Kapitel 1

Motivation und Einführung

Als auf der NATO-Tagung 1978 in Garmisch-Partenkirchen die Begriffe Software-Krise und Software-Engineering als Technik zur Lösung geprägt wurden sind, hatte McIlroy in seinem Vortrag die Vision, daß sich die Software-Produktion grundlegend ändern könnte. Statt alles jedesmal aufs Neue von Grund auf zu entwickeln (analog zum Handwerk), sollte es doch möglich sein, Standard-Komponenten mit präzisen Spezifikationen zu entwickeln, die nur noch zusammenzusetzen sind (analog zur industriellen Produktion).

Entscheidend bei der Entwicklung und dem Zusammenbau von Komponenten sind die Schnittstellen bzw. Protokolle, die in Form von Abstraktionen zusammengefaßt werden. Leider fehlten den klassischen Programmiersprachen (z.B. Fortran, Cobol, Algol, Pascal und C) jegliche Unterstützung zur Formulierung von Abstraktionen und der Gewährleistung von Schnittstellensicherheit.

Bei einigen Programmiersprachen der 70er Jahre (z.B. Mesa, CLU, Modula-2 und Ada) wurde ein Modulkonzept eingeführt, das eine erste Unterstützung brachte. Allerdings war dabei die Einschränkung, daß zu einer Abstraktion nur eine Implementierung gleichzeitig im Programm vorhanden sein konnte. Es war also nicht möglich, das gleiche Protokoll für Objekte unterschiedlichen Typs zu verwenden.

Dieses Defizit wurde mit der Einführung der objekt-orientierten Programmiersprachen geschlossen (z.B. Simula, Smalltalk, Eiffel, C++ und Java). Leider kann bei der großen Mehrheit dieser Sprachen eine Abstraktion nur einen Objekt-Typ umfassen – es ist somit nicht möglich, daß im Rahmen einer Abstraktion mehrere Objekt-Typen definiert und miteinander in Beziehung gebracht werden. Ausnahmen sind hier z.B. Oberon, Modula-3 und BETA.

Dabei verdient der Punkt Beachtung, daß Abstraktionen nicht nur innerhalb eines Programms oder abgeschlossenen Software-Systems relevant sind. Vielmehr basieren alle Formen der Interaktion auf dem Internet auf Abstraktionen. Wenn somit eine geeignete Infrastruktur zur Verfügung steht (z.B. CORBA von der OMG), dann können bei der Etablierung entsprechender Standards beispielsweise Bestellungen, Rechnungen und Kontobewegungen über ein globales Netz abgewickelt werden.

Entscheidend sind aber nicht nur die sprachlichen Voraussetzungen, sondern ein ganzes Bündel von Software-Entwicklungstechniken zur Verbesserung der Software-Qualität, die sich aufgrund einer geeigneten Programmiersprache besser und kontrollierter umsetzen lassen. Entsprechend der Definition von Bertrand Meyer ist eine Unterscheidung in äußere und innere Qualität möglich. Die äußere Qualität ist die, die von dem Benutzer der Software wahrgenommen werden kann. Dies betrifft die Korrektheit (inwieweit entspricht die Software der Spezifikation), die Robustheit (inwiefern werden abnormale Randbedingungen toleriert und überlebt) und schließlich ergonomische Aspekte. Diese äußere Qualität ist kein Himmelsgeschenk, sondern läßt sich nur auf Basis ingenieurmäßiger Techniken erreichen, die in den inneren Qualitäten der Software ihren Ausdruck finden.

Schlüsselfaktoren nach Bertrand Meyer sind dabei:

Korrektheit Die Übereinstimmung der Software mit der Spezifikation ist sicherlich das primäre Ziel. Leider ist es nicht nur sehr schwierig, einen entsprechenden Nachweis zu führen, sondern überhaupt erst eine konsistente Spezifikation zu erstellen.

Robustheit Robustheit ergänzt Korrektheit dahingehend, daß auch bei Randbedingungen, die von der Spezifikation nicht vorgegeben sind, in sinnvoller Weise reagiert wird.

Erweiterbarkeit Da Spezifikationen laufend erweitert und geändert werden, sollte es möglich sein, solche Änderungen mit einem Minimum an Aufwand in die Software aufzunehmen.

Wiederverwendbarkeit Häufig werden viele Software-Produkte parallel oder hintereinander entwickelt – da zahlt es sich aus, wenn der Entwicklungsaufwand durch das erneute Einsetzen erprobter Komponenten reduziert werden kann.

Kompatibilität Software-Komponenten sollten problemlos miteinander kombinierbar sein – auch wenn sie unabhängig voneinander entwickelt worden sind.

Effizienz Obwohl Effizienz typischerweise kein primäres Ziel ist, kann es über Tauglichkeit und wirtschaftliche Einsetzbarkeit entscheiden.

Portabilität Software sollte normalerweise soweit wie möglich unabhängig von konkreten Plattformen sein.

Leichte Benutzbarkeit Software sollte in konsistenter und leicht erlernbarer Weise für Personen mit verschiedenem Hintergrund benutzbar sein.

Funktionalität Die richtige Wahl zwischen einer spartanischen Oberfläche und einem Wirrwarr überflüssiger Möglichkeiten ist nicht einfach zu finden.

Rechtzeitigkeit Nicht zuletzt ist es auch wichtig, daß Software pünktlich fertig wird.

Integrität Im vernetzten Umfeld ist es wichtig, daß unzulässige Zugriffe oder Eingriffe abgewehrt werden.

Die Techniken, die weitgehend diesen Zielen dienen, lassen sich vereinfacht unter dem Stichwort "objekt-orientierte Entwicklung" zusammenfassen, wenngleich es weder allgemein akzeptierte präzise Definitionen noch etablierte Standards dafür gibt. Der Begriff der Objekt-Orientierung selbst ist relativ unglücklich – es wäre wohl passender gewesen, den Begriff "abstraktions-orientiert" zu verwenden.

Die Hauptzielrichtung objekt-orientierter Entwicklungstechniken liegt darin, jede Software-Komponente in eine abstrakte Welt einzubetten, um Abhängigkeiten zur realen Welt (und der gegebenen Spezifikation) zu minimieren oder gar ganz zu entfernen. Das Problem liegt darin, geeignete abstrakte Welten zu finden und zu strukturieren, die zu dem vorgegebenen Problemfeld passen – jedoch gleichzeitig so allgemein sind, so daß sie invariant gegenüber Änderungen des Problemfeldes sind.

Wenn abstrakte Welten und Strukturen genügend allgemein sind, werden daraus geeignete Kandidaten für den Aufbau einer objekt-orientierten Bibliothek. Letztendlich ist das Ziel, daß bei einer neuen Anforderung es nur notwendig ist, Abbildungen vorhandener Abstraktionen aus der Bibliothek auf konkrete Gegebenheiten geeignet miteinander zu kombinieren.

Sofern dieses Ziel noch nicht erreicht ist, empfiehlt es sich, nicht die neuen Anwendungen unnötig aufzublähen, sondern stattdessen nach den neuen notwendigen Abstraktionen und Verallgemeinerungen zu suchen, die für die neue Anwendung noch fehlen. Das Problem ist hier natürlich immer, daß dies jeweils einen höheren Aufwand erfordert, der sich erst in der Zukunft auszahlen wird – sei es bei der späteren Wartung der Software oder bei der Entwicklung weiterer ähnlicher Anwendungen.

Das Ziel der Vorlesung ist es, eine Einführung in die dafür notwendigen Techniken zu geben und sie anhand einer Reihe von allgemein nützlichen Abstraktionen der Ulmer Oberon-Bibliothek zu demonstrieren. Diese Bibliothek wurde deswegen gewählt, weil die zugrundeliegende Programmiersprache angenehm einfach ist und damit keine zusätzliche Hürde darstellt. Ferner gibt es leider nur sehr wenige objekt-orientierten Bibliotheken in öffentlich frei verfügbarer Form – die Ulmer Oberon-Bibliothek ist da eine der ganz wenigen Ausnahmen.

Darüber hinaus sind diese Techniken weitgehend unabhängig von der verwendeten Programmiersprache. Sie können also beispielsweise auch für Eiffel oder C++ eingesetzt werden.

Kapitel 2

OO-Techniken in Oberon

2.1 Einführung

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 nicht notwendig, da insbesondere 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.

2.2 Das Modulkonzept von Oberon

Obwohl sich das Modulkonzept von Oberon von Modula-2 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.

Die interessantere Vereinfachung ergibt sich aus dem Verzicht auf nicht qualifizierte 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 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. 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 Records 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, im späteren Report verschwand jedoch diese Restriktion. Beim Ulmer Oberon-Übersetzer wurde die Restriktion beibehalten.

2.3 Typerweiterungen

Recordtypen können als Erweiterung eines anderen Recordtyps 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 2.1 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 die Zuweisung eines Basistyps an eine Erweiterung nicht zugelassen, da dann Komponenten undefiniert bleiben würden:

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

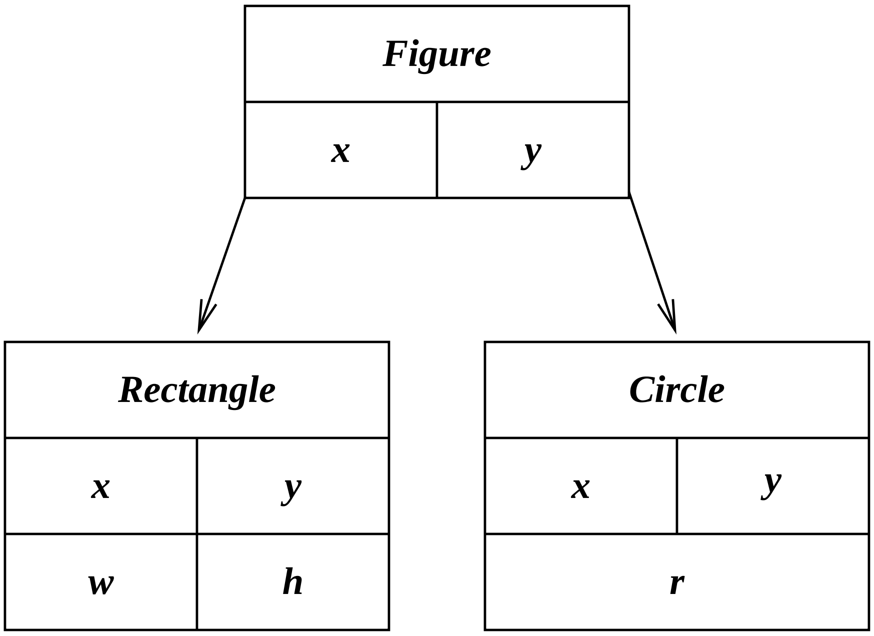

Abbildung 2.1: Einfache Typenhierarchie mit den zugehörigen Komponenten

```
(* ... *)

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 Record 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 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*.

Erweiterte Typen können erneut erweitert werden: Ein Recordtyp U ist eine Erweiterung des Recordtyps 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 Recordtyp T genau dann ein Basistyp eines Recordtyps 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 Records 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.

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 Records 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" Records 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 Records 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:

```
-- 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++ ist ein entsprechender nur Mechanismus geplant. 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 und etwas später in sehr ähnlicher Form in Ada. Wie Bertrand Meyer gezeigt hat, 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 ist Verwaltung 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);

   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;
```

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

```
   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. 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.

Wenn im obigen Beispiel *WriteLines* absolut sichergehen kann, daß alle Objekte der Liste Erweiterungen von *LineObject* sind, dann kann der Programmtext weiter vereinfacht 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 es kann mit konventioneller Technik analog zu Modula-2 ein Prozedurparameter explizit angegeben werden.

2.6 Aufträge und Auftragnehmer

Die Technik des *late binding* wird auf Basis von Record-Komponenten erreicht, die einen Prozedurtyp besitzen. 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 AddMessage THEN
            WITH message: AddMessage DO
               Append(collection, message.object);
            END;
         ELSIF message IS FirstMessage THEN
            First(collection);
         ELSIF message IS NextMessage THEN
            WITH message: 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 2.2 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. 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 darüberlie-

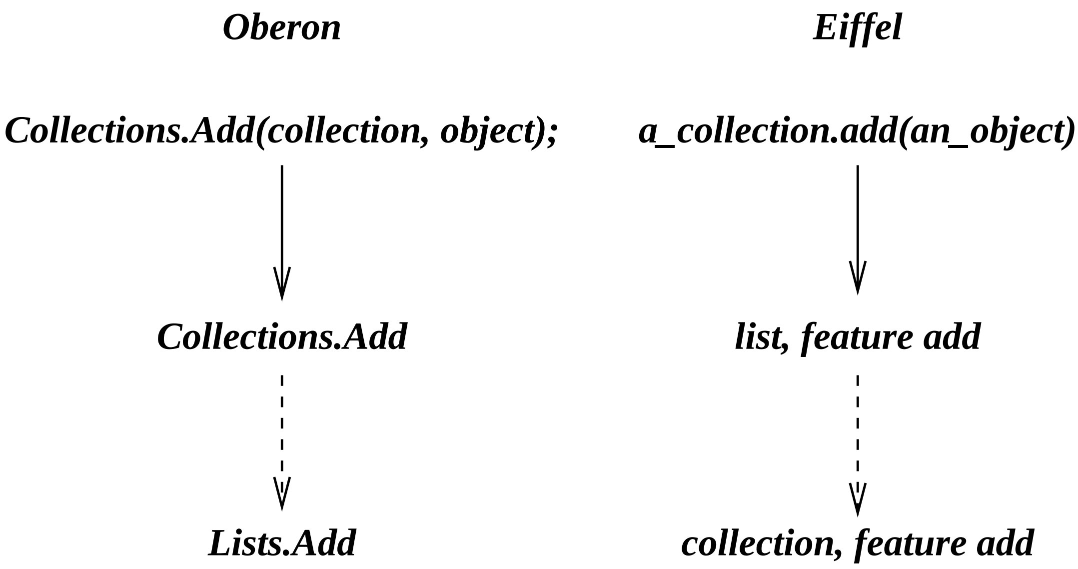

Abbildung 2.2: Unterschiedlicher Kontrollfluß bei Oberon und klassischen OO-Sprachen

genden Klasse aufrufen kann.

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: 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; VAR 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 von Delegation selbst unbekannter Operationen, jedoch verbleiben einige Nachteile:

- 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 Schnittstellenrecords. 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 Schnittstellenrecord 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(primary, object);
         Collections.Add(secondary, object);
```

```
      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 Schnittstellenrecords von der Menge der Fähigkeiten ergibt sich ganz deutlich: Der Schnittstellenrecord 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, das als schwer zu lösendes Problem im Bereich der objekt-orientierten Sprachen gilt. 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:[2]

```
DEFINITION Colours;

   IMPORT Figures;

   TYPE
      Colour = INTEGER;

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

END Colours.
```

