließlich bei Adobe Systems eine Programmiersprache zur Beschreibung von Grafiken entwickelt [Adobe Systems Inc.86]. Obwohl POSTSCRIPT zunächst nur als Sprache zum Drucken gedacht war, erwies sie sich später auch als sehr geeignet für grafische Benutzeroberflächen. Statt auf Bitmap-Operationen ist POSTSCRIPT auf das *paint-stencil*-Modell begründet. Vereinfacht formuliert kann hierbei zunächst eine Schablone in Form von Linien und Kurven vorgegeben werden. Anschließend kann die Fläche ausgefüllt werden oder entlang der Form eine Linie gezogen werden oder die Schablone zur Grundlage weiterer Operationen gemacht werden. Da in der Beschreibung keine Raster vorkommen und selbst die Zeichen aus Zeichensätzen auf Basis von Kurven definiert sind, ist sie vollkommen geräteunabhängig und skalierbar. Darüber hinaus enthält POSTSCRIPT alle Elemente einer normalen Programmiersprache (die etwas an Forth erinnert, aber nicht von ihr abstammt).

Ähnlich wie Oberon eine Programmiersprache und auch ein Betriebssystem bezeichnet, steht NeWS sowohl für das Fenstersystem (und damit für den Server) als auch für die zugehörige Programmiersprache, die eine Erweiterung von POSTSCRIPT ist. Prinzipiell können in NeWS vollständige Applikationen geschrieben werden. Diese Programme lassen sich zum Server in Quelltextform herunterladen, der sie anschließend interpretativ ausführt. Da in POSTSCRIPT viele Elemente fehlen, die im Rahmen einer Benutzeroberfläche wichtig sind, wurden in NeWS Ereignisse, Tasks und objekt-orientierte Konzepte hinzugefügt.

Im Normalfall wird jedoch nicht die gesamte Applikation in NeWS formuliert. Statt dessen wird nur die Kommunikation zwischen dem NeWS-Server und einem Klienten auf Basis von NeWS durchgeführt. Da dennoch ganze Programmteile und Bibliotheken in NeWS zum Server heruntergeladen werden können, ist es möglich, die routinemäßige Kleinarbeit im Server erledigen zu lassen (z.B. das Ausfüllen eines Formulars oder die Auswahl in einem Menü), um sich anschließend die Ergebnisse (und nicht die unzähligen Ereignisse) wieder zurückschicken zu lassen.

Der überraschende Erfolg davon ist, daß NeWS wirklich *extensible* im Sinne von Entfernung ist. So ist es kein Problem, bei NeWS den Klienten und den Server auf unterschiedlichen Erdteilen auf dem Internet laufen zu lassen — bei X-Windows wäre dies (zumindest zur Zeit) undenkbar.

Ein weiterer Vorteil findet sich darin, daß die Bibliotheken für die gängigen Elemente einer Benutzeroberfläche (*gadgets* und *widgets*) in NeWS realisiert sind. Im Gegensatz zu X-Windows kann man diese Bibliotheken auch dann benutzen, wenn man keine in C geschriebenen Bibliotheken einbinden kann oder will. Von daher lag es nahe, eine Schnittstelle zu NeWS in Oberon zur Verfügung zu stellen.

7.8.2.2. Die Kommunikation zwischen Oberon und NeWS

Zwischen einem Klienten und dem NeWS-Server gibt es im Normalfall nur einen einzigen bidirektionalen Kommunikationskanal. Auf dem Hinweg zum Server können Anweisungen in NeWS geschickt werden. Die Programmteile, die in den NeWS-Server auf diese Weise heruntergeladen worden sind, können wiederum jederzeit Rückmeldungen auf diesem Kanal ausgeben, die auf der Seite des Klienten wieder eingelesen werden können.

In der Ulmer Oberon-Bibliothek bietet sich als Abstraktion für den Kommunikationskanal *Streams* an. Entsprechend einfach kann eine Interaktion zwischen dem Oberon-Programm und dem NeWS-Server aussehen:

```
Write.LineS(news, "(It works\n) print");
Streams.Touch(news);
Read.LineS(news, line);
(* line sollte hier "It works" enthalten *)
```

Das NeWS-Protokoll erlaubt die Übertragung in normaler ASCII-Form, jedoch auch aus Effizienzgründen (um den Netzverkehr noch weiter zu reduzieren) in kompaktierter Form. So gibt es für alle Schlüsselwörter und weiteren lexikalischen Elemente spezielle Kodierungen. Das Modul *NeWS* aus der Ulmer Oberon-Bibliothek enthält eine komplette lexikalische Analyse von NeWS und kann diese Kodierungen alle fortlaufend durchführen. Im obigen Beispiel wurde für die Zeichenkette "It works\n" ein spezielles Symbol für eine Zeichenkette zusammen mit der Länge generiert und später ein spezielles Symbol für **print**. Statt 18 Zeichen wurden somit nur 11 übertragen.

Häufig gehört eine Reihe von Fenstern und Elementen zu einer Applikation. Da der Benutzer die Freiheit hat, auf welchem Fenster er als nächstes arbeitet oder welches Element er benutzt, ist aus der Sicht des Klienten eine Reihe sehr unterschiedlicher Rückmeldungen zu erwarten. Um Rückmeldungen aus verschiedenen Quellen leichter voneinander unterscheiden zu können, können Marken (*tags*) vergeben werden. Eindeutige Marken werden von *NeWS.GetUniqueTag* generiert und dann als Parameter beim Kreieren eines Elements über NeWS an den Server übergeben. In NeWS kann dann mit der Operation **tagprint** eine Marke beim Beginn einer Rückmeldung ausgegeben werden. Marken werden vom NeWS-Server speziell kodiert und lassen sich von den anderen Ausgaben unterscheiden. Typischerweise folgt einer Marke eine Reihe von Informationen (z.B. welcher Text in einem bestimmten Feld eingegeben worden ist), die mit **typedprint** ebenfalls kodiert werden.

Auf der Seite von NeWS läßt sich durch lokale Synchronisierungsmaßnahmen sicherstellen, daß es zu keinen Interferenzen zwischen Rückmeldungen verschiedener Elemente kommen kann. Somit kann die Eingabe auf der Seite des Klienten als Folge von Rückmeldungen betrachtet werden, die aus einer Marke und einigen zugehörigen Informationen bestehen (siehe Abbildung 40, wobei die Kreise für Marken stehen und die Quadrate für zugehörige Informationen).

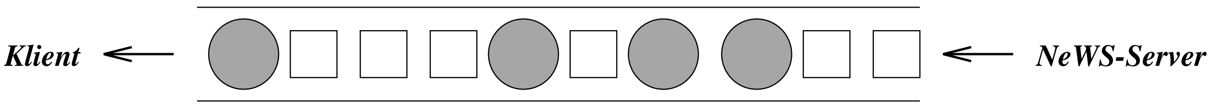

Abb. 40: Folgen von Rückmeldungen

Prinzipiell gleicht dies wieder dem traditionellen Bild analog zu X-Windows, bei dem der Klient vom Server eine Folge von Ereignissen erhält. Der wesentliche Unterschied ist hier, daß die Art der Ereignisse und die zugehörigen Informationen individuell für eine bestimmte Applikation entwickelt werden können und nicht analog zum X-Protokoll fest vorgegeben sind.

7.8.2.3. Die parallele Bearbeitung von Rückmeldungen

Typisch für frühe Applikationen waren zentrale Schleifen, die alle eingehenden Ereignisse untersuchten und die zugehörigen Operationen aufriefen. Später wurde (wie z.B. in einer Reihe gängiger Bibliotheken zu X-Windows) die zentrale Schleife in eine Bibliothek ausgelagert. Dann ist es notwendig, daß zuvor für alle interessanten Ereignisse Prozeduren als Bearbeiter angegeben werden. Dies hat den Nachteil, daß den Prozeduren beim Aufruf der lokale Kontext fehlt, den sie sich erst wieder auf Basis globaler Variablen holen müssen.

Diese Ansätze sind nicht befriedigend, und es ist offensichtlich, daß es vorteilhafter ist, korrespondierend zu den parallel existierenden Objekten in der Benutzeroberfläche auch parallel laufende Tasks zu haben. So könnte beispielsweise ein Editor für jedes Textfenster, das eröffnet wird, auch eine zugehörige Task starten, die sich um die Ereignisse für das neue Textfenster kümmert.

Zu jedem der Ereignisse, für die sich eine Task interessiert, ist die zugehörige Marke bekannt (entweder weil sie das als Parameter erhalten hat oder selbst das zugehörige Element mitsamt der Marke kreiert hat). Auf dieser Basis kann eine Bedingung kreiert werden, die es erlaubt, auf eine bestimmte Rückmeldung zu warten:

```
PROCEDURE CreateCondition(VAR condition: Conditions.Condition;
                          s: Streams.Stream;
                          tag: Tag; minlen: INTEGER);
```

s ist hier die Verbindung zum NeWS-Server, *tag* die Marke und *minlen* die Mindestlänge (gezählt in Objekten), die als Information zu der Marke gehört. Die Semantik dieser Bedingung entspricht der zuerst vorgestellten Variante in §7.4.3.2 mit der *accept*-Operation. Dem *accept* entspricht hier das Abholen der Objekte, wenn die Bedingung wahr geworden ist:

```
PROCEDURE GetListOfObjects(s: Streams.Stream; tag: Tag;
                           VAR list: ListOfObjects) : BOOLEAN;
```

Die einzelnen Objekte der Liste können ganze Zahlen, Fließkommazahlen oder Zeichenketten (beliebiger Länge) sein, die sequentiell betrachtet werden können:

```
PROCEDURE Length(list: ListOfObjects) : INTEGER;
PROCEDURE IsInteger(list: ListOfObjects) : BOOLEAN;
PROCEDURE GetInt(list: ListOfObjects; VAR intval: LONGINT) : BOOLEAN;
(* analoge Operationen fuer Gleitkommazahlen und Zeichenketten *)
```

Die Anwendung dieser Operationen kann an einem kleinen Beispiel demonstriert werden. Die folgende Task soll ein Fenster verwalten, das es erlaubt, einige Konfigurationsparameter zu verstellen. Insgesamt gibt es drei Parameter, für die jeweils eine Textzeile eingegeben werden kann. Dazu kommen die drei Knöpfe *Apply*, *Cancel* und *Done*. *Apply* und *Done* führen zum Akzeptieren der möglicherweise modifizierten Parameter. Bei *Cancel* und *Done* soll das Fenster verschwinden.

Entsprechend den drei Knöpfen gibt es die drei Ereignisse *apply*, *cancel* und *done*, wobei *apply* und *done* die drei Zeilen mit den Parametern als Information mitliefern:

```
PROCEDURE ConfWindow(news: Streams.Stream);
   CONST
      nfields = 3; (* Anzahl der Konfigurationsparameter *)
   VAR
      applyTag, cancelTag, doneTag: NeWS.Tag;
      applyCond, cancelCond, doneCond: Conditions.Condition;
      anythingPressed: Conditions.ConditionSet;
      finished: BOOLEAN;

   PROCEDURE UpdateConf(tag: NeWS.Tag);
      VAR
         list: NeWS.ListOfObjects;
         field: ARRAY nfields OF ConfParameter;
         index: INTEGER;
   BEGIN
      IF NeWS.GetListOfObjects(news, tag, list) THEN
         index := 0;
         WHILE (index < nfields) &
               NeWS.GetString(list, field[index]) DO
            INC(index);
         END;
         IF index = nfields THEN
            (* neue Konfigurationsparameter eintragen *)
         END;
      END;
   END UpdateConf;

   PROCEDURE DummyAccept(tag: NeWS.Tag);
      VAR
         list: NeWS.ListOfObjects;
   BEGIN
      IF NeWS.GetListOfObjects(news, tag, list) THEN END;
   END DummyAccept;

BEGIN
   NeWS.GetUniqueTag(news, applyTag);
   NeWS.GetUniqueTag(news, cancelTag);
   NeWS.GetUniqueTag(news, doneTag);
   (* entsprechendes Fenster kreieren und die drei Marken
```

```
      als Parameter uebergeben
   *)
   NeWS.CreateCondition(applyCond, news, applyTag, nfields);
   NeWS.CreateCondition(cancelCond, news, cancelTag, 0);
   NeWS.CreateCondition(doneCond, news, doneTag, nfields);
   Conditions.CreateSet(anythingPressed);
   Conditions.Incl(anythingPressed, applyCond);
   Conditions.Incl(anythingPressed, cancelCond);
   Conditions.Incl(anythingPressed, doneCond);

   finished := FALSE;
   WHILE ~finished DO
      Tasks.WaitForOneOf(anythingPressed);
      IF Conditions.Test(applyCond, news) THEN
         UpdateConf(applyTag);
      ELSIF Conditions.Test(cancelCond, news) THEN
         DummyAccept(cancelTag); finished := TRUE;
      ELSIF Conditions.Test(doneCond, news) THEN
         UpdateConf(doneTag); finished := TRUE;
      ELSE
         (* schwerwiegender Fehler, z.B. Zusammenbruch
            des Kommunikationskanals zum NeWS-Server
         *)
         finished := TRUE;
      END;
   END;
   (* das Fenster wieder abbauen *)
END ConfWindow;
```

Wichtig ist hier, daß Rückmeldungen, die verschiedene Marken besitzen, in einer anderen Reihenfolge abgeholt werden können, als sie über den Kommunikationskanal geschickt worden sind. Nur bei Rückmeldungen mit der gleichen Marke bleibt die ursprüngliche Reihenfolge erhalten.

7.8.2.4. Die Implementierung von *NeWS*

Zu jeder Verbindung mit einem *NeWS*-Server wird eine Hash-Tabelle mit eingegangenen, aber noch nicht bearbeiteten Rückmeldungen unterhalten:

```
CONST
   tagtabsize = 32;
TYPE
   TagEntry = POINTER TO TagEntryRec;
   TagBucketTable = ARRAY tagtabsize OF TagEntry;
   TagEntryRec =
      RECORD
         tag: Tag;
         head, tail: ListOfObjects;   (* linear list of object lists *)
         next: TagEntry;              (* next entry with same hash value *)
      END;
```

Eine Bedingung, die es erlaubt, auf das Warten einer bestimmten Rückmeldung zu warten, enthält den Verweis auf den Kanal, die Nummer der Marke und die Mindestlänge:

```
TYPE
   TagCondition = POINTER TO TagConditionRec;
   TagConditionRec =
      RECORD
         (Conditions.ConditionRec)
         s: Stream;
         tag: Tag;
         minlen: INTEGER;
      END;
VAR
   tagconddomain: Conditions.Domain;
```

Entsprechend einfach ist das Kreieren von Bedingungen:

```
PROCEDURE CreateCondition(VAR condition: Conditions.Condition;
                          s: Streams.Stream; tag: Tag; minlen: INTEGER);
   VAR
      newcond: TagCondition;
BEGIN
   IF (s IS Stream) & (s(Stream).mode = cooked) THEN
      WITH s: Stream DO
         NEW(newcond); Conditions.Init(newcond, tagconddomain);
         newcond.s := s; newcond.tag := tag; newcond.minlen := minlen;
      END;
   ELSE
      newcond := NIL;
   END;
   condition := newcond;
END CreateCondition;
```

Die zugehörige Testoperation ruft zunächst *FillBuf* auf, das die gesamte Eingabe vom NeWS-Server abholt, soweit sie sich einlesen läßt, ohne blockiert zu werden. *FindTagEntry* sucht nach der Marke der Bedingung innerhalb der Hash-Tabelle. Das Auftreten von Fehlern, nach denen das Ereignis nicht mehr eintreten kann, muß zur Terminierung von entsprechenden Warteoperationen führen. Entsprechend wird bei einem Zusammenbruch der Verbindung zum NeWS-Server hier **TRUE** zurückgeliefert:

```
PROCEDURE TestTagCondition(domain: Conditions.Domain;
                           condition: Conditions.Condition;
                           errors: RelatedEvents.Object) : BOOLEAN;
   VAR
      index: INTEGER;
      tagentry: TagEntry;
BEGIN
   WITH condition: TagCondition DO
      FillBuf(condition.s);
      FindTagEntry(condition.s, condition.tag, tagentry);
      IF (tagentry.head # NIL) &
          ((tagentry.head.length >= condition.minlen) OR
           (tagentry.head # condition.s.currentList)) THEN
         RETURN TRUE
      ELSE
         IF condition.s.socket.eof OR condition.s.socket.error THEN
            Error(errors, connectionFinished); RETURN TRUE
         END;
         RETURN FALSE
      END;
   END;
END TestTagCondition;
```

Wenn *NeWS* nur *TestTagCondition* in der Schnittstelle zu *Conditions* anbieten würde, wäre eine *busy loop* unvermeidlich. Dies kann glücklicherweise durch die Definition von Vorbedingungen vermieden werden:

```
PROCEDURE GetTagPreConditions(
               domain: Conditions.Domain;
               condition: Conditions.Condition;
               VAR preconds: Conditions.ConditionSet;
               errors: RelatedEvents.Object) : BOOLEAN;
BEGIN
   WITH condition: TagCondition DO
      Conditions.CreateSet(preconds);
      IF TestTagCondition(domain, condition, errors) THEN
         RETURN FALSE
      ELSIF condition.s.socket.eof OR condition.s.socket.error THEN
         Error(errors, connectionFinished); RETURN FALSE
      ELSE
         Conditions.Incl(preconds, condition.s.readyForReading);
         RETURN TRUE
      END;
   END;
END GetTagPreConditions;
```

GetTagPreConditions liefert genau dann **TRUE** zurück, wenn es irgendwelche Vorbedingungen gibt. Die Rückgabe von Vorbedingungen ist nur dann sinnvoll, wenn das Ereignis noch nicht eingetreten ist und davon auszugehen ist, daß es noch eintreffen wird. Hier wird als Vorbedingung *condition.s.readyForReading* zurückgeliefert. Dabei handelt es sich um eine Bedingung von *StreamConditions*, die bei UNIX-Kommunikationskanälen von *UnixFileConditions* zur Verfügung gestellt wird. *UnixFileConditions* wird in kooperativer Weise von allen Modulen genutzt, die Bedingungen auf Basis von UNIX-Dateideskriptoren benötigen, und stellt *Conditions* gegenüber eine *select*-Schnittstelle zur Verfügung, die auf *SysSelect*[1] basiert. Auf dieser Basis ist ein effizientes Warten auf das Eintreffen von bestimmten Marken möglich.

7.9. Zusammenfassung

Während in einer Vielzahl anderer Programmiersprachen Parallelität ein essentieller Bestandteil der Sprache ist, wurde beim Ulmer Oberon-System der Versuch unternommen, Parallelität über die Bibliothek einzuführen. Hierzu gibt es nur sehr rare auf Koroutinen basierende Beispiele wie etwa einige C++-Klassen für die diskrete Ereignissimulation [Stroustrup87] und *Processes* für Modula-2 [Wirth82, §30]. Sowohl die Varianten, die Parallelität über die Programmiersprache unterstützen, als auch die Bibliotheken haben weitgehend gemeinsam, daß sie nur Sprachmittel zum Starten neuer Tasks und zur Synchronisierung zwischen Tasks unterstützen. Allesamt sind sie kaum erweiterbar — so ist beispielsweise der Scheduler integraler Bestandteil dieser Systeme, und es gibt keine Möglichkeit, ihn zu ersetzen oder zu erweitern.