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

```
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;

   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 Schnittstellenrecords mit Hilfe von *Disciplines* an ein Objekt anzuhängen. Allerdings werden in dieser Form weder nicht überdefinierbare Operationen (wie die Operationen von *Co-*

lours) noch ggf. die Operationen, die auf Basis von weiteren Schnittstellenrecords 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 Records 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 Schnittstellenrecords 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 bei Trellis/Owl verwendet.

Während sich mit den vorgestellten Konventionen auch außergewöhnliche Techniken (Delegationen, Farbenproblem) 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 zu integralen Bestandteilen einer Programmiersprache zu machen, führt dies zu einer völligen Überladung (siehe C++). 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.

Kapitel 3

Organisation und Aufbau von Bibliotheken

3.1 Einführung

Bibliotheken sollten mehr sein als nur ein Sammelsurium von Hinterlassenschaften aus früheren Zeiten, da sich eine sinnvolle Nutzung nur dann ergibt, wenn der Aufwand des Findens, Verstehens und Einbauens deutlich geringer ist als der des Neuschreibens. Um dies zu ermöglichen, ist ein großes Maß an Ordnungsprinzipien und Konventionen unerläßlich.

Die Problematik läßt sich leicht an einigen Negativbeispielen der historisch gewachsenen und heute weitgehend standardisierten C-Bibliothek demonstrieren:

- Während *gets* den Zeilentrenner aus der Eingabezeile entfernt, verbleibt bei der verallgemeinerten Variante *fgets* der Zeilentrenner im Zielpuffer. Schlimmer noch: Während *fgets* neben der Angabe der Dateiverbindung auch die Pufferlänge verlangt und damit sicherstellt, daß der Puffer nicht überläuft, verzichtet hingegen *gets* auf diese Angabe und öffnet daher ein Sicherheitsrisiko, falls die Länge der Eingabezeile die des Zielpuffers überschreitet.

- Die Positionierung der Parameter ist innerhalb der *stdio* inkonsistent: Während z.B. *fseek* und *ftell* die Dateiverbindung als ersten Parameter erwarten, wird sie bei *fgets* oder *fputc* als letztes angegeben.

- Fehler werden in mannigfaltiger Form zurückgegeben: Während *fgets* 0 als Fehlerindikation versteht, liefert *getc* -1 im Fehlerfalle.

Entsprechend ergibt sich auch eine der Hauptforderungen von Bertrand Meyer für Bibliotheken (aus dem Buch "Reusable Software"):

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

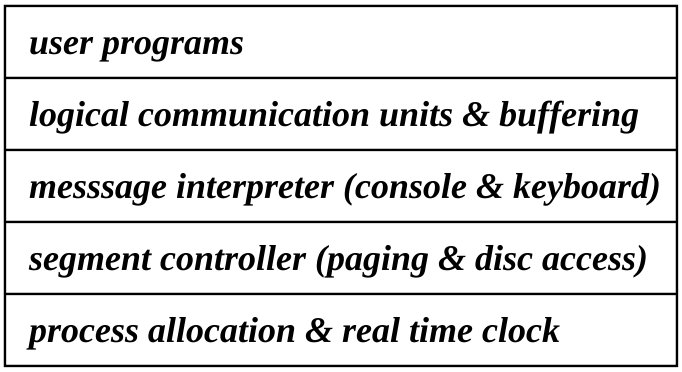

Abbildung 3.1: THE-System von Dijkstra

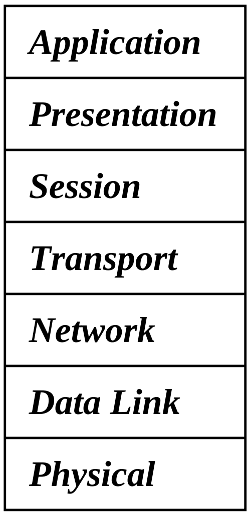

Abbildung 3.2: ISO-Referenz-Modell

So stellt sich heraus, daß es nicht nur sehr anspruchsvoll ist, einzelne Bibliotheksmodule zu entwerfen, sondern noch viel schwerer, sie zu einer harmonischen Einheit zusammenzuführen.

Neben durchgehenden Konventionen, die den Umgang erleichtern, sind auch die Strukturierung in Schichten (entsprechend den Abhängigkeiten und den sich natürlich ergebenden Gruppierungen), die Gestaltung in Hierarchien, systematische Konventionen und eine geeignete Dokumentation Voraussetzungen für eine effektiv einsetzbare Bibliothek.

3.2 Strukturierung in Schichten

Spätestens mit der Publizierung von Dijkstra über "The Structure of 'THE'-Multiprogramming System" im Jahre 1968 wurde eine Strukturierung eines Systems in Schichten entsprechend der Abhängigkeiten populär (siehe Abbildung 3.1). Er zerlegte sein Betriebssystem in fünf Schichten, bei der jede Ebene auf dem Weg von unten nach oben bestimmte Hardware-Ressourcen in abstrakter und virtueller Form zur Verfügung stellte. Bei dieser Strukturierung war sichergestellt, daß jede Ebene nur Aufrufe an sich selbst und die untergeordneten Ebenen absetzt. Damit eröffnet sich nicht nur eine leichtere Verstehbarkeit eines solchen Systems – die einzelnen Ebenen lassen sich auch leichter testen oder austauschen.

Die gleiche Motivation führte auch zu den Schichten aus dem ISO-Referenz-Modell für Netzwerk-Software (siehe Abbildung 3.2). Genauso wie bei dem THE-System von Dijkstra sind mit den einzelnen Schichten Schnittstellen verbunden, und jede Schicht implementiert ein Protokoll mit seiner eigenen Ebene auf der Seite des Kommunikationspartners, das den anderen Schichten verborgen bleibt. Auf diese Weise ähneln Kommunikationspakete russischen Puppen, bei denen jede Schicht nach und nach ihre privaten Informationen generiert (bzw. auf der

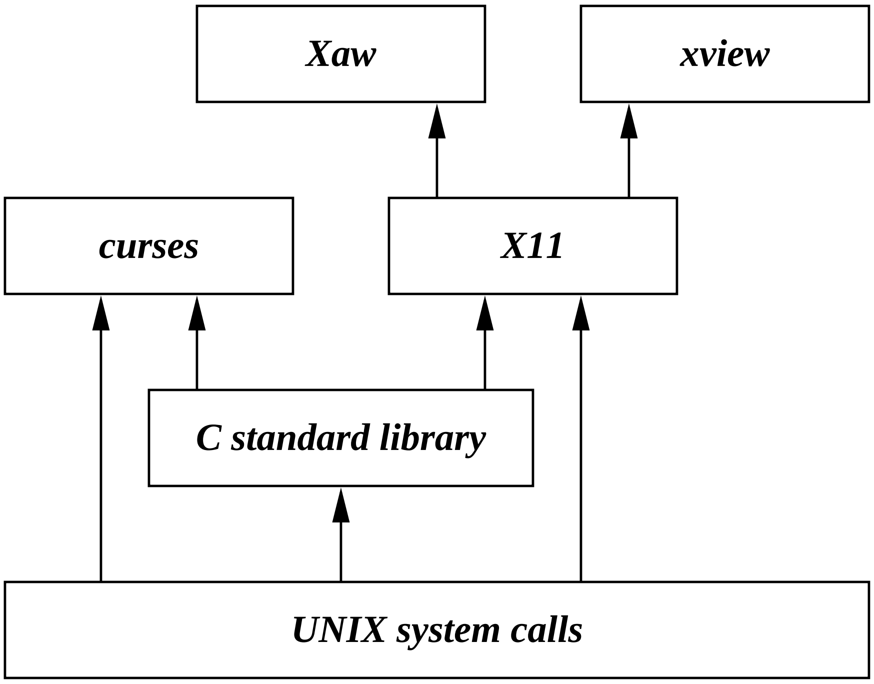

Abbildung 3.3: Auswahl der Schichten gängiger C-Bibliotheken

anderen Seite liest), um dann das Paket der übergeordneten Ebene einzubauen (bzw. herauszunehmen und weiterzugeben).

Genauso wie bei Software-Systemen liegt es auch nahe, bei Bibliotheken Schichten einzuführen. Die Abbildung 3.3 zeigt einen Ausschnitt der Schichten der Bibliotheken in C. Typisch für die Bibliotheksstruktur klassischer Programmiersprachen ist der Aufbau, der mit systemnahen Modulen unten beginnt und nach oben mit eher system-unabhängigen Schichten fortgesetzt wird. Zwar läßt sich beispielsweise eine der Schichten ersetzen, dennoch sind die Abhängigkeiten zu UNIX von *curses* und *X11* unübersehbar. Selbst bei der Beschränkung auf obere Schichten ohne direkten Import der Systemschnittstelle besteht immer das Risiko, daß Systemabhängigkeiten indirekt eingeführt werden. So verwendet und exportiert beispielsweise die C-Standard-Bibliothek eine Reihe von Datentypen, die in der Ebene der Systemaufrufe definiert werden.

Durch die Verwendung objekt-orientierter Techniken lassen sich jedoch die Schnittstellendefinitionen von den zugehörigen Implementierungen trennen, und damit ist es möglich, dieses Beziehungsgeflecht vollkommen umzustülpen. Wie Abbildung 3.4 zeigt, ist es nun möglich, alle Abstraktionen mitsamt den systemunabhängigen Implementierungen in die unterste Schicht zu verlagern. Es läßt sich sogar erreichen, daß eine Applikation nur aus dieser Schicht importiert und damit eine garantierte Systemunabhängigkeit besitzt. Weiter darüber folgt eine systemabhängige Ebene, die eine möglichst sprachnahe Schnittstelle zum System offeriert. Dabei können verschiedene Module aus der systemunabhängigen Schicht importiert werden wie z.B. Mechanismen zur Ausnahmenbehandlung. In der obersten Ebene werden dann Implementierungen der systemunabhängigen Abstraktionen hinzugefügt, die von der Systemschnittstelle aus der mittleren

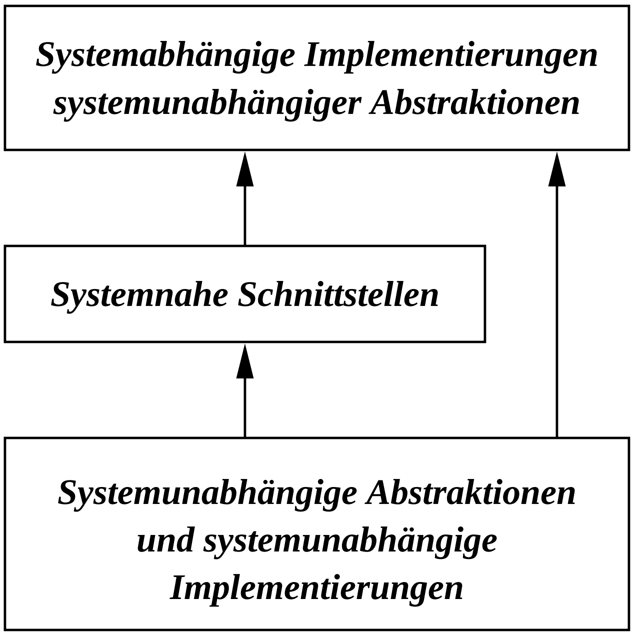

Abbildung 3.4: Trennung zwischen systemabhängigen und systemunabhängigen Schichten

Ebene abhängen.

Natürlich beschränkt sich eine objekt-orientierte Bibliothek sich nicht in jedem Fall auf drei Schichten. Sehr häufig entstehen Schichten, indem bestimmte Funktionalitäten zu Paketen geschnürt werden (z.B. eine Schicht für Container, eine für Persistenz und die nächste für verteilte Systeme). Oder es existieren mannigfaltige externe Abhängigkeiten (z.B. zu einem bestimmten Betriebssystem, einer bestimmten Netzwerkarchitektur, einer Datenbank usw.), die dann entsprechende Schichtenpaare hinterlassen (je eine als direkte Schnittstelle und eine mit darauf basierenden Implementierungen, wobei dies manchmal auch gleich zusammengefaßt wird).

Darüber hinaus ist es denkbar, verschiedene Schicht-Systeme parallel zu unterhalten, die die gleiche Menge an Modulen unterschiedlich unterteilen, z.B. auf der einen Seite funktional orientiert und auf der anderen Seite nach Abhängigkeiten sortiert. Allerdings wird dies normalerweise nicht praktiziert und allenfalls im Rahmen der Dokumentation virtuell erzeugt.

Die Strukturierung in Schichten wird üblicherweise nicht direkt von den objekt-orientierten Programmiersprachen unterstützt. Allerdings gibt es bei einzelnen Programmiersprachen (insbesondere Eiffel (*cluster*) und BETA (*fragments*)) vorgegebene Meta-Sprachen, während bei anderen Programmiersprachen auf die Dienste des Betriebssystems oder anderer spezieller Pakete zurückgegriffen wird. Im einfachsten Fall werden einzelne Schichten in entsprechenden Verzeichnissen untergebracht – so auch bei Oberon.

Die Ulmer Oberon-Bibliothek ist in folgenden acht Schichten organisiert:

base Allgemeine Abstraktionen und Mechanismen mitsamt den systemunabhängigen Implementierungen.

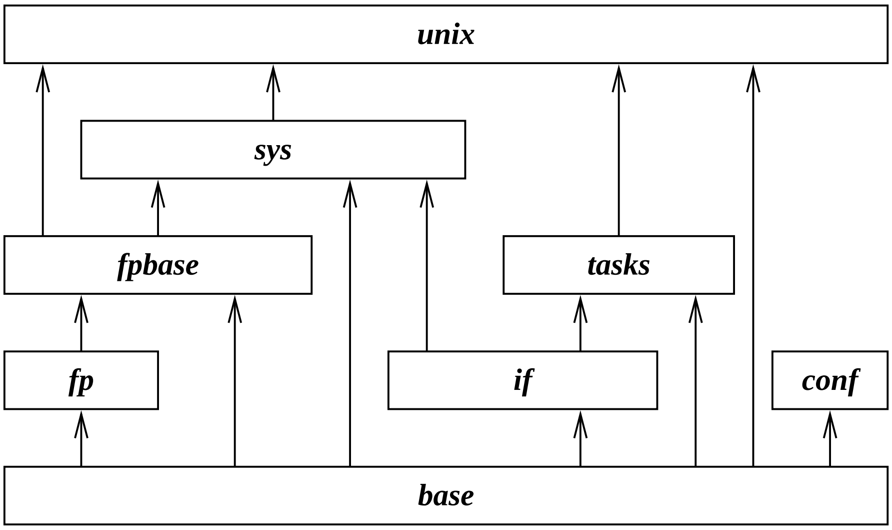

Abbildung 3.5: Importbeziehungen zwischen den Schichten

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.

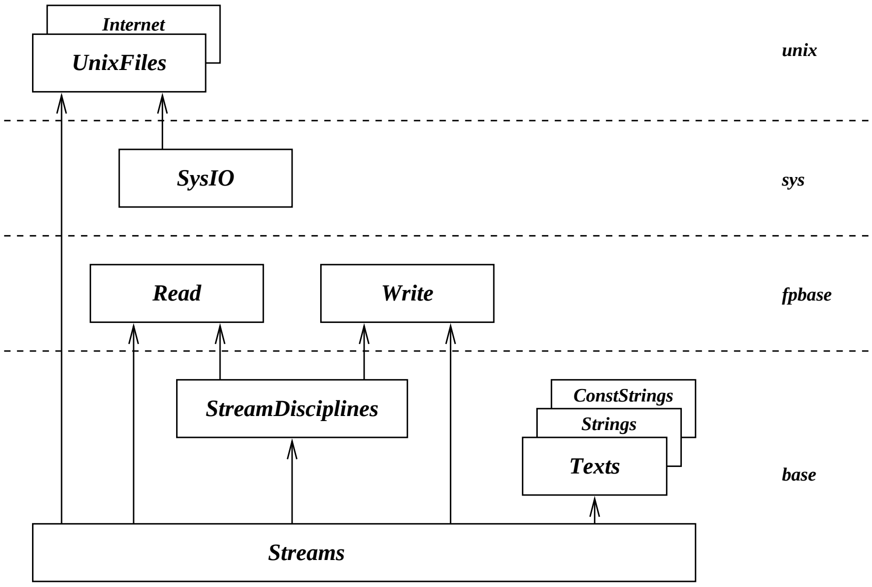

Abbildung 3.6: Modulhierarchie im Ein- und Ausgabebereich

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

3.3 Hierarchien

Das nächste wichtige Ordnungsprinzip bei Bibliotheken sind Hierarchien, wobei es sich nicht immer um Hierarchien im strengen Sinne handelt, sondern häufig nur um gerichtete Graphen ohne Zyklen (DAGs). Entscheidend ist somit die topologische Sortierbarkeit. Die beiden folgenden Beziehungen sind dabei von besonderem Interesse:

Importbeziehung: Auf der Basis von Importbeziehungen lassen sich die irreflexiven Ordnungsrelationen $\rightarrow$ und $\rightarrow^+$ zwischen Modulen definieren (Schnittstelle und **MODULE** werden als zusammengehörig betrachtet):

- $A \rightarrow B$, wenn B das Modul A importiert.
- $A \rightarrow^+ B$, wenn $A \rightarrow B$ oder $\exists M_1, \ldots, M_n$ (mit $n > 0$), so daß $A \rightarrow M_1 \rightarrow \cdots \rightarrow M_n \rightarrow B$.

Zwar lassen viele Programmiersprachen (einschließlich Oberon) Import-Zyklen zu – dennoch sollten sie in jedem Fall vermieden werden (und es ist auch in jedem Fall möglich), da ansonsten die Initialisierungsreihenfolge undefiniert ist.

In den später folgenden Diagrammen, die Modulhierarchien darstellen, beziehen sich die Pfeile immer auf die Relation $\rightarrow$ (wie z.B. in Abbildung 3.6). Wenn zwischen zwei Modulen A und B die Relation $A \rightarrow^+ B$ definiert ist, wird A unterhalb von B angeordnet.

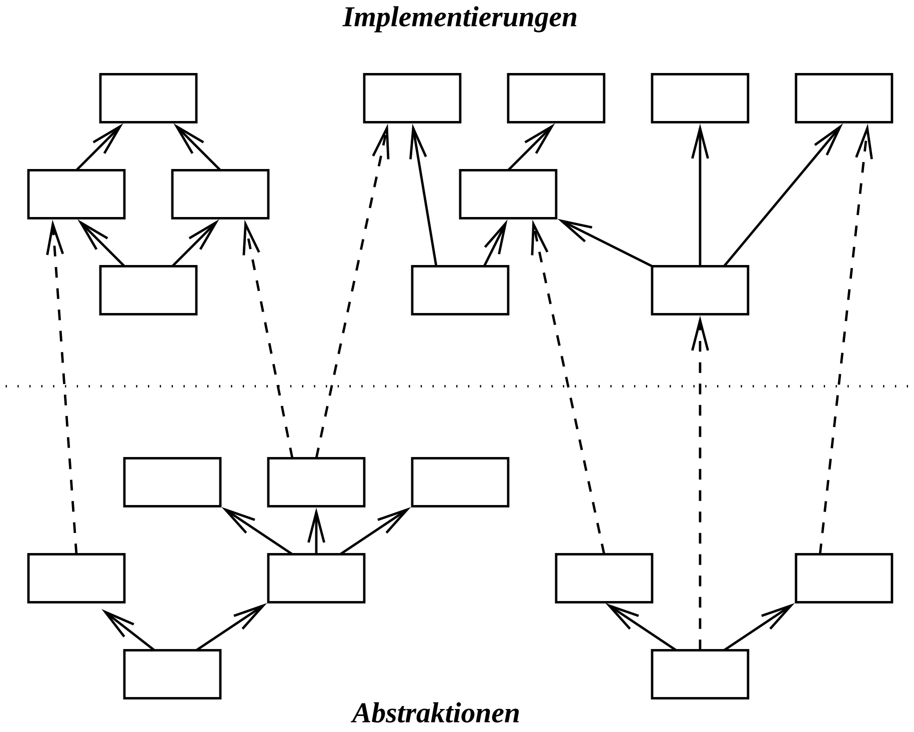

Abbildung 3.7: Abstraktionen und Implementierungen

Natürlich ist es auch möglich, die Importbeziehungen von Schichten zu betrachten und darzustellen, wie es zum Beispiel in Abbildung 3.5 für die Ulmer Oberon-Bibliothek zu sehen ist.

Erweiterungsbeziehung: Analog können Ordnungsrelationen $\rightarrow$ und $\rightarrow^+$ zwischen Typen (Zeiger und Records) definiert werden (siehe auch §2.3):

- $T \rightarrow U$, wenn U eine direkte Erweiterung von T ist.
- $T \rightarrow^+ U$, wenn U eine Erweiterung von T ist.

Wenn $M(T)$ das Modul ist, indem T definiert ist, dann gilt

$$\begin{array}{rcl} T \rightarrow U & \Rightarrow & M(T) \rightarrow M(U) \\ T \rightarrow^+ U & \Rightarrow & M(T) \rightarrow^+ M(U), \quad \text{falls} \quad T \neq U \end{array}$$

Im Falle von *single inheritance* handelt es sich um Hierarchien im strengen Sinne, bei *multiple inheritance* ergibt diese Relation einen DAG.

Die später folgenden Diagramme, die Typhierarchien darstellen, werden analog zu den Modulhierarchien angeordnet (Basistypen unten, abgeleitete Typen weiter oben), wenngleich in der Literatur überwiegend die umgekehrte Variante bevorzugt wird.

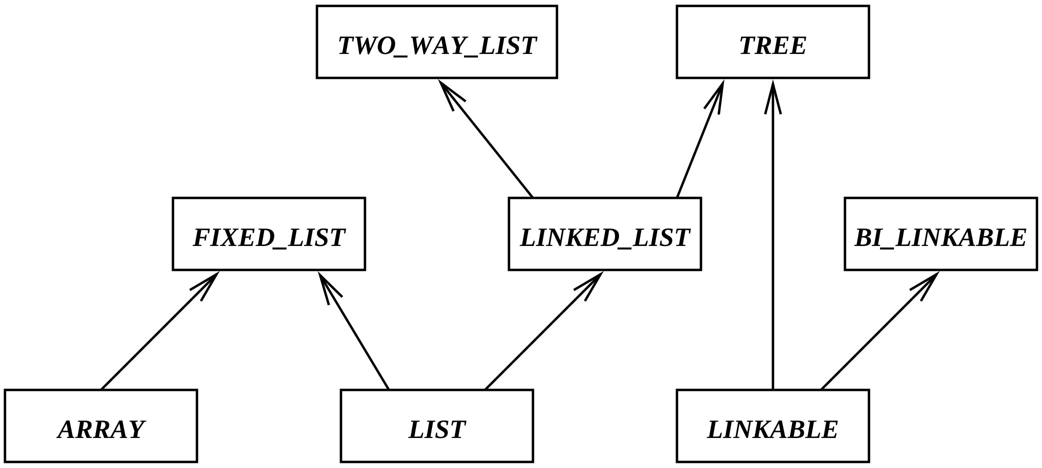

Abbildung 3.8: Implementierungs-orientierte Klassenhierarchie bei Eiffel

Die beiden wichtigsten Modulvarianten sind Abstraktionen (*deferred classes*) und implementierende Module. Generell ist es bei objekt-orientierten Bibliotheken interessant, wie Abstraktionshierarchien (auf Basis von Erweiterungsbeziehungen) und Implementierungshierarchien (auf Basis von Importbeziehungen ohne entsprechende Erweiterungen) aufgebaut und miteinander verknüpft sind.

Im idealen Fall läßt sich jedes Modul einer der beiden Kategorien exklusiv zuordnen und in eine entsprechende Hierarchie einordnen. Während das Ziel bei einer Hierarchie von Abstraktionen darin liegt, allgemeine abstrakte Schnittstellen zu offerieren, die in jede gewünschte Richtung verfeinert werden können, liegt der Hauptzweck bei Implementierungshierarchien in der Wiederverwendung von Programmtext. So können vielerlei Algorithmen als Baukasten zur Verfügung stehen und bei mehr als einer Implementierung einer Abstraktion eingesetzt werden. Abbildung 3.7 zeigt die sich daraus ergebende Struktur, wobei Implementierungsbeziehungen gestrichelt dargestellt sind.

Diese ideale Situation wird leider häufig nicht erreicht, da bei den klassischen objekt-orientierten Programmiersprachen die Versuchung zu groß ist, eine Implementierung bereits bei der Abstraktion unterzubringen, die dann im Verfahren der Überdefinition alternativ implementiert werden kann. Auch werden häufig Erweiterungsbeziehungen verwendet, um den Programmtext von Implementierungen wiederzuverwenden.

Ein Beispiel dafür liefert eine frühe Version der Eiffel-Bibliothek (siehe Abbildung 3.8), bei der sich der Aufbau an den Implementierungen orientiert. So offeriert *TREE* die Funktionalität eines Mehrwegbaumes, wobei es eine Erweiterung von *LINKED_LIST* ist, um den Programmtext für lineare Listen wiederzuverwenden. Wenn hingegen streng zwischen Abstraktionen und Implementierungen unterschieden wird, könnte der in Abbildung 3.9 demonstrierte Aufbau entstehen, der bemerkenswerterweise völlig ohne *multiple inheritance* auskommt.

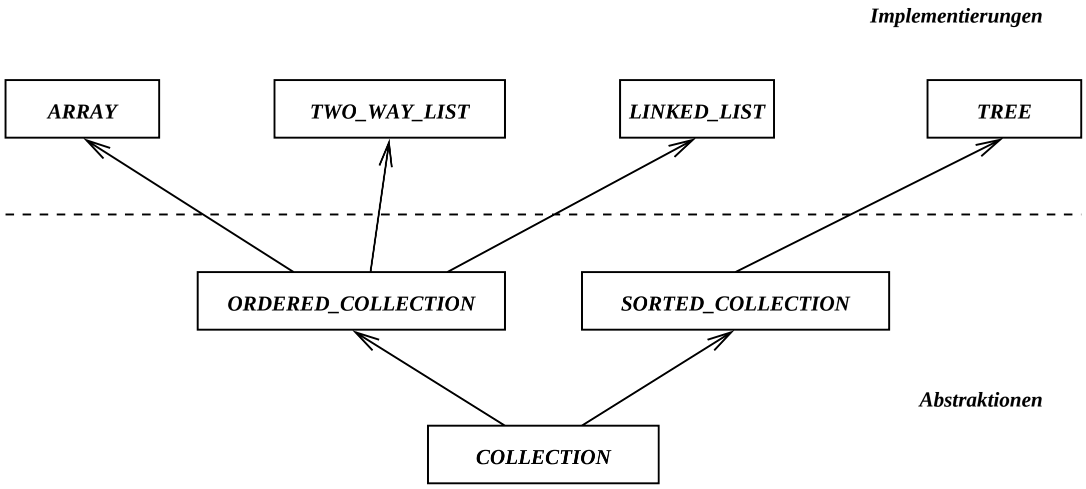

Abbildung 3.9: Alternative mit zweigeteilter Hierarchie

3.4 Stilistische Konventionen

3.4.1 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.

Dies wird recht gut deutlich an einem Beispiel aus Bertrand Meyers Buch "Reusable Software". Zunächst hatte die Bibliothek nur wenige Klassen und bei jeder wurden die Namen in enger Anlehnung an die jeweilige Klasse gewählt:

Klasse	Operationen		
STACK	top	push	pop
ARRAY	entry	enter	
QUEUE	oldest	add	remove_oldest
H_TABLE	value	insert	delete

Wenn jedoch die Größe der Bibliothek anwächst, ist diese Lösung nicht glücklich, weil der Lernaufwand für jede neue betrachtete Klasse immens ist. Standardisierte Namen erleichtern hingegen die Orientierung, ohne daß dazu jeweils der entsprechende Eintrag im Referenzmanual im Vorfeld studiert werden muß. Und so endete die Eiffel-Bibliothek bei folgender Namensvergabe:

Klasse	Operationen		
STACK	item	put	remove
ARRAY	item	put	
QUEUE	item	put	remove
H_TABLE	item	put	remove

Genauso wie in der Eiffel-Bibliothek wurde auch bei der Ulmer Oberon-Bibliothek sehr auf durchgehende Namenskonventionen geachtet, die sich allerdings analog erst im Laufe der Zeit ergeben haben, wenngleich einige Konventionen auch auf Modula-2 zurückgehen:

- 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.

Ein weiteres Problem sind potentielle Namenskonflikte bei den Modulnamen, wenn mehrere Parteien unabhängig voneinander Beiträge zu einer größeren Bibliothek leisten. Sehr sinnvoll sind hier die hierarchischen Modulnamen, die beispielsweise Perl und Java anbieten. Sie lassen insbesondere auch gleichzeitig eine funktions- oder anbieterorientierte Strukturierung in Schichten erkennen. Im Prinzip ist es auch möglich, bei anderen Programmiersprachen die Modul- oder Typnamen zu hierarchisieren, indem Konventionen längere Präfixe vorschreiben.

Dies geschieht auch teilweise in der Ulmer Oberon-Bibliothek. So beginnen alle Module aus der Schicht *sys* mit dem Präfix *Sys*, UNIX-spezifische Module mit *Unix* und das Internet unterstützende Module mit *Inet*.

3.4.2 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. Variablenparameter für Ausnahmenbehandlungen.

3.4.3 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. Dieses Prinzip wird in der Ulmer Oberon-Bibliothek durch Disziplinen unterstützt. So können auch Module, die Operationen auf fremden Objekten durchführen (wie z.B. *Read.Int* für ein Objekt vom Typ *Streams.Stream*), ihre Optionen beim Objekt notieren (siehe *StreamDisciplines*).

3.4.4 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.

3.4.5 Anordnung der Deklarationen in einer Schnittstelle

Viele Schnittstellen bedienen mehrere Typen von Klienten. So offeriert ein Modul, das eine Abstraktion definiert, sowohl einige Deklarationen für Implementierer als auch natürlich Operationen für Benutzer einer Abstraktion. Es ist sehr hilfreich, wenn dies sowohl räumlich getrennt wird (soweit möglich) als auch in der Referenzdokumentation klar hervorgehoben wird.

Bei Oberon können leider nicht Prozedur-Deklarationen beliebig mit anderen Deklarationen vermischt werden, so daß sich dieses Prinzip nur beschränkt realisieren läßt. Da zu der Schnittstelle für Implementierer häufig eine große Reihe von Typdefinitionen dazugehören (Schnittstellen-Record und Prozedurtypen), werden häufig im Anschluß die Prozeduren für Implementierer angegeben (insbesondere *Init*).

3.4.6 Formatierung

Damit fremder Programmtext leicht verstanden werden kann, ist es sehr hilfreich, wenn eine einheitliche Formatierung und Darstellung des Programmtextes verwendet wird. Dies hängt natürlich auch von den verwendeten Werkzeugen ab, da es nicht mehr überall üblich ist, Programmtext als ASCII-Text abzuspeichern. Insbesondere das Züricher Oberon-System weicht davon ab, und es ist dort möglich, Programmtext wie den Text eines Dokuments zu setzen (mit Zeichensätzen, Hervorhebungen, Schriftgrößen, Diagrammen und Farben).

Bei der Ulmer Oberon-Bibliothek gelten beispielsweise folgende Richtlinien:

- Programmtext wird eingerückt entsprechend der Blockstruktur von Oberon und entsprechend der Verschachtelung von Anweisungen, wobei eine Einrückweite von 3 Zeichen eingehalten wird. Beispiel:

```
MODULE Hello;

   IMPORT Write;

   PROCEDURE DoHello;
   BEGIN
      Write.Line("Hallo!");
   END DoHello;

BEGIN
   DoHello;
END Hello.
```

- Zeilen sollten normalerweise nicht länger als 80 Zeichen lang sein. Wenn eine Zeile umgebrochen wird, werden Fortsetzungszeilen jeweils eingerückt. Typischerweise wird die Zeile nach einem Operator oder einem Komma getrennt.

- Leerzeichen gibt es grundsätzlich nach Satzzeichen (Komma oder Semikolon), jedoch nicht davor. Operatoren werden beidseitig von Leerzeichen umgeben, und bei Einklammerungen (bei Ausdrücken, Prozeduren, Indizierungen und Mengen) wird ansonsten auf Leerzeichen verzichtet. Beispiel:

```
result := P(x, y + z * (17 - a[i]));
```

- Da Oberon nur wenige Vorrangstufen in seiner Ausdrucks-Grammatik besitzt, kann ohne Verlust an Klarheit auf überflüssige Klammern verzichtet werden.

- Ganz zu Beginn des Programmtextes kann ein Copyright-Vermerk und die Entwicklungshistorie in einem Kommentar vermerkt werden. Kommentare zu Gruppen von Prozeduren, Variablen oder Anweisungen stehen jeweils davor und werden nicht weiter eingerückt. Kommentare, die ein Modul oder eine Prozedur beschreiben, folgen jeweils dem Kopf und werden eingerückt. Kommentare zu einzelnen Konstanten, Variablen, Feldkomponenten und Anweisungen können nachfolgend auf der gleichen Zeile stehen oder bei mangelndem Platz eingerückt auf der nächsten Zeile folgen.

3.5 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 bei den ggf. zwischenliegenden Modulen $M_1 ... M_n$ nur die öffentlichen Importbeziehungen betrachtet werden (d.h. die in Oberon im **DEFINITION** sind).

Konfliktsituationen durch einander fremde Module entstehen durch globale Ressourcen. Dazu gehören Raum, Zeit, globale Variablen und das umgebende System.

3.5.1 Kooperative Ausnutzung des Adressraumes

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 den moderneren UNIX-Varianten neben dem traditionellen *brk(2)* die Systemaufrufe *mmap(2)* und *shmat(2)*, die es erlauben, an (fast) beliebigen Stellen im Adreßraum Speicher zu allokieren, Dateibereiche 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.

3.5.2 Kooperative Ausnutzung der Rechenzeit

Traditionelle Programmiersprachen und -systeme bieten nur einen einzigen Kontrollfluß an. Dies ist extrem einschränkend, da es dann beispielsweise nicht ohne weiteres möglich ist, parallel zur Interaktion mit dem Benutzer mit anderen Prozessen über das Netzwerk zu kommunizieren. Entsprechend ergeben sich Konflikte bezüglich der Zeit aus einer begrenzten Parallelisierbarkeit.

Heute sollte jede Programmumgebung Parallelität in der einen oder anderen Form anbieten, da sonst viele Techniken schlichtweg unmöglich sind – es sei denn auf Kosten der Unabhängigkeit der einzelnen Module (wenn z.B. die Benutzerinteraktion gezwungen ist, regelmäßig das Netzwerk auf neue Pakete abzufragen) oder alles muß in Form von Ereignis-Häppchen in möglichst kurzer Zeit abgearbeitet werden. Dies wird beispielsweise bei Java standardmäßig unterstützt – bei anderen Systemen sieht es teilweise ziemlich düster aus.

Oberon sieht in der Sprachdefinition nur einen einzigen Kontrollfluß vor. Diese Restriktion wurde im Ulmer Oberon-System durch die Einführung von Koroutinen gelockert. Da jedoch immer nur eine Koroutine aktiv sein kann, müssen auch Koroutinen untereinander kooperieren, indem sie die Ausführung der anderen Koroutinen nicht unnötig blockieren.

3.5.3 Konflikte durch globale Variablen

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.

3.5.4 Konflikte durch den Zugriff auf die Umgebung

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. 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.

3.5.5 Kooperativität im heterogenen Sprachumfeld

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.

Kapitel 4

Dokumentation von Bibliotheken

Eine Bibliothek ist ohne eine begleitende Dokumentation undenkbar, da sie die primäre Schnittstelle zwischen den Anwendern und den Entwicklern einer Bibliothek darstellt. Die Vielfalt und Breite existierender Dokumentationen für Bibliotheken ist erstaunlich – sowohl in der Strukturierung als auch in der Qualität. Leider werden häufig gewohnte Dokumentationstechniken aus dem Bereich der normalen Software-Entwicklung übernommen, ohne sie an die ungleich umfangreicheren Bedürfnisse für Bibliotheken anzupassen.

4.1 Dokumentationsvarianten

Prinzipiell gibt es drei Arten von Dokumentationen, die Bibliotheken begleiten:

Referenz-Manuals sind mit Abstand die wichtigste Art von Dokumentation. Sie sollten ein gutes Nachschlagen ermöglichen und sind typischerweise nach Modulen, Klassen oder Typen geordnet. Um praktikabel zu sein, sollten sie ähnlich strengen Formalismen unterworfen werden, wie sie bei dem Programmtext Anwendung finden. Sehr geeignet ist z.B. die Struktur der UNIX-Manualseite mit ihren festen Abschnitten von **NAME** bis **BUGS**.

Guides führen in die Bibliothek oder in einen Teilbereich der Bibliothek umfassend ein. Hier liegt der Schwerpunkt in der didaktischen Aufbereitung und weniger bei der guten Nachschlagbarkeit. Zwar kann letzteres teilweise auch durch einen umfangreichen Index geleistet werden, jedoch nie so gut wie eine ausgefeilte Manualseite, die zuverlässig *alle* relevanten Informationen zu der Verwendung eines Moduls, Klasse oder Typs geben sollte. Die didaktische Aufbereitung sollte insbesondere auch das Modell und die Prinzipien hinter der Bibliothek (oder Teilen davon) erläutern, so daß Aufbau und Stil der Bibliothek vertrauter werden.

Tutorials erlauben einen ersten Einstieg in die Benutzung einer Bibliothek, bei dem die Betonung auf leichte Erfaßbarkeit liegt auf Kosten der vollständi-

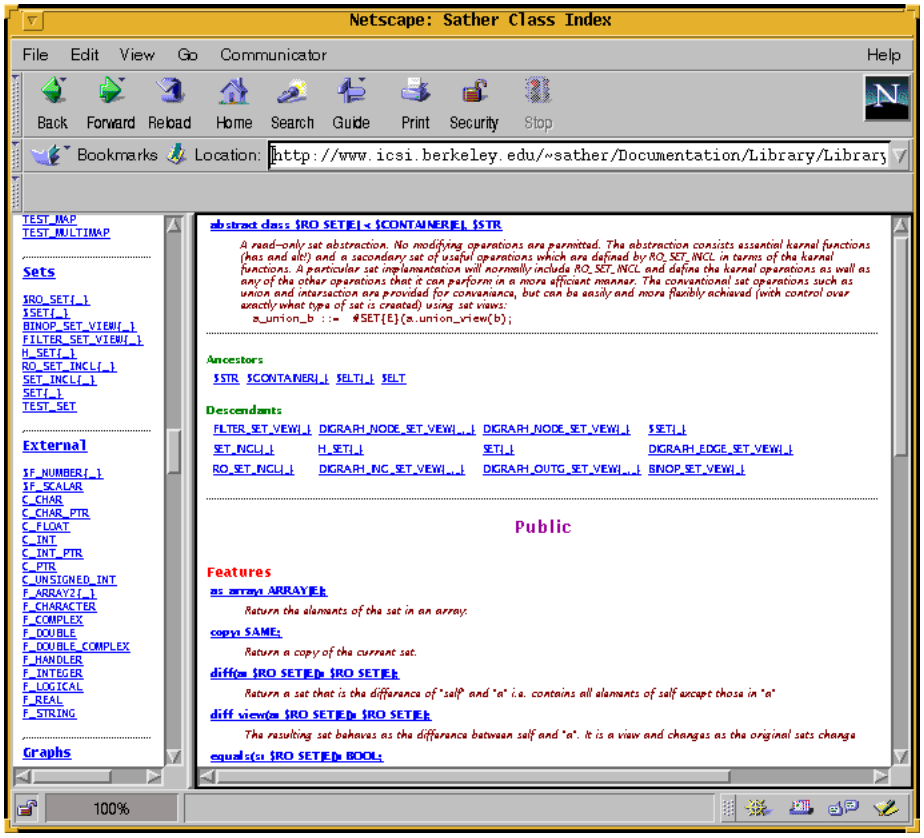

Abbildung 4.1: Online-Dokumentation der Sather-Bibliothek

gen Abdeckung. Häufig gibt es auch Tutorials zu einzelnen Bibliotheksbereichen.

Leider gibt es nur in den allerwenigsten Fällen alle drei Varianten. Grundsätzlich sollte es Referenz-Manuals geben, weil

- sie am leichtesten korrigiert, ergänzt und angepaßt werden können, wenn Änderungen bei den Quellen erfolgen. Einem Entwickler von Bibliotheksmodulen kann die Pflege der zugehörigen Manualseite zugemutet werden, jedoch typischerweise nicht die von Guides und Tutorials, da dort der Aufwand jeweils ungleich größer ist und sie daher (bei kommerziellen Bibliotheken) von technischen Autoren gepflegt werden.
- nur sie am schnellsten dem erfahrenen Bibliotheks-Nutzer die gewünschten Informationen liefern können.

4.2 Repräsentierung der Dokumentation

Grundsätzlich kann Dokumentation in Form von Papier (Schrankware) oder am Rechner online vorliegen, wobei natürlich vorzugsweise beide Varianten unterstützt werden. Zweifellos ist Hypertext insbesondere bei Referenz-Manuals

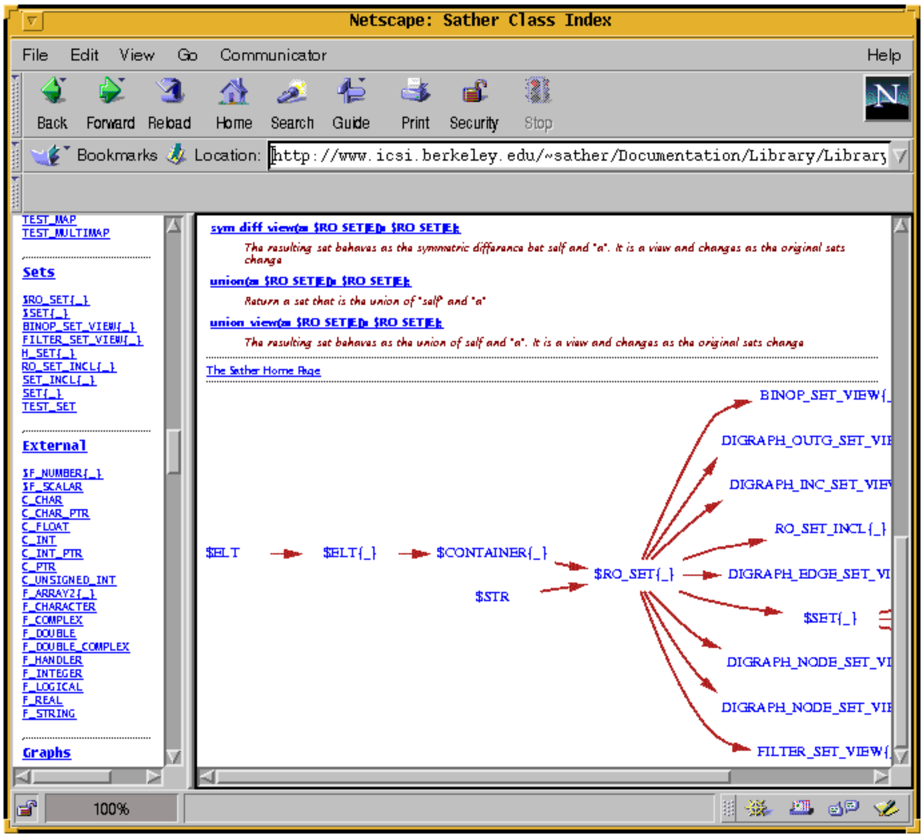

Abbildung 4.2: Erweiterungshierarchie in der Dokumentation

die attraktivste Repräsentierung, da neben der traditionellen Vorteile der Referenz-Manuals noch Navigierbarkeit, das leichte Verfolgen von Verweisen und die Durchsuchbarkeit hinzukommen können.

Eine gute Referenz-Dokumentation sollte in ihrer Struktur der Bibliothek und der zugrundeliegenden Programmiersprache gleichen. So ist es insbesondere wichtig, daß die Orientierung entsprechend der Schichten, Pakete und Hierarchien wiedergegeben wird. Ein gutes Beispiel dafür ist die Online-Dokumentation der Sather-Bibliothek, die am International Computer Science Institute an der Universität in Berkeley entwickelt worden ist.

Wie auf der Abbildung 4.1 zu sehen, wird links eine Struktur in Paketen vorgegeben (z.B. *Sets*, *External* und *Graphs*) und rechts die Beschreibung einer abstrakten Klasse dargestellt (in der Abbildung *RO_SETS*). Nach einer einführenden Beschreibung folgt die Liste der übergeordneten Klassen, von denen *RO_SETS* abgeleitet ist, danach folgen die abgeleiteten Klassen (jeweils verbunden mit Hypertext-Verweisen). Danach wird jede exportierte Operation oder Komponente (genannt *features* in Eiffel und Sather) aufgelistet und erläutert.

Weiter unten (wie in Abbildung 4.2 zu sehen) endet die Referenz-Seite mit einer graphischen Darstellung der Klassenhierachie, in die *RO_SETS* eingebettet ist.

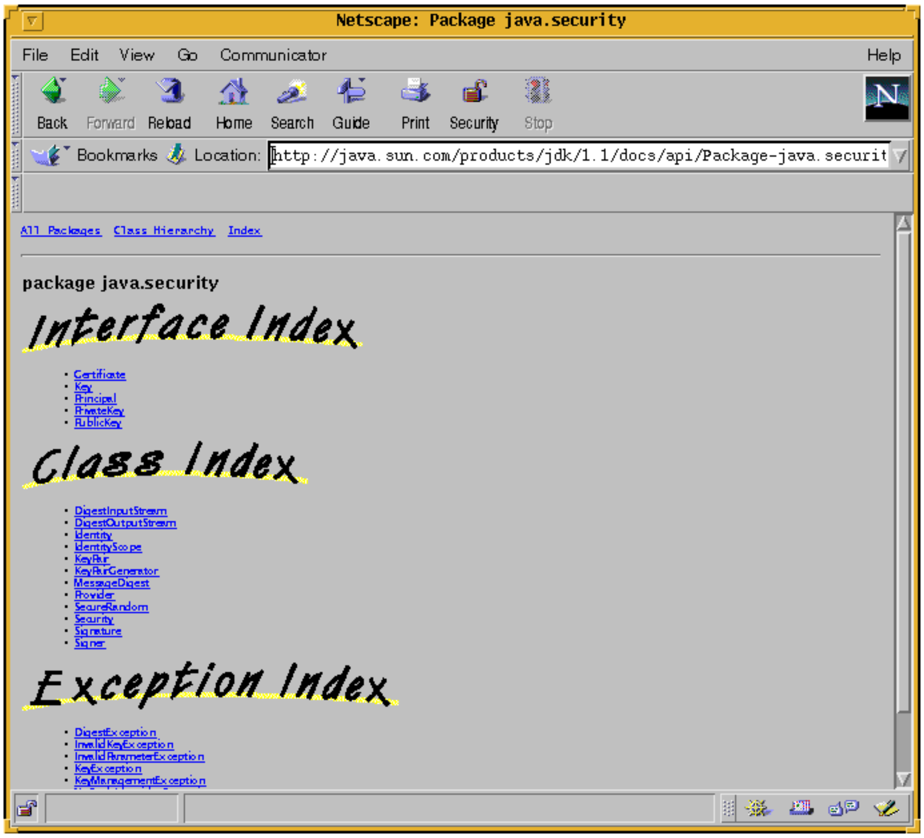

Abbildung 4.3: Dreigeteilter Modulindex eines Java-Pakets

Für die Dokumentation von Java-Bibliotheken gibt es eine Repräsentierung, die sich ebenfalls sehr eng an die Struktur von Java anlehnt. Entsprechend den hierarchischen Modulnamen in Java ergeben sich die Pakete. Bei den einzelnen Modulen in Java gibt es drei verschiedene Varianten: *interfaces* (abstrakte Schnittstellen analog zu *deferred classes* in Eiffel oder *abstract virtual classes* in C++), *classes* (Implementierungen von abstrakten Schnittstellen oder Ableitungen anderer Klassen) und *exceptions* (Hierarchie von Ausnahmen, die zur Fehlerbehandlung dienen).

Anstatt wie bei der Dokumentation zur Sather-Bibliothek mit einer zweigeteilten Anzeige zu arbeiten (*frames*), sind die Seiten bei Java hierarchisch organisiert, wobei es immer recht einfach ist, in die höhere Ebene zurückzugelangen. Die oberste Ebene liefert einen Überblick über alle Pakete und auf der darunterliegenden Ebene sind in sortierter Form (nach den drei Varianten) die einzelnen Module zu finden (siehe Abbildung 4.3).

In der dritten und untersten Ebene wird dann ein einzelnes Modul vorgestellt (siehe Abbildung 4.4). Die Seite beginnt mit einer Darstellung der Erweiterungshierarchie, die dank *single inheritance* genügend einfach ist. Die Seite wird dann mit einem Index der Variablen, Konstruktoren und Methoden fortgesetzt und mit ausführlichen Erläuterungen abgeschlossen.

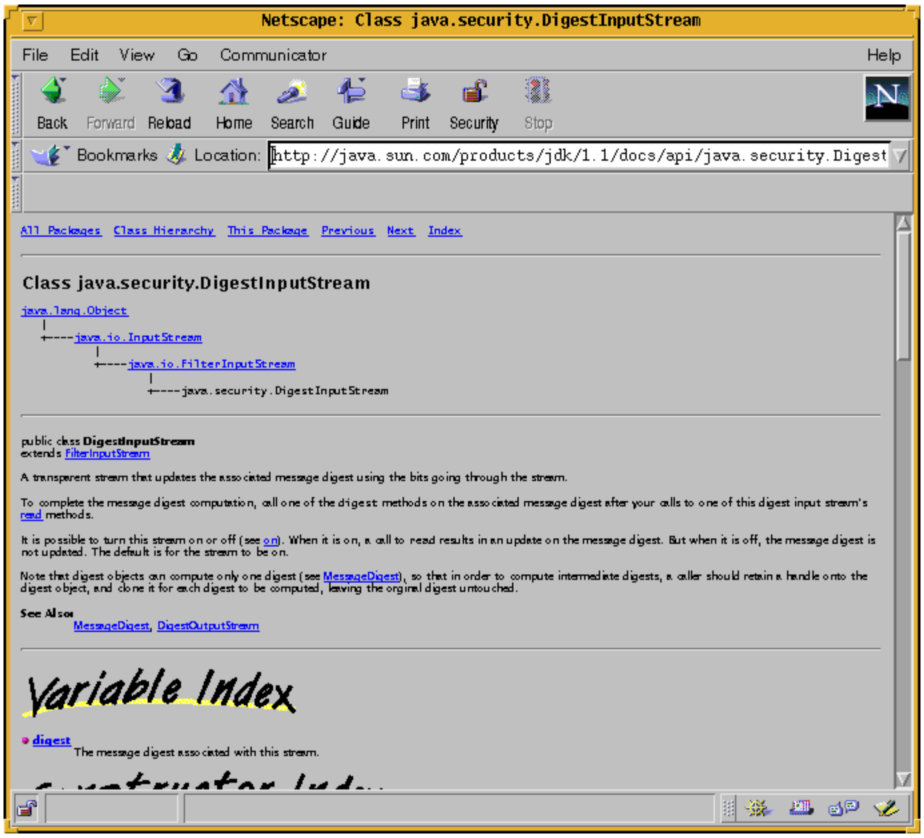

Abbildung 4.4: Referenzseite eines Java-Moduls

4.3 Generierung der Dokumentation

Wie bei allen Systemen, die hochgradig strukturiert sind und bei denen zahllose Details zuverlässig korrekt repräsentiert werden sollten, ist es hilfreich, wenn sie weitgehend automatisch generiert oder verifiziert werden. Somit bleiben die Fragen, wie die Dokumentation zunächst entwickelt wird und auf welche Weise sie nachher in die endgültige Form (hoffentlich automatisch) transferiert wird.

4.3.1 Originale Form der Dokumentation

Hierbei gibt es zwei Schulen: Die eine Seite bringt die Referenz-Dokumentation direkt in den Quellen unter, und die andere zieht es vor, dies in separaten Dokumenten zu verwalten. Wichtig ist, daß die Verbindung von Quellen mit der Referenz-Dokumentation nicht mit *literate programming* zu verwechseln ist, wie Knuth es für TeX eingeführt hat, da eine Referenz-Dokumentation nur beschreibt, wie ein Modul zu benutzen ist und nicht wie sein Innenleben funktioniert. Für beide Schulen gibt es gute Gründe:

- Wenn der Programmtext und die zugehörige Referenz-Dokumentation ein gemeinsames Dokument bilden, ist die Gefahr geringer, daß beide nicht miteinander synchronisiert sind. Außerdem wird damit dem Programmierer unmittelbar die Verantwortung zur Verwaltung der Referenz-Dokumentation übertragen.

- Andererseits sind in vielen Fällen die textuellen Aufbereitungsmöglichkeiten innerhalb der Programme sehr beschränkt – so ist es bei einigen Systemen nicht möglich, Tabellen unterzubringen. Natürlich gibt es auch hierfür Lösungen: So ist bei einigen Oberon-Systemen die Einfügung beliebigen formatierten Textes bishin zu Animationen möglich (entsprechende Arbeiten gibt es z.B. von Mössenböck) und alternativ könnten auch textbasierte Formatierungssysteme wie z.B. TEX eingebettet werden. Allerdings ist es nicht ganz einfach, diese Varianten ohne weiteres in HTML zu verwandeln.
- Häufig wird die Referenz-Dokumentation modifiziert, ohne daß der eigentliche Programmtext geändert werden muß, z.B. um den Text didaktisch besser aufzubereiten oder ganz schlicht Schreibfehler zu korrigieren. Dies führt dann dazu, daß das betroffene Modul und alle davon abhängigen Module vollkommen unnötig frisch übersetzt werden müssen.

4.3.2 Werkzeuge zur Generierung

Wenn die Dokumentation direkt den Quellen entnommen wird, werden häufig speziell markierte Bereiche oder bestimmte Kommentare betrachtet. Bei Perl gibt es in einer Quelle im Grunde genommen zwei Sprachen (Perl und POD, das für *plain old documentation* steht), die einfach lexikalisch voneinander zu trennen sind. Somit kann der Perl-Interpreter einfach die POD-Texte überlesen, während die POD-Konvertierer den Perl-Programmtext ignorieren kann. POD ist sehr einfach und kann somit in eine Vielzahl von Formaten ohne größeren Aufwand konvertiert werden: einfacher ASCII-Text, UNIX-Manualseiten, LATEX und HTML. So sieht z.B. der Anfang von *File::Basename* aus:

```
package File::Basename;

=head1 NAME

fileparse - split a pathname into pieces

basename - extract just the filename from a path

dirname - extract just the directory from a path

=head1 SYNOPSIS

    use File::Basename;

    ($name,$path,$suffix) = fileparse($fullname,@suffixlist)
    fileparse_set_fstype($os_string);
    $basename = basename($fullname,@suffixlist);
    $dirname = dirname($fullname);
```

```
    ($name,$path,$suffix) = fileparse("lib/File/Basename.pm","\.pm");
    fileparse_set_fstype("VMS");
    $basename = basename("lib/File/Basename.pm",".pm");
    $dirname = dirname("lib/File/Basename.pm");

=head1 DESCRIPTION

These routines allow you to parse file specifications into useful
pieces using the syntax of different operating systems.
[...]
=over

=item C<basename>

The basename() routine returns the first element of the list produced
by calling fileparse() with the same arguments, except that it always
quotes metacharacters in the given suffixes.  It is provided for
programmer compatibility with the UNIX shell command basename(1).

=item C<dirname>

The dirname() routine returns the directory portion of the input file
specification.  When using VMS or MacOS syntax, this is identical to the
second element of the list produced by calling fileparse() with the same
input file specification.  (Under VMS, if there is no directory information
in the input file specification, then the current default device and
directory are returned.)  When using UNIX or MSDOS syntax, the return
value conforms to the behavior of the UNIX shell command dirname(1).  This
is usually the same as the behavior of fileparse(), but differs in some
cases.  For example, for the input file specification F<lib/>, fileparse()
considers the directory name to be F<lib/>, while dirname() considers the
directory name to be F<.>).

=back

=cut

require 5.002;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(fileparse fileparse_set_fstype basename dirname);
#use strict;
#use vars qw($VERSION $Fileparse_fstype $Fileparse_igncase);
$VERSION = "2.5";
[...]
```

Ein Gleichheitszeichen zu Beginn der Zeile eröffnet eine POD-Direktive und startet ggf. den POD-Bereich, der durch `=cut` beendet wird. Somit wird der Teil danach (ab `require 5.002`) wieder vom Perl-Interpreter berücksichtigt.

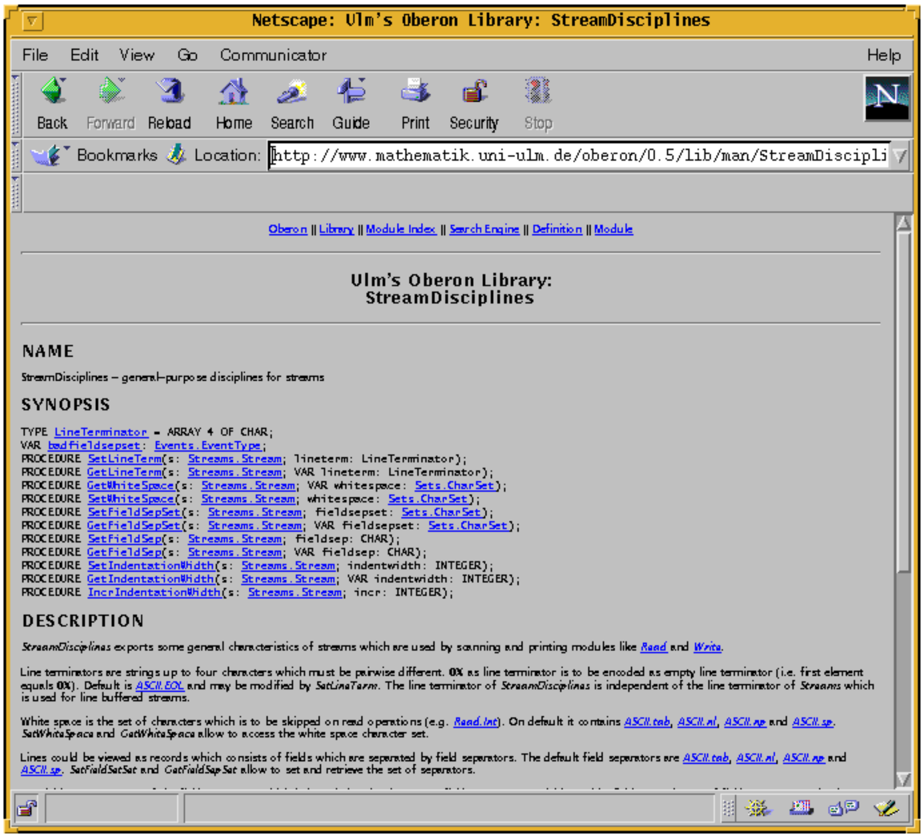

Abbildung 4.5: Dokumentation der Ulmer Oberon-Bibliothek

Bei Eiffel ist eine solche strikte Trennung nicht üblich. Stattdessen werden die Quellen der Schnittstellen direkt in einer leicht manipulierten Form präsentiert. Mitgeliefert werden bei Eiffel zwei Werkzeuge namens *short* und *flat*. Beide erhalten als Argument ein Modul, wovon *short* nur die Schnittstelle (also die nach außen sichtbaren Teile) ausgibt, während *flat* die Ausgabe um alle exportierten Teile der übergeordneten Klassen erweitert. Somit ist bei *flat* mit einem Blick sichtbar, welche Operationen alle möglich sind, ohne daß eine größere Menge von Referenz-Seiten nacheinander studiert werden muß. Dies ist insbesondere wertvoll, weil bei Eiffel im Zuge von Erweiterungen Operationen umgetauft werden können und somit die Dokumentation einer übergeordneten Schnittstelle garnicht mehr direkt zutrifft.

Bei der Ulmer Oberon-Bibliothek gab es von Beginn an eine parallele Dokumentation zu jedem Modul in Form von UNIX-Manualseiten (zu formatieren mit dem ms-Paket unter troff). Sobald das Web und HTML zur Verfügung standen, wurden Perl-Skripte entwickelt, die sowohl die Manualseiten als auch die Quellen in HTML konvertieren konnten (siehe Abbildung 4.5). Damit steht nicht nur die Dokumentation im Web zur Verfügung, sondern auch die Quellen, wobei dort jede Verwendung importierter Prozeduren oder anderer Deklarationen mit einem Verweis auf die entsprechende Dokumentation versehen ist. Umgekehrt ist es auch möglich, zielgerichtet von der Referenz-Dokumentation an die entsprechende Stelle der zugehörigen Quelle zu wechseln. Dies wurde entscheidend durch die Pflicht zur Benutzung qualifizierter Namen in Oberon erleichtert und damit wird der Einsatz aufwendiger CASE-Tools zur Analyse des Programmtexts erspart, wie er teilweise bei C++ unumgänglich ist.

Kapitel 5

Objekt-Modelle von Bibliotheken

Objekte sind alle Instanzen in einer Programmiersprache, die dynamisch kreiert werden können und auf die typischerweise mittels Zeiger explizit (oder implizit, aber immer noch der der Zeigersemantik folgend) zugegriffen wird.[1] Sowohl durch die Programmiersprache als auch durch eine Bibliothek werden eine Reihe von Grundeigenschaften für Objekte festgelegt, die unabhängig von einzelnen Abstraktionen sind und daher die gesamte Bibliothek und den Umgang damit festlegen.

Allerdings ist es in einigen Fällen garnicht so einfach festzustellen, inwieweit die Programmiersprache bzw. ein Compiler Eigenschaften vorgibt und inwiefern sie durch die Bibliothek realisiert werden. Bei Eiffel gibt es die Klasse *ANY*, von der per Sprachdefinition alle anderen Klassen abgeleitet sind. Darüberhinaus existieren einige Klassen, die wie normale Klassen aus der Bibliothek aussehen, jedoch nur virtuell bestehen, da keine zugehörigen Quellen existieren und die Unterstützung direkt vom Compiler geleistet wird. Dies gilt z.B. für die Klasse *ARRAY* bei ISE Eiffel, um die bei konventionellen Programmiersprachen gewohnte Effizienz für Arrays zu gewährleisten, obwohl sie nicht unmittelbar in der Programmiersprache definiert sind.

Diese Unterscheidung ist dann nicht mehr so relevant, wenn die Programmiersprache bzw. deren zugehörige Compiler und die grundlegenden Bibliotheken (*kernel libraries*) als natürliche Einheit betrachtet werden. Allerdings muß dann auch ggf. die gesamte Einheit standardisiert werden und nicht nur ein Teil davon. Dies wird jetzt beispielsweise für Java versucht.

Zusammenfassend unter dem Begriff Objekt-Modell sollen hier alle Eigenschaften und Techniken präsentiert werden, die für alle (oder fast alle) Objekte relevant sind, beginnend vom Lebenszyklus der Objekte, allgemein verfügbarer Operationen, Ausnahmenbehandlungen (insbesondere, wenn sie objekt-bezogen sind), die Verknüpfung von Objekten mit Abstraktionen (statisch und/oder dynamisch), Delegationen, Abhängigkeiten bis hin zu Typen als Objekte (*1st class objects*). Sie gehören in Ergänzung zur Programmiersprache zu dem Grundwortschatz, der von jedem Anwender vollständig zu erkennen und erlernen ist, bevor

[1] Dies ist eine minimalistische Definition, da nichts über Methoden und andere OO-Techniken gesagt wird. Sie hilft jedoch bei den hybriden Programmiersprachen wie z.B. C++.

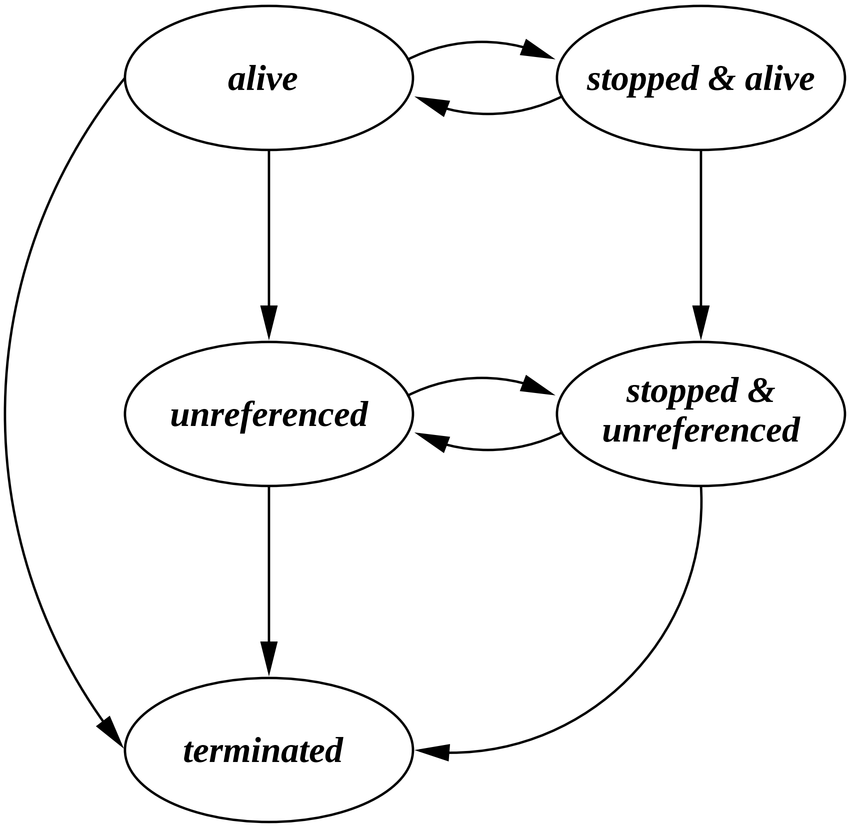

Abbildung 5.1: Von *Resources* vorgegebener Lebenszyklus

an erste Entwicklungen zu denken ist.

5.1 Lebenszyklus von Objekten

Der zunächst wichtigste Umgang mit Objekten umfaßt deren Lebenszyklus von der Allokation des zugehörigen Speichers bis zu dessen Freigabe. Typischerweise erfolgt die Allokation implizit (z.B. Eiffel und C++) oder explizit (z.B. Oberon) durch einen Konstruktor. Die Mehrheit der objekt-orientierten Programmiersprachen (eine prominente Ausnahme ist C++) gibt dann den für das Objekt allokierten Speicher wieder vollautomatisch frei, wenn es nicht mehr möglich ist, darauf zuzugreifen (*garbage collection*).

Doch gerade das Ableben eines Objekts kann sich etwas aufwendiger darstellen. In Eiffel und einigen anderen Programmiersprachen ist es möglich, ein Objekt, das gerade deallokiert werden soll, noch einige Aktionen ausführen zu lassen (z.B. die Freigabe externer Ressourcen wie offene Dateiverbindungen). Dies hat die Konsequenz, dass sich das Objekt potentiell wieder in lebende Strukturen einhängen kann und diese "Attacke" überlebt. Wegen dieser Komplexität, die die *garbage collection* nicht unerheblich komplizierter macht bzw. in ihren Möglichkeiten einschränkt, steht diese Möglichkeit auch nicht überall zur Verfügung.

Einige Systeme unterscheiden zwischen *heavyweight* und *lightweight* Zeigern, wobei das Überleben eines Objektes nur von den *heavyweight* Zeigern abhängt. Wenn ein Objekt dann automatisch deallokiert wird, werden alle entsprechenden *lightweight* Zeiger auf **NIL** gesetzt. Dies ist dann nützlich, wenn Objekte in eine Reihe von buchhalterischen Datenstrukturen enthalten sind, die die Lebensdauer nicht beeinflussen sollen.

Bei Oberon gibt es von der Programmiersprache her nur die Unterstützung für die *garbage collection*, jedoch nicht für Operationen kurz vor der Deallokation oder die Unterscheidung von Verweisen. Dessen ungeachtet ist es möglich, diese Mechanismen bei Bedarf über die Bibliothek zu realisieren – in der Ulmer Oberon-Bibliothek geschieht dies durch das Modul *Resources*, das in Bezug auf den Lebenszyklus einen kooperativen Umgang mit Objekten erlaubt.

Jedes Objekt, das eine Erweiterung von *Disciplines.Object* ist, befindet sich in einem der fünf folgenden Zustände, die von *Resources* definiert werden (siehe Abbildung 5.1):

alive Dies ist der Anfangszustand aller Objekte, der solange bleibt, wie das Objekt für Operationen zur Verfügung steht und schwergewichtige Zeiger darauf verweisen.[2]

unreferenced Dieser Zustand wird erreicht, wenn der letzte der schwergewichtigen Zeiger auf das Objekt entfernt wird. In dieser Situation können noch Aufräumarbeiten durchgeführt werden, die insbesondere die leichtgewichtigen Zeiger kappen sollten, damit die *garbage collection* das Objekt aufräumen kann.

terminated ist der finale Zustand, wenn ein Objekt nicht mehr zu benutzen ist. Ggf. sind letzte Verweise (egal welcher Art) darauf wegzunehmen, damit es der *garbage collection* anheim fallen kann.

stopped & alive Dieser Zustand tritt ein, wenn das Objekt zwar noch lebt und in allen Datenstrukturen verbleiben sollte, jedoch eine Kommunikation zur Zeit nicht möglich ist. Damit ist der Effekt zwischenzeitlich aufgerufener Operationen undefiniert (möglicherweise werden sie ignoriert, aufgesammelt oder blockieren den Aufrufer, bis das Objekt wieder ansprechbar ist).

stopped & unreferenced kann theoretisch ebenfalls eintreten, wenn während der Zeit des Aufräumens die Verbindung temporär abbricht.

Leichte Zeiger werden auf gewohnte Weise einfach durch entsprechende Zuweisungen gesetzt, wobei nur ggf. die Verpflichtung entsteht, bei einem Übergang zu *unreferenced* oder *terminated* die Verbindungen zu kappen. Schwergewichtige Zeiger werden zusätzlich durch *Resources.Attach* und *Resources.Detach* begleitet, die über eine private Disziplin die Anzahl der schwergewichtigen Verweise hoch- und runterzählen. Landet der Zähler wieder auf 0, so wird automatisch der Zustandswechsel nach *dereferenced* ausgelöst. Ähnliche Referenz-Zähler gibt es auch in einigen C-Bibliotheken, die auf diese kooperative Weise dann feststellen, wann ein Destruktor aufgerufen werden kann. Bei Oberon ist dies Ver-

[2] Falls nie ein schwergewichtiger Zeiger auf ein Objekt gesetzt wird, bleibt ein Objekt ebenfalls leben. Auf diese Weise ist der Umgang mit schwergewichtigen Zeigern nur dort notwendig, wo kooperative Mechanismen erforderlich sind.

fahren allerdings nur notwendig, wenn wirklich für ein Objekt beide Zeigervarianten verwendet werden. Grundsätzlich haben natürlich alle Referenz-Zähler-Implementierungen den Nachteil, daß sie bei zyklischen Datenstrukturen nicht helfen. Ein anderes Problem besteht in der Möglichkeit der nicht korrekten Verwendung von *Resources.Attach* und *Resources.Detach*, die in Oberon jedoch durch die Verwendung von Schlüsseln zuverlässig verhindert wird.

Die Unterscheidung zwischen lebenden und toten Objekten nicht nur auf Basis der Erreichbarkeit im Sinne der *garbage collection* bringt enorme Vorteile im Bereich verteilter Systeme, da dann ein flexibler und allgemeiner Mechanismus zur Verfügung steht, der bei einem Zusammenbrechen von Verbindungen (oder deren temporären Ausfall) eingesetzt werden kann.

5.2 Fundamentale Objekt-Eigenschaften

Eine Reihe von fundamentalen Eigenschaften und unterstützten Operationen sind für jedes Objekt vorgegeben, sei es durch die Programmiersprache, durch die von der Programmiersprache vorgegebene implizite Basisklasse (z.B. *ANY* in Eiffel) oder durch die Bibliothek durch einen per Konvention gewählten Basistyp.

5.2.1 Kopierbarkeit

Die Unterstützung für Zuweisungen wird typischerweise von der Programmiersprache gegeben und kann (z.B. in C++) für bestimmte Objekttypen überdefiniert werden. Prinzipiell gibt es drei Arten von Zuweisungen bzw. Kopieroperationen:

Zeigerzuweisung Hier wird nur ein Zeigerwert kopiert, wobei kein neues Objekt angelegt wird und das alte unverändert bleibt.

Shallow Copy / Cloning Bei einer nicht tiefgehenden Kopie wird nur ein Objekt dupliziert, jedoch nicht die Datenstrukturen, auf die von dem Objekt aus verwiesen werden.

Deep Copy Hier werden rekursiv ganze Datenstrukturen dupliziert.

Während die Zeigerzuweisung in jedem Fall notwendig ist und auch zur Verfügung gestellt wird, stellt sich bei den beiden Kopiervarianten die Frage, ob sie überhaupt sinnvoll sind, da in beiden Fällen private Daten und Datenstrukturen ohne Kenntnis des betroffenen Moduls dupliziert werden können. Wenn kein Bruch der Abstraktions- und Modulgrenzen zugelassen werden soll, dann darf eine Duplizierung nur in Kooperation mit allen betroffenen Parteien durchgeführt werden.

Beispiel: Wenn ein Objekt *s* vom Typ *Streams.Stream* dupliziert werden kann, dann gäbe es Schwierigkeiten, wenn es private Verweise auf Systemresourcen gibt (z.B. ein Dateideskriptor bei *UnixFiles*): Die Operationen auf dem ersten Objekt *s* und dem geklonten *s0* würden sich gegenseitig beeinflußen, da sie auf der gleichen Datei arbeiten und wenn der eine geschlossen werden würde, könnte auf dem anderen nicht mehr gearbeitet werden. All dies kann nicht in genereller Weise vorhergesehen und berücksichtigt werden.

Eiffel löst dies Problem dadurch, daß die Operationen grundsätzlich (über *ANY*) zur Verfügung stehen, jedoch für einzelne abgeleitete Klassen überdefiniert werden können. Somit kann die notwendige Kontrolle ausgeübt werden.

In Oberon hingegen ist das Klonen per Zuweisung nur zulässig, wenn keine privaten Komponenten enthalten sind. Allerdings gibt es in der Ulmer Oberon-Bibliothek eine weitere etwas aufwendigere Methode zum Klonen für persistente Objekte, da ein abgesichertes persistentes Objekt auch dann zum Leben wieder erweckt werden kann, wenn sein Original noch lebt. Da die Persistenz-Eigenschaft voll von allen betroffenen Abstraktionen und Implementierungen kontrolliert wird, ist dies ebenfalls kein Bruch der Modul- und Abstraktionsgrenzen.

5.2.2 Vergleichbarkeit

Einige Programmiersprachen (wie z.B. Eiffel) bieten für alle Objekte Operatoren für den Test auf Gleichheit und Identität an. Genauso wie bei Zuweisungen ist es möglich zwischen der Identität der Zeiger, der dahinterliegenden Komponenten oder der vollständigen dahinterliegenden Datenstrukturen zu unterscheiden. Eiffel unterstützt hier ebenfalls wieder Überdefinitionen – bietet jedoch auch parallel noch eine nicht überdefinierbare Variante an, die dann im Prinzip die Modulgrenzen bricht.

Spätestens bei den Vergleichen ist es aber im Prinzip zweifelhaft, ob all diese Operationen tatsächlich benötigt werden und mit welcher Semantik sie belegt sind. Schließlich ist für viele Objekttypen mehr als ein Vergleichsoperator denkbar. Für manche mögen beispielsweise dann zwei Autos "gleich" sein, wenn Modell, Baureihe, Ausstattung und Farbe übereinstimmen, während andere auf die Identität der Fahrzeugnummern bestehen würden (also auf dem von dem Werk vorgegebenen eindeutigen Schlüssel).

5.2.3 Ein- und Ausgabeoperationen

Die generelle potentiell für den menschlichen Leser bestimmte Ein- und Ausgabe von Objekten läßt sich genauso wenig zufriedenstellend von dem System vorgeben wie bei den Vergleichsoperatoren, da dies immer von dem Kontext abhängt und mehrere Varianten denkbar sind. Nur bei Debugging-Zwecken mag eine Ausgabe aller Komponenten sinnvoll sein.

Anders sieht es bei einer Persistenz-Eigenschaft aus, die es ermöglicht, ein Objekt so abzulegen und nachher wiederzuladen, daß es in semantisch identischer Weise[3] neu geschaffen wird.

[3] Semantisch identisch heißt, daß ohne Bruch der Abstraktionsgrenzen kein Unterschied festzustellen ist. Das heißt, daß einzelne verborgene Komponenten durchaus unterschiedlich sein können. Wenn beispielsweise sich hinter dem Objekt ein Dateideskriptor auf eine bestimmte Datei verbirgt, kann dieser durchaus einen unterschiedlichen Wert haben und trotzdem auf die gleiche Datei verweisen.

5.3 Ausnahmenbehandlungen

5.3.1 Einführung

Eine Reihe von Programmiersprachen (z.B. C++ und Eiffel) 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) 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).

Folgt man Bertrand Meyer, 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.

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 5.3.1: 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.

Während eines Programmlaufs führen Fehler zu Vertragsverletzungen und damit zu Ausfällen. Ausfälle ergeben Ausnahmesituationen, die bei Entdeckung

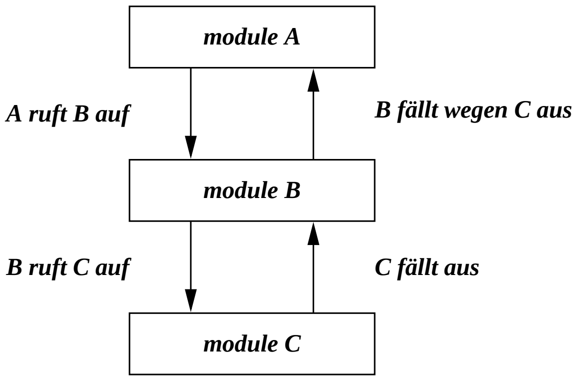

Abbildung 5.2: Transitive Ausfälle

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. 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 Flaviu Cristian ab, der hierfür präzise mathematische Definitionen gegeben hat.

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.

5.3.2 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. 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.
- 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.

5.3.3 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. Bei einigen Programmiersprachen ist es auch in diesem Zusammenhang sinnvoll, lokale Blöcke als lokale Prozeduren zu betrachten (insbesondere C++ und Modula-3).

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.

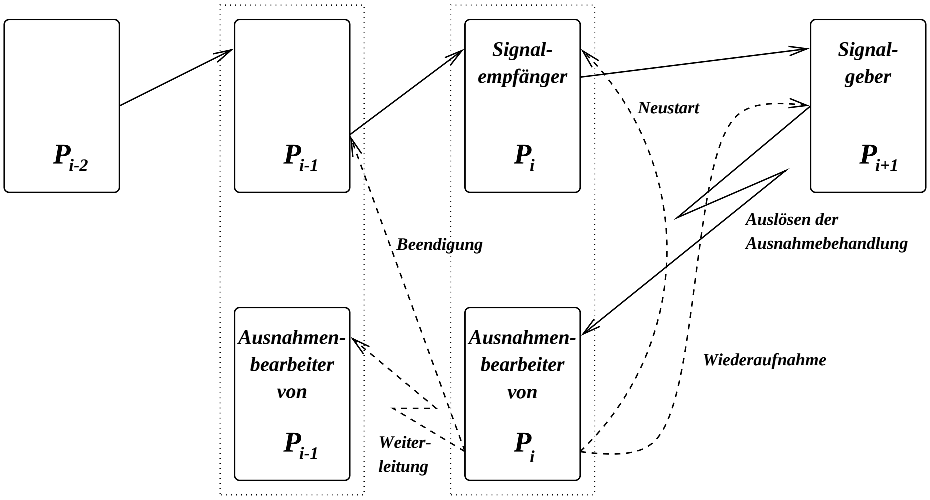

Abbildung 5.3: Optionen eines Ausnahmenbearbeiters

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). 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) unterstützen objektbezogene Ausnahmenbearbeiter.

Ausnahmenbearbeitern stehen – nach einigen normalen Anweisungen – folgende Optionen offen (siehe Abbildung 5.3.3):

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*). 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.

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.

Von der Sprache unterstützte Ausnahmenbehandlungen können einige der genannten Probleme der traditionellen Technik lösen. Als ein Beispiel mag in Modula-3 die Prozedur *PutChar* aus dem Modul *Wr* dienen:

```
(* 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:

```
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 ein Modell auf Basis von Ereignissen und Interessenten als günstiger (wie es in der Ulmer Oberon-Bibliothek als *Events* realisiert ist),[4] 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.

[4] Beispiele hierzu gibt es unter
http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/events.html

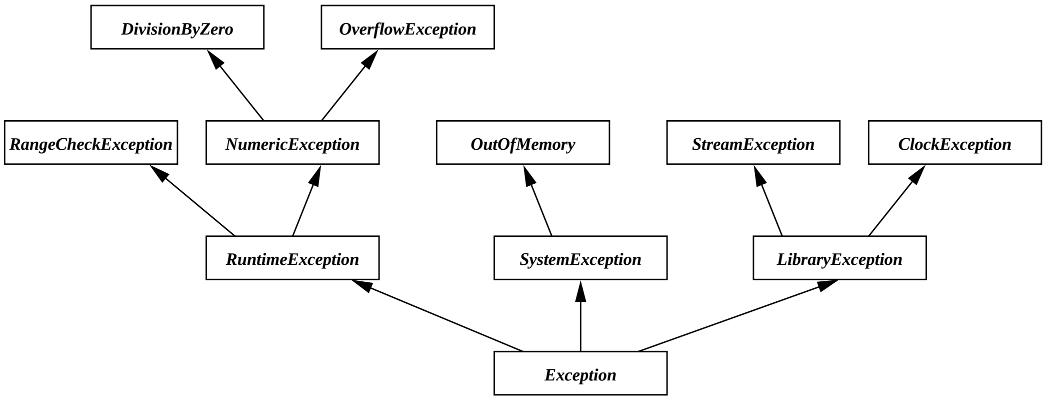

Abbildung 5.4: Beispiel für eine Hierarchie von Ausnahmen

5.3.4 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. 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 5.3.4 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.

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.

5.3.5 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. 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.

5.3.6 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 sich 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.

5.3.7 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.

5.4 Zuordnung von Abstraktionen an Objekte

In den klassischen objekt-orientierten Programmiersprachen werden alle von einem Objekt unterstützten Abstraktionen und alle Komponenten (seien sie nun öffentlich oder privat) zum Zeitpunkt des Kreierens in Abhängigkeit der angegebenen Klasse (oder Typs) festgelegt. Bei *single inheritance* sind damit die unterstützten Abstraktionen auf eine einzelne Klasse und deren Serie von Basisklassen beschränkt, während hingegen bei *multiple inheritance* noch die Möglichkeit besteht, (ggf. mit Hilfe einer Typerklärung) beliebig viele Abstraktionen simultan zu unterstützen. Allerdings sind später nach der Kreation keine Änderungen mehr möglich.

Es gibt jedoch auch Techniken, die Unterstützung weiterer Abstraktionen später zuzulassen:

- Disziplinen erlauben das Anheften beliebiger Informationen für jede interessierte Partei bei einem Objekt (siehe §2.8) bzw. an dessen Typ (siehe §5.6).
- Ein allgemeiner Auftragsnehmer (siehe §2.6) könnte die Registrierung beliebig vieler Auftragnehmer zulassen und Aufträge einem nach den anderen übergeben, bis einer der Auftragnehmer den Auftrag akzeptiert und erledigt. Dieses Verfahren wird beispielsweise von *Messages* unterstützt.
- Manche Programmiersprachen (wie z.B. Perl oder CLOS) gestatten prinzipiell den Zugriff auf die internen Datenstrukturen, die den Aufruf von Methoden regeln.

Wenn es den Unterschied zwischen statisch und dynamisch zugeordneten Abstraktionen gibt, führt dies typischerweise auch dazu, daß beide Arten von Abstraktionen unterschiedlich sprachlich erfaßt werden. Entsprechend ist es dann notwendig, schon bei der Entwicklung einer Abstraktion zu entscheiden, ob sie statisch (primär) oder dynamisch (sekundär) einem Objekt zugeordnet werden soll.

Während eine statische Zuordnung auch eine statische Sicherheit und weitgehende Überprüfbarkeit durch den Compiler gewährleistet, ist dies bei dynamischen Zuordnungen nicht sichergestellt, so daß offen ist, welche dynamischen zuzuordnenden Abstraktionen von einem Objekt tatsächlich unterstützt werden. Entsprechend ist es analog zu Typentests sinnvoll, Testoperationen innerhalb der Bibliothek zu offerieren, die überprüfen, ob ein Objekt eine bestimmte Abstraktion unterstützt.[5]

[5] In der Ulmer Oberon-Bibliothek geschieht dies durch *Services.Installed*.

5.5 Delegationen und Abhängigkeiten zwischen Objekten

Bei einer Delegation reicht ein Objekt alle Operationen (in möglicherweise modifizierter Form) an ein oder ggf. mehrere andere Objekte weiter. Das delegierende Objekt wird dabei als Filter-Objekt oder Proxy bezeichnet. Delegationen sind insbesondere sinnvoll bei

- Filterungen (so eliminiert z.B. *Comments* alle Kommentare bei der Eingabe),
- ergänzenden Operationen (so offeriert z.B. *CopyingStreams* Positionierungen auf reinen Eingabeverbindungen, die das nicht unterstützen),
- Verteilern (wie z.B. *Maps*, das in Abhängigkeit seiner lokalen Position Operationen an eine der untergeordneten Verbindungen weiterleitet),
- Vervielfältigern (wie z.B. in dem Beispiel *DribbleCollections* in §2.6) und
- verteilten Systemen, da dort Objekte in anderen Adressräumen durch Proxies vertreten werden, die die Operationen zum Original-Objekt weiterleiten.

Durch Delegationen entstehen Abhängigkeiten, die den Lebenszyklus der beteiligten Objekte betreffen:

- Wenn das Objekt hinter einem Filter-Objekt terminiert, sind normalerweise Operationen auf dem Filter-Objekt nicht mehr sinnvoll. Entsprechend liegt es nahe, das Leben des Filter-Objekts in diesem Augenblick ebenfalls zu beenden.
- Wenn alle Filter-Objekte (typischerweise wohl nur ein einziges), die auf ein Objekt zeigen, terminieren und es sonst keine weiteren schwergewichtigen Verweise auf dieses Objekt gibt, ist es sinnvoll, auch in umgekehrter Richtung die Terminierung zu propagieren.

Diese Problematik wird von *Resources* unterstützt. Mit Hilfe der Operation *Resources.DependsOn* können entsprechende Abhängigkeiten deklariert werden, wobei *Resources* einerseits einen schwergewichtigen Verweis von dem abhängenden Objekt auf das bezogene Objekt setzt und umgekehrt dafür sorgt, daß die Terminierung des bezogenen Objekts zur Terminierung des abhängenden Objekts führt.

Ein wichtiges Problem bei Filter-Objekten ist die Berücksichtigung sekundärer Abstraktionen, da Module, die Filter-Objekte implementieren, selbst nur eine primäre Abstraktion unterstützen. Eine Lösung hierfür bietet *Forwarders* an, das die Registrierung von sekundären Abstraktionen ermöglicht und diese im Bedarfsfall informiert, wenn ein Filter-Objekt neu erzeugt wird.

Unterstützt werden die durch *Forwarders* registrierten Filter-Objekt-Ketten auch durch *RelatedEvents*, das in so einem Fall automatisch, Ereignisse von dem Ende der Kette an die Front bringt – also in genau umgekehrt zur Bearbeitungsreihenfolge von Operationen. Dies geschieht unabhängig von den weiteren

Einstellungen, die auch eine Weiterleitung von objektbezogenen Ereignissen entlang der Aufrufkette nicht ausschließen (in diesem Fall würden sich Ereignisse in beide Richtungen ausbreiten, bis alle beteiligten Objekte sie mitbekommen haben).

Ein Problempunkt bei Delegationen kann die Lokation von Disziplinen sein. In vielen Fällen sind Module nur an einer Disziplin pro Filterkette interessiert und möchten nur ungern den Verwaltungsaufwand betreiben, der notwendig würde, wenn sie sich bei *Forwarders* registrieren. Hierfür gibt es *IndirectDisciplines*, das die gleiche Schnittstelle wie *Disciplines* aufweist, jedoch die gesamte Kette nach einer Disziplin absucht.

5.6 Typen oder Markierungen als Objekte

Selbst wenn die Programmiersprache Typentests wie z.B. Oberon zur Verfügung stellt, ist dies noch lange nicht so allgemein wie die Einführung von Typen (oder Klassen) als *1st class objects*. So ist es beispielsweise in Oberon nicht möglich herauszufinden, ob der Typ eines Objekts eine Erweiterung des Typs eines anderen Objekts ist. Smalltalk hat zuerst Metaklassen eingeführt und andere Sprachen wie z.B. Self sehen überhaupt keinen Unterschied mehr zwischen Klassen und Objekten.[6]

Wenn die Programmiersprache selbst nicht die notwendige Unterstützung liefert, ist es prinzipiell kein Problem, sie im Rahmen einer Bibliothek nachzuholen, wobei dann allerdings Konventionen entstehen, die konsequent einzuhalten sind, um diese Möglichkeiten umfassend ausnutzen zu können. Um diese Konventionen nicht auf alle zu kreierenden Objekte auszudehnen, werden in der Ulmer Oberon-Bibliothek Typen als *1st class objects* nur für Erweiterungen von *Services.Object*[7] angeboten, das selbst eine Erweiterung von *Disciplines.Object* ist.

Und so sehen die Konventionen in der Ulmer Oberon-Bibliothek aus:

- Jedes Modul muß jede von ihm definierte Erweiterung von *Services.Object* bei *Services* registrieren. Beispiel: Angenommen *Collections* würde *Collections.Collection* als Erweiterung von *Services.Object* definieren, dann würde dies folgendermaßen aussehen:

  ```
  VAR type: Services.Type; (* globale private Variable *)
  (* ... *)
  (* im Initialisierungsteil von Collections *)
  Services.CreateType(type, "Collections.Collection", "");
  ```

  Bei *Services.CreateType* wird als dritter Parameter der Name des Basistyps angegeben, wobei statt *Services.Object* einfach eine leere Zeichenkette angegeben wird. Dies würde bei einer Erweiterung von *Collections.Collection* wie z.B. bei *Lists* anders aussehen:

[6] Eine kurze Übersicht über die Variationen auf diesem Feld liefert die Antwort zur Frage 1.4 in den FAQ zu comp.object, zu finden unter
http://www.cyberdyne-object-sys.com/oofaq/oo-faq-S-1.4.html.

[7] Die Motivation für den Namen *Services* folgt in Kürze.

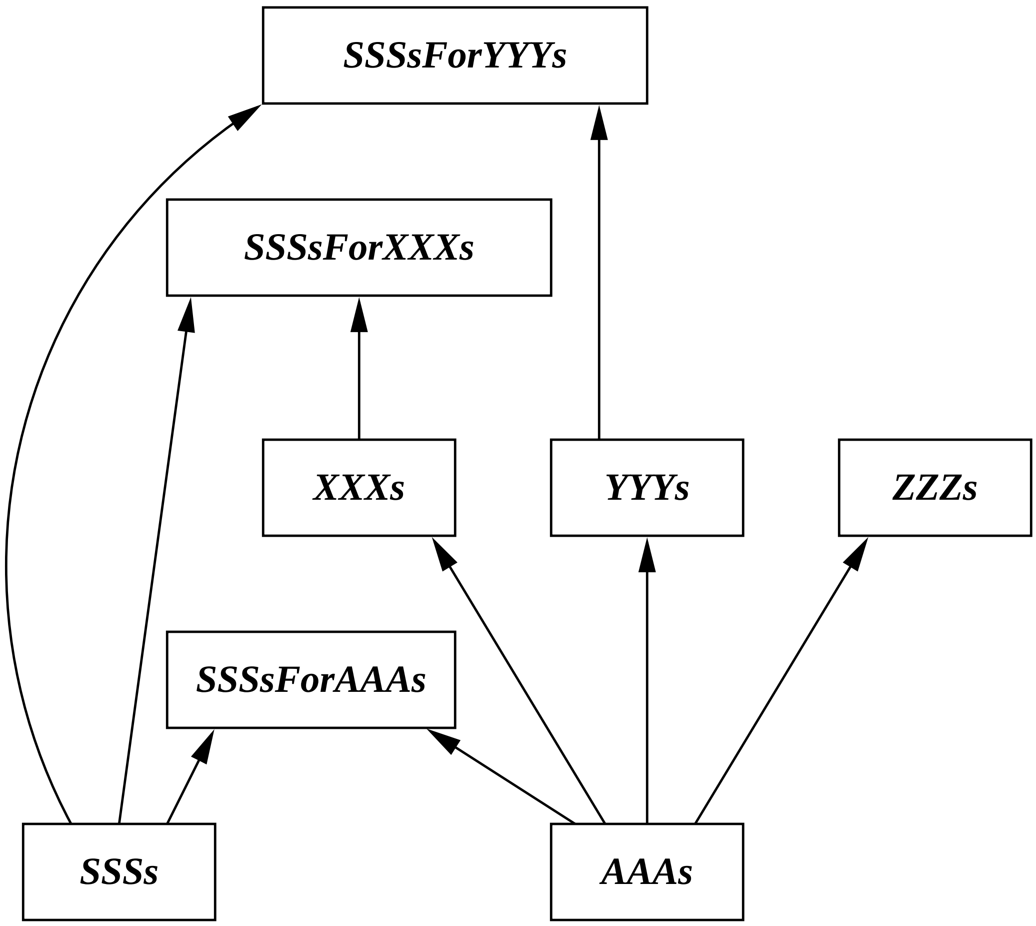

Abbildung 5.5: Importhierarchie bei einer auf *Services* aufbauenden sekundären Abstraktion *SSSs*

```
VAR type: Services.Type; (* globale private Variable *)
(* ... *)
(* im Initialisierungsteil von Lists *)
Services.CreateType(type, "Lists.List", "Collections.Collection");
```

Auf diese Weise lernt *Services* nach und nach die gesamte Typhierarchie, die von *Services.Object* abgeleitet ist, kennen. Natürlich ist die korrekte Aufrufreihenfolge wichtig, da (*Services* darauf besteht, daß der Basistyp bereits registriert ist). Solange allerdings keine Zyklen bei der Importrelation eingebaut worden sind, wird dies durch die Initialisierungsreihenfolge gewährleistet.

- Jedes neu kreierte Objekt einer Erweiterung von *Services.Object* muß mit dem zugehörigen Typ verknüpft werden. Folgende Zeilen müßten dann beispielsweise in *Lists.Create* enthalten sein:

```
Services.Init(list, type); (* zuerst mit dem Typ und dann ... *)
Collections.Init(list, if, caps); (* ... mit der Schnittstelle verknuepfen *)
```

- Damit die Verknüpfung mit dem Typ nicht versehentlich in einem Konstruktor versäumt wird, ist es sinnvoll, dies in der Initialisierungsprozedur der Basisabstraktion zu überprüfen. Folgende Zeilen wären somit in *Collections.Init* aufzunehmen:

```
VAR ctype: Services.Type; (* lokale Variable von Collections.Init *)
(* ... *)
Services.GetType(collection, ctype);
ASSERT((ctype # NIL) & Services.IsExtensionOf(ctype, type));
```

Wenn all diese Vorbereitung stattgefunden haben, ist es möglich, den Typ eines jeden Objekts zu ermitteln und als Objekt zu behandeln, wenn der Typ des Objekts eines Erweiterung von *Services.Object* ist. Da *Services.Type* eine Erweiterung von *Disciplines.Object* ist, besteht auch insbesondere die Möglichkeit, Disziplinen an Typen anzuheften.

Dieser Mechanismus ist von besonderem Interesse für sekundäre Abstraktionen, die nicht von dem Modul unterstützt werden, das für die Implementierung der primären Abstraktion zuständig ist, und für die andererseits auch keine allgemein gültige Implementierung existiert. Stattdessen wird die Implementierung der sekundären Abstraktion von beliebig vielen hinzukommenden Modulen realisiert, die den entsprechenden Dienst jeweils für bestimmte Typen und deren Ableitungen anbieten. Dies wird von *Services* direkt unterstützt und daher ergab sich auch der Name dieses Moduls. Ein mögliches Szenario hierfür illustriert die Abbildung 5.5:

SSSs: Eine sekundäre Abstraktion, deren Implementierung von dem Typ eines Objekts abhängt und daher nicht generell vorgegeben werden kann. Da diese Abstraktion jedoch erst später hinzugekommen ist, können zu ihrer Unterstützung nicht die Implementierungen primärer Abstraktionen aufgebläht werden.

AAAs: Eine primäre Abstraktion mit den Implementierungen *XXXs*, *YYYs* und *ZZZs*, die lange vor der Entstehung von *SSSs* eingefroren worden sind.

SSSsForXXXs: Dieses Modul implementiert die sekundäre Abstraktion *SSSs* für Objekte vom Typ *XXXs.XXX* oder einer Erweiterung davon. Analog unterstützt *SSSsForYYYs* die sekundäre Abstraktion für *YYYs*.

SSSsForAAAs: Hier wird die Unterstützung für alle Erweiterungen von *AAAs.AAA* angeboten, für die es keine spezielle Implementierung gibt. In diesem Szenario betrifft dies *ZZZs*.

Genauso wie bei primären Abstraktionen bieten sich auch hier Schnittstellen-Records an (siehe §2.7), die von *SSSs* öffentlich definiert und über eine *Support*-Operation mit dem jeweiligen Typ verbunden werden. Als Beispiel (Importhier-

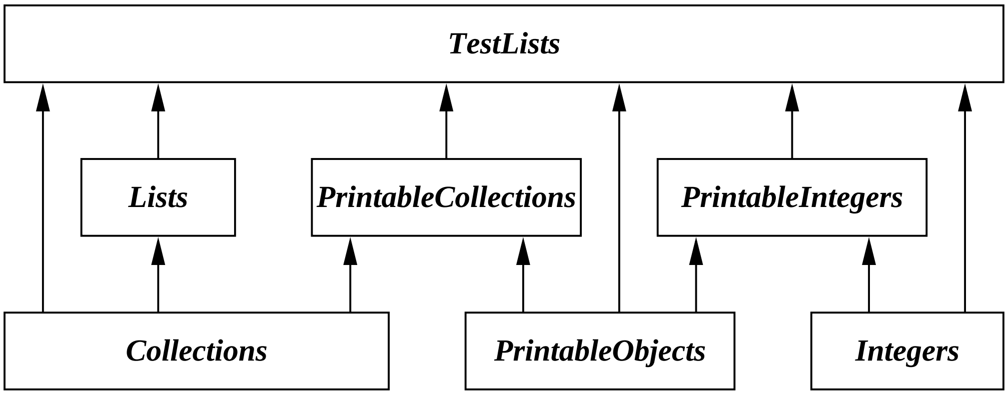

Abbildung 5.6: Importhierarchie zur sekundären Abstraktion *PrintableObjects*

archie in Abbildung 5.6) sei hier eine Variante von *PrintableObjects* gegeben, die auf *Services* basiert:[8]

```
DEFINITION PrintableObjects;

   IMPORT Services, Streams;

   (* secondary abstraction for printable objects
      which is usually to be supported by type-dependent modules
   *)

   TYPE
      PrintProc = PROCEDURE (s: Streams.Stream; object: Services.Object);

   PROCEDURE Support(for: Services.Type; print: PrintProc);
      (* supply print method for all instances of 'for' and its extensions *)

   PROCEDURE SetDefault(s: Streams.Stream; default: ARRAY OF CHAR);
      (* sets the per-stream default output text for objects
         which do not have an associated print method
      *)

   PROCEDURE Print(s: Streams.Stream; object: Services.Object);
      (* call the print method of 'object', or, if not present,
         the per-stream default text, or, if not present, a question mark
      *)

END PrintableObjects.
```

In *Support* wird die Schnittstelle dann an den Typ über eine Disziplin geheftet:

[8]Das Beispiel ist in vollständiger Form unter
http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Collections-14/
zu finden.

```
PROCEDURE Support(for: Services.Type; print: PrintProc);
   (* supply print method for all instances of 'for' and its extensions *)
   VAR
      typeDisc: TypeDiscipline;
BEGIN
   Services.Define(for, service, NIL);
   NEW(typeDisc); typeDisc.id := typeDiscID;
   typeDisc.print := print;
   Disciplines.Add(for, typeDisc);
END Support;
```

Bei der Anwendung wird dann jeweils nach dem unterstützten Basistyp des übergebenen Objekts gesucht und dort die Schnittstelle der Disziplin wieder entnommen:

```
PROCEDURE Print(s: Streams.Stream; object: Services.Object);
   (* call the print method of 'object', or, if not present,
      the per-stream default text, or, if not present, a question mark
   *)
   VAR
      baseType: Services.Type;
      typeDisc: TypeDiscipline;
      streamDisc: StreamDiscipline;
BEGIN
   Services.GetSupportedBaseType(object, service, baseType);
   IF (baseType # NIL) &
         Disciplines.Seek(baseType, typeDiscID, typeDisc) THEN
      typeDisc.print(s, object);
   ELSIF Disciplines.Seek(s, streamDiscID, streamDisc) THEN
      Write.StringS(s, streamDisc.defaultText);
   ELSE
      Write.StringS(s, defaultText);
   END;
END Print;
```

Ein Diensteanbieter wie beispielsweise *PrintableCollections* muß dann nur noch seine Implementierung mittels *PrintableObjects.Support* registrieren:

```
MODULE PrintableCollections;

   (* implements PrintableObjects for Collections *)

   IMPORT Collections, Disciplines, PrintableObjects, Services,
      Streams, Write;

   PROCEDURE PrintCollection(s: Streams.Stream; object: Services.Object);
      VAR
         member: Disciplines.Object;
         first: BOOLEAN;
   BEGIN
      WITH object: Collections.Collection DO
```

```
         Write.CharS(s, "(");
         Collections.First(object);
         first := TRUE;
         WHILE Collections.Next(object, member) DO
            IF first THEN
               first := FALSE;
            ELSE
               Write.StringS(s, ", ");
            END;
            IF member IS Services.Object THEN
               PrintableObjects.Print(s, member(Services.Object));
            ELSE
               Write.CharS(s, "?");
            END;
         END;
         Write.StringS(s, ")");
      END;
   END PrintCollection;

   PROCEDURE Init;
      VAR type: Services.Type;
   BEGIN
      Services.SeekType("Collections.Collection", type);
      PrintableObjects.Support(type, PrintCollection);
   END Init;

BEGIN
   Init;
END PrintableCollections.
```

5.7 Zusammenfassung

In der allgemeinsten und einfachsten Form sind Objekte nur reine Bezugspunkte, die mit beliebig vielen Abstraktionen statisch oder dynamisch verknüpft werden können und bei denen es nicht notwendigerweise eine Instanz gibt, die das gesamte Objekt mitsamt allen beteiligten Datenstrukturen kennt. Stattdessen wird ein Objekt im Laufe seines Lebens möglicherweiser einer Vielzahl von Modulen als Bezugspunkt genannt oder bekannt, von denen jedes seine sehr eingeschränkte Sicht der Dinge besitzt. Aus diesem Grund sind zahlreiche kooperative Konventionen notwendig, damit alle Operationen, die andere Parteien betreffen könnten, den Interessenten zur Kenntnis gelangen.

Grundsätzlich werden Kooperationen immer durch entsprechende Module repräsentiert, bei denen sich alle Interessenten oder Betroffenen registrieren und über die dann später Notifikationen verlaufen. In den demonstrierten Beispielen der Ulmer Oberon-Bibliothek wurde dies unter anderem bei folgenden Punkten realisiert:

- *Resources* verwaltet die Objekt-Zustände im Rahmen des Lebenszyklus, der insbesondere die Terminierung und die Unterbrechung der Kommunikation einschließt. Diese Kooperation erlaubt es, Speicherlecks und Hänger zu vermeiden.

- *RelatedEvents* erlaubt eine objekt-bezogene Ausnahmenbehandlung, bei der alle interessierten Parteien alles an Informationen über einen Ausfall erhalten können.

- *Forwarders* ermöglicht eine Koordinierung bei dem Aufbau von Filterketten, die eine entsprechende Ausnahmenbehandlung, die Verwaltung zusätzlicher privater Datenstrukturen und die Unterstützung sekundärer Abstraktionen ermöglicht.

- *Services* unterstützt sekundäre Abstraktionen, indem es Objekte mit den dafür zur Verfügung stehenden sekundären Abstraktionen und deren Implementierungen zusammenbringt.

Wenn entsprechende Kooperationsmechanismen fehlen, dann geht die Eigenschaft der Erweiterbarkeit verloren, die Ergänzungen ermöglicht, ohne bestehende Module zu verändern.

Kapitel 6

Persistenz

Für Persistenz gibt es zahlreiche Definitionen – recht einfach, umfassend und allgemeingültig ist die von Booch (*Object-Oriented Design*):

> "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 beschränken sich viele Definitionen nur auf den ersten Teil und ignorieren die Möglichkeit, daß Objekte auch von einem Adreßraum zum anderen wandern können.

Traditionell haben sich viele Programme selbst um das Sichern und Einlesen ihrer Daten gekümmert. Allerdings ist damit ein erheblicher Programmieraufwand verbunden, viele Fehlermöglichkeiten kommen dadurch hinzu und diese Methode ist völlig ungeeignet in einem Umfeld, bei dem keine einzelne Instanz mehr ohne weiteres alle Daten eines Objektes kennt.

Zur Unterstützung der Persistenz gibt es eine Reihe von Ansätzen:

Persistenz auf Basis von Spracherweiterungen: Bei zahlreichen klassischen Programmiersprachen wurde das Typsystem in orthogonaler Weise in Richtung von Persistenz und Datenbanken (insbesondere relationale Datenbanken) erweitert. Prominent sind hier insbesondere Pascal/R (Joachim W. Schmidt und andere) mit den Nachfolgern Modula/R und DBPL und E (auf C++ aufbauend von Richardson und anderen). Diese Lösungen bestechen insbesondere in ihrer Eleganz im Vergleich zu den herzlich verunglückten Embedded-SQL-Varianten. Trotz der Eleganz sind die Möglichkeiten sehr beschränkt: Es werden die Limitierungen relationaler Datenbanken übernommen (dynamische Datenstrukturen müssen entsprechend abgebildet werden) und die Verknüpfung mit einer konkreten Datenbank erlaubt auch nicht viel Flexibilität.

Persistenz auf Basis von Systembibliotheken: Viele 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. Dieser Ansatz wird von der großen Mehrheit der objekt-orientierten Programmiersysteme verwendet wie z.B. bei ISE Eiffel, Modula-3 oder BETA.

Der Hauptnachteil liegt in der fehlenden Portabilität: Die Persistenz als unterstützte Eigenschaft ist nicht in der Sprachdefinition vorgesehen und somit wird die Persistenz in jeweils unterschiedlichen Formen implementiert, die wegen ihrer Systemabhängigkeit nicht auf andere Systeme übertragen werden können.

Typischerweise sichern diese Systeme nicht nur ein einzelnes Objekt, sondern analog zur tiefen Kopie (siehe §5.2.1) auch alle davon abhängigen Objekte. Dies ist weitgehend wünschenswert, kann aber auch zu Problemen führen, wenn größere temporäre Datenstrukturen ungewollt mitgesichert werden (z.B. interne Buffer einer offenen Dateiverbindung). Einige Systeme offerieren eine Lösung durch die Möglichkeit der Überdefinition (oder Ergänzung) der System-Methoden zum Sichern und Einlesen von Objekten.

Persistenz auf Basis von Betriebssystemen: Prozesse und Adreßräume 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 (siehe beispielsweise das Monads-Projekt von Keedy[1]). Diese Lösung ist sehr elegant und ermöglicht auch verteilte Systeme. Leider steht sie nur auf einigen wenigen experimentellen Betriebssystemen zur Verfügung.

Persistenz auf Basis systemunabhängiger Bibliotheken: Wenn kein spezielles Betriebssystem zur Verfügung steht, die Programmiersprache nicht unnötig erweitert werden soll und die Unabhängigkeit von dem Programmiersystem wünschenswert ist, dann muß die Lösung in einer systemunabhängigen und damit portablen Bibliothek gesucht werden. Hierfür müssen zwar möglicherweise Performance-Einbußen hingenommen werden, jedoch sollte die dadurch gewinnbare Flexibilität darüber hinwegtrösten.

6.1 Grundtechniken

Generell stellt sich zunächst die Frage nach dem Medium, in dem Objekte länger leben bzw. bewegt werden können. Statt konkret an Dateisysteme oder gar Datenbanken zu denken, empfiehlt es sich, die notwendige Grundeigenschaft zu verlangen, die für alle weiteren Varianten die Voraussetzung bildet:

[1] Mehr dazu unter http://www.informatik.uni-ulm.de/abt/rs/staff/projects/monads/PersistentVMemoryE.html

> Ein Objekt ist genau dann persistent, wenn es in eine Sequenz von Bytes verwandelt werden kann und umgekehrt aus der Sequenz von Bytes wieder ein Objekt erzeugt werden kann, das semantisch identisch zum Ausgangs-Objekt ist.[2]

Wenn ein Objekt einmal in eine Bytesequenz verwandelt worden ist, kann es anschließend beliebig in Dateien und Datenbanken abgelegt oder über das Netzwerk zu anderen Prozessen und Adreßräumen übermittelt werden. Dies wird insbesondere durch eine geeignete Abstraktion für Bytesequenzen erleichtert, wie sie z.B. durch *Streams* bei der Ulmer Oberon-Bibliothek gegeben ist.

Im Umfeld objekt-orientierter Programmiersprachen liegt es nahe, die Objekte selbst die Verantwortung für die Konvertierungen übernehmen zu lassen. Allerdings ergeben sich dabei zwei Probleme:

- Angenommen, persistente Objekte hätten die Methoden *Read* und *Write*. Der Aufruf von *Write* ist kein Problem – jedoch: Wie soll eine Methode *Read* aufgerufen werden, wenn das Objekt noch nicht da ist und nur als Bytesequenz vorliegt?
- Ein kleineres Problem ergibt sich dadurch, daß an einem Objekt beliebig viele Module mit privaten Informationen beteiligt sein können. Entsprechend werden bei den Konvertierungen kooperative Mechanismen benötigt, die alle interessierten Parteien einschließen.

Das zweite Problem kann bei klassischen Programmiersprachen durch die Verpflichtung zum Aufruf der übergeordneten Methode (Super-Methode) gelöst werden oder allgemein durch entsprechende Registrierungstechniken. Das erste und entscheidende Problem läßt sich nur durch die Einführung von Typinformationen lösen und setzt damit die Existenz von Typen als *1st class objects* voraus.[3]

Für den Fall, daß die Programmiersprache Typen nicht direkt als Objekte zugänglich macht, reicht es insbesondere nicht aus, beim Einlesen nur genügend Speicher für eine Kopie zu organisieren und die Bytesequenz zurück in den Speicher zu kopieren. Denn hierbei würde die interne Typinformation verloren gehen, für die kein portabler Zugang zur Verfügung steht (sonst hätten wir Typen als *1st class objects*). Somit müssen die Objekte beim Einlesen wieder auf die gewohnte Weise kreiert werden, und da der **NEW**-Operator den Typen auf statisch festgelegte Weise direkt oder indirekt erhält, kann dies nicht auf allgemeine Weise geschehen, sondern nur über für jeden persistenten Typ registrierte Konstruktoren.

Um dies zu erreichen, ist es notwendig, nicht nur die Objektdaten zu sichern, sondern auch noch die Typinformation (siehe Abbildung 6.1). Dabei ist es am geeignetsten, einen global eindeutigen Typnamen in Form einer Zeichenkette zu verwenden.

Somit werden bei einer reinen Bibliothekslösung folgende Methoden benötigt:

[2]Hinweis: Diese Definition ist nicht allgemeingültig und nur bei reinen Bibliothekslösungen sinnvoll, da beispielsweise bei dem Monads-System keine Konvertierungen oder Transfers notwendig sind.

[3]Es gibt Lösungen, die nur mit Modulen als *1st class objects* auskommen, wenn pro Modul nur ein persistenter Typ definiert wird. Jedoch führt ansonsten prinzipiell kein Weg daran vorbei.

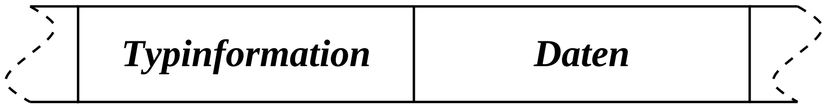

Abbildung 6.1: Darstellung eines Objekts in einer Bytesequenz

Create: Eine Methode in Abhängigkeit der Typinformation, die ein entsprechendes Objekt kreiert und initialisiert.

Read: Konvertierung einer Bytesequenz in den Inhalt eines Objekts, das bereits existiert.

Write: Konvertierung eines Objekts in eine Sequenz von Bytes.

Theoretisch können *Create* und *Read* zusammengelegt werden – dann wären jedoch keine kooperativen Lösungen möglich, wenn mehr als eine Partei private Daten zu dem Objekt unterhält.

Mit der im Rahmen einer Bibliothek realisierten Persistenzeigenschaft würde dann folgender manuell zu programmierender Verwaltungsaufwand notwendig werden:

- Jedes Modul (bzw. Klasse) muß zu seiner Initialisierungszeit seine persistenten Datentypen registrieren zusammen mit den zugehörigen Methoden *Create*, *Read* und *Write*. Bei Abstraktionen ohne eigene Datenstrukturen werden keine Methoden benötigt.

- Wenn Typen als *1st class objects* im Rahmen der Bibliothek realisiert sind, dann müssen neu kreierte Objekte mit den zugehörigen Typen verbunden werden.

6.2 Schnittstelle

In der Ulmer Oberon-Bibliothek wird die Eigenschaft der Persistenz durch das Modul *PersistentObjects* definiert, das den Typ *PersistentObjects.Object* exportiert, der eine Erweiterung von *Services.Object* ist. Persistenz wurde als primäre Abstraktion definiert, da

- diese Eigenschaft alle betroffenen Parteien einschließt und deswegen normalerweise nicht nachträglich über eine sekundäre Abstraktion definiert werden kann und weil

- so die angenehme statische Gewißheit besteht, ob Objekte persistent sind oder nicht.

Der für *PersistentObjects* notwendige Verwaltungsaufwand besteht analog zu Erweiterungen von *Services.Object* aus der Registrierung der Typen (*Services.CreateType*), der Verknüpfung von Objekten mit Typen (*Services.Init*) und zusätzlich den angesprochenen Schnittstellenprozeduren. Um den Verwaltungsaufwand zu reduzieren, wird die Registrierung bei *Services* von *PersistentObjects* übernommen.

Für einen einfachen Objektdatentyp für ganze Zahlen sieht das beispielsweise folgendermaßen aus:[4]

Gegeben sei der Datentyp:

```
TYPE
   Integer = POINTER TO IntegerRec;
   IntegerRec = RECORD (PersistentDisciplines.ObjectRec) val: INTEGER END;
```

Als Basistyp wird hier *PersistentDisciplines.Object* verwendet, da damit persistente Disziplinen unterstützt werden – genauso wie typischerweise *Disciplines.Object* gegenüber *Objects.Object* vorzuziehen ist. Dann sind folgende Änderungen im Vergleich zu einer direkten Erweiterung von *Services.Object* notwendig:

1. Im Initialisierungsteil des Moduls, das einen persistenten Typ definiert, ist statt

   ```
   Services.CreateType(type, "Integers.Integer", "");
   ```

   nun

   ```
   PROCEDURE Init;
      VAR
         persif: PersistentObjects.Interface;
   BEGIN
      NEW(persif);
      persif.create := CreateInteger;
      persif.read := ReadInteger;
      persif.write := WriteInteger;
      persif.createAndRead := NIL;
      PersistentObjects.RegisterType(type,
         "Integers.Integer", "PersistentDisciplines.Object", persif);
   END Init;
   ```

   aufzurufen.

2. Im Konstruktor wird statt mit

   ```
   Services.Init(new, type);
   ```

   die Verknüpfung mit

   ```
   PersistentObjects.Init(new, type);
   ```

[4] Der Programmtext ist den Beispielen *Collections-14* (noch ohne Persistenz auf direkter Basis von *Services.Object* und *Collections-16* (mit Persistenz) entnommen – zu finden unter http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Collections-14/ und http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Collections-16/.

hergestellt.

3. Hinzu kommen die drei Schnittstellenprozeduren für *PersistentObjects*:

```
PROCEDURE CreateInteger(VAR object: PersistentObjects.Object);
   VAR
      integer: Integer;
BEGIN
   Create(integer, 0);
   object := integer;
END CreateInteger;

PROCEDURE ReadInteger(s: Streams.Stream;
                      object: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH object: Integer DO
      RETURN NetIO.ReadInteger(s, object.val)
   END;
END ReadInteger;

PROCEDURE WriteInteger(s: Streams.Stream;
                       object: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH object: Integer DO
      RETURN NetIO.WriteInteger(s, object.val)
   END;
END WriteInteger;
```

   Das Modul *NetIO* unterstützt die Ein- und Ausgabe aller Basistypen von Oberon in einer (für jeden *Stream*) adaptierbaren Weise, mit der ggf. Inkompatibilitäten zwischen verschiedenen Hardware-Architekturen ausgeglichen werden können.

Der Verwaltungsaufwand läßt sich bei Abstraktionen weiter reduzieren, die keine eigenen Daten unterhalten, die zu sichern und zu restaurieren sind. Dies trifft beispielsweise für eine Version von *Collections* zu, die persistent ist. Hier kann ganz einfach bei der Schnittstelle **NIL** übergeben werden und damit auf die Implementierung der Schnittstellenprozeduren verzichtet werden:

```
PersistentObjects.RegisterType(type,
   "Collections.Collection", "PersistentDisciplines.Object", NIL);
```

Auf solche Weise vorbereitete Objekte können dann mit *PersistentObjects.Write* in eine Sequenz von Bytes (über einen *Stream*) verwandelt oder von einem *Stream* mit *PersistentObjects.Read* wieder restauriert werden:

```
ok := PersistentObjects.Write(s, collection);
```

Beim Einlesen empfiehlt es sich, etwas vorsichtig zu sein, da nicht sichergestellt ist, daß genau das vorgefunden wird, was einmal früher korrekt abgelegt worden ist. Wenn beispielsweise

```
VAR collection: Collections.Collection;
(* ... *)
ok := PersistentObjects.Read(s, collection);
```

aufgerufen wird und das eingelesene Objekt keine Kollektion ist, dann hat *PersistentObjects.Read* keine Gelegenheit, **FALSE** zurückzuliefern – stattdessen gibt es einen Laufzeitfehler: *type guard failure*.

Um dies zu vermeiden, gibt es zwei Möglichkeiten:

1. Einzulesendes Objekt als *PersistentObjects.Object* deklarieren und anschließend mit einem Typentest überprüfen:

   ```
   VAR
      collection: Collections.Collection;
      object: PersistentObjects.Object;
   (* ... *)
   IF PersistentObjects.Read(s, object) &
         (object IS Collections.Collection) THEN
      collection := object(Collections.Collection);
      (* ... *)
   END;
   ```

2. Eleganter geht es mit *PersistentObjects.GuardedRead*:

   ```
   VAR
      collection: Collections.Collection;
      guard: Services.Type;
   (* ... *)
   Services.SeekType("Collections.Collection", guard);
   (* ... *)
   ok := PersistentObjects.GuardedRead(s, guard, collection);
   ```

   Hier führt *PersistentObjects* selbst einen Typentest mit *guard* durch, bevor eine Zuweisung an *collection* stattfindet und liefert **FALSE** zurück, falls das eingelesene Objekt keine Erweiterung des *guard* ist.

6.3 Sicherung und Restaurierung größerer Datenstrukturen

Die vorhandene Schnittstelle kann auch bequem im Falle beliebiger dynamischer Datenstrukturen verwendet werden, da die Schnittstellenprozeduren zum Schreiben und Lesen natürlich *PersistentObjects.Write* und *PersistentObjects.Read* rekursiv aufrufen dürfen.

Falls Zeiger möglicherweise **NIL** sind, können auch *PersistentObjects.WriteObjectOrNIL* und *PersistentObjects.ReadObjectOrNIL* verwendet werden. So könnte beispielsweise eine Sicherungsprozedur für einen binären Baum aussehen:

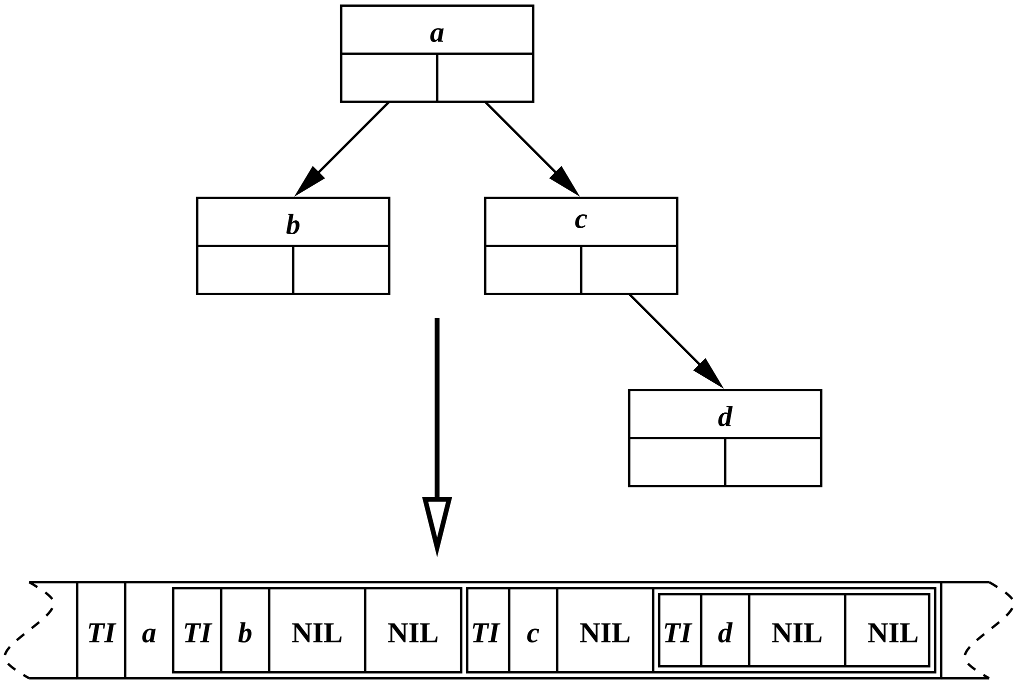

Abbildung 6.2: Serialisierung eines binären Baumes

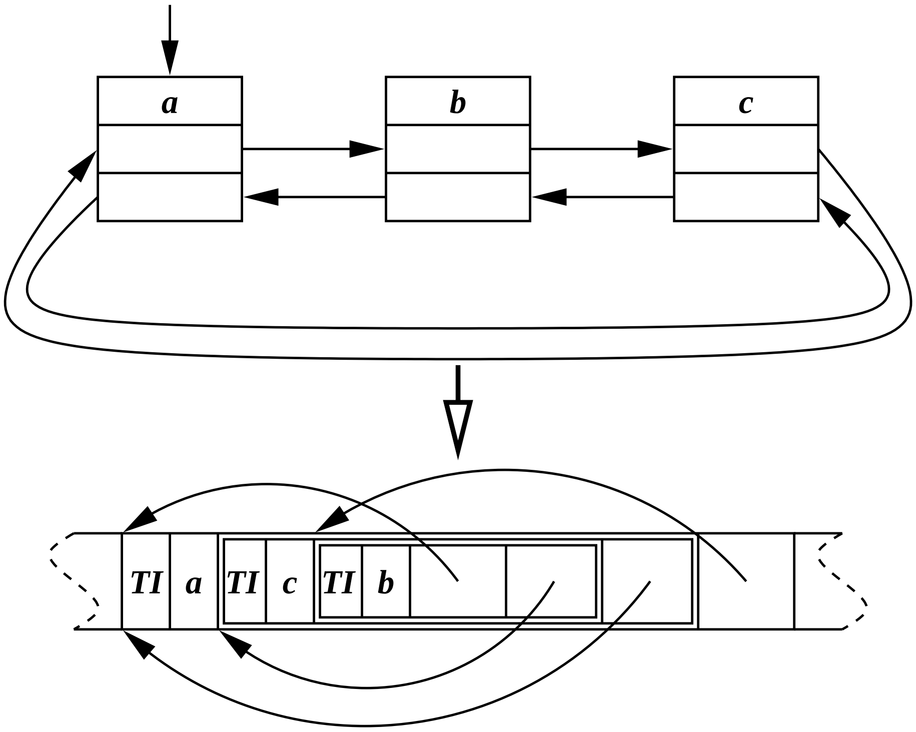

Abbildung 6.3: Serialisierung einer zyklischen Datenstruktur

```
PROCEDURE WriteTree(s: Streams.Stream;
                    tree: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH tree: Tree DO
      RETURN PersistentObjects.WriteObject(s, tree.object) &
             PersistentObjects.WriteObjectOrNIL(s, tree.left) &
             PersistentObjects.WriteObjectOrNIL(s, tree.right)
   END;
END WriteTree;
```

Abbildung 6.2 zeigt das Schema eines entsprechend serialisierten binären Baumes.

Die Rekursion führt natürlich bei zyklischen Datenstrukturen zu Problemen, die mit dem Modul *LinearizedStructures* elegant vermieden werden können. *LinearizedStructures* verwaltet über eine private Disziplin bei dem *Stream* die Liste der bereits gesicherten (bzw. eingelesenen) Objekte und gibt entsprechend bei Wiederholungen Verweise aus und kann sie beim Einlesen wieder interpretieren. Entsprechend ist es beispielsweise möglich, einen zweiseitig verketteten Ring zu sichern (siehe auch Abbildung 6.3):

```
PROCEDURE WriteRing(s: Streams.Stream;
                    ring: PersistentObjects.Object) : BOOLEAN;
BEGIN
   WITH ring: Ring DO
      RETURN LinearizedStructures.WriteObject(s, ring.object) &
             LinearizedStructures.WriteObject(s, ring.prev) &
             LinearizedStructures.WriteObject(s, ring.next)
END WriteRing;
```

Dabei sind *LinearizedStructures.WriteObject* und *LinearizedStructures.ReadObject* analog zu *PersistentObjects.WriteObjectOrNIL* und *PersistentObjects.ReadObjectOrNIL* tolerant gegenüber **NIL**-Zeigern.

Manchmal ist es aber auch einfacher, eine gesamte dynamische Datenstruktur in einer Prozedur zu sichern (bzw. zu restaurieren), wenn die beteiligten Objekttypen privat definiert werden. Dies spart den Aufwand für mehrere persistente Datentypen. Als Beispiel sei hier die Sicherungsprozedur von *Lists* aus dem Beispiel *Collections-16*[5] gegeben:

```
CONST
   objectSY = "O"; endSY = "E"; (* encoding for persistent representation *)

(* ... *)

PROCEDURE WriteList(s: Streams.Stream;
                    list: PersistentObjects.Object) : BOOLEAN;
   VAR
      member: Linkable;
BEGIN
   WITH list: List DO
```

[5]Zu finden unter http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Collections-16/.

```
      member := list.head;
      WHILE member # NIL DO
         IF member.object IS PersistentObjects.Object THEN
            IF ~NetIO.WriteChar(s, objectSY) OR
                  ~PersistentObjects.Write(s,
                     member.object(PersistentObjects.Object)) THEN
               RETURN FALSE
            END;
         END;
         member := member.next;
      END;
   END;
   RETURN NetIO.WriteChar(s, endSY)
END WriteList;
```

Wenn die Anzahl der gesicherten Objekte in so einem Fall variabel ist, empfiehlt es sich, die Zahl der Objekte oder – wie im Beispiel – spezielle Symbole auszugeben, die das korrekte Einlesen nachher ermöglichen. In diesem Beispiel wurde auf diese Weise vermieden, den internen Datentyp *Linkable* als persistentes Objekt zu definieren. Außerdem ist diese Lösung effizienter, da sie ohne Rekursion auskommt.

6.4 Zusammenfassung

Entscheidend ist bei einer Definition der Persistenzeigenschaft in einer systemunabhängigen Bibliothek die Beschränkung auf die wesentliche Eigenschaft – das Konvertieren eines Objekts in eine Bytesequenz und zurück – und der Verzicht auf darüberhinausgehende Mechanismen wie Transaktionen, Objektidentitäten und Datenbanken. Auf diese Weise wurden beide Punkte der Definition von Booch frei Haus geliefert, da sich Bytesequenzen sowohl in Dateien und Datenbanken ablegen lassen als auch über das Netz transferiert werden können.

Auf *PersistentObjects* aufbauend gibt es entsprechend in der Ulmer Oberon-Bibliothek auch zahlreiche weitere Mechanismen, die über die Basiseigenschaft hinausgehende Anwendungen unterstützen, so zum Beispiel *TransStreams* (Transaktionen auf *Streams*, wird als Filterobjekt eingesetzt), *Containers* (Objektdatenbanken) und *Messages* (persistente Aufträge und damit einer der Grundvoraussetzungen für verteilte Systeme).

Kapitel 7

Parallelität und Synchronisierung

Parallelität im Bereich von Programmiersprachen ist unter 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. Aus diesem Grund wurden parallelisierende Übersetzer entwickelt, die es erlauben, die Programme in der lesbareren sequentiellen Form zu belassen. Einige neuere Programmiersprachen wie Fortran-90 und High Performance Fortran erlauben eine Reihe zusätzlicher Angaben, die entsprechende Optimierer und Parallelisierer unterstützen.
- 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 objektorientierte Programmiersprachen, wie z.B. Simula, Actor und ABCL/1, 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, wird auch häufig von Quasi-Parallelität gesprochen. 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 Stack, eine aktuelle Position im Programm und ein Satz Register. 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.

Der besondere Vorteil von Koroutinen ist, daß keine Unterstützung von Seiten des Betriebssystems erforderlich ist.

- Verschiedene Betriebssysteme (insbesondere die neueren UNIX-Varianten wie Mach, Solaris 2 und Digital UNIX) 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 *Communicating Sequential Processes* 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.

Unabhängig von der Art der Parallelisierung sind Sprachmittel zur Synchronisierung zu sehen – mit etwas Glück und durch den Einsatz geeigneter Abstraktionen können auch beide orthogonal (also voneinander völlig unabhängig) gelöst werden. Die größte Flexibilität wird natürlich gewonnen, wenn beides – soweit wie möglich – innerhalb einer Bibliothek gelöst wird, wenngleich dies nicht immer sehr einfach zu portieren ist und auch Effizienzeinbußen mit sich bringen kann.

7.1 Koroutinen

7.1.1 Einführung

Koroutinen erlauben die Nebenläufigkeit von mehreren Programmteilen auf sehr einfache Weise und wurden als Sprachmittel zuerst 1963 von Conway und Erdwinn eingeführt. 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 Stack, eine eigene aktuelle Position und einen eigenen Registersatz. 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 kaum teurer als ein durchschnittlicher Prozeduraufruf.

7.1.2 Koroutinen in Modula-2

Modula-2 offeriert Koroutinen auf Basis von zwei Operationen aus dem Modul **SYSTEM**:

```
PROCEDURE NEWPROCESS(proc: PROC;
                     addr: ADDRESS; size: CARDINAL;
                     VAR new: ADDRESS);
PROCEDURE TRANSFER(VAR source, destination: ADDRESS);
```

NEWPROCESS kreiert eine neue Koroutine mit einem Stack, 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 Stackgröße sehr systemabhängig. Da der Stack mit Anfangsadresse und Größenangabe fest vom Aufrufer vorgegeben werden muß, ist es konzeptionell unmöglich, Koroutinen mit wachsendem Stack 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 Stacks abgelegt werden.
- Die Verwaltungsinformation wird allokiert zum Zeitpunkt des Aufrufs von **TRANSFER**. Geeignet wäre hierfür die Spitze des Stacks.

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 bzw. die zuletzt aktive 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;
```

7.1.3 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:

```
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 Stack übergeben noch die Größe des Stacks festgelegt werden.

CRSPAWN ist dadurch natürlich etwas aufwendiger als **NEWPROCESS** und führt folgende Schritte durch:

- Kreieren eines neuen Stacks. Die Anfangsgröße des Stacks wird von dem Umfang der lokalen Daten der Koroutinenprozedur und einem vorgegebenen Wert aus *Coroutines.defaultsize* bestimmt. *Coroutines.defaultsize*

[1] Seit einiger Zeit gibt es deswegen auch mehrere Varianten des Züricher Oberon-Systems mit Koroutinen.

ist eine unter normalen Umständen ausreichende Größe, die natürlich leicht erhöht werden kann. In diesem Fall bezieht sich die Vergrößerung gleichmäßig auf alle danach erzeugten Koroutinen. Der Stack 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 Stacks von Koroutinen wachsen können.

- Die Verwaltungsinformation für die Koroutine wird am Anfang des Stacks angelegt und initialisiert. Die Verwaltungsinformation enthält die Stackverwaltungsregister 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 Stack 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. Die Zeiger auf die zuletzt aktive Koroutine und die aktuelle Koroutine werden laufend unterhalten und sind 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
         *)
      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:[2]

[2]Als vollständiges Beispiel steht es unter http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/ProdCons/ zur Verfügung.

```
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;
```

7.1.4 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 die Rolle des 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 (Services.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 als Parameter den Verweis auf den Iterator selbst vor, dann eine beliebige Erweiterung von *Objects.Object* und einen weiteren kleinen Parameter, der zur Auswahl verschiedener Iterationsmodi dienen kann (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 *Get* auf der Konsumentenseite **FALSE** zurückliefert.

Es lohnt sich, Iteratoren bei zahlreichen Abstraktionen wie z.B. bei *Collections* zum Durchwandern von Datenstrukturen einzusetzen, weil dadurch

- eine Datenstruktur mehrfach gleichzeitig durchlaufen werden kann und
- die Implementierung erleichtert wird, insbesondere wenn die Datenstruktur und damit die Traverse rekursiv ist.

Entsprechend wäre es sinnvoll, bei *Collections* statt *First* und *Next* eine Operation *GetIterator* zur Verfügung zu stellen:[3]

```
PROCEDURE GetIterator(collection: Collection;
                      VAR iterator: Iterators.Iterator);
```

Bei der zugehörigen Implementierung *Lists* wird der Iterator durch die Prozedur *IterateList* repräsentiert, die dann durch die Schnittstellenprozedur *GetIterator* zum Leben erweckt wird:

```
PROCEDURE IterateList(it: Iterators.Iterator;
                      list: Objects.Object;
                      param: SHORTINT); (* ignored *)
   VAR
```

[3] Das Beispiel ist vollständig zu finden unter http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Collections-17/

```
      linkable: Linkable;
BEGIN
   WITH list: List DO
      linkable := list.head;
      WHILE linkable # NIL DO
         Iterators.Yield(it, linkable.object);
         linkable := linkable.next;
      END;
   END;
END IterateList;

PROCEDURE GetIterator(collection: Collections.Collection;
                      VAR iterator: Iterators.Iterator);
BEGIN
   Iterators.Create(iterator, IterateList, collection, 0);
END GetIterator;
```

Der vierte Parameter von *Iterators.Create* wird als dritter Parameter an *IterateList* übergeben und dient zur vereinfachten Parameterisierung des Iterators, so kann beispielsweise die Reihenfolge spezifiziert werden. In diesem Beispiel wird davon aber kein Gebrauch gemacht.

In *PrintableCollections* könnte dann ein Iterator auf folgende Weise zum Durchlaufen einer Kollektion verwendet werden:

```
PROCEDURE PrintCollection(s: Streams.Stream; object: Services.Object);
   VAR
      member: Disciplines.Object;
      first: BOOLEAN;
      it: Iterators.Iterator;
BEGIN
   WITH object: Collections.Collection DO
      Write.CharS(s, "(");
      Collections.GetIterator(object, it);
      first := TRUE;
      WHILE Iterators.Get(it, member) DO
         IF first THEN
            first := FALSE;
         ELSE
            Write.StringS(s, ", ");
         END;
         IF member IS Services.Object THEN
            PrintableObjects.Print(s, member(Services.Object));
         ELSE
            Write.CharS(s, "?");
         END;
      END;
      Write.StringS(s, ")");
   END;
END PrintCollection;
```

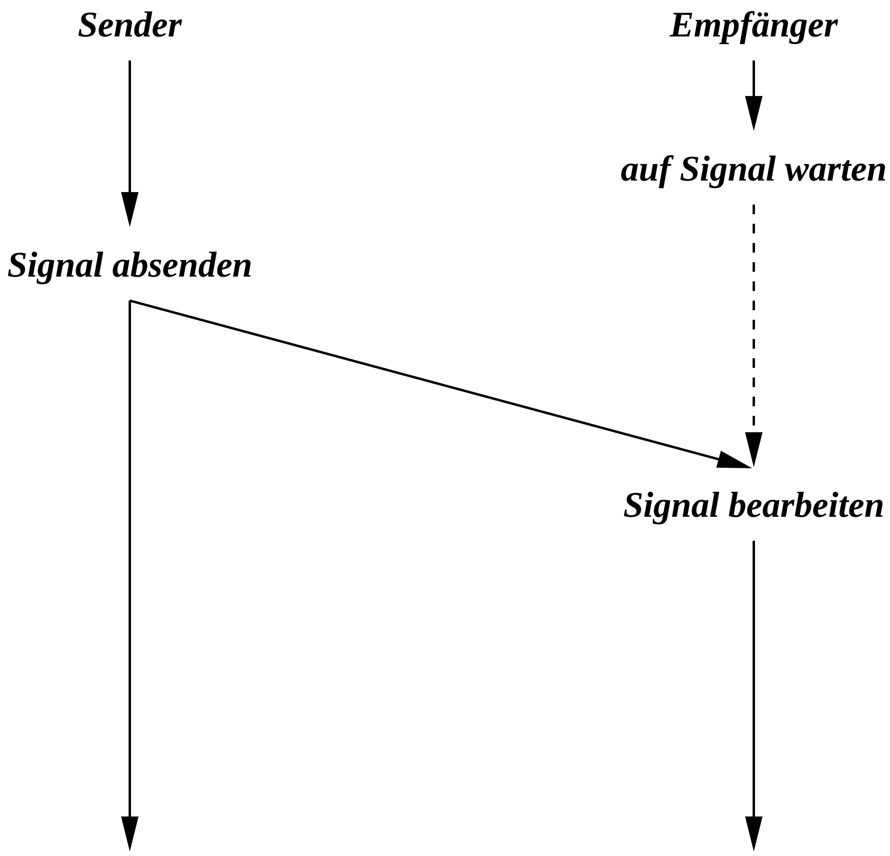

Abbildung 7.1: Kommunikation über asynchrone Signale

7.2 Synchronisierungsmechanismen

Synchronisierungen werden primär zur Kommunikation zwischen den parallel ausgeführten Einheiten und zum gegenseitigen Ausschluß bei Zugriff auf gemeinsame Ressourcen benötigt. Es gibt einige wenige prinzipielle Synchronisierungsmechanismen, die sich in einer sehr großen Vielzahl von Sprachmitteln wiederfinden. Ihre Verwendung ist prinzipiell unabhängig von der Art der Parallelisierung, wenngleich dies natürlich nicht notwendigerweise für ihre Implementierung gilt.

7.2.1 Gegenseitiger Ausschluß

Dijkstras P- und V-Operationen gehören zu den ersten Synchronisierungsmechanismen, die einen gegenseitigen Ausschluß auf denkbar einfache Weise realisieren. So einfach wie sie sind, so problematisch können sie sein:

- Solange der Zugriff auf ein Semaphor nicht eng umgrenzt ist (z.B. durch ein Modul), ist es nicht einfach zu verifizieren, daß es nicht zu Deadlocks kommt durch eine vergessene V-Operation, ein mehrfaches P der gleichen Partei oder durch eine ungünstige Reihenfolge der P-Operationen wie z.B.

bei