Entsprechend schlecht ist es um die Skalierbarkeit dieser Systeme bestellt, da sie typischerweise nur Parallelität auf Basis von Koroutinen oder Multiprozessormaschinen anbieten — jedoch nicht auf Basis verteilter Systeme.[2] In der Ulmer Oberon-Bibliothek wurden alle Elemente, die für ein entsprechendes System notwendig sind, klar separiert und so allgemein und kooperativ wie möglich gehalten. So sind Synchronisierung (Bedingungen und der Warte-Algorithmus), Synchronisierungsmechanismen (die Implementierungen verschiedener Bedingungen, z.B. *LocalSemaphores*), Maßnahmen zur Parallelisierung (z.B. Koroutinen oder verteilte Systeme) und Scheduling (*Tasks* und *Schedulers*) getrennt definiert und implementiert worden.

Die Trennung zwischen Synchronisierungsmechanismen und Schedulern hat natürlich einen Preis. In konventionellen Systemen wird beim Generieren eines Ereignisses festgestellt, ob und wer darauf wartet, und ggf. werden einige Tasks aus Warteschlangen in die Schlange der rechenbereiten Tasks überführt. Dieses Verfahren kann effizienter sein als das teilweise wiederholte Testen von Bedingungen. Allerdings zwingt es dazu, daß das Generieren von Ereignissen untrennbar mit Scheduling verbunden ist. Damit ist die Unabhängigkeit von der Art der Parallelisierung und die Synchronisierung mit Ereignissen in der Außenwelt mehr möglich.

Bei genauerer Betrachtung zeigt sich, daß der zu zahlende Preis nicht allzu hoch ist. Er hängt ab von der Anzahl der Bedingungen, die in Warteanweisungen (z.B. *Tasks.WaitFor*) angegeben worden sind. Bei *Tasks* finden Tests im Rahmen von Taskgruppen statt, die eine Untermenge aller vorhandenen Tasks sein können und bei *Conditions* können — wie gezeigt — innerhalb des Warte-Algorithmus eine Reihe von Bedingungen aussortiert werden. Die Testprozeduren selbst bestehen normalerweise nur aus sehr wenigen Instruktionen und benötigen entsprechend kaum Rechenzeit. Die Mechanismen lassen sich problemlos über verteilte Systeme skalieren, da jeder Prozeß nur seine eigenen Bedingungen testen muß.

Die Abbildung 41 gibt einen Überblick über die wichtigsten Module aus diesem Bereich. Die Bedingungen und der Warte-Algorithmus (*Conditions*) ließen sich vollkommen unabhängig von der Technik der Parallelisierung formulieren und sind somit für alle Bereiche anwendbar. Auf der Basis von Bedingungen konnte eine Reihe von Synchronisierungsmechanismen vorgegeben werden wie etwa *EventConditions*, *TimeConditions* und *StreamConditions*. *Coroutines* stellt einen Teil der Schnittstelle zu Koroutinen zur Verfügung (der andere Teil wird von **SYSTEM** exportiert). Darauf aufbauend konnten allgemeine Abstraktionen für Tasks und Scheduler entwickelt werden. Ein konkreter Scheduler ist mit *RoundRobin* gegeben.

Als Beispiel für die Möglichkeit der Synchronisierung mit der Außenwelt finden sich in der Hierarchie einige Implementierungen von *StreamConditions*. *UnixFileConditions* definiert den Bereich aller Bedingungen, die mit UNIX-Dateideskriptoren zu tun haben, und implementiert die zugehörige *Select*-Operation für *Conditions*, die auf *SysSelect* basiert. *UnixFileConditions* wird in kooperativer Weise von allen *Stream*-Implementierungen

---

[1] *SysSelect* offeriert die Schnittstelle zu dem UNIX-Systemaufruf *select*, der es erlaubt, auf eine Menge von Ereignissen im Ein- und Ausgabebereich zu warten.

[2] Es gibt natürlich Programmiersprachen und Bibliotheken, die verteilte Systeme unterstützen (wie z.B. Emerald). Allerdings ist dem Autor kein einziges System bekannt, das den gesamten Bereich in erweiterbarer Weise abdeckt.

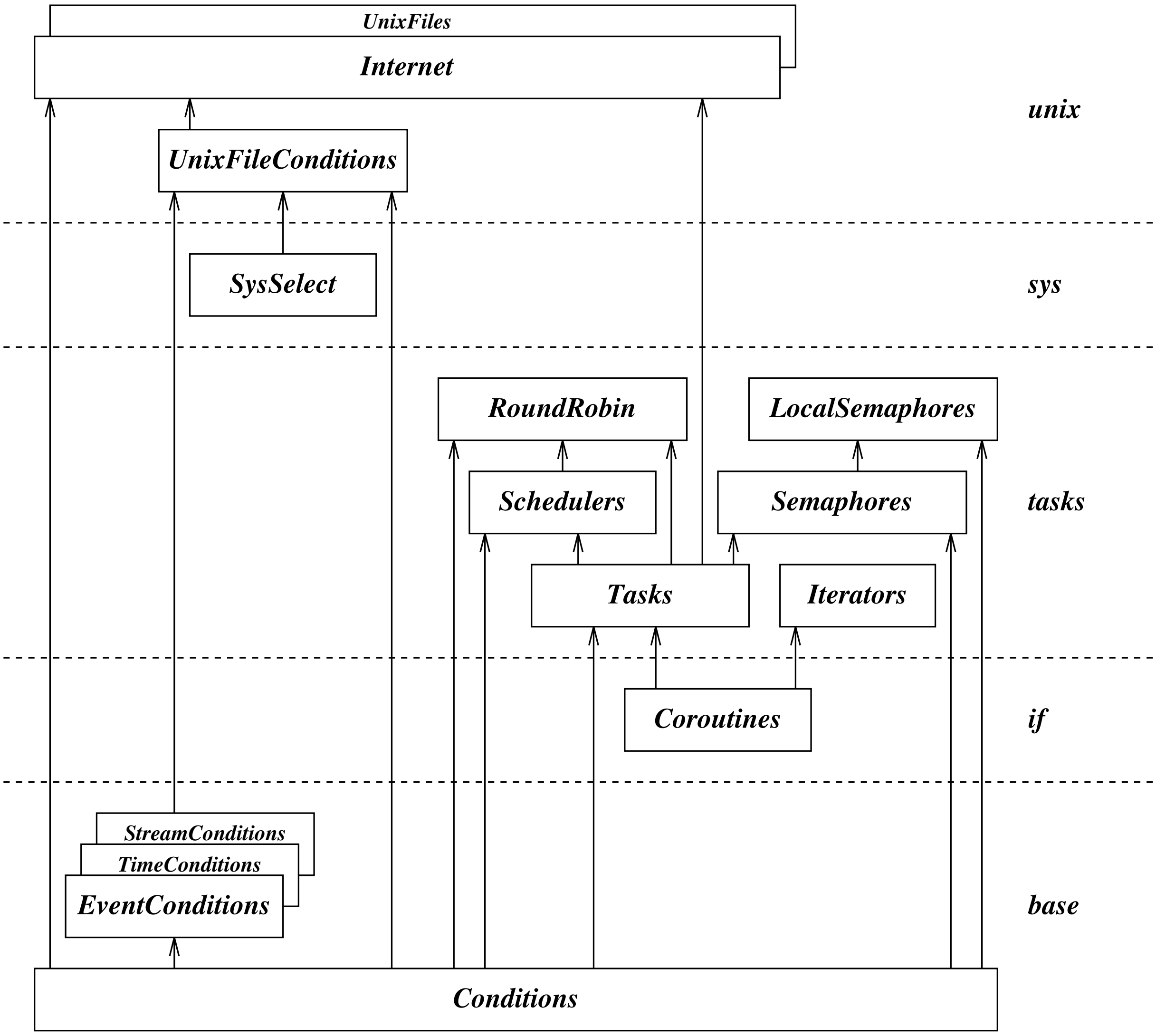

Abb. 41: Modulhierarchie für Parallelität

verwendet, die auf UNIX-Dateideskriptoren basieren, wie z.B. *UnixFiles* und *Internet*.

Beim Entwurf von Grundoperationen zur Parallelisierung und Synchronisierung stellt sich zunächst die entscheidende Frage, ob sie auf Asynchronität oder Synchronität beruhen sollen. Viele Programmiersprachen bieten aus guten Grund sofort Synchronität an wie z.B. Ada und CSP, während vergleichsweise wenige Sprachen asynchrone Operationen offerieren wie z.B. Actors. Grundsätzlich kann man synchrone Operationen auf asynchronen aufbauen und umgekehrt. Während die Beweisbarkeit und Verstehbarkeit auf Basis von synchronen Operationen erleichtert wird, sind die Vorgänge auf der untersten Ebene asynchron.

Man kann asynchrone und synchrone Operationen als zwei Abstraktionsebenen betrachten, wobei die synchronen Operationen auf den asynchronen Operationen aufbauen. Während bei einer Programmiersprache die asynchrone Abstraktionsebene zum Verschwinden gebracht werden kann, muß sie in einer Bibliothek irgendwo existieren. In der Ulmer Oberon-Bibliothek sind dies die Bedingungen, und darauf lassen sich — wie gezeigt — Mechanismen aufbauen, die den gewohnten synchronen Operationen entsprechen.

8. Persistenz

8.1. Einführung

In den klassischen Programmiersprachen wie z.B. Modula-2 oder C wird die räumliche und zeitliche Lokalisierung von Daten innerhalb recht enger Grenzen festgelegt. So sind die Daten unmittelbar mit dem Ablauf eines Programms verbunden, da sie aus der Sicht der Programmiersprache weder das Programm überdauern noch zu einem anderen Programm, das gleichzeitig läuft, übertragen werden können.

Natürlich lassen sich diese Schwächen teilweise mit Hilfe von einfacher Ein- und Ausgabe auf Dateien oder Kommunikationskanälen überwinden. Jedoch sind diese Techniken relativ umständlich, fehleranfällig und unzureichend für größere dynamische Datenstrukturen. Speziell bei objekt-orientierten Programmiersprachen kommen noch die Schwierigkeiten des Typsystems hinzu, die auf konventionelle Weise nicht gelöst werden können.

Es gibt einige Ansätze, um diese Problematik zu lösen:

8.1.1. Persistenz auf Basis von Spracherweiterungen

Bei einigen Sprachen wurde das Typsystem orthogonal in Richtung von Persistenz und Datenbanken erweitert, wie beispielsweise in Pascal/R [Jarke82], PS-algol [Atkinson83], Modula/R [Reimer83, Koch83], DBPL [Eckhardt85] und der C++-Erweiterung E [Richardson93]. Einige dieser Varianten bieten neben der eigentlichen Persistenz auch eine Erweiterung der Ausdrücke von der Aussagenlogik zur Prädikatenlogik der ersten Stufe an. Einen Überblick hierüber gibt [Atkinson87]. Kennzeichnend für diese Lösungen ist, daß sie alle von relativ umfangreichen Laufzeitbibliotheken begleitet sind, deren Aufrufe vom Übersetzer generiert werden.

8.1.2. Persistenz auf Basis von Systembibliotheken

Einige Lösungen verzichten auf die Erweiterung der Sprache und fügen statt dessen Module oder Klassen hinzu, die alle notwendigen Operationen definieren. Die Implementierung dieser Module oder Klassen hängt weitgehend von den internen Speicherverwaltungsstrukturen ab, die vom Übersetzer generiert worden sind. Diesem Ansatz entsprechen zahlreiche objekt-orientierte Programmiersprachen wie beispielsweise Eiffel [Meyer88, §15.5.6], Modula-3 [Nelson91, §3.6], BETA [Agesen93] und EuLisp in Verbindung mit Telos [Broadbery93].

8.1.3. Persistenz auf Basis von Betriebssystemen

Prozesse und der Adreßraum unterliegen der Verwaltung durch ein Betriebssystem. Während bei konventionellen Betriebssystemen Prozesse und Daten beim Ende eines Prozesses oder spätestens beim Herunterfahren des Systems verlorengehen, gibt es einige verteilte Betriebssysteme, die auf Basis von virtuellen Speichertechniken erlauben, Objekte langfristig an bestimmten Adressen abzulegen. Damit entfällt die Notwendigkeit, die Objekte zu transferieren. Statt dessen findet jeder Prozeß, der die notwendigen Zugangsrechte hat — unabhängig von seiner eigenen Lokalisierung — das Objekt an seiner festen Adresse.

8.2. Persistenz auf Basis von systemunabhängigen Bibliotheken

Wenn kein spezielles Betriebssystem zur Verfügung steht, die Programmiersprache nicht unnötig erweitert werden sollte und die Unabhängigkeit vom Übersetzer wünschenswert ist, dann muß nach einer Verfeinerung konventioneller Techniken gesucht werden. Im Umfeld objekt-orientierter Programmiersprachen liegt es nahe, die Objekte selbst dafür die Verantwortung übernehmen zu lassen in Form von zwei zusätzlichen Methoden zur Ein- und Ausgabe analog zu *encode* und *decode* des Ansatzes von [Herlihy82].

Bei diesem Ansatz gibt es zwei schwerwiegende Probleme:

- Beim Schreiben ist — dank *late binding* — die Lokalisierung der richtigen Prozedur kein Problem. Welche Prozedur ist jedoch für das Einlesen zu verwenden? Vor dem Aufruf der Einlese-Operation liegt das Objekt selbst noch nicht vor, und somit gibt es keine Möglichkeit, die Technik des *late binding* auszunutzen.
- Speziell in Oberon gibt es nicht notwendigerweise eine zentrale Instanz, die ein Objekt vollständig kennt. Entsprechend müssen beim Ausgeben und Einlesen eines Objektes alle Module beteiligt werden, die zur Typhierarchie des Objekts gehören.

8.2.1. Die Erhaltung der Typinformation

Auf den ersten Blick ließe sich der erste Punkt dadurch entkräften, daß man die Größe des dynamischen Objekts ermittelt und es dann einfach binär aus dem Speicher kopiert. Beim Einlesen könnte man natürlich entsprechend Speicher anlegen und das Objekt wieder zurückkopieren. Abgesehen davon, daß sich selbst diese Operationen in einigen Programmiersprachen nicht auf portable Weise durchführen lassen, wird bei diesem Verfahren die systemspezifische Typinformation nicht gerettet.

Alle objekt-orientierten Programmiersprachen benötigen systemspezifische Informationen zu einem Objekt, die es zur Laufzeit ermöglichen, die richtige Operation (per *late binding*) aufzurufen, Typentests durchzuführen und die automatische Speicherverwaltung arbeiten zu lassen. Diese Typinformationen sind weder notwendigerweise Bestandteil des eigentlichen Objekts, noch ist sichergestellt, daß sie ein Programm, das sie später liest, in irgendeiner Form wiederverwerten kann.

Wenn Objekte wieder einzulesen sind, dann sollte man auf die in der Sprache vorgesehenen Methoden, neue Objekte zu kreieren, zurückgreifen. In Oberon kommt dazu nur **NEW** in Frage. Damit **NEW** das gewünschte Objekt erzeugen kann, muß der Aufrufer von **NEW** bereits den Typ kennen und ihn statisch als Typ deklariert haben.

Bei dem Züricher Oberon-Projekt wurde bei der Entwicklung des erweiterbaren Editors *Write* eine vergleichsweise einfache Technik entwickelt, um dieses Problem zu lösen [Szyperski92b]: Jedes Modul, das einen persistenten Datentyp für *Write* zur Verfügung stellt, besitzt ein *command* (Kommando: eine parameterlose exportierte Prozedur, die aufgerufen werden kann, wenn der Modulname und der Prozedurname z.B. in Form von Zeichenfeldern bekannt sind), das ein Objekt des exportierten Datentyps einlesen kann. Bei der Ausgabe wird zunächst der Modul- und Prozedurname des Kommandos ausgegeben und erst dann die Informationen, die zum Objekt selbst gehören. Beim Einlesen ist es dann nur noch notwendig, den Namen des Kommandos einzulesen und es aufzurufen.

Dieser Technik verbleiben nur noch zwei Nachteile: Kommandos müssen über globale Variablen parametrisiert werden, und ein einzelnes Kommando zum Einlesen muß das gesamte Objekt kennen. Die Verwendung von Kommandos läßt sich vermeiden, wenn statt dessen jedes Modul, das einen persistenten Datentyp exportiert, eine parametrisierte Einlese-Prozedur registriert.

8.2.2. Verteilte Ein- und Ausgabe

Spätestens bei der notwendigen Verteilung der Ein- und Ausgabe wird deutlich, daß man eine zentrale Instanz benötigt, die Registrierungen entgegennimmt und auf der anderen Seite die öffentlichen Ein- und Ausgabe-Operationen für persistente Objekte anbietet.[1] Das Registrieren und das Zuordnen beim Einlesen basiert auf Namen, wobei es am zweckmäßigsten ist, die Typnamen selbst zu verwenden.

Wenn bei der Registrierung nicht nur der Name des eigentlichen Typs, sondern zudem auch noch der des Basistyps mitgegeben wird, dann ist die zentrale Instanz in der Lage, die Typhierarchie zusammenzustellen. Bei der Aus- und Eingabe kann dann nicht nur die Operation, die direkt zu dem angegebenen Typ gehört, aufgerufen werden, sondern auch noch die Operationen für all die übergeordneten Typen der Hierarchie.

8.2.3. Die Verwendung qualifizierter Typnamen

Sinnvollerweise sollten die Typnamen voll qualifiziert angegeben werden (d.h. Modulname, gefolgt von einem Trenner und dem eigentlichen Typnamen). Dies stellt die Eindeutigkeit sicher (soweit die Modulnamen selbst eindeutig sind) und erlaubt auch die Technik des dynamischen Nachladens oder späten Initialisierens.

Der wesentliche Vorteil dieser Technik ist, daß die beiden Parteien (das schreibende und das lesende Programm) weder zum gleichen Programm gehören müssen, noch notwendigerweise auf dem gleichen Rechner bzw. Rechnerarchitektur laufen. Es muß bei beiden Parteien nur das gleiche Modul im Quelltext vorgelegen haben, und die Kodierung der Daten muß natürlich rechnerunabhängig erfolgen. Selbst unterschiedliche Versionen eines Moduls und des Datentyps sind kein Problem, wenn beispielsweise eine Versionsnummer mit zur Kodierung gehört und neuere Modulversionen in der Lage sind, ältere Kodierungen zu lesen.

---

[1] Auf Basis von Überdefinitionen bzw. Verfeinerungen im Sinne klassischer OO-Techniken ließe sich dies teilweise vermeiden. Günstiger wäre hier allerdings die Technik von BETA, bei der die Kontrolle bei den Basisklassen beginnt und dann mit **INNER** an die abgeleiteten Klassen delegiert wird. Auf jeden Fall empfiehlt es sich, die Typinformationen zentral zu verwalten und entsprechend auch zentral Operationen für die Ein- und Ausgabe anzubieten — in BETA könnte dies die Basisklasse für persistente Objekte übernehmen.

Natürlich steht und fällt diese Technik mit der Eindeutigkeit von Modulnamen. Einige Programmiersysteme (z.B. Lace, das zu Eiffel gehört [Meyer92, §D.10]) erlauben es, mehrere unterschiedliche Module mit dem gleichen Namen in ein Programm einzubinden. In diesen Fällen müßte man auf einen anderen Namensraum zurückgreifen, der jedem Modul einen eindeutigen Namen zuordnet. Dies führt jedoch zu einer Reihe von neuen Schwierigkeiten, die kaum ohne spezielle Systemunterstützung gelöst werden können. Insbesondere wird dann fraglich, ob auch noch nicht registrierte Typen eingelesen werden können, deren zugehörige Module nachträglich geladen oder initialisiert werden müßten.

Sowohl die Züricher Oberon-Systeme als auch das Ulmer Oberon-System garantieren (innerhalb eines Programms) die Eindeutigkeit von Modulnamen, so daß qualifizierte Typnamen genügend eindeutig sind. Natürlich kann es dann immer noch passieren, daß zwei getrennt entwickelte Programme ein Modul des gleichen Namens besitzen, das persistente Datentypen exportiert. Das kann dazu führen, daß versucht wird, Objekte einzulesen, die sonst nicht akzeptiert werden würden.

8.2.4. Zusammenhängende Objekte

Das Laufzeitsystem von Smalltalk ermöglicht es, das gesamte System zu sichern (in Form eines sogenannten *snapshot* oder *image*) und zu einem späteren Zeitpunkt wieder zu laden [Goldberg84 §§1.4,23.4]. Dieses Verfahren ist jedoch nicht geeignet für das Abspeichern von Daten, die zu einem eingegrenzten Bereich gehören, oder gar zur Interprozeßkommunikation.

Eiffel [Meyer92, §30] und Modula-3 [Nelson91, §3.6] offerieren systemspezifische Operationen, die ab einem gegebenen Zeiger alle davon abhängigen Objekte abspeichern. Allerdings ist das jeweilige Format systemspezifisch und daher nicht zur Interprozeßkommunikation geeignet.

Bei systemunabhängigen Bibliotheken steht dieser Weg nicht zur Verfügung. Damit verbleibt die Verantwortung, zusammenhängende Objekte zu serialisieren, zu schreiben und später wieder einzulesen und richtig miteinander in Verbindung zu bringen, beim Programmierer. Wenn man jedoch bedenkt, daß zusammengehörende Objekte häufig mit Hilfe von Abstraktionen zu Sammlungen zusammengefaßt sind, dann ist klar, daß der Aufwand nur je einmal für die unterschiedlichen Sorten von Sammlungen investiert werden muß. Dann kann eine Sammlung von Objekten selbst als persistentes Objekt betrachtet werden.

Im Unterschied zu den Techniken in Eiffel und Modula-3 sind dann die einzelnen Module, die persistente Datentypen exportieren, in der Lage, zwischen temporären und für die Persistenz interessanten Komponenten zu unterscheiden. Denn häufig gibt es Zeiger, die zwar im laufenden Programm zur Verwaltung wichtig sind, jedoch für die Partei, die die Objekte später einliest, uninteressant sind.

Als Beispiel könnte man annehmen, daß eine Datenstruktur einen Zeiger auf einen geöffneten Kanal besitzt, der auf eine wohlbekannte Datei verweist. Bei der Technik von Eiffel und Modula-3 würden dann nicht nur eigentliche Datenstruktur, sondern auch noch die internen Daten von *Streams* mitgesichert werden. Da *Streams* eine interne Verkettung aller Kanäle unterhält, würde das zur Sicherung sämtlicher gepufferter Daten führen.

8.2.5. Die Identität von Objekten

In den gängigen objekt-orientierten Programmiersprachen gibt es keine Objekt-Identität (wenn man von der Lokalisierung absieht). Persistente Objekte können durch das Sichern und ggf. wiederholte Einlesen geklont werden. Solange keine Identitäten extra geschaffen werden, läßt sich nachher nicht mehr feststellen, ob zwei Objekte durch Klonen aus dem gleichen Ursprungsobjekt hervorgegangen sind oder nicht.

Im Bereich langfristiger Datenhaltung und Datenbanken kann eine Objekt-Identität jedoch sehr nützlich sein. Wenn es möglich ist, aufgrund der Objekt-Identität ein Objekt zu finden und zu laden, dann müssen Objekte nicht immer überall mitkopiert und mittransportiert werden, um zur Verfügung zu stehen. Statt dessen sollte es dann nur noch an einer einzigen Stelle existieren, wenn man von Sicherungskopien absieht. In einem verteilten System müßten dann ggf. Operationen über das Netz zum dem identifizierten Objekt weitergeleitet werden.

Im Züricher Oberon-System 3 lassen sich alle persistenten Objekte genau einer Bibliothek (*library*) und innerhalb der Bibliothek genau einem Index zuordnen [Gutknecht93b]. Das Tupel (Bibliotheksname, Index) dient als identifizierender Schlüssel für Objekte. Bibliotheken dienen dann als eine Sammlung von Objekten, die in ihrer Gesamtheit eingelesen oder ausgelagert werden können.

Das Konzept der Bibliothek ist zwar sehr geeignet als Kollektion von Objekten, die direkt dem Endanwender zugänglich sind, jedoch bleiben viele Anwendungsgebiete verschlossen, da keine Trennung zwischen persistenten und identifizierbaren Objekten existiert. So sind beispielsweise identifizierbare Objekte nicht zur Interprozeßkommunikation geeignet. Andererseits ist es jedoch kein Problem, auf Basis von persistenten Objekten eine Abstraktion für identifizierbare Objekte zu schaffen.

8.3. Die Schnittstelle von *PersistentObjects*

Entsprechend dem vorgestellten Ansatz einer systemunabhängigen Unterstützung für persistente Objekte definiert sich Persistenz als die Möglichkeit, ein Objekt in eine Sequenz von Bytes zu verwandeln und umgekehrt. Um die dazu benötigten Operationen mit den Objekten zu verbinden, ist es am günstigsten, einen speziellen Basistyp (*PersistentObjects.Object*) zu verwenden. Natürlich wäre es auch denkbar, dies über Disziplinen zu erreichen. Da Persistenz jedoch eine statische Eigenschaft eines Objekttyps ist, würde dies keinen Vorteil bringen.

Es empfiehlt sich in jedem Fall, *PersistentObjects.Object* als Erweiterung von *Objects.Object* zu definieren, um die Kompatibilität zu den Operationen zu gewährleisten, die nichts über ein Objekt voraussetzen. *PersistentObjects.Object* wurde auch als Erweiterung von *Disciplines.Object* definiert, obwohl die Disziplinen von *Disciplines* nicht persistent sind und auch nicht nachträglich dazu gemacht werden sollten. In der Praxis zeigt sich, daß es sehr sinnvoll ist, sowohl temporäre als auch persistente Disziplinen bei persistenten Objekten zu haben. Aus diesem Grund kam noch ein zusätzliches Modul für persistente Disziplinen (*PersistentDisciplines*) hinzu.

Auf Basis dieser Vorüberlegungen wurden von Detlef Birkholz im Rahmen einer Diplomarbeit *PersistentObjects* und *PersistentDisciplines* entwickelt [Birkholz94]. Module, die einen persistenten Datentyp exportieren, müssen einerseits zu ihrer Initialisierungszeit den Typ registrieren und zum anderen jedes neu angelegte Objekt dieses Typs mit der Registrierung verbinden:

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

CONST
   read = 0; write = 1; sizeOf = 2; (* capabilities *)
TYPE
   CapabilitySet = SET;
   ReadProc = PROCEDURE (s: Streams.Stream; o: Object) : BOOLEAN;
   WriteProc = PROCEDURE (s: Streams.Stream; o: Object) : BOOLEAN;
   SizeOfProc = PROCEDURE (s: Streams.Stream; o: Object) : Streams.Count;

   Interface = POINTER TO InterfaceRec;
   InterfaceRec =
      RECORD
         (Objects.ObjectRec)
         read: ReadProc;
         write: WriteProc;
         sizeOf: SizeOfProc;
      END;

TYPE
   CreateProc = PROCEDURE (VAR o: Object);
   Datatype = POINTER TO DatatypeRec;
   DatatypeRec = RECORD (Disciplines.DisciplineRec) END;

PROCEDURE RegisterType(name, baseName: ARRAY OF CHAR;
                       create: CreateProc;
                       if: Interface;
                       caps: CapabilitySet) : Datatype;

PROCEDURE Connect(object: Object; type: Datatype);
```

Neben *read* und *write* gehört zur Schnittstelle auch die optionale *sizeOf*-Prozedur. Da es durchaus denkbar ist, daß die lesende Partei einen Typ nicht kennt, sollte sie in der Lage sein, das unbekannte Objekt (bzw. einen Teil des Objektes bei Projektionen, die u.U. unterstützt werden) zu überlesen. Dies geschieht sinnvollerweise durch eine Längenangabe vor dem eigentlichen Objekt. Wenn *PersistentObjects* die persistenten Objekte auf einen Kanal ausgibt, der Positionierungen zuläßt, dann ist es natürlich kein Problem, nach der Ausgabe eines Objekts rückwärts zu positionieren und die Längenangabe "hineinzuflicken". Bei Kanälen ohne Möglichkeit zur Positionierung muß die Länge im voraus bekannt sein. Sie kann entweder umständlich durch das Absichern in einen temporären Puffer ermittelt werden oder eben auf Basis der optional vorhandenen Schnittstellenprozedur *sizeof*. Das funktioniert allerdings nur, wenn bei einem verteilten Sicherungsszenario alle Parteien dies unterstützen.

Die bei *RegisterType* übergebene *create*-Prozedur ist dafür verantwortlich, ein Objekt des persistenten Typs zu kreieren. Dies kann nicht durch die *read*-Prozedur geschehen, da das Einlesen verteilt von einer Reihe von Prozeduren übernommen wird entsprechend der Typhierarchie. Gesichert und gelesen wird in der Reihenfolge der Basistypen, d.h. zuerst die direkte Erweiterung von *PersistentObjects.Object*, dann die Erweiterung davon usw. Diese Reihenfolge ermöglicht es, Projektionen durchzuführen, wenn beispielsweise der Typ selbst zwar nicht bekannt ist, jedoch ein Basistyp davon. Da das Modul, das beim Lesen zuerst aufgerufen wird, den Typ der Erweiterung nicht kennt, kann es das Objekt auch nicht kreieren. Aus diesem Grund muß das Objekt frühzeitig entsprechend der Typinformation angelegt werden, bevor irgendwelche Objektdaten eingelesen werden.

Die Ein- und Ausgabe persistenter Objekte erweist sich dann als relativ einfach:

```
PROCEDURE Read(s: Streams.Stream; VAR object: Object) : BOOLEAN;
PROCEDURE Write(s: Streams.Stream; object: Object) : BOOLEAN;
```

8.3.1. Ein Beispiel für einen persistenten Datentyp

Als einfaches Beispiel kann ein Literaturhinweis dienen:

```
TYPE
   Book = POINTER TO BookRec;
   BookRec =
      RECORD
         (PersistentDisciplines.ObjectRec)
         title: ARRAY 80 OF CHAR;
         author: ARRAY 20 OF CHAR;
         year: INTEGER;
      END;
```

Während der Initialisierungsphase ist der Typ zu registrieren:

```
MODULE Books;

   (* ... *)

   VAR
      bookType: PersistentObjects.Datatype;
      if: PersistentObjects.Interface;
      caps: PersistentObjects.CapabilitySet;

   (* ... *)

BEGIN
   if.read := ReadBook;
   if.write := WriteBook;
   if.sizeOf := SizeOfBook;
   caps := {PersistentObjects.read, PersistentObjects.write,
            PersistentObjects.sizeOf};
   bookType := PersistentObjects.RegisterType(
                  "Books.Book", "PersistentDisciplines.Object",
                  CreateBook, if, caps);
END Books.
```

CreateBook braucht nur **NEW** aufzurufen und das Objekt mit *bookType* zu assoziieren:

```
PROCEDURE CreateBook(VAR object: PersistentObjects.Object);
   VAR
      book: Book;
BEGIN
   NEW(book);
   PersistentObjects.Connect(book, bookType);
   object := book;
END CreateBook;
```

Für die Ein- und Ausgabe der Basistypen von Oberon speziell zum Zwecke der Kommunikation gibt es das Modul *NetIO*. Die Operationen von *NetIO* hängen vom jeweiligen Kanal ab.[1] So ist es möglich, zwischen zwei

[1] Dies ist möglich mit Hilfe einer entsprechenden Disziplin.

Prozessen auf unterschiedlichen Rechnerarchitekturen Konvertierungen durchzuführen, während zum gleichen Zeitpunkt Ein- und Ausgaben zu einem anderen Kanal ohne Konvertierungen stattfinden. Auf dieser Basis läßt sich dann relativ einfach *ReadBook* und *WriteBook* realisieren:

```
PROCEDURE WriteBook(s: Streams.Stream; object: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH object: Book DO
      RETURN NetIO.WriteString(s, object.title) &
             NetIO.WriteString(s, object.author) &
             NetIO.WriteInt(s, object.year)
   END;
END WriteBook;

PROCEDURE ReadBook(s: Streams.Stream; object: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH object: Book DO
      RETURN NetIO.ReadString(s, object.title) &
             NetIO.ReadString(s, object.author) &
             NetIO.ReadInt(s, object.year)
   END;
END ReadBook;

PROCEDURE SizeOfBook(s: Streams.Stream; object: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH object: Book DO
      RETURN NetIO.SizeOfString(s, object.title) &
             NetIO.SizeOfString(s, object.author) &
             NetIO.SizeOfInt(s, object.year)
   END;
END SizeOfBook;
```

An dem Beispiel sieht man auch, warum es sinnvoll ist, daß *SizeOfBook* nicht nur *object*, sondern auch *s* als Parameter erhält. Da die Kodierung von dem Kanal abhängt, steht auch der Speicherbedarf in Abhängigkeit vom Kanal.

8.3.2. Projektionen

Wenn zwei verschiedene Programme miteinander kommunizieren, ist es denkbar, daß nicht alle persistenten Datentypen des einen Programms dem anderen bekannt sind. Grundsätzlich ist es möglich, mit Hilfe der zu Beginn stehenden Längeninformation ein Objekt unbekannten Typs zu überlesen. Wenn ein Programm ein Objekt erhält, dessen Typ nicht bekannt ist, wäre es jedoch denkbar, daß einer der in der Hierarchie übergeordneten Typen bekannt ist. In diesem Fall könnte analog zu der Zuweisung bei Verbunden eine Projektion durchgeführt werden. Dazu ist es notwendig, daß nicht nur der qualifizierte Typname zu Beginn der Bytefolge steht, sondern die gesamte Typhierarchie zugänglich ist.

Natürlich sind Techniken zur Komprimierung der Typinformationen denkbar, wenn man davon ausgeht, daß eine Reihe von Objekten in einer festgelegten Reihenfolge gelesen werden oder zusätzliche Kommunikationskanäle zur Verfügung stehen [Mössenböck93, §8.3]. Bislang unterstützt *PersistentObjects* jedoch nur zwei Modi: Entweder geht jedem Objekt die vollständige Typenhierarchie voraus (Voreinstellung) oder nur der einzelne Typname. Im letzteren Fall sind später beim Einlesen keine Projektionen mehr möglich.

Neben der Möglichkeit, für einen gegebenen Kanal den Modus zu setzen, kann beim Lesen überprüft werden, ob eine Projektion durchgeführt worden ist, und, wenn ja, wie die zugehörigen Typen heißen:

```
CONST noProjection = 1; useTypeList = 2;
TYPE Mode = SHORTINT; (* noProjection | useTypeList *)

PROCEDURE SetProjectionMode(s: Streams.Stream; mode: Mode);
PROCEDURE GetProjectionMode(s: Streams.Stream; VAR mode: Mode);
PROCEDURE IsProjection(object: Object) : BOOLEAN;
PROCEDURE GetProjectionName(object: Object;
                            VAR trueName, projName: ARRAY OF CHAR);
```

8.4. Verteilte Systeme auf Basis von persistenten Objekten

Ursprünglich wurde Persistenz als die Eigenschaft von Daten definiert, deren Lebensdauer unabhängig von der der zugreifenden Programme ist [Atkinson83, Atkinson87]. Jedoch kann man neben der zeitlichen Unabhängigkeit auch die räumliche Beweglichkeit in Betracht ziehen: Wenn die Repräsentierung von Daten bereits unabhängig von einem einzelnen Programm ist, liegt es nahe, daß es auch möglich ist, Daten zwischen zwei Prozessen auszutauschen, die eventuell sogar auf unterschiedlichen Rechnern laufen. Dies entspricht auch der Definition von Persistenz von Booch [Booch91]:

> ‘‘Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object’s location moves from the address space in which it was created).’’

Leider wird dennoch in vielen Fällen nur eine der beiden Möglichkeiten zur Mobilität unterstützt. So bieten Eiffel und Modula-3 Persistenz nur im Sinne von Langlebigkeit an, während einige Programmiersprachen nur Persistenz im Rahmen verteilter Systeme unterstützen, wie beispielsweise eine Erweiterung von Smalltalk [Decouchant86, Decouchant89] und Emerald [Black87].

Im Rahmen von *PersistentObjects* definiert sich Persistenz als die Eigenschaft, ein Objekt in eine Sequenz von Bytes und wieder zurück verwandeln zu können. Da *PersistentObjects* auf *Streams* basiert, gibt es keine feste Verbindung zu einem Dateisystem oder einem verteilten System. Genauso wie es leicht möglich ist, ein Objekt in eine Datei abzusichern, kann das gleiche Objekt über eine Netzverbindung zu einem anderen Prozeß geschickt werden.

8.4.1. Prozeduraufrufe über Prozeßgrenzen hinweg

Viele verteilte Systeme wie z.B. Cedar [Birrell84] basieren auf der Technik von Prozeduraufrufen, die über Prozeß- und Rechnergrenzen hinweg (*remote procedure calls*) erfolgen können. Üblicherweise werden dabei mehrere Operationen zu einer Schnittstelle zusammengefaßt, die einem Modul nahekommt. Typischerweise werden aus der Schnittstellenbeschreibung zwei Schnittstellenmodule generiert, eines für die Seite des Klienten und eines für die Seite des Servers. Entsprechend der Abbildung 42 verpackt das Schnittstellenmodul für die Seite des Klienten beim Aufruf einer Operation alle Parameter in ein Paket und verschickt es über das Netz zum Server. Auf der Seite des Servers wird im Schnittstellenmodul das Paket wieder dekodiert und die eigentliche Operation durchgeführt. Die Resultate werden wiederum verpackt und zurückgeschickt. Erst wenn der Klient das Antwortpaket erhält, endet der Aufruf der Operation.

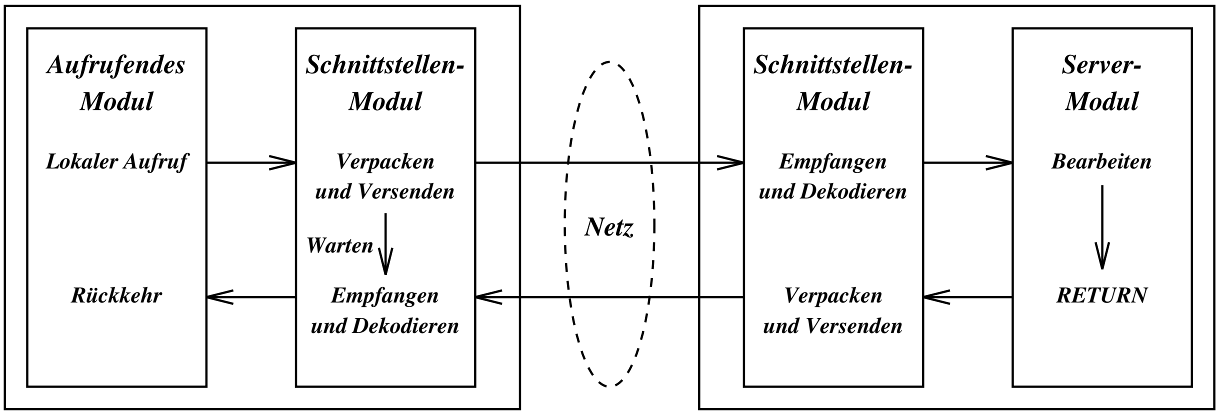

Abb. 42: Prozeduraufrufe über das Netz

Grundsätzlich ließe sich diese Technik relativ leicht auf der Basis von persistenten Objekten realisieren. Die Parameter einer Operation können in Form eines Auftrags zu einem Verbund zusammengefaßt werden. Wenn ein Auftragsverbund persistent ist, dann kann er über eine Netzverbindung gesendet und wieder — mit Erledigungsvermerken versehen — zurückgeschickt werden.

Wie Levy und Tempero in ihrem Beitrag über verteilte Programmierung darlegen [Levy91], hängt der Aufbau und damit die Flexibilität eines verteilten Systems sehr von der Technik ab, mit der Operationen über das Netz erfolgen. Wenn man von Erledigungsvermerken absieht, werden alle Parameter bei entfernten Prozeduraufrufen via *call-by-value* übergeben. So ist es insbesondere nicht ohne weiteres möglich, beliebige Objekte anzugeben, auf denen der Server Operationen durchführen kann.

8.4.2. Verteilte Objekte

Die Alternative ist, nicht bestimmte Operationen als entfernte Prozeduren zu betrachten, sondern anzunehmen, daß die Objekte selbst verteilt sein können. Entscheidend ist dabei, daß der Aufruf einer Operation für ein Objekt unabhängig von dessen Lokalisierung erfolgen sollte.

Bei Emerald wird diese Technik vom Laufzeitsystem unterstützt. In Oberon muß die Weiterleitung einer Operation zum Zielobjekt von der Bibliothek aus erfolgen. Dies geschieht durch den Einsatz von Stellvertreter-Objekten (siehe Abbildung 43). Ein Stellvertreter-Objekt kann als lokaler Repräsentant eines Objektes betrachtet werden, das zu einem anderen Prozeß (ggf. sogar auf einem anderen Rechner) gehört. Jeder Auftrag, der bei einem Stellvertreter-Objekt eintrifft, wird über eine Netzverbindung zum eigentlichen Objekt weitergeleitet. Anschließend wartet das Stellvertreter-Objekt auf den Empfang des zurückgesendeten Auftrages mit den Erledigungsvermerken.

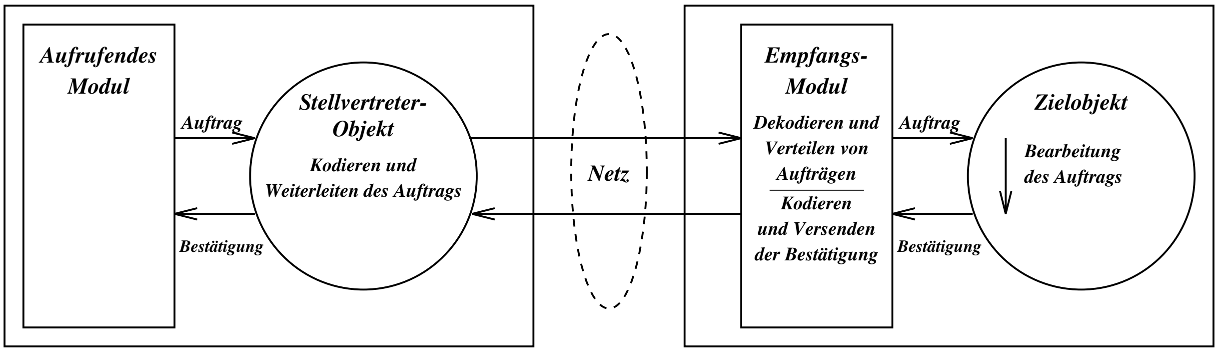

Abb. 43: Versenden von Aufträgen an ferne Objekte

Die darunterliegende Technik ähnelt zwar teilweise den entfernten Prozeduraufrufen, jedoch sind folgende Unterschiede wesentlich:

- Bei dem Aufruf einer entfernten Prozedur ist statisch festgelegt, daß der Server über eine Netzverbindung erreicht wird. Damit ist normalerweise eine statische Struktur des verteilten Systems gegeben.
- Es ist viel schwerer, einen abstrakten Datentyp auf Basis von entfernten Prozeduraufrufen zu realisieren. Es ist unklar, auf welcher Basis ein Zustand für einen Klienten bei einem Server unterhalten werden soll, wenn nur gelegentlich Auftragspakete eintreffen. So erfährt der Server nicht ohne weiteres, daß einer der Klienten beendet worden ist.

Die Technik auf Basis von Stellvertreter-Objekten ist vergleichbar zu den der *proxy objects* im Hermes-System von Black und Artsy, das auf Modula-2+ und einer vorhandenen Bibliothek für entfernte Prozeduraufrufe basiert [Black90]. Im Unterschied zu Modula-2+ können in Oberon Parameterlisten als Objekte “erster Klasse” behandelt werden, d.h. es ist möglich, sie in Auftragsverbunde zu verpacken. Wenn in Modula-2+ das sichere Typsystem nicht verlassen werden soll, ist man genötigt, für jeden Objekt-Typ ein spezielles Stellvertreter-Modul generieren zu lassen. Ferner muß grundsätzlich das Stellvertreter-Modul aufgerufen werden, auch wenn sich das Objekt lokal aufhalten könnte.

8.4.3. Persistente Aufträge mit verteilter Verantwortung

Im Rahmen der Ulmer Oberon-Bibliothek sind persistente Aufträge die Voraussetzung für verteilte Objekte. *Messages* exportiert einen entsprechenden Basistyp für Aufträge:

```
TYPE
   Message = POINTER TO MessageRec;
   MessageRec =
      RECORD
         (PersistentObjects.ObjectRec)
         processed: BOOLEAN;
         done: BOOLEAN;
         errors: RelatedEvents.Object;
      END;

PROCEDURE Init(message: Message);
```

Grundsätzlich sollten Aufträge ignoriert werden, wenn ein Auftragsbearbeiter mit einem Auftrag nichts anfangen kann. Aus diesem Grund gibt es einige Erledigungsvermerke für den Auftraggeber, damit ersichtlich ist, ob der Auftrag angenommen worden ist, und wenn ja, ob er erfolgreich ausgeführt werden konnte. Ein Auftragsbearbeiter, der sich um einen Auftrag kümmert, sollte zunächst *processed* auf **TRUE** setzen und später in Abhängigkeit vom Erfolg der Operation entweder *done* den Wert **TRUE** zuweisen oder die Ereignisse aufgrund von Ausnahmenbehandlungen in Bezug zu *errors* setzen. Die Prozedur *Init* initialisiert diese Komponenten.

Objektbezogene Ereignisse werden von *RelatedEvents* über eine lokale nicht-persistente Disziplin verwaltet. Um lokal relevante Ereignisse von für einen Kommunikationspartner interessanten Ereignissen unterscheiden zu können, haben Aufträge eine *errors*-Komponente für das Aufsammeln von auftragsbezogenen Ereignissen. Das Modul *ForeignEvents* erlaubt es, Ereignisse zu kodieren und später wieder zu dekodieren. Allerdings kann *ForeignEvents* nur bei besonderer Unterstützung des zugehörigen Ereignistyps das Ereignis mit all seinen speziellen Erweiterungen festhalten. Liegt eine Unterstützung nicht vor, so wird nur die ausführliche Fehlermeldung erhalten.

Wenn es maximal nur einen Auftragsbearbeiter für ein Objekt geben kann, dann können aktive Disziplinen (d.h. Disziplinen, die Aufträge ausführen können), nicht unterstützt werden. Von daher ist es angemessen, daß einem Objekt beliebig viele Auftragsbearbeiter zugeordnet werden können:

```
TYPE Handler = PROCEDURE (object: PersistentDisciplines.Object; message: Message);
PROCEDURE InstallHandler(object: PersistentDisciplines.Object; handler: Handler);
```

Wichtig ist, daß *InstallHandler* von einem Modul aus aufgerufen wird, das entweder direkt zu der Typhierarchie des Objekts gehört oder eine Disziplin an das Objekt hängt. Der Aufruf muß entweder direkt oder indirekt vom Konstruktor aus bzw. beim Erzeugen der Disziplin erfolgen. Dies stellt sicher, daß beim Dekodieren des Objekts *InstallHandler* wiederum aufgerufen wird.

Das Verschicken eines Auftrages erfolgt dann mit *Send*:

```
PROCEDURE Send(object: PersistentDisciplines.Object; VAR message: Message);
```

Abbildung 44 zeigt die zugehörige Datenstruktur zusammen mit den Sichtbarkeitsgrenzen. *Messages* hängt seine eigene Datenstruktur als temporäre Disziplin an das Objekt, während die zusätzlichen Auftragsbearbeiter zu persistenten Disziplinen gehören. Wenn ein Auftrag an *Messages.Send* übergeben wird, werden alle Auftragsbearbeiter der Reihe nach aufgerufen. Durch die Erledigungsvermerke kann festgestellt werden, ob einer der bereits aufgerufenen Auftragsbearbeiter schon aktiv geworden ist.

Der Parameter *message* muß ein **VAR**-Parameter sein für den Fall, daß *object* ein Stellvertreter-Objekt ist und das eigentliche Zielobjekt in einem anderen Prozeß lebt. In diesem Fall wird die Rückantwort von *PersistentObjects* neu kreiert und dann eingelesen. Natürlich besteht dadurch das Risiko, daß eine andere Erweiterung von *Messages.Message* zurückkommt. Dies kann allerdings nur durch Programmierfehler bei der anderen Partei passieren und läßt sich ggf. lokal durch Typentests überprüfen.

Im Beispiel des Studenten (siehe §1.8) könnten auf dieser Basis die Operationen für einen Studenten realisiert werden:

```
DEFINITION Students;

   IMPORT PersistentDisciplines;

   TYPE
      Student = PersistentDisciplines.Object;
      Name = ARRAY 64 OF CHAR;
      Address = ARRAY 128 OF CHAR;
      Course = ARRAY 32 OF CHAR;

   PROCEDURE Create(VAR student: Student;
                    name: Name; address: Address;
                    course: Course);

   PROCEDURE GetName(student: Student; VAR name: Name) : BOOLEAN;
   PROCEDURE SetName(student: Student; name: Name) : BOOLEAN;

   PROCEDURE GetAddress(student: Student; VAR address: Address) : BOOLEAN;
   PROCEDURE SetAddress(student: Student; address: Address) : BOOLEAN;
```

```
      PROCEDURE GetCourse(student: Student; VAR course: Course) : BOOLEAN;
      PROCEDURE SetCourse(student: Student; course: Course) : BOOLEAN;

   END Students.
```

Im Vergleich zu den anderen Techniken in Oberon liegt hier nicht mehr das gleiche Maß an statischer Typsicherheit vor. Da *Students* sowohl für lokal vorhandene *Student*-Objekte als auch für Stellvertreter-Objekte auf gleiche Weise funktionieren soll, wird der Typ der Erweiterung nicht bekanntgegeben. Somit gibt es auch keine Möglichkeiten zu sinnvollen Typentests (*object* **IS** *Students.Student* ist genau dann wahr, wenn *object* eine Erweiterung von *PersistentDisciplines.Object* ist) noch die statische Sicherheit, daß das Objekt die Operationen auch versteht.

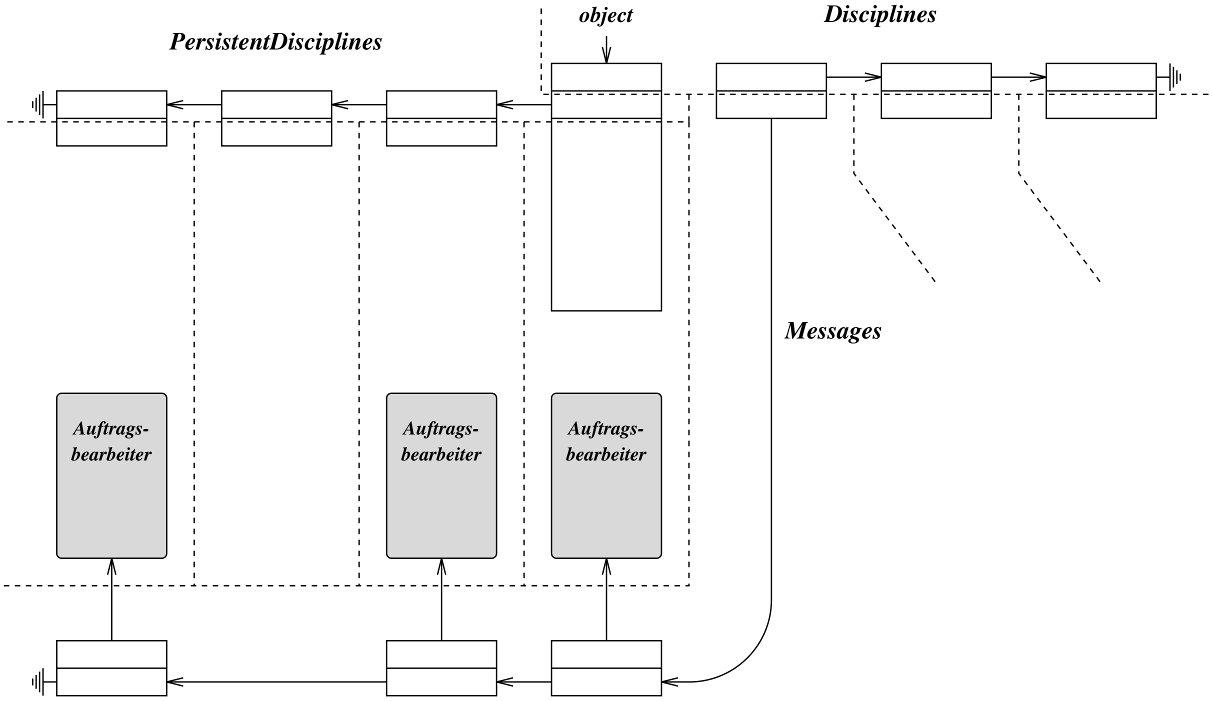

Abb. 44: Sichtbarkeitsbereiche bei Objekten mit verteilter Verantwortung

Dies gleicht hier der fehlenden statischen Typsicherheit in Smalltalk. Statt einen Abbruch des Programmlaufs zu erzwingen, würde man in der Ulmer Oberon-Bibliothek entsprechende Ereignisse generieren und ggf. **FALSE** als Rückgabewert zurückliefern.

Entsprechend dürfen auch keine Komponenten nach außen sichtbar sein. Statt dessen müssen für alle Datenfelder, die öffentlich zugänglich sein sollen, Zugriffsfunktionen eingerichtet werden.

Die Implementierung von *Students* besteht im wesentlichen aus drei Teilen: den exportierten Operationen, die entsprechende Aufträge weiterschicken, den Schnittstellenprozeduren für *PersistentObjects* und einem Auftragsbearbeiter, der Aufträge entgegennimmt. Dabei müssen alle Auftragstypen und *RealStudent* (garantiert ein Student im Gegensatz zu *Student*) als persistente Datentypen eingerichtet werden:

```
CONST
   set = 0; get = 1; (* Set or Get operation? *)
TYPE
   Mode = SHORTINT; (* set or get *)
   Message = POINTER TO MessageRec;
   MessageRec = RECORD (Messages.MessageRec) END;
   AccessNameMsg = POINTER TO AccessNameMsgRec;
   AccessNameMsgRec =
      RECORD
         (MessageRec)
         mode: Mode;
         name: Name;
      END;
```

```
      (* AccessAddressMsg, AccessCourseMsg *)
   VAR
      accessNameMsgType,
      accessAddressMsgType,
      accessCourseMsgType: PersistentObjects.Datatype;

   TYPE
      RealStudent = POINTER TO RealStudentRec;
      RealStudentRec =
         RECORD
            (PersistentDisciplines.ObjectRec)
            name: Name;
            address: Address;
            course: Course;
         END;
   VAR
      realStudentType: PersistentObjects.Datatype;

   PROCEDURE InitPersistentType(VAR type: PersistentObjects.Datatype;
                                name, baseName: ARRAY OF CHAR;
                                create: PersistentObjects.CreateProc;
                                read: PersistentObjects.ReadProc;
                                write: PersistentObjects.WriteProc);
      CONST
         caps = {PersistentObjects.read, PersistentObjects.write};
      VAR
         if: PersistentObjects.Interface;
   BEGIN
      NEW(if);
      if.read := read; if.write := write;
      type := PersistentObjects.RegisterType(name, baseName,
                  create, if, caps);
   END InitPersistentType;
```

Für *AccessNameMsg* könnten die Schnittstellenprozeduren folgendermaßen aussehen:

```
   PROCEDURE CreateAccessNameMsg(VAR obj: PersistentObjects.Object);
      VAR
         msg: AccessNameMsg;
   BEGIN
      NEW(msg); PersistentObjects.Connect(obj, accessNameMsgType);
      obj := msg;
   END CreateAccessNameMsg;

   PROCEDURE ReadAccessNameMsg(s: Streams.Stream;
                               obj: PersistentObjects.Object) : BOOLEAN;
   BEGIN
      WITH obj: AccessNameMsg DO
         RETURN NetIO.ReadShortInt(s, obj.mode) &
                NetIO.ReadString(s, obj.name)
      END;
   END ReadAccessNameMsg;

   PROCEDURE WriteAccessNameMsg(s: Streams.Stream;
                                obj: PersistentObjects.Object) : BOOLEAN;
   BEGIN
      WITH obj: AccessNameMsg DO
         RETURN NetIO.WriteShortInt(s, obj.mode) &
                NetIO.WriteString(s, obj.name)
      END;
   END WriteAccessNameMsg;
```

Der Auftragsbearbeiter kann anhand des gemeinsamen Basistyps *Students.Message* schnell überprüfen, ob der Auftrag für ihn bestimmt ist, und dann weiter selektieren:

```
PROCEDURE Handler(obj: Messages.Object; VAR message: Messages.Message);
BEGIN
   IF message.processed OR ~(message IS Message) THEN
      RETURN
   END;
   WITH obj: RealStudent DO
      WITH message: Message DO
         message.processed := TRUE;
         IF message IS AccessNameMsg THEN
            WITH message: AccessNameMsg DO
               CASE message.mode OF
               | get:   message.name := obj.name;
               | set:   obj.name := message.name;
               END;
               message.done := TRUE;
            END;
         ELSIF (* ... *)
         END;
      END;
   END;
END Handler;
```

CreateStudent wird sowohl als Schnittstellenprozedur für *PersistentObjects* als auch von *Create* genutzt:

```
PROCEDURE CreateStudent(VAR object: PersistentObjects.Object);
   VAR
      student: RealStudent;
BEGIN
   NEW(student);
   PersistentObjects.Connect(student, realStudentType);
   Messages.InstallHandler(student, Handler);
   object := student;
END CreateStudent;

PROCEDURE Create(VAR student: Student;
                 name: Name; address: Address;
                 course: Course);
   VAR
      realStudent: RealStudent;
BEGIN
   CreateStudent(realStudent);
   realStudent.name := name;
   realStudent.address := address;
   realStudent.course := course;
   student := realStudent;
END Create;
```

Die anderen exportierten Operationen müssen dann nur noch entsprechende Aufträge kreieren und abschicken:

```
PROCEDURE GetName(student: Student; VAR name: Name) : BOOLEAN;
   VAR
      msg: AccessNameMsg;
      message: Messages.Message;
BEGIN
   NEW(msg); Messages.Init(msg);
   RelatedEvents.Forward(msg.errors, student);
   msg.mode := get; msg.name[0] := 0X;
   message := msg;
   Messages.Send(student, message);
   RETURN (message IS AccessNameMsg) & message.done
END GetName;
```

Mit *RelatedEvents.Forward* wird hier sichergestellt, daß alle Fehlerereignisse in Zusammenhang mit dem Auftrag an *student* weitergeleitet werden. Dies ist selbst dann sinnvoll, wenn der Auftragsbearbeiter keine Ereignisse generieren sollte, da es möglicherweise zu Ereignissen wegen Kommunikationsproblemen kommen könnte.

Analog zum Beispiel in §1.8 können weitere Module der gleichen Art für die Sicht des Prüfungsamts oder Studentensekretariats hinzugefügt werden. Im Unterschied zu *Students* würden sie ihre Daten nicht direkt bei dem Objekt selbst verwalten, sondern persistente Disziplinen dafür verwenden.

8.4.4. Export und Import von Objekten

Auf der Basis von *Messages* lassen sich entfernte Objekte auf Basis von Delegation über Stellvertreter-Objekte erreichen. Dieser Dienst wird in der Ulmer Oberon-Bibliothek von *RemoteObjects* offeriert. *RemoteObjects* basiert auf *Networks*, das eine allgemeine Abstraktion für zuverlässige Netzwerkverbindungen offeriert, und ist damit selbst auch systemunabhängig.

Die Adressierung von entfernten Objekten erfolgt mit Tupeln, die aus einer Netzwerkadresse (*Networks.Address*) und einer Objekt-Nummer bestehen:

```
TYPE
   Object = POINTER TO ObjectRec;
   ObjectRec = RECORD (PersistentDisciplines.ObjectRec) END;
   Address = POINTER TO AddressRec;
   AddressRec = RECORD (PersistentDisciplines.ObjectRec) END;
   ObjectID = LONGINT;

PROCEDURE CreateAddress(VAR address: Address;
                        port: Networks.Address;
                        objectID: ObjectID);
```

Wenn ein Prozeß Objekte exportieren möchte, muß dafür gesorgt werden, daß Aufträge von außen eingehen können und daß die Adressen lokaler Objekte potentiellen Klienten mitgeteilt werden. Analog zum Telefonieren muß hierfür ein Anschluß vorhanden sein, bevor Anrufe eingehen können. Da häufig mehrere Netzwerke zur Verfügung stehen mit unterschiedlichen Geschwindigkeiten und Mengen von erreichbaren Kommunikationspartnern, sind mehrere Anschlüsse sehr sinnvoll. *AddPort* übergibt einen Anschluß zur Verwendung an *RemoteObjects*, so daß hier anschließend Aufträge entgegengenommen werden können:

```
PROCEDURE AddPort(port: Networks.Address;
                  VAR errors: RelatedEvents.Object) : BOOLEAN;
```

Nach *Export* ist ein Objekt von außen zugänglich:

```
CONST
   serial = 0; parallel = 1;
TYPE
   Mode = SHORTINT; (* serial or parallel *)

PROCEDURE Export(object: PersistentDisciplines.Object;
                 mode: Mode; VAR objectID: ObjectID);
```

Da Aufträge von außen asynchron eintreffen, bleibt die Frage, wer für die Synchronisierung verantwortlich ist. Wenn als Modus *serial* angegeben wird, wartet *RemoteObjects* die Erledigung des alten Auftrags ab, bevor es neue auf das Objekt losläßt. Bei *parallel* bleibt die Synchronisierung in der Verantwortung des Auftragsbearbeiters. Dies kann dann mit *Semaphores* geschehen. Es besteht dann die Möglichkeit, einander nicht störende Operationen auf einem Objekt parallel ablaufen zu lassen.

Die von *Export* zurückgelieferte Objekt-Nummer kann dann mit jedem Port kombiniert werden, der einmal via *AddPort* eingetragen worden ist, um eine gültige Objekt-Adresse zu erhalten. *GetPorts* liefert über einen Iterator die Menge aller Ports zurück:

```
PROCEDURE GetPorts(VAR it: Iterators.Iterator);
```

Objekt-Adressen sind persistent und können dann auf geeignete Weise (z.B. über spezielle Server, die Angebote registrieren und an Interessenten weitergeben) weitergeleitet werden. Kommt ein anderer Prozeß zu einer Objekt-Adresse, so kann mit *Import* ein lokales Stellvertreter-Objekt erzeugt werden:

```
PROCEDURE Import(VAR object: Object; address: Address;
                 VAR errors: RelatedEvents.Object) : BOOLEAN;
```

Eine auf diese Weise hergestellte Beziehung wird solange aufrechterhalten, bis sie von einer der beiden Seiten explizit wieder gelöst wird. Mit *Drop* kann der Importeur sein künftiges Desinteresse am Stellvertreter-Objekt bekunden, und mit *Withdraw* werden weitere Zugriffe von außen auf ein Objekt verhindert:

```
PROCEDURE Drop(object: Object);
PROCEDURE Withdraw(object: PersistentDisciplines.Object);
```

Solange die Verbindung auf beiden Seiten aufrechterhalten wird, ist weder das eigentliche Objekt noch das Stellvertreter-Objekt von der automatischen Speicherverwaltung bedroht, da *RemoteObjects* in seinen internen Datenstrukturen Verweise darauf hat.

Darüber hinaus stehen noch Operationen zur Verfügung, die alle Verbindungen und Dienste schließen:

```
PROCEDURE RemoveAllPorts;
PROCEDURE CloseAllConnections;
```

Zusätzlich zu den Aufträgen an ein Stellvertreter-Objekt werden auch alle Operationen von *PersistentDisciplines* weitergeleitet. Es ist also möglich, bei entfernten Objekten persistente Disziplinen anzuhängen und später wieder abzurufen. Dies geht, da *PersistentDisciplines* es ermöglicht, alle objektbezogenen Operationen zusammen (und nicht etwa nur teilweise) überzudefinieren. *RemoteObjects* kümmert sich dann auch um die Abbildung von lokalen in fremde Disziplin-Nummern, die zur Identifizierung von Disziplinen dienen.

Dabei ist zu beachten, daß immer nur die Auftragsbearbeiter des Prozesses, bei dem das eigentliche Objekt lebt, aufgerufen werden und nicht etwa desjenigen Prozesses, der eine persistente Disziplin anderswo angehängt hat. Wenn Aufträge an ein Objekt eines Prozesses P_1 von einem Modul eines Prozesses P_2 bearbeitet werden sollen, dann muß P_2 ein Hilfsobjekt an P_1 exportieren, damit eine entsprechende Weiterleitung realisiert werden kann.

Wenn die Forderung eingehalten wird, daß es keine öffentlichen Komponenten gibt und alle Operationen über *Messages* abgewickelt werden, dann gibt es keine Unterschiede bei der Benutzung lokaler oder fremder Objekte, wenn man von den Zeitunterschieden und den hinzugekommenen Fehlermöglichkeiten absieht. Der Aufruf einer Operation bei einem Stellvertreter-Objekt führt zur Blockierung der aufrufenden Task, bis das Ergebnis eingetroffen ist oder es zu einem Fehler kommt.

Natürlich können die Operationen analog zum Rendezvous-Konzept auch in synchroner Weise entgegengenommen werden. Dazu braucht der Auftragsbearbeiter die eingehenden Aufträge nur in Ereignisse zu verpacken, sie an *Events.Raise* zu übergeben und selbst auf ein Antwortereignis zu warten. Für das Objekt sollte dann eine eigene Task kreiert werden, die anschließend über *EventConditions* auf eingehende Aufträge wartet. Wenn der bei *Messages* eingetragene Auftragsbearbeiter unterschiedliche Ereignisse für unterschiedliche Typen von Aufträgen generiert, kann die zum Objekt gehörende Task analog zur **select**-Anweisung angeben, welche Aufträge im Augenblick zu akzeptieren sind. Sinnvollerweise sollte hier der Modus bei *RemoteObjects* auf *parallel* gesetzt werden, damit analog zum Rendezvous-Konzept die eingehenden Aufträge voll miteinander konkurrieren können.

8.4.5. Die Implementierung von *RemoteObjects*

Auf der Server-Seite von *RemoteObjects* gibt es vier Gruppen von Tasks, die über Bedingungen und die gemeinsamen Datenstrukturen miteinander kommunizieren:

- Für jeden Port, der über *AddPort* eingetragen worden ist, gibt es einen *PortListener*, der auf neue Verbindungen wartet, sie eröffnet und für sie jeweils eine neue Task kreiert.
- Für jede Verbindung gibt es einen *ChannelListener*. *RemoteObjects* kann jede Verbindung zweiseitig verwenden und bemüht sich auch, mit einer Verbindung für jedes Paar an Prozessen auszukommen. Entsprechend findet ein *ChannelListener* sowohl Aufträge als auch Antworten in der Eingabe. Jede eingehende Antwort, für die ein entsprechender Auftrag existiert, wird an den Auftraggeber weitergegeben. Neue Aufträge werden an *ObjectListener* oder *ResponseListener* weitergegeben.
- Für jedes exportierte Objekt, das eine Serialisierung der Zugriffe wünscht, gibt es einen *ObjectListener*. Ein *ObjectListener* wartet jeweils auf einen neuen Auftrag, führt *Messages.Send* aus und schickt die Antwort zurück.
- Für die exportierten Objekte, auf die parallel zugegriffen werden darf, gibt es einen Vorrat an *ResponseListener*. Jeder von ihnen wartet auf irgendeinen Auftrag, der ihnen von einem *ChannelListener*

zugeteilt wird, und führt ihn analog zu *ObjectListener* aus. Im Gegensatz zu den *ObjectListener* arbeitet ein *ResponseListener* für verschiedene Objekte. Je nach Bedarf werden neue *ResponseListener* erzeugt oder wieder freigegeben.

Zusätzlich zu den Aufträgen, die für ein exportiertes Objekt bestimmt sind, gibt es auch einige interne Aufträge von *RemoteObjects*, die es erlauben, die Verbindung zu testen, die Validität von Objekt-Nummern zu überprüfen, eine Verbindung auf "sanfte" Weise zu schließen und die Operationen von *PersistentDisciplines* weiterzuleiten.

8.4.6. Nichtpersistente Objekte in verteilten Systemen

Im obigen Beispiel des Studenten war von Anfang an vorgesehen, daß Objekte vom Typ *Student* exportiert und importiert werden sollten, um innerhalb verteilter Systeme eingesetzt zu werden. Dabei gab es zwei Nachteile: Ein Teil der gewohnten statischen Typsicherheit ging verloren (welche Operationen werden von dem Objekt unterstützt?) und natürlich der Zusatzaufwand, der durch *Messages* entsteht.

Glücklicherweise ist es nicht notwendig, sofort auf Basis von *Messages* zu arbeiten. Wenn eine klare Trennung zwischen Abstraktion und den zugehörigen Implementierungen vorliegt, kann man auch eine Implementierung hinzufügen, die alle Operationen in Aufträge verwandelt und sie über *Messages* verschickt. Der zugehörige Auftragsbearbeiter delegiert dann alle eingehenden Aufträge an ein anderes Objekt, das der gleichen Abstraktion entspricht.

Ein Beispiel für diese Technik ist *RemoteStreams*. Mit Hilfe von *RemoteStreams* ist es möglich, einen beliebigen Kanal als Parameter eines Auftrags an ein fremdes Objekt zu schicken. Irgendwelche Ein- und Ausgaben des fremden Objektes auf dem übergebenen Kanal werden dann in Aufträge verpackt, über das Netz geschickt und dann auf der Seite des Exporteurs durchgeführt.

Entsprechend dieser Zielrichtung gestaltet sich auch die Schnittstelle von *RemoteStreams*:

```
DEFINITION RemoteStreams;

   IMPORT Streams;

   PROCEDURE Export(net, stream: Streams.Stream) : BOOLEAN;
   PROCEDURE Import(net: Streams.Stream;
                    VAR stream: Streams.Stream;
                    bufmode: Streams.BufMode) : BOOLEAN;

END RemoteStreams.
```

Wenn ein Kanal Komponente eines persistenten Auftrags ist, wird in der zugehörigen *write*-Schnittstellenprozedur (für *PersistentObjects*) *Export* aufgerufen und bei der *read*-Schnittstellenprozedur *Import*.

8.5. Zusammenfassung

Im Vergleich zu vielen anderen Programmiersprachen und Programmiersystemen wurde in der Ulmer Oberon-Bibliothek Persistenz sowohl im Sinne von Langlebigkeit als auch im Sinne von Transportfähigkeit eingeführt. Dies wurde erreicht, indem die Abstraktion von persistenten Objekten auf der von byte-orientierten Kanälen (*Streams*) beruht. Byte-orientierte Kanäle dienen damit als universelles Transportmedium für persistente Objekte. Sie können auf diese Weise in Dateien, Datenbanken oder über Netzwerkverbindungen in andere Prozesse gelangen.

Die Verantwortung für das Lesen und Schreiben von Objektdaten wird in verteilter Form an alle Module weitergereicht, die eine eigene Sicht auf ein persistentes Objekt unterhalten. Dies folgt den Anforderungen an Unabhängigkeit und Dezentralisierung.

Auf der Basis von *Messages* und *RemoteObjects* ist es möglich, auf Objekten unabhängig von ihrer Lokalisierung zu operieren. Als zentrale Eigenschaft der Programmiersprache ist hierfür die Delegation notwendig. Dies ist in Oberon möglich auf Basis von Auftragsverbunden, die leicht über das Netz verschickt werden können, wenn sie Erweiterungen von *PersistentObjects.Object* sind.

Sicherlich ist es häufig notwendig, für das Ein- und Ausgeben persistenter Datentypen etwas mehr Programmtext zu formulieren und zu überprüfen, als dies bei Vergleichssystemen der Fall ist. Dennoch lohnt sich der Aufwand, da damit die Unterscheidung in temporäre und sichernswerte Komponenten vom Entwickler aus getroffen werden kann. Selbst *RemoteStreams*, das alle Operationen von *Streams* unterstützt und daher eine Reihe verschiedener Auftragstypen intern besitzt, kommt inklusive aller Kommentare mit ca. 500 Programmzeilen aus. Im Gegenteil, ohne eine von einer Bibliothek realisierte Lösung, die auf erweiterbaren

Abstraktionen beruht, wäre es kaum denkbar, wie man gleichzeitig sowohl räumliche als auch zeitliche Persistenz erreichen könnte.

Abbildung 45 gibt einen Überblick der Module der Ulmer Oberon-Bibliothek, die im Rahmen von Persistenz und verteilten Systemen entstanden sind. Auf der untersten Ebene stehen *PersistentObjects* und *PersistentDisciplines*, die auf *Streams* basieren und damit universell eingesetzt werden können.

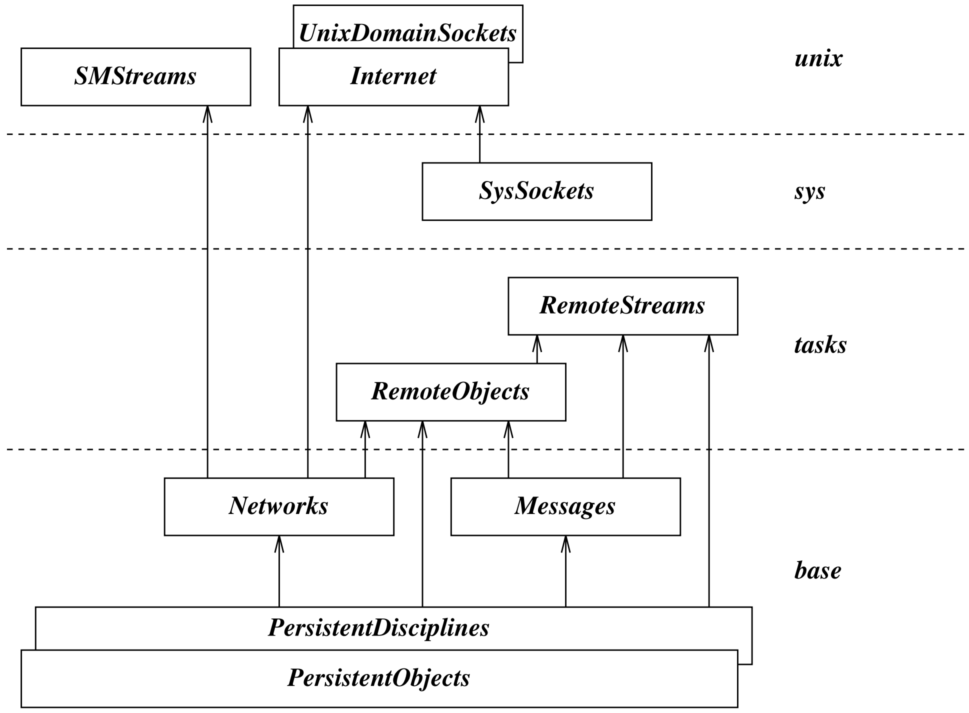

Abb. 45: Modulhierarchie für persistente Objekte

Networks offeriert eine Abstraktion für zuverlässige Netzwerkverbindungen, die ein wenig an die Schnittstelle von BSD-Sockets erinnert, jedoch etwas vereinfacht worden ist.[1] Für *Networks* gibt es die beiden naheliegenden Implementierungen *Internet* und *UnixDomainSockets*, die auf *SysSockets* und damit auf BSD-Sockets beruhen. Zusätzlich gibt es die von Till Franke entwickelte Implementierung *SMStreams*, die eine gegenüber BSD-Sockets schnellere Kommunikation zwischen zwei Prozessen auf dem gleichen Rechner über einen gemeinsamen Speicherbereich mit zwei Ringpuffern ermöglicht.

Mit *Messages* wurde ein allgemeiner Mechanismus entwickelt, auf dessen Basis persistente Aufträge an alle Auftragsbearbeiter eines Objekts weitergeleitet werden können. Nur auf diese Weise können nicht nur die Operationen des Haupt-Auftragsbearbeiters (aus dem Modul, das das Objekt kreiert hat) delegiert werden, sondern auch die der aktiven Disziplinen (d.h. Disziplinen, die einen Auftragsbearbeiter besitzen). *RemoteObjects* basiert nur auf der systemunabhängigen Abstraktion *Networks* und dem Mechanismus von *Messages* und ist damit selbst vollkommen unabhängig von einem bestimmten Betriebssystem.

Natürlich ist die Unterstützung für verteilte Systeme noch nicht abgeschlossen. Notwendig wären noch verteilte Namens-Server und Autorisierungs-Server. Mit Hilfe von Namens-Servern können Objekte leichter exportiert und importiert werden, ohne daß sich beide Partner vorher kennen. Auf Basis von Namens-Servern können auch Objekt-Identitäten definiert werden, die eine Migration mit der Chance des Wiederfindens zulassen würden.

Damit wäre das Fundament dafür geschaffen, daß Software-Systeme nicht nur unabhängig und dezentral organisiert werden können, sondern auch noch problemlos skalierbar sind.

[1] Mit den BSD-Sockets ist insbesondere gemeinsam, daß die Eröffnung eines Kommunikationskanals asymmetrisch ist, während die Kommunikation selbst symmetrisch verläuft.

9. Zusammenfassung

Ziel dieser Arbeit war es, die Erweiterbarkeit von Programmiersprachen durch Bibliotheken an dem konkreten Beispiel der Ulmer Oberon-Bibliothek zu zeigen.

Hierzu wurde nach einer kritischen Bewertung der gängigen Techniken im Bereich der Modularisierung und objekt-orientierten Programmierung ein Forderungskatalog bezüglich der Grundtechniken aufgestellt, die von einer für große Software-Systeme geeigneten Programmiersprache unterstützt werden müssen. Die Forderungen weichen in mehreren Punkten ab von der Linie der klassischen objekt-orientierten Programmiersprachen (wie etwa Smalltalk und Eiffel). Dies betrifft insbesondere die Forderung, daß Erweiterbarkeit und die Modularisierung zwei zueinander orthogonale Konzepte sein müssen, so daß jedes Modul seine private Sicht auf ein Objekt unterhalten kann. Es wurde dabei hervorgehoben, daß es auch möglich sein muß, eine private Sicht zur Laufzeit aufzubauen.

Weiter wurde gezeigt, daß Oberon im wesentlichen diesen Forderungen trotz des ungewöhnlich geringen Sprachumfangs genügt. Oberon unterstützt insbesondere die Orthogonalität von Erweiterungen und Modularisierungen, wobei das Hinzufügen von Sichten zur Laufzeit auf Basis von Disziplinen realisiert werden kann. Obwohl es in der Literatur einige den Disziplinen ähnliche Konzepte gibt, erlauben sie nicht die wirklich dezentrale Unterhaltung von Sichten.

Beim Aufbau größerer Software-Systeme und insbesondere bei Bibliotheken sollten implizite Abhängigkeiten vermieden werden, die auch zwischen zueinander fremden Modulen entstehen können. Hierfür wurden kooperative Mechanismen entwickelt, die alle denkbaren Konfliktsituationen bezüglich Raum, Zeit und externen Ressourcen vermeiden.

Auf Basis der vorgestellten Grundtechniken wurde gezeigt, wie sich so fundamentale Mechanismen wie Ausnahmenbehandlungen, Synchronisierung, Persistenz und die Unterstützung verteilter Systeme vollständig auf Basis von Bibliotheken in vollkommen portabler Form realisieren lassen, ohne dabei an Ausdruckskraft und Eleganz zu verlieren. Wenn man die bekannten objekt-orientierten Programmiersprachen und ihre zugehörigen Systeme betrachtet (z.B. BETA, C++ inkl. Varianten, Eiffel, Emerald, Modula-3, Simula oder Smalltalk), so sind fast ohne Ausnahme diese Mechanismen entweder über die Programmiersprache, über Schnittstellen zu nicht portablen Laufzeitbibliotheken oder überhaupt nicht realisiert worden. Zu den wenigen Ausnahmen gehört das Züricher Oberon-System in Bezug auf Persistenz, die allerdings nur in vergleichsweise geringem Umfang unterstützt wird (keine verteilte Verantwortung und keine Mobilität).

Während Programmiersprachen nur unter großem Aufwand zu ändern oder zu erweitern sind, beruhen die in der Ulmer Oberon-Bibliothek definierten Mechanismen auf erweiterbaren Abstraktionen. Dies führt zu vielfältigen neuen Möglichkeiten, die bei durch die Sprache realisierten Mechanismen kaum denkbar wären.

Das bemerkenswerteste Beispiel hierzu ist die Unterstützung verteilter Systeme, die durch die Erweiterung anderer Abstraktionen entscheidend erleichtert wurde:

- Da Persistenz auf den Abstraktionen für Ein- und Ausgabe beruht, gibt es keinen Unterschied, ob ein Objekt in einer Datei gespeichert oder über das Netz geschickt wird.
- Weil Synchronisierungsbedingungen frei definiert werden können, ist die Synchronisierung mit externen Kommunikationspartnern möglich.
- Da die Zugriffsoperationen auf persistente Disziplinen delegierbar sind, ist es auch möglich, private Daten bei exportierten Objekten zu unterhalten, die anderen Prozessen gehören.

Weitere Beispiele sind die objektbezogene Ausnahmenbehandlung, die Flexibilität bei der Repräsentierung persistenter Objekte und die Möglichkeit, neue Netzwerke in verteilte Systeme zu integrieren.

Als ein wesentliches Konzept stellte sich die Dezentralisierung heraus. Sie gehört zu den Voraussetzungen, die Erweiterungen ohne die Änderung bestehender Module ermöglichen. Die Sprache Oberon unterstützt dies durch die mögliche Trennung von Abstraktionen und Implementierungen, durch die Möglichkeit, daß jedes an einer Typerweiterung beteiligte Modul private Daten unterhalten kann, und durch die Abwesenheit eines Hauptprogramms. Hinzugekommen sind

- die Disziplinen, die dezentralisierte Sichten auf ein Objekt erlauben,
- die Ereignisse, die als Informationsverteiler dienen,
- die Tasks, die einen zentralen Kontrollfluß vermeiden helfen,
- die Aufträge mit verteilter Verantwortung und

- die Unterstützung für verteilte Systeme.

Natürlich ist die Entwicklung der Ulmer Oberon-Bibliothek noch nicht abgeschlossen. Es fehlen insbesondere noch Abstraktionen für Namensräume in Verbindung mit dezentralisiert arbeitenden Namens- und Autorisierungs-Servern. Ferner wäre es gerade im Bereich verteilter Systeme sehr hilfreich, wenn Module *on the fly* verschickt und geladen werden könnten, damit mobile Objekte überall unterstützt werden können. Wegweisende Arbeiten hierzu sind [Black90] und [Franz94].

A. Die Sprachvarianten von Oberon

A.1. Einführung

Nach der ersten Veröffentlichung eines Reports für die Sprache Oberon in [Wirth88a] gab es einige Revidierungen von Wirth und den Versuch von Mössenböck, klassische objekt-orientierte Techniken in Oberon zu integrieren. Der Ulmer Oberon-Übersetzer folgte zunächst der 1988 veröffentlichten Fassung und übernahm später teilweise die Revisionen von Wirth. Die Abbildung 46 gibt einen Überblick der Entwicklungslinien von Oberon. Die folgenden Abschnitte spezifizieren die genauen Unterschiede zwischen den beiden Oberon-Reports aus Zürich und Ulmer Oberon.

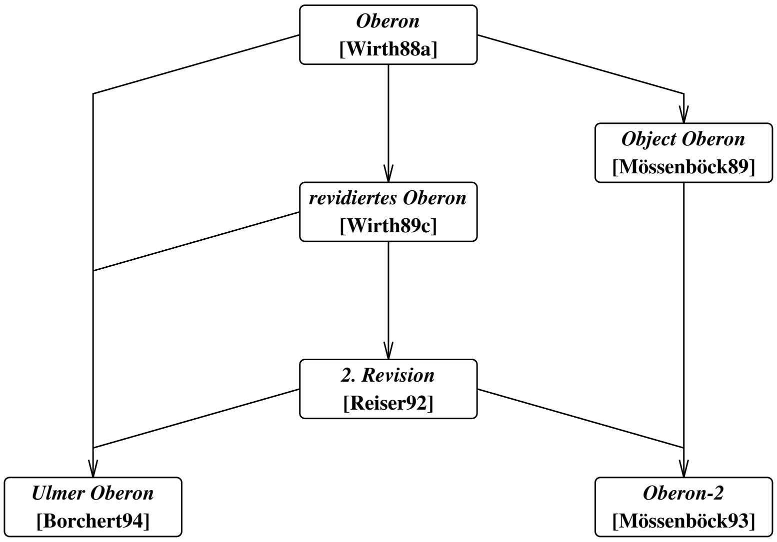

Abb. 46: Entwicklungsgeschichte der Oberon-Varianten

Die durch einen Pfeil angegebenen Verweise zu Beginn der folgenden Punkte beziehen sich immer auf die Abschnitte der zugehörigen Sprachbeschreibungen.

A.2. Die Unterschiede zwischen [Wirth88a] und [Wirth89c]

Nach der ersten Veröffentlichung über Oberon in [Wirth88a] folgten einige geringfügige Korrekturen in [Wirth89f]. Im Herbst 1989 erschien der Report für revidiertes Oberon in [Wirth89c]. Wirth selbst gibt in dem begleitenden Artikel [Wirth89d] eine Liste der Unterschiede an, die jedoch leider nicht vollständig ist. Die folgende Liste ist nach der Relevanz der Änderungen geordnet:

1. ↑4, 11. Die beiden verschiedenen Arten von Übersetzungseinheiten (Schnittstelle und Modul) wurden zu einer zusammengefaßt. Zu exportierende Namen werden durch einen Stern markiert:

   ```
   identdef = ident ["*"] .
   ConstantDeclaration = identdef "=" ConstExpression .
   TypeDeclaration = identdef "=" type .
   IdentList = identdef { "," identdef } .
   ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters] .
   CompilationUnit = module .
   ```

   Bei einer Importliste müssen alle Module mit einem Stern markiert werden, die danach in den exportierten Deklarationen verwendet werden:

   ```
   import = identdef [ ":" ident ] .
   ```

2. ↑6.5. Die Syntax für Prozedurtypen wurde dahingehend geändert, daß bei Parameterlisten analog zu normalen Prozedurdeklarationen nicht nur die Typnamen, sondern auch die Namen der Parameter angegeben werden. Die Namen der Parameter sind zwar ohne Bedeutung — haben aber dokumentarischen Wert:

ProcedureType = PROCEDURE [FormalParameters] .

3. ↑10. Die Deklarationen von Konstanten, Typen und Variablen dürfen in beliebiger Reihenfolge erfolgen. Vorher war festgelegt, daß Typen nicht vor Konstanten und Variablen nicht vor Typen deklariert werden dürfen. Nach wie vor müssen diese Deklarationen den Prozedurdeklarationen vorangehen:

```
DeclarationSequence = { CONST { ConstantDeclaration ";" } |
                        TYPE { TypeDeclaration ";" } |
                        VAR { VariableDeclaration ";" } }
                        { ProcedureDeclaration ";" | ForwardDeclaration ";" } .
```

4. ↑10.1. Die Regel, daß bei Variablenparametern der Typ des formalen Parameters ein Basistyp des aktuellen Parameters sein muß, wurde mit Ausnahme von Verbundtypen dahingehend verschärft, daß Typenidentität verlangt wird.
5. ↑10.1. Die Regel, daß bei einem Parametertyp von **ARRAY OF BYTE** alle Arten von Parametern zugelassen sind, gilt nur noch für Variablenparameter.
6. ↑10.1. Die Restriktion, daß auf offene Feldparameter — abgesehen von der Weitergabe als Parameter — nur elementweise zugegriffen werden darf, wurde fallengelassen. Damit ist insbesondere die Zuweisung von Zeichenkettenkonstanten an offene Feldparameter vom Typ **ARRAY OF CHAR** legal.
7. ↑9.1. Die Restriktion, daß Zuweisungen bei Verbundtypen nicht legal sind, wenn sie unbekannte private Komponenten enthalten, wurde fallengelassen.
8. ↑6.4. Bei der Revision wird darauf hingewiesen, daß Allokationen fehlschlagen können. Somit ist beim Aufruf von **NEW**(*p*) damit zu rechnen, daß *p* anschließend den Wert **NIL** hat.
9. ↑10.2. Die Standardprozedur **COPY** wurde hinzugefügt. **COPY**(*x*, *v*) kopiert die Zeichenkette *x* (eine Variable oder eine Konstante) nach *v* (eine Variable).
10. ↑Appendix. Analog zu Modula-2 wurde bei der Revidierung ein internes Modul namens **SYSTEM** hinzugefügt, das eine Reihe systemnaher Operationen unterstützt.
11. ↑10.2, Appendix. Die Standardfunktionen **ADR** und **SIZE** wurden Bestandteil des neu geschaffenen internen Moduls **SYSTEM**.
12. ↑3.3. Der Apostroph darf nicht mehr als Zeichenkettenbegrenzer verwendet werden.

CharConstant = """ character """ | digit {hexDigit} "X" .

13. ↑4. Es wurde der ausdrückliche Hinweis hinzugefügt, daß kein Name mehr als ein Objekt innerhalb eines Sichtbereiches (*scope*) bezeichnen darf. Dies betont, daß überlappende Sichtbereiche analog zum folgenden Beispiel nicht legal sind:

```
PROCEDURE P;
   CONST x = 1;

   PROCEDURE Q;
      CONST y = x;
      VAR x: INTEGER;
   END Q;
END P;
```

14. ↑6.1. Der revidierte Report weist darauf hin, daß der vordefinierte Datentyp **CHAR** nicht nur den ASCII-Bereich 0X..7FX abdeckt, sondern den gesamten Bereich 0X..0FFX.
15. ↑10.2. Der revidierte Report legt klar, daß bei der Standardfunktion **LEN** die Indizierung der Dimensionen eines Feldes bei 0 beginnt und somit **LEN**(*v*) äquivalent zu **LEN**(*v*, 0) ist.

A.3. Die Unterschiede zwischen [Wirth89c] und [Reiser92]

In [Reiser92] ist im Anhang ein Report von Oberon enthalten, in dem einige Änderungen nach [Wirth89c] eingeflossen sind. Obwohl dieser Report 1992 in diesem Buch veröffentlicht worden ist, dürfte er schon aus dem Jahr 1990 stammen.

1. ↑6.5. Variablen von Prozedurtypen dürfen nicht nur eine Prozedur, sondern auch **NIL** als Wert besitzen.
2. ↑Appendix, 12. Der Typ **BYTE** gehört nicht mehr zu den Standardtypen, sondern wird statt dessen von **SYSTEM** exportiert.
3. ↑10.2, Appendix. Die Funktion **SIZE** wird nicht mehr von dem internen Modul **SYSTEM** exportiert, sondern steht als Standardfunktion zur Verfügung.
4. ↑11. Die Syntax der Importliste wurde dahingehend geändert, daß Umbenennungen mit ":=" statt ":" angegeben werden:

   ```
   import = "ident [ ":=" ident ] .
   ```

5. ↑11. Die Notwendigkeit, diejenigen Module in der Importliste mit einem Stern zu markieren, die durch exportierte Deklarationen verwendet werden, entfällt.
6. ↑8.2.2. Während die Operatoren **DIV** und **MOD** vorher für beliebige ganzzahlige Typen definiert waren, sind sie es nun nur für positive Divisoren.

A.4. Die Unterschiede zwischen [Reiser92] und der Ulmer Oberon-Variante

Der Ulmer Oberon-Übersetzer wurde in der ersten Hälfte des Jahres 1989 entwickelt und folgte damals genau der Definition aus [Wirth88a]. Im Oktober 1989 wurde der Report [Wirth89c] zugänglich, der nur teilweise übernommen worden ist. Spätere Änderungen wurden 1990 durch persönliche Kontakte mit Zürich zum Teil bekannt und konnten später teilweise auch in [Reiser92] wiedergefunden werden. Leider mußte bei der Veröffentlichung des Züricher Oberon-Systems festgestellt werden, daß sich der Oberon-Übersetzer von Wirth nicht überall eng an den Report hält und einige Unterschiede aufweist. Diese konnten natürlich nicht alle ermittelt werden.

Es muß darauf hingewiesen werden, daß die Ulmer Oberon-Variante keine Ulmer Neuschöpfung ist, sondern nur eine selektive Adaption der Züricher Revisionen von Oberon. Es wurde somit kein einziges auch noch so geringes Sprachkonstrukt neu zu Oberon hinzugefügt, so daß hier weiterhin der Name Oberon gerechtfertigt ist.

Hier sind die Unterschiede im Vergleich zu dem zuletzt bekannt gewordenen Züricher Oberon-Report, die aus der Sicht der Ulmer Variante geschildert werden:

1. ↑4, 11. Es gibt zwei Sorten von Übersetzungseinheiten: Schnittstellen und Module. Entsprechend fallen auch die Export-Markierungen mit dem Stern weg:

   ```
   CompilationUnit = module | definition .
   definition = DEFINITION ident ";" [ImportList] DefSequence END ident "." .
   DefSequence = { CONST { ConstantDeclaration ";" } |
                   TYPE { TypeDeclaration ";" } |
                   VAR { VariableDeclaration ";" } }
                   { ProcedureHeading ";" } .
   identdef = ident .
   ```

   Dies entspricht der zuerst veröffentlichten Fassung — abgesehen davon, daß die Reihenfolge der Deklarationen genauso freizügig ist wie in der revidierten Fassung von Oberon.
2. ↑10.1. Bei Variablenparametern darf nicht nur bei Verbundtypen der Typ des aktuellen Parameters eine Erweiterung des formalen Parameters sein, sondern auch bei Zeigertypen. Dies entspricht der Regelung aus [Wirth88a].
3. ↑9.1. Zuweisungen bei Verbundtypen sind nicht legal, wenn darin unbekannte private Komponenten enthalten sind. Dies entspricht der Regelung aus [Wirth88a].
4. ↑6.4. Es wird zugesichert, daß nach **NEW**(p) der Wert von p ungleich **NIL** ist. Wenn kein Speicherplatz mehr zur Verfügung steht, wird bei dem Ulmer Oberon-System eine Ausnahmenbehandlung eingeleitet, die im Normalfall zur Terminierung des Programms führt. Dies entspricht [Wirth88a] und ist vorteilhaft, da nicht jeder Aufruf von **NEW** von einem Erfolgstest begleitet werden muß.
5. ↑12. Der Datentyp **BYTE** wird nicht von **SYSTEM** exportiert und gehört statt dessen zu den Standardtypen.

6. ↑12. Bei dem formalen Parametertyp **ARRAY OF BYTE** sind alle Arten von aktuellen Parametern zugelassen. Dies gilt auch, wenn es sich um einen Werteparameter handelt. Diese Regelung stimmt mit [Wirth88a] überein.
7. ↑12. Die Funktion **SIZE** gehört nicht zu den Standardfunktionen und wird statt dessen von **SYSTEM** exportiert.
8. ↑8.2.2. **DIV** und **MOD** sind auch dann definiert, falls der rechte Operand negativ ist. Entsprechend [Wirth88a] und [Wirth89c] gelten die folgenden Formeln:

$$x = (x \ \mathbf{DIV} \ y) * y + (x \ \mathbf{MOD} \ y)$$

$$0 \leq (x \ \mathbf{MOD} \ y) < y \quad or \quad y < (x \ \mathbf{MOD} \ y) \leq 0$$

9. ↑3. Kommentare dürfen geschachtelt werden. Dies steht in keinem der Oberon-Reports von Wirth, entspricht aber der Züricher Oberon-Implementierung [Wirth92, S.385].
10. ↑12. Es gibt einige Unterschiede im internen Modul **SYSTEM**: **CC** wird nicht unterstützt, und einige Typen, Funktionen und Prozeduren sind hinzugekommen. Die Details hierzu können [Borchert94] entnommen werden.

B. Glossar

Abstraktion; eine verallgemeinerte Sichtweise auf zahlreiche denkbare konkrete Phänomene. Abstraktionen werden durch Schnittstellen definiert, zu denen es eine Reihe von konkreten Implementierungen geben kann.

Auftrag (*message*); eine durch einen Verbund (*message record*) repräsentierte Operation, die an einen oder mehrere Bearbeiter weitergeleitet wird.

Auftragnehmer (*message handler*); ein anderes Wort für Auftragsbearbeiter.

Ausfall; die Nichteinhaltung eines Vertrags.

Ausnahme (*exception*); die Repräsentierung einer erkannten Ausnahmesituation. Während eines Programmlaufs führen Fehler zu Vertragsverletzungen und damit zu Ausfällen. Ausfälle führen zu Ausnahmesituationen, die bei Entdeckung einer Ausnahmenbehandlung unterzogen werden.

automatische Speicherverwaltung (*garbage collection*); eine Speicherverwaltung, bei der die Freigabe von Speicher automatisch erfolgt.

Basistyp; T_1 ist ein Basistyp von T_2 genau dann, wenn der Typ T_2 eine Erweiterung des Typs T_1 ist.

Bearbeiter (*handler*); eine Prozedur, die Aufträge oder Ereignisse entgegennimmt und bearbeitet.

Bedingung (*condition*); eine Funktion, die angibt, ob ein bestimmtes Ereignis (im allgemeineren Sinne) eingetreten ist. Über *Tasks.WaitFor* kann auf das Eintreten eines oder mehrerer Ereignisse gewartet werden.

Beendigung (*termination*); bei der Ausnahmenbehandlung die Beendigung des Signalempfängers, die aus der Sicht des Aufrufers von einer normalen Rückkehr nicht zu unterscheiden ist.

Bibliothek (*library*); eine Sammlung von Modulen.

Binäre Ein- und Ausgabe (*binary i/o*); die Ein- und Ausgabe von Datenstrukturen in ihrer maschinenabhängigen Repräsentierung.

Blockierung; eine ausführende Einheit wird blockiert, wenn sie eine Operation aufruft, die längere Zeit in Anspruch nimmt.

Delegation; die Weiterleitung von Operationen an ein anderes Objekt.

Disziplin; eine zur Laufzeit hinzufügbare Sicht eines Moduls auf ein nicht vom ihm kreiertes Objekt. Es gibt sowohl temporäre als auch persistente Disziplinen. Mit Disziplinen können Operationen und Auftragsbearbeiter verbunden sein.

dynamischer Typ; der tatsächliche Typ eines Objekts zur Laufzeit, der eine Erweiterung des statischen Typs ist. Durch die freizügigeren Kompatibilitätsregeln bei Zuweisungen und Parameterübergaben können dynamischer und statischer Typ voneinander abweichen.

Erledigungsvermerk; eine Komponente eines Auftragsverbundes, bei der notiert wird, ob der Auftrag bearbeitet worden ist oder nicht.

Ereignis (*event*); eine Mitteilung. Jedem Ereignis sind Informationen und ein Ereignistyp zugeordnet. Für jeden Ereignistyp können Interessenten eingetragen werden. Ereignisse dienen somit als Informationsverteiler.

Erweiterbarkeit (*extensibility*); ist die Fähigkeit eines Moduls oder einer Abstraktion, an neue Anforderungen angepaßt werden zu können, ohne daß Veränderungen daran erforderlich sind.

Erweiterung (*extension, inheritance*); die Verfeinerung einer Abstraktion, eines Moduls oder eines Typs.

externes Ereignis (*external event*); ein Ereignis, das außerhalb des eigenen Prozesses initiiert worden ist.

Feldtrenner (*field separator*); ein Zeichen, das eine Zeile in mehrere Felder auftrennt.

Formatierte Ein- und Ausgabe (*formatted i/o*); Ein- und Ausgabe in einem für Menschen leicht lesbaren Format. Sie ist auch häufig als maschinenunabhängige Repräsentierung sinnvoll.

Füllzeichen (*white space*); Zeichen, die zur Trennung und zur Aufbereitung der Ein- oder Ausgabe dienen. Häufig ist kein Unterschied dadurch gegeben, ob ein Füllzeichen einzeln oder mehrfach hintereinander vorkommt.

Funktion; eine Prozedur mit Rückgabewert. Im Gegensatz zum Funktionsbegriff der Mathematik können

Funktionen Seiteneffekte haben.

Implementierung; ein Modul, das eine Abstraktion für ein spezielles Phänomen konkretisiert.

Importbeziehung (*import relation*); eine Importbeziehung zwischen zwei Modulen ist gegeben, wenn das eine als Klient die exportierten Dienste des anderen verwendet.

Interessent; ein für einen Ereignistyp eingetragener Bearbeiter.

internes Modul; ein Modul, das nur in virtueller Form durch den Übersetzer existiert. In Modula-2 und Oberon ist **SYSTEM** ein internes Modul.

Iterator; eine Abstraktion für das Durchlaufen einer Datenstruktur. Zu einem Iterator gibt es einen Produzenten, der die Objekte liefert, und einen Konsumenten, der sie abholt.

Kanal (*channel, stream*); eine Verbindung, über die Ein- und Ausgaben erfolgen können.

Keller (*stack*); eine Datenstruktur, bei die zuletzt eingefügten Objekte zuerst entnommen werden (LIFO). Zu jedem Kontrollfluß gibt es genau einen Keller, bei dem die Zustandsinformationen der aufgerufenen Prozeduren (*activation records*) die Objekte bilden.

Klasse; im klassischen objekt-orientierten Sinne die Schnittstelle für eine Menge von Objekten. Klassen sind somit spezielle Arten von Modulen oder Abstraktionen und in vielen Sprachen auch direkt mit dem Konzept der Typen unifiziert.

Klient; **1**: ein Modul, das die Dienstleistung eines anderen Moduls importiert. — **2**: eine ausführende Einheit, die die Dienste eines Servers in Anspruch nimmt.

kombinierte Erweiterung (*multiple inheritance*); eine Erweiterung, die sich auf zwei oder mehr Abstraktionen, Module oder Typen bezieht.

Kommando; eine parameterlose exportierte Prozedur. Der Name eines Kommandos setzt sich aus dem Modulnamen, einem Punkt und dem Namen der Prozedur zusammen. Im Züricher Oberon-System kann ein Kommando durch das Anklicken des Kommandonamens in einem Textfenster aufgerufen werden.

Kontrakt; ein Synonym für Vertrag.

Kontrollfluß (*program counter, locus*); eine sequentielle Abarbeitungsfolge mit einer aktuellen Position. Kontrollflüsse können entweder in quasi-paralleler Form als Koroutinen realisiert sein oder laufen echt parallel als leichtgewichtige Prozesse.

Kooperativität; die konfliktfreie Koexistenz von zueinander fremden Modulen.

Koroutine; ein quasi-parallel arbeitender Kontrollfluß eines (ggf. leichtgewichtigen) Prozesses. Es ist immer nur eine Koroutine zu einem bestimmten Zeitpunkt aktiv.

Laufzeitbibliothek (*run time library*); eine (häufig in Assembler oder in C) geschriebene Bibliothek, deren Aufrufe vom Übersetzer generiert werden. Laufzeitbibliotheken sind sowohl abhängig von dem Übersetzer als auch der Zielmaschine und dienen zur Implementierung komplizierterer Konstrukte, die nicht mit wenigen Maschineninstruktionen realisiert werden können.

Laufzeitfehler (*run time error*); eine zur Laufzeit entdeckte Verletzung der Regeln der Programmiersprache.

Laufzeitstart (*run time start*); der Teil einer Laufzeitbibliothek, der bei einem Programmstart zuerst ausgeführt wird.

leichtgewichtiger Prozeß (*thread*); Kontrollfluß, der in echter Parallelität zu anderen Kontrollflüssen innerhalb eines Adreßraumes agiert. Jedem leichtgewichtigen Prozeß können mehrere Koroutinen angehören.

Methode; eine objektbezogene Operation.

Modul; eine Übersetzungseinheit, die eine öffentlich bekannte Schnittstelle implementiert. Module dienen zur Begrenzung der Sichtbereiche (*information hiding*) und sollten nicht mit Klassen oder Typen verwechselt werden.

Nachbedingung (*postcondition*); bei einem Kontrakt anzugebende Bedingung, die eine aufgerufene Operation sicherzustellen hat, vorausgesetzt, sie wurde mit einer erfüllten Vorbedingung aufgerufen.

Neustart (*retry*); bei der Ausnahmenbehandlung der Abbau des Kellers des aktuellen Kontrollflusses bis zum

Signalempfänger, gefolgt von der erneuten Ausführung des Signalempfängers.

Objekt; ein Bezugspunkt, zu dem jedes Modul seine private Sicht unterhalten kann.

Partei; steht immer für eine bestimmte Sichtweise, die durch ein Modul oder eine ausführende Einheit repräsentiert wird.

Persistenz; die Eigenschaft eines Objekts, sowohl langfristig existieren als auch räumlich bewegt werden zu können.

Priorität; sowohl Ereignissen als auch allen ausführenden Einheiten sind Prioritäten zugeordnet. Durch das Hochsetzen der Priorität kann das Eintreffen von Ereignissen verzögert werden.

private Komponente; die Komponente eines Verbundes, die nur von einem Modul aus sichtbar ist.

Programm; eine bezüglich der Importbeziehung abgeschlossene Menge von Modulen, die in ausführbarer Form zusammengebunden worden ist.

Prozeß; vom Betriebssystem aus identifizierbare Rechtegemeinschaft von leichtgewichtigen Prozessen, die sich einen Adreßraum teilen.

Puffer (*buffer*); eine Datenstruktur, die Ein- und Ausgabe-Operationen entgegennehmen und mit Verzögerung weiterleiten kann. Puffer dienen zur Effizienzsteigerung und zur Vermeidung unnötiger Blockierungen.

Qualifikation; die Angabe des exportierenden Moduls bei der Verwendung von importierten Deklarationen.

Quasi-Parallelität; von Quasi-Parallelität wird gesprochen, wenn es mehrere Kontrollflüsse nur in Form von Koroutinen gibt und somit immer nur eine davon zu einem bestimmten Zeitpunkt aktiv sein kann.

Scheduler; eine Task, die festlegt, welche der rechenbereiten Tasks ihrer Gruppe als nächstes zur Ausführung gelangt. Ein Scheduler gehört selbst immer einer anderen Taskgruppe an, als der, die er betreut.

Schnittstelle (*interface*); entweder ein eigenständiges Dokument (Vertrags-Modell) oder eine Sicht auf eine zugehörige Implementierung (Sicht-Modell), die die öffentliche Vertragsgrundlage zwischen den zugehörigen Implementierungen und den Klienten bildet.

Schnittstellenverbund (*interface record*); ein Verbund, der aus einer Reihe von Prozedurkomponenten besteht, die die Operationen einer Abstraktion für einen bestimmten Objekttyp repräsentieren.

Semaphor; ein Mechanismus, über den sichergestellt werden kann, daß maximal nur eine Partei auf eine gemeinsame Ressource zugreifen kann (*mutual exclusion*).

Sichtbereich (*scope*); der Bereich des Programmtexts, innerhalb dessen ein Name einer bestimmten Deklaration bekannt ist. In Oberon dienen Module und Prozeduren zur Begrenzung von Sichtbereichen.

Sichtmodell; beim Sichtmodell wird die Schnittstelle aus einer Implementierung extrahiert.

Signalempfänger; das Modul oder die Prozedur, die den zuständigen Ausnahmenbearbeiter definiert hat.

Signalgeber; die Prozedur, die eine Ausnahmenbehandlung initiiert.

Standardfunktionen, -prozeduren und -typen; die Funktionen, Prozeduren und Typen, die zur Sprachdefinition gehören und ohne Importanweisungen verwendet werden können.

statischer Typ; der im Programmtext angegebene Typ eines Objekts.

Symboldatei; eine vom Übersetzer generierte Schnittstellendefinition, die zur Überprüfung bei der Übersetzung der Klienten und Implementierungen dient.

Systembibliothek; ein Synonym für Laufzeitbibliothek.

Task; eine wartefähige Einheit in der Ulmer Oberon-Bibliothek. Ihr sind eine oder mehrere Koroutinen zugeordnet. Tasks sind in Gruppen organisiert, und für jede Gruppe gibt es einen Scheduler.

Typentest; ein Objekt kann zwei verschiedene Typen haben: den statisch bekannten und den tatsächlichen zur Laufzeit, der eine Erweiterung des statisch bekannten Typs sein muß. Ein Typentest erlaubt die Überprüfung des zur Laufzeit bestimmten Typs.

Typzusicherung (*type guard*); die Zusicherung, daß ein Objekt einen bestimmten Typ zur Laufzeit besitzt.

Übersetzer (*compiler*); ein Programm, das Module in Maschinensprache und Schnittstellen in Symboldateien übersetzt.

Übersetzungseinheit (*compilation unit*); abgeschlossener Programmtext, der separat übersetzt werden kann. Üblicherweise wird jede Übersetzungseinheit durch den Inhalt einer Datei repräsentiert. Sowohl Schnittstellen als auch Module sollten Übersetzungseinheiten sein.

Unterbrechung (*interrupt*); das asynchrone Eintreffen externer Ereignisse, die zur Suspendierung des aktuellen Kontrollflusses und zum Aufruf eines speziellen Bearbeiters führen.

Verbund (*record*); ein Verbund ist eine Datenstruktur, die mehrere unterschiedliche Komponenten zusammenfaßt.

Vertrag; die Repräsentierung einer Abstraktion durch eine Schnittstelle zusammen mit der Angabe von Vor- und Nachbedingungen.

Vertragsmodell; beim Vertragsmodell wird die Schnittstelle durch ein eigenständiges Dokument repräsentiert.

Vorbedingung (*precondition*); **1**: (bei Kontrakten) eine einer Operation zugeordnete Bedingung, die der Aufrufer vor dem Aufruf sicherzustellen hat. — **2**: (bei Wartebedingungen) eine Bedingung, die erfüllt werden muß, bevor eine andere Bedingung wahr werden kann. Wenn es mehrere Vorbedingungen gibt, dann muß mindestens eine davon erfüllt werden.

Weiterleitung (*propagation*); bei der Ausnahmenbehandlung die Weitergabe einer Ausnahme durch das erneute Initiieren einer Ausnahmenbehandlung.

Wiederaufnahme (*resumption*); bei der Ausnahmenbehandlung die Fortsetzung des Signalgebers am Punkt des Ausfalls.

Wiederverwendbarkeit (*reusability*); die Eigenschaft von Software, auch bei zukünftigen Projekten unter zumutbaren Aufwand (ggf. mit einigen Änderungen) erneut benutzt werden zu können. Wiederverwendbarkeit ist die schwächere Form von Erweiterbarkeit.

Zeichenkette (*string*); **1**: eine Konstante, die aus einer Sequenz von Zeichen besteht, die in doppelte Anführungszeichen gefaßt ist. — **2**: ein Feld, das **CHAR** als Elementtyp hat.

Zeile (*line*); eine Folge von Zeichen, die durch einen Zeilentrenner abgeschlossen worden ist.

Zugriffsschutz (*protection*); Mechanismen einer Programmiersprache, die unbeabsichtigte Zugriffe auf private Bereiche anderer Module verhindern.

C. Literaturverweise

[Adobe Systems Inc.86]
Adobe Systems Inc., *POSTSCRIPT Language Reference Manual,* Addison Wesley, 1986.

[Agesen93]
Ole Agesen, Svend Frølund, Michael Hoffmann Olsen, "Language level support for persistence in BETA," in *Object-Oriented Environments: The Mjølner Approach*, ed. by J.L. Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson, pp. 198-206, Prentice Hall, Hertfordshire, UK, 1993.

[Allen87]
Randy Allen, David Callahan, Ken Kennedy, "Automatic Decomposition of Scientific Programs for Parallel Execution," *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 63-76, Munich, Germany, January 1987.

[ANSI83]
ANSI, AJPO, "The Programming Language Ada Reference Manual," Lecture Notes in Computer Science 155, Springer-Verlag, Feb. 17, 1983. ANSI/MIL-STD-1815A-1983

[Atkinson83]
M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, R. Morrison, "An Approach to Persistent Programming," *The Computer Journal*, vol. 26, no. 4, pp. 360-365, 1983.

[Atkinson87]
Malcolm P. Atkinson, O. Peter Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, vol. 19, no. 2, pp. 105-190, June 1987.

[Avižienis85]
Algirdas Avižienis, "The *N*-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491-1501, December 1985.

[Bach86]
Maurice J. Bach, *The Design of the UNIX Operating System,* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[Biggerstaff89]
Ted J. Biggerstaff, Alan J. Perlis, "Introduction," in *Software Reusability*, ed. by Ted J. Biggerstaff, Alan J. Perlis, pp. xv-xxv, Addison Wesley, New York, 1989.

[Birkholz94]
Detlef Birkholz, *Persistente Objekte in Oberon,* Diplomarbeit an der Universität Ulm, SAI, Ulm, Februar 1994.

[Birrell84]
Andrew D. Birrell, Bruce Jay Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, February 1984.

[Birtwistle73]
G.M. Birtwistle, O.J. Dahl, B. Myhrhaug, K. Nygaard, *Simula Begin,* Auerbach Publishers Inc., Philadelphia, 1973.

[Black87]
Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, Larry Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 65-76, January 1987.

[Black90]
Andrew P. Black, Yeshayahu Artsy, "Implementing Location Independent Invocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 107-119, January 1990.

[Bloom79]
Toby Bloom, "Evaluating Synchronization Mechanisms," *Proceedings of the Seventh Symposium on Operating Systems Principles*, pp. 24-32, ACM SIGOPS, New York, December 1979.

[Booch91]
Grady Booch, *Object-Oriented Design: With Applications,* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[Borchert92]
Andreas F. Borchert, "Streams Tutorial," in *Ulm's Oberon System Reference*, ed. by Andreas F. Borchert, pp. 19-37, Universität Ulm, SAI, Ulm, Germany, March 1992.

[Borchert94]
Andreas Borchert, "Report of Ulm's Oberon Compiler," in *Ulm's Oberon System Reference*, ed. by Andreas Borchert, pp. 53-60, Universität Ulm, February 1994.

[Broadbery93]
Peter Broadbery, Christopher Burdorf, "Applications of Telos," *Lisp and Symbolic Computation*, vol. 6, pp. 139-158, 1993.

[Campbell73]
R.H. Campbell, A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Lecture Notes on Computer Science 16*, pp. 89-102, Springer-Verlag, 1973.

[Conway63]
Melvin E. Conway, "Design of a Separable Transition-Diagram Compiler," *Communications of the ACM*, vol. 6, no. 7, pp. 396-408, July 1963.

[Cook88]
Steve Cook, "OOPSLA'87 Panel P2: Varieties of Inheritance," *ACM SIGPLAN Notices*, vol. 23, no. 5, pp. 35-43, May 1988. OOPSLA'87 Addendum to the Proceedings

[Cook89]
W. R. Cook, "A Proposal for Making Eiffel Type-safe," *The Computer Journal*, vol. 32, no. 4, pp. 305-311, 1989.

[Cornelius88]
B. J. Cornelius, "Problems with the Language Modula-2," *Software—Practice and Experience*, vol. 18, no. 6, pp. 529-543, June 1988.

[Cristian82]
Flaviu Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers*, vol. C-31, no. 6, pp. 531-540, June 1982.

[Cristian84]
Flaviu Cristian, "Correct and Robust Programs," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 2, pp. 163-174, March 1984.

[Cristian85]
Flaviu Cristian, "Exceptions, défaillances et erreurs," *Technique et Science Informatiques*, vol. 4, no. 4, pp. 385-390, 1985.

[Cristian89]
Flaviu Cristian, "Exception handling," in *Dependability of resilient computers*, ed. by T. Anderson, pp. 68-97, BSP Professional Books, Oxford, 1989.

[Cui92]
Qian Cui, John Gannon, "Data-Oriented Exception Handling," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 393-401, May 1992.

[Decouchant86]
Dominique Decouchant, "Design of a Distributed Object Manager for the Smalltalk-80 System," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 444-452, November 1986.

[Decouchant89]
Dominique Decouchant, "A Distributed Object Manager for the Smalltalk-80 System," in *Object-Oriented Concepts, Databases, and Applications*, ed. by Won Kim, Frederick H. Lochovsky, pp. 487-520, Addison Wesley, Reading, Massachusetts, 1989.

[Deutsch89]
L. Peter Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," in *Software Reusability*, ed. by Ted J. Biggerstaff, Alan J. Perlis, pp. 57-71, Addison Wesley, New York, 1989.

[Dijkstra68]
Edsger W. Dijkstra, "The Structure of the "THE"-Multiprogramming System," *Communications of the ACM*, vol. 11, no. 5, pp. 341-346, May 1968.

[Dijkstra71]
Edsger W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, pp. 115-138, 1971.

[Dijkstra75]
Edsger W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453-457, August 1975.

[Dijkstra82]
Edsger W. Dijkstra, "How Do We Tell Truths that Might Hurt?," in *Selected Writings On Computing: A Personal Perspective*, pp. 129-131, Springer-Verlag, New York, 1982.

[Dony88]
Christophe Dony, "An Object-oriented Exception Handling System for an Object-oriented Language," *European Conference on Object-Oriented Programming*, pp. 146-161, Springer-Verlag, Oslo, Norway, August 1988.

[Dony90]
Christophe Dony, "Exception Handling and Object-Oriented Programming: towards a synthesis," *ACM SIGPLAN Notices*, vol. 25, no. 10, pp. 322-330, October 1990. Proceedings of the ECOOP/OOPSLA '90

[Eckhardt85]
H. Eckhardt, J. Edelmann, J. Koch, M. Mall, J.W. Schmidt, "Draft Report on the Database Programming Language DBPL," DBPL Memo 091-85, Johann Wolfgang Goethe-Universität, Fachbereich Informatik, Frankfurt, September 1985.

[Egorov92]
Igor I. Egorov, Ruslan P. Bogatyrev, Dmitry L. Petrovichev, "Yet Another Approach for Exception Handling in Modula-2," *The ModulaTor*, vol. 2, no. 4, pp. 1-28, May 1992.

[Egorov93]
Igor I. Egorov, Ruslan P. Bogatyrev, Dmitry L. Petrovichev, "Yet Another Approach to Modula-2 Implementation of Exception Handling Mechanism," *Structured Programming*, vol. 14, pp. 23-26, 1993.

[Ellis90]
Margaret A. Ellis, Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison Wesley, Murray Hill, NJ, 1990.

[Franz94]
Michael Franz, "Technological Steps toward a Software Component Industry," *Lecture Notes in Computer Science 782*, pp. 259-281, Springer-Verlag, Zürich, March 1994.

[Gehani92]
N.H. Gehani, "Exceptional C or C with Exceptions," *Software—Practice and Experience*, vol. 22, no. 10, pp. 827-848, October 1992.

[Gehani93]
Narain H. Gehani, "Capsules: A Shared Memory Access Mechanism for Concurrent C/C++," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 795-811, July 1993.

[Goldberg84]
Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison Wesley, Reading, Massachusetts, 1984.

[Goldberg89]
Adele Goldberg, David Robson, *Smalltalk-80: The Language,* Addison-Wesley, Reading, Massachusetts, 1989 (reprinted with corrections).

[Goodenough75]
John B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683-696, December 1975.

[Gosling89]
James Gosling, David S.H. Rosenthal, Michelle Arden, *The NeWS Book,* Springer-Verlag, New York, 1989.

[Gries81]
David Gries, *The Science of Programming,* Springer-Verlag, New York, 1981.

[Griffiths91]
Laurie Griffiths, "Modula-2 Is Three Times Less Error Prone Than C," *Second International Modula-2 Conference*, pp. 332-338, Loughborough University of Technology, Loughborough, UK, September 1991.

[Gutknecht93a]
Jürg Gutknecht, "The Tragedy of Programming Language Development," *Structured Programming*, vol. 14, pp. 49-55, 1993.

[Gutknecht93b]
Jürg Gutknecht, *Oberon System 3 — A Realm of Persistent Objects,* ETH Zürich, Zürich, June 1993.

[Hansen75]
Per Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 199-207, June 1975.

[Harbison92]
Samuel P. Harbison, *Modula-3,* Prentice Hall, 1992.

[Hendler86]
James Hendler, "Enhancement for multiple-inheritance," *ACM SIGPLAN Notices*, vol. 21, no. 10, pp. 98-106, October 1986.

[Herlihy82]
M. Herlihy, B. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 527-551, October 1982.

[Hoare73]
C. A. R. Hoare, N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica*, vol. 2, pp. 335-355, 1973.

[Hoare74]
C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549-557, October 1974.

[Hoare81]
Charles Antony Richard Hoare, "The Emperor's Old Clothes," *Communications of the ACM*, vol. 24, no. 2, pp. 75-83, February 1981.

[Hoare85]
C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall, London, 1985.

[Hohlfeld84]
Bernhard Hohlfeld, "Programmverifikation in lauffähigen PASCAL-Programmen," *8. Fachtagung Programmiersprachen und Programmentwicklung*, pp. 26-40, Springer-Verlag, Zürich, März 1984.

[Horowitz83]
Ellis Horowitz, *Fundamentals of Programming Languages,* Springer-Verlag, Berlin, 1983.

[Ichbiah79]
J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, B.A. Wichmann, "Rationale for the Design of the ADA Programming Language," *ACM SIGPLAN Notices*, vol. 14, no. 6B, June 1979.

[Jarke82]
Matthias Jarke, Joachim W. Schmidt, "Query Processing Strategies in the Pascal/R Relational Database Management System," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 256-264, Orlando, June 1982.

[Jones92]
Rick Jones, "Extended type checking in Eiffel," *Journal of Object-Oriented Programming*, pp. 59-62, May 1992.

[Kelly89]
John P.J. Kelly, ‘‘Software design diversity,’’ in *Dependability of resilient computers*, ed. by T. Anderson, pp. 138-166, BSP Professional Books, Oxford, 1989.

[Kernighan78]
Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[King89]
Roger King, ‘‘My Cat is Object-Oriented,’’ in *Object-Oriented Concepts, Databases, and Applications*, ed. by Won Kim, Frederick H. Lochovsky, pp. 23-30, Addison Wesley, New York, 1989.

[Knudsen87]
Jørgen Lindskov Knudsen, ‘‘Better Exception-Handling in Block-Structured Systems,’’ *IEEE Software*, vol. 4, pp. 40-49, May 1987.

[Knudsen88]
Jørgen Lindskov Knudsen, Ole Lehrmann Madsen, ‘‘Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages,’’ *European Conference on Object-Oriented Programming*, pp. 21-40, Springer-Verlag, Oslo, Norway, August 1988.

[Koch83]
J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, C.A. Zehnder, ‘‘Modula/R Report,’’ LIDAS Memo 091-83, ETH Zürich, Institut für Informatik, Zürich, September 1983.

[Koch92]
Frank A. Koch, ‘‘Produkthaftung für Software — Haftungsentlastung durch Qualitätssicherung,’’ *Handbuch der modernen Datenverarbeitung*, vol. 29, no. 163, pp. 54-61, Forkel-Verlag, Wiesbaden, Januar 1992.

[Koenig90]
Andrew Koenig, Bjarne Stroustrup, ‘‘Exception Handling for C++,’’ *Journal of Object-Oriented Programming*, vol. 3, no. 2, pp. 16-33, July/August 1990.

[Kofler93]
Thomas Kofler, ‘‘Robust Iterators in ET++,’’ *Structured Programming*, vol. 14, pp. 62-85, 1993.

[Krueger92]
Charles W. Krueger, ‘‘Software Reuse,’’ *ACM Computing Surveys*, vol. 24, no. 2, pp. 131-183, June 1992.

[Lalis94]
Spiros Lalis, Beverly A. Sanders, ‘‘Adding Concurrency to the Oberon System,’’ in *Programming Languages and System Architectures*, ed. by Jürg Gutknecht, pp. 328-344, Springer-Verlag, Berlin, March 1994.

[LaLonde86]
Wilf R. LaLonde, Dave A. Thomas, John R. Pugh, ‘‘An Exemplar Based Smalltalk,’’ *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 322-330, November 1986.

[Leavens91]
Gary T. Leavens, ‘‘Modular Specification and Verification of Object-Oriented Programs,’’ *IEEE Software*, pp. 72-80, July 1991.

[Leffler89]
Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System,* Addison Wesley, Reading, MA, 1989.

[Levy91]
Henry M. Levy, Ewan D. Tempero, ‘‘Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation,’’ *Software—Practice and Experience*, vol. 21, no. 1, pp. 77-90, January 1991.

[Lieberman86]
Henry Lieberman, ‘‘Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems,’’ *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 214-223, November 1986.

[Lieberman88]
Henry Lieberman, Lynn Andrea Stein, David Ungar, "Of Types and Prototypes: The Treaty of Orlando," *ACM SIGPLAN Notices*, vol. 23, no. 5, pp. 43-44, May 1988.

[Liskov77]
Barbara Liskov, Alan Snyder, Russell Atkinson, Craig Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, vol. 20, no. 8, pp. 564-576, August 1977.

[Liskov79]
Barbara H. Liskov, Alan Snyder, "Exception Handling in CLU," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6, pp. 546-558, November 1979.

[Liskov81]
Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, Alan Snyder, "CLU Reference Manual," Lecture Notes in Computer Science 114, Springer-Verlag, Berlin, 1981.

[MacLaren77]
M. Donald MacLaren, "Exception Handling in PL/I," *ACM SIGPLAN Notices*, vol. 12, no. 3, pp. 101-104, March 1977.

[Madsen88]
Ole Lehrmann Madsen, Birger Møller-Pedersen, "What object-oriented programming may be — and what it does not have to be," *European Conference on Object-Oriented Programming*, pp. 1-20, Springer-Verlag, Oslo, Norway, August 1988.

[Madsen93]
Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard, *Object-Oriented Programming in the BETA Programming Language,* Addison Wesley, Wokingham, England, 1993.

[McIlroy76]
M.D. McIlroy, "Mass Produced Software Components," in *Software Engineering — Concepts and Techniques*, ed. by Peter Naur, Brian Randell, J.N. Buxton, pp. 88-95, Petrocelli Charter, New York, 1976. Proceedings of the NATO Conference held at Garmisch, Germany, 1968

[Melliar-Smith77]
P.M. Melliar-Smith, B. Randell, "Software Reliability: The Role of Programmed Exception Handling," *ACM SIGPLAN Notices*, vol. 12, no. 3, pp. 95-100, March 1977. Proceedings of the ACM Conference on Language Design for Reliable Software

[Meyer86]
Bertrand Meyer, "Genericity versus Inheritance," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 391-405, November 1986.

[Meyer88]
Bertrand Meyer, *Object-oriented Software Construction,* Prentice Hall, New York, 1988.

[Meyer90]
Bertrand Meyer, "Tools for the New Culture: Lessons from the Design of the Eiffel Libraries," *Communications of the ACM*, vol. 33, no. 9, pp. 69-88, September 1990.

[Meyer92]
Bertrand Meyer, *Eiffel: The Language,* Prentice Hall, New York, 1992.

[Meyer94]
Bertrand Meyer, *Reusable Software — The Base Object-Oriented Component Libraries,* Prentice Hall, Hertfordshire, UK, 1994.

[Millo79]
Richard A. De Millo, Richard J. Lipton, Alan J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM*, vol. 22, no. 5, pp. 271-280, May 1979.

[Milner89]
Robin Milner, *Communication and Concurrency,* Prentice Hall, Hertfordshire, UK, 1989.

[Moon86]
David A. Moon, "Object-Oriented Programming with Flavors," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 1-8, November 1986.

[Moon89]
David A. Moon, "The Common Lisp Object-Oriented Programming Language Standard," in *Object-Oriented Concepts, Databases, and Applications*, ed. by Won Kim, Frederick H. Lochovsky, pp. 49-78, Addison Wesley, New York, 1989.

[Moylan93]
P.J. Moylan, "The case against C," *The ModulaTor*, vol. 3, no. 6, pp. 1-11, July 1993.

[Mössenböck89]
Hanspeter Mössenböck, Josef Templ, "Object Oberon — A Modest Object-Oriented Language," *Structured Programming*, vol. 10, no. 4, pp. 199-207, 1989.

[Mössenböck93]
Hanspeter Mössenböck, *Objektorientierte Programmierung in Oberon-2,* Springer-Verlag, Berlin, 1993.

[Nelson91]
Systems Programming with Modula-3, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Noble69]
J.M. Noble, "The Control of Exceptional Conditions in PL/1 Object Programs," *Proceedings IFIP Congress 1968*, pp. 565-571, North-Holland, Amsterdam, 1969.

[Padiou90]
G. Padiou, "Nested Coroutines for Exception Handling in Modula-2," *Structured Programming*, vol. 11, pp. 79-83, 1990.

[Parnas72]
D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

[Parnas86]
David Lorge Parnas, Paul C. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251-257, February 1986.

[Raj91]
Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, Eric Jul, "Emerald: A General-Purpose Programming Language," *Software—Practice and Experience*, vol. 21, no. 1, pp. 91-118, January 1991.

[Reimer83]
Manuel Reimer, "Implementation of the Database Programming Language Modula/R on the Personal Computer Lilith," in *Database Techniques for Professional Workstations*, ed. by Carl August Zehnder, pp. 49-64, ETH Zürich, Institut für Informatik, Zürich, September 1983. Report 55

[Reiser91]
Martin Reiser, *The Oberon System,* Addison Wesley, New York, 1991.

[Reiser92]
Martin Reiser, Niklaus Wirth, *Programming in Oberon,* Addison-Wesley, New York, 1992.

[Richardson93]
Joel E. Richardson, Michael J. Carey, Daniel T. Schuh, "The Design of the E Programming Language," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 3, pp. 494-534, July 1993.

[Ritchie84]
Dennis M. Ritchie, "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1577-1593, October 1984.

[Rovner86]
Paul Rovner, "Extending Modula-2 to Build Large, Integrated Systems," *IEEE Software*, vol. 3, no. 6, pp. 46-57, November 1986.

[Sakkinen92]
Markku Sakkinen, "The Darker Side of C++ Revisited," *Structured Programming*, vol. 13, pp. 155-177, 1992.

[Schaffert86]
Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt, "An Introduction to Trellis/Owl," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 9-16, November 1986.

[Shaw77]
Mary Shaw, William A. Wulf, Ralph L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," *Communications of the ACM*, vol. 20, no. 8, pp. 553-564, August 1977.

[Shopiro89]
Jonathan E. Shopiro, "An Example Of Multiple Inheritance in C++: A Model of the Iostream Library," *ACM SIGPLAN Notices*, vol. 24, no. 12, pp. 32-36, December 1989.

[Snyder86]
Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 38-45, November 1986.

[SPARC International, Inc.92]
SPARC International, Inc., *The SPARC Architecture Manual — Version 8,* Prentice Hall, Englewood Cliffs, 1992.

[Spector84]
Alfred Spector, David Gifford, "The Space Shuttle Primary Computer System," *Communications of the ACM*, vol. 27, no. 9, pp. 874-900, September 1984.

[Stein87]
Lynn Andrea Stein, "Delegation is Inheritance," *ACM SIGPLAN Notices*, vol. 22, no. 10, pp. 138-146, October 1987.

[Stein89]
Lynn Andrea Stein, Henry Lieberman, David Ungar, "A Shared View of Sharing: The Treaty of Orlando," in *Object-Oriented Concepts, Databases, and Applications*, ed. by Won Kim, Frederick H. Lochovsky, pp. 31-48, Addison Wesley, New York, 1989.

[Stroustrup86]
Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading, Massachusetts, 1986.

[Stroustrup87]
Bjarne Stroustrup, Jonathan E. Shopiro, "A Set of C++ Classes for Co-routine Style Programming," *Proceedings of USENIX C++ Workshop*, pp. 417-439, USENIX Association, Santa Fe, November 1987.

[Stroustrup91]
Bjarne Stroustrup, *The C++ Programming Language,* Addison Wesley, Reading, Massachusetts, 1991. Second Edition

[Szyperski92a]
Clemens A. Szyperski, "Import is Not Inheritance: Why We Need Both: Modules and Classes," *European Conference on Object-Oriented Programming*, pp. 19-32, Springer-Verlag, Utrecht, The Netherlands, June/July 1992.

[Szyperski92b]
Clemens A. Szyperski, "Write-ing Applications: Design of an Extensible Text Editor as an Application Framework," *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92)*, pp. 247-261, Prentice-Hall, March 1992.

[Templ93]
Josef Templ, "A Systematic Approach to Multiple Inheritance Implementation," *ACM SIGPLAN Notices*, vol. 28, no. 4, pp. 61-66, April 1993.

[Tomlinson89]
Chris Tomlinson, Mark Scheevel, "Concurrent Object-Oriented Programming Languages," in *Object-Oriented Concepts, Databases, and Applications*, ed. by Won Kim, Frederick H. Lochovsky, pp. 79-124, Addison Wesley, New York, 1989.

[Tsichritzis87]
D. Tsichritzis, E. Fiume, S. Gibbs, O. Nierstrasz, ‘‘KNOs: *KNPowledge Acquisition, Dissemination, and Manipulation Objects,’’ ACM Transactions on Office Information Systems*, vol. 5, no. 1, pp. 96-112, January 1987.

[Wegner90]
Peter Wegner, ‘‘Concepts and Paradigms of Object-Oriented Programming,’’ *ACM OOPS Messenger*, vol. 1, no. 1, pp. 7-87, 1990.

[Wilde92]
Norman Wilde, Ross Huitt, ‘‘Maintenance Support for Object-Oriented Programs,’’ *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1038-1044, December 1992.

[Wirfs-Brock89]
Rebecca Wirfs-Brock, Brian Wilkerson, ‘‘Object-Oriented Design: A Responsibility-Driven Approach,’’ *ACM SIGPLAN Notices*, vol. 24, no. 10, pp. 71-75, October 1989.

[Wirfs-Brock90]
Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Wirth82]
Niklaus Wirth, *Programming in Modula-2,* Springer-Verlag, Berlin, 1982.

[Wirth85]
Niklaus Wirth, *Programming in Modula-2,* Springer-Verlag, Berlin, 1985. Third Edition

[Wirth87]
Niklaus Wirth, ‘‘Extensions of Record Types,’’ *SIGCSE Bulletin*, vol. 19, no. 2, pp. 2-9, June 1987.

[Wirth88a]
Niklaus Wirth, ‘‘The Programming Language Oberon,’’ *Software—Practice and Experience*, vol. 18, no. 7, pp. 671-690, July 1988.

[Wirth88b]
Niklaus Wirth, ‘‘Type Extensions,’’ *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 2, pp. 204-214, April 1988.

[Wirth88c]
Niklaus Wirth, Jürg Gutknecht, ‘‘The Oberon System,’’ Report 88, ETH Zürich, Institut für Computersysteme, July 1988.

[Wirth88d]
Niklaus Wirth, ‘‘From Modula to Oberon,’’ *Software—Practice and Experience*, vol. 18, no. 7, pp. 661-670, July 1988.

[Wirth89a]
Niklaus Wirth, Jürg Gutknecht, ‘‘The Oberon System,’’ *Software—Practice and Experience*, vol. 19, no. 9, pp. 857-893, September 1989.

[Wirth89b]
Niklaus Wirth, ‘‘Designing a System from Scratch,’’ *Structured Programming*, vol. 10, no. 1, pp. 10-18, 1989.

[Wirth89c]
Niklaus Wirth, ‘‘The Programming Language Oberon,’’ Report 111, ETH Zürich, Institut für Computersysteme, September 1989.

[Wirth89d]
Niklaus Wirth, ‘‘From Modula to Oberon,’’ Report 111, ETH Zürich, Institut für Computersysteme, September 1989.

[Wirth89e]
Niklaus Wirth, ‘‘Modula-2 and Object-Oriented Programming,’’ *Proceedings of the First International Modula-2 Conference*, pp. 7-13, Jožef Stefan Institute, Ljubljana, Slovenia, October 1989.

[Wirth89f]
Niklaus Wirth, “Corrections: The Programming Language Oberon,” *Software—Practice and Experience*, vol. 19, no. 1, January 1989.

[Wirth92]
Niklaus Wirth, Jürg Gutknecht, *Project Oberon,* Addison-Wesley, New York, 1992.

[Wirth94]
Niklaus Wirth, “Gedanken zur Software-Explosion,” *Informatik-Spektrum*, vol. 17, no. 1, pp. 5-10, Februar 1994.

[Yonezawa86]
Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama, “Object-Oriented Concurrent Programming in ABCL/1,” *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 258-268, November 1986.

D. Verzeichnis der Abbildungen