```
(* Partei 1 *)  P(mutex1); P(mutex2); (* ... *) V(mutex2); V(mutex1);
(* Partei 2 *)  P(mutex2); P(mutex1); (* ... *) V(mutex1); V(mutex2);
```

- Es gibt bei Dijkstras Operationen keine Kontrollmöglichkeiten, nach welchen Kriterien der nächste Anwärter ausgesucht wird, nachdem ein Semaphor freigegeben worden ist. Dies kann beispielsweise zum Verhungern führen, d.h. daß zwar eine der beteiligten Parteien theoretisch wieder zur Ausführung käme – praktisch aber keine Chance dazu hat.
- Wenn nur Semaphore zur Verfügung stehen, muß die Kommunikation zwischen den Parteien zwangsläufig über gemeinsam zugängliche Ressourcen abgewickelt werden. Dies ist nicht nur vergleichsweise ineffizient, sondern auch umständlich.

Weitere Varianten nach den P- und V-Operationen von Dijkstra sind kritische Regionen (mit impliziten P- und V-Operationen) von Brinch Hansen in Concurrent Pascal, Pfad-Ausdrücke von Campbell und Habermann und Kapseln von Gehani in Concurrent C++. Wenngleich durch diese Verfeinerungen Deadlocks und das Verhungern eher vermieden werden können, bleibt die Trennung zwischen Synchronisierung und Kommunikation und damit der Zwang zur Nutzung gemeinsamer Ressourcen, um Informationen auszutauschen.

7.2.2 Asynchrone Signale

Ein relativ einfacher und mächtiger Mechanismus sind asynchrone Signale. Wie in Abbildung 7.1 dargestellt, kann ein Sender jederzeit an eine andere Partei (im Diagramm als Empfänger bezeichnet) ein Signal absenden. Der Sender kann anschließend seine Tätigkeit ungehindert fortsetzen, während das Signal (typischerweise mit Informationen behaftet) zeitversetzt bei dem Empfänger eintrifft.

Der Empfänger hat sich möglicherweise vorher darauf vorbereitet und angegeben, daß er auf eine bestimmte Sorte von Signalen (möglicherweise auf beliebige Signale) zu reagieren bereit ist und solange wartet, bis eines eintrifft. Wenn der Empfänger wegen dem eingetroffenen Signal aufgeweckt wird, kann er sich damit beschäftigen. Eine Rückantwort ist in diesem Grundmechanismus nicht vorgesehen, jedoch anschließend unter Ausnutzung der gleichen Technik mit vertauschten Rollen natürlich realisierbar. Verschiedene Möglichkeiten bestehen, wenn der Empfänger nicht vorbereitet ist: So könnte das Signal einfach ignoriert werden oder in einem Puffer warten, bis sich der Empfänger später dafür interessiert.

Wenn asynchrone Signale nicht nur der Informationsübermittlung dienen, sondern auch einen bestimmten Status bei gemeinsamen Ressourcen anzeigen, dann sind sie mit Vorsicht zu genießen, da eine Bedingung bezüglich der gemeinsamen Ressource, die zum Absendezeitpunkt noch zutraf, noch vor dem Eintreffen beim Empfänger sich durch die weiteren Aktivitäten des Absenders ändern kann. Ein Nachweis des Gegenteils ist nicht immer leicht zu führen.

Asynchrone Signale werden seltener als Grundlage eines Sprachmittels verwendet – die in mehrfacher Hinsicht ungewöhnliche objekt-orientierte Programmiersprache Actor von Tomlinson ist eine Ausnahme.

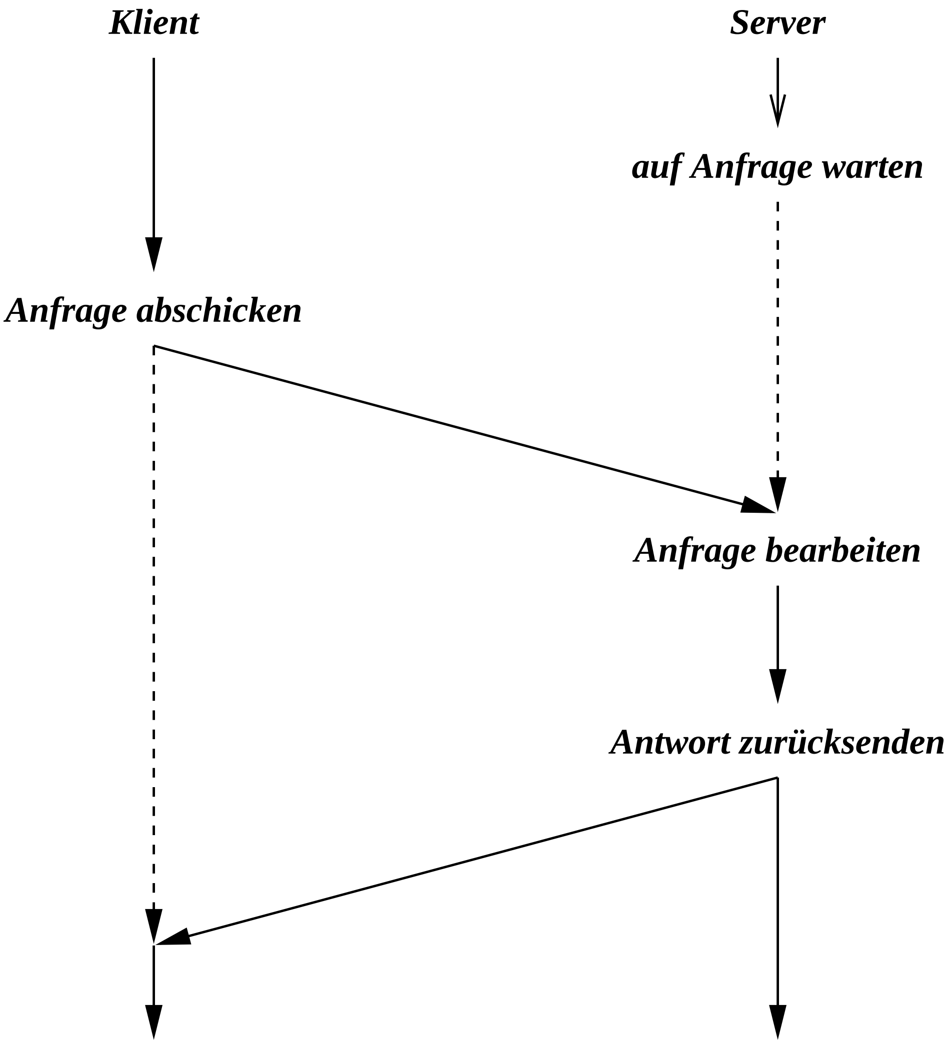

Abbildung 7.2: Synchrone Kommunikation

7.2.3 Synchrone Kommunikation

Die synchrone Kommunikation ähnelt zunächst weitgehend einem Prozeduraufruf, bei dem der Klient (oder Aufrufer) solange blockiert wird, bis der Server (oder Aufgerufene) die Anfrage entgegen genommen hat, sie bearbeitet hat und die Antwort des Servers bei dem Klienten wieder angekommen ist, so daß mit der Antwort weitergearbeitet werden kann.

Anders als bei einem Prozeduraufruf existiert der Server vorher schon und er hat explizit erklärt, daß er bestimmte Anfragen akzeptiert. So kann ein synchron arbeitender Server beispielsweise eine Ressource verwalten, die für mehrere Parteien zugänglich sein soll. Da er sich jeweils nur mit einer Anfrage beschäftigen kann, ist das Problem des gegenseitigen Ausschlusses gelöst.

Vorteilhaft bei der synchronen Kommunikation ist nicht nur die bequeme Möglichkeit, eine Rückantwort zu erhalten, sondern auch die Sicherheit des Servers, daß eine Bedingung, die zum Absendezeitpunkt der Anfrage zutraf, zumindest nicht von dem Klienten modifiziert worden ist. Somit kann synchrone Kommunikation den Nachweis der Korrektheit erleichtern.

Nachteilig kann der Zeitverlust auf der Seite des Klienten sein, wenn die Antwort uninteressant ist oder wenn der Klient während der Wartezeit andere sinnvolle Dinge erledigen könnte. Wie Abbildung 7.3 zeigt, kann diese Problematik durch das Einschalten eines Agenten gelöst werden, der anstelle des Klienten wartet. Abgesehen von dem zusätzlichen Aufwand für den Agenten und die vermehrte Kommunikation ist dies dann äquivalent zu asynchronen Signalen.

Ein erster, wenngleich noch sehr unvollkommener, Ansatz zur synchronen Kommunikation waren die Monitore von Hoare. Später entwickelte Hoare CSP (*Communicating Sequential Processes*), welches in eleganter Weise Kommunikation und Synchronisierung miteinander verbindet. CSP motivierte das Ada-Team zur Entwicklung von Rendezvous, die dem vorgestellten Mechanismus genau entsprechen.

7.3 Sprachmittel zur Synchronisierung

7.3.1 Kriterien von Bloom

Auf der Grundlage der vorgestellten Synchronisierungsmechanismen gibt es unzählige Sprachmittel auf Basis von Programmiersprachen, Bibliotheken oder Betriebssystemen. Um diese miteinander vergleichen zu können veröffentlichte Bloom 1979 einen Kriterienkatalog, der drei Bereiche betrachtet:

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.

Ergänzend könnten noch Prüfbarkeit (nicht nur theoretisch, sondern auch unter Betrachtung des Aufwands) und Erweiterbarkeit genannt werden. Ein sehr wichtiger Punkt ist dann noch die Frage, ob eine Synchronisierung nur mit anderen parallel ausführenden Einheiten des gleichen Programms oder auch mit externen Parteien möglich ist.

Zweifelos können durch Bibliotheken realisierte Synchronisierungsmechanismen flexibler und leichter erweiterbar sein, wenngleich dies möglicherweise Effizienzeinbußen mit sich bringen kann. Im Vergleich zu einer Lösung der Programmiersprache haben sie eher die Möglichkeit, die Synchronisierung mit externen Parteien zu ermöglichen und ggf. Mechanismen des Betriebssystems einzubinden.

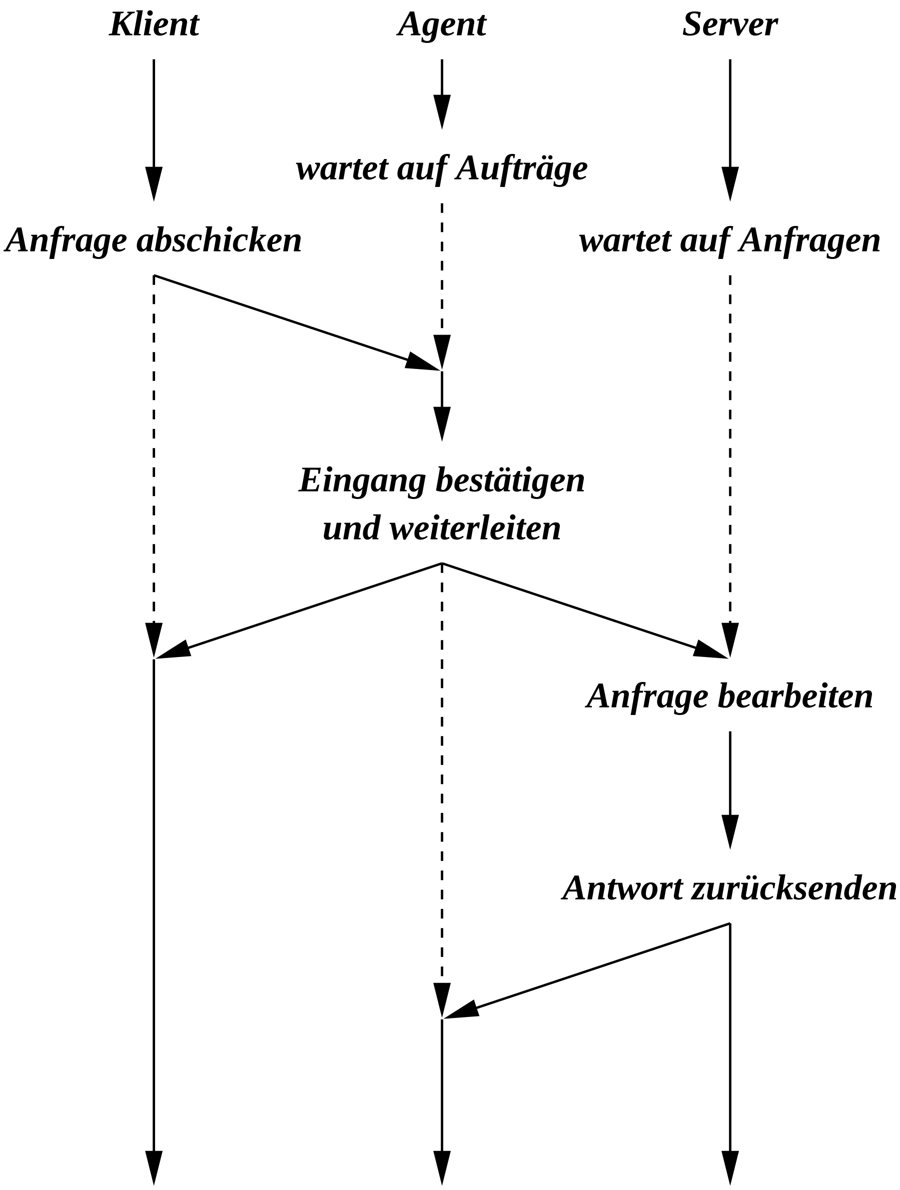

Abbildung 7.3: Asynchrone Signale auf Basis synchroner Kommunikation

7.3.2 Bedingungen

Bedingungen sind ein im Rahmen der Ulmer Oberon-Bibliothek entwickeltes Sprachmittel, das eine gewisse Nähe zu asynchronen Signalen hat. Jede konkrete Bedingung bezieht sich auf ein bestimmtes Ereignis[4], das möglicherweise schon eingetreten ist oder in der Zukunft möglicherweise zu erwarten ist. Jederzeit ist es möglich, eine Bedingung zu befragen, ob das Ereignis bereits eingetreten ist oder nicht. Dabei ist allerdings unter Umständen eine Zeitverzögerung zu berücksichtigen, d.h. daß eine Bedingung erst einige Zeit später wahr wird, nachdem das zugehörige Ereignis eingetreten ist. Wenn ein Ereignis eintritt, können damit verbundene Informationen den Weg zu der sich darauf beziehenden Bedingung nehmen, so daß auf dieser Basis eine asynchrone Kommunikation möglich ist.

Im Unterschied zu dem vorgestellten Modell der asynchronen Kommunikation findet kein explizites Versenden eines Signals oder einer Anfrage statt. Wenn der Empfänger eine Bedingung kreiert, führt dies implizit zu dem Versenden des Signals, wo immer das Ereignis auch eintritt. Damit stellt sich auch nicht die Frage, was passiert, wenn ein Signal abgesendet wird, ohne daß der Empfänger darauf vorbereitet ist.

Für Bedingungen gibt es in der Ulmer Oberon-Bibliothek die Abstraktion *Conditions* und zahlreiche Implementierungen, die jederzeit ergänzt werden können. Der Umgang entspricht etwa folgendem Muster:

- Zunächst kann eine Partei für sich konkrete Bedingungen kreieren. Wichtig ist dabei, daß jede Partei ihre eigenen Bedingungen kreiert – selbst wenn mehrere Parteien an dem gleichen Ereignis interessiert sind.

  ```
  VAR xxxcond, yyycond: Conditions.Condition;
  (* ... *)
  XXXConditions.Create(xxxcond);
  YYYConditions.Create(xxxcond);
  ```

- Jederzeit ist es möglich, den aktuellen Wert einer Bedingung abzufragen:

  ```
  IF Conditions.Test(xxxcond, errors) THEN
     (* das Ereignis ist eingetreten oder
        wird moeglicherweise nie eintreten (errors ueberpruefen!)
     *)
  ELSE
     (* das Ereignis ist noch nicht eingetreten *)
  END;
  ```

- Unter Verwendung eines Schedulers (in der Ulmer Oberon-Bibliothek über *Tasks* erreichbar, kann die aktuelle Task (entspricht einer Koroutine, mehr dazu etwas später) bis zum Eintreffen eines Ereignisses (oder dem Feststellen, daß das Ereignis nicht eintreffen kann) suspendiert werden:

[4] Ereignisse sind hier nicht im eingeschränkten Sinne des Bibliotheksmoduls *Events* zu verstehen.

```
Tasks.WaitFor(xxxcond);
```

- Es ist auch möglich, auf das Eintreffen eines Ereignisses aus einer Menge von Ereignissen zu warten:

  ```
  VAR condset: Conditions.ConditionSet;
  (* ... *)
  Conditions.CreateSet(condset);
  Conditions.Incl(condset, xxxcond);
  Conditions.Incl(condset, yyycond);
  Tasks.WaitForOneOf(condset);
  ```

  Statt *Tasks.WaitForOneOf* kann auch *Tasks.Select* verwendet werden, wenn die Menge der wahr gewordenen Bedingungen von Interesse ist.

Zwar sind die Operationen für Bedingungen einheitlich, aber das Umgangsschema mit Bedingungen kann durchaus unterschiedlich sein. Es gibt dabei vier Grundvarianten:

Globale Ereignisse sind für alle Parteien jederzeit abfragbar. Insbesondere ist es auch möglich *nach* dem Eintreten des Ereignisses noch die zugehörige Bedingunge zu kreieren und zu erfahren, daß es stattgefunden hat. Ein Beispiel hierfür sind *TimeConditions*, die das Eintreten eines bestimmten Zeitpunkts bei einer Uhr als Ereignis betrachten.

lokale Ereignisse haben genau zwei Parteien, bei denen eine Partei das Ereignis geschehen läßt, an dem die andere Partei interessiert ist. Ein Beispiel hierfür sind *StreamConditions* und typischerweise wartet die eine Partei darauf, etwas lesen zu können, das die andere Partei schreibt.

Gruppenereignisse können für beliebig viele Parteien interessant sein. Um das Eintreten des Ereignisses mitzubekommen, ist es notwendig, sich (implizit durch das Kreieren der entsprechenden Bedingung) zu registrieren. Typischerweise ist es dann auch noch notwendig, den Empfang der Information über ein eingetretenes Ereignis zu quittieren (damit es nicht mehr länger aufbewahrt werden muß) und explizit die Registrierung zurückzunehmen, wenn kein Interesse mehr vorliegt (damit die Informationen üeber die Ereignisse sich nicht ungenutzt aufsammeln). Typischerweise werden Gruppenereignisse zur Kommunikation einer Gruppe von Tasks untereinander verwendet. In der Bibliothek bietet sich dafür *EventConditions* an.

Konkurrierende Ereignisse treten bei einer Synchronisierung zum gegenseitigen Ausschluß für den Zugriff auf eine Ressource auf. Hier gibt es beliebig viele Parteien, die an einem Zugriff interessiert sind, und das ersehnte Ereignis ist, daß die eigene Partei den Zugriff erhält. Das klassische Beispiel dafür sind *Semaphores*.

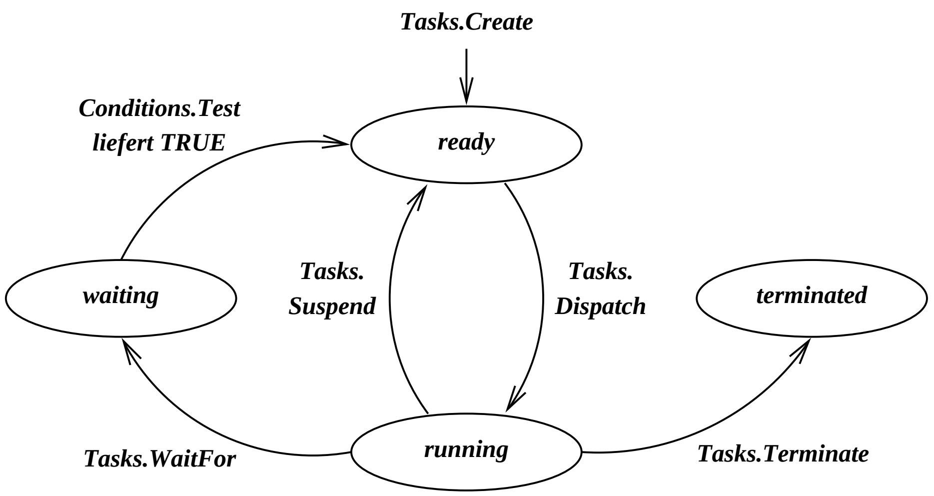

Abbildung 7.4: Zustandsübergänge für Tasks

7.3.3 Scheduling

Während für das Scheduling von Threads und Prozessen das Betriebssystem zuständig ist, muß dies bei Koroutinen von der Programmiersprache oder der Bibliothek gelöst werden. In der Ulmer Oberon-Bibliothek ist die Schnittstelle hierfür *Tasks*, wobei es möglich ist, selbst eigene Scheduler zu definieren und sogar mehrere gleichzeitig zu verwenden in Form einer hierarchischen Struktur. Im Normalfall dürften allerdings die Voreinstellungen reichen, die einen Scheduler auf Basis des Round-Robin-Schemas vorsehen[5]

Da Koroutinen nicht durch die Bibliothek verwaltet werden, gibt es für den Scheduler handhabbare konkurrierende Einheiten den Datentyp *Tasks.Task*. Normalerweise ist eine Task direkt einer Koroutine zugeordnet – aber dies muß nicht sein, da auch häufig Koroutinen kreiert werden, die den Schedulern nicht direkt bekannt sind.[6] Wenn eine Koroutine mit *Tasks.Create* zur Task erhoben und damit einem Scheduler bekannt gemacht wird, dann ist es nicht mehr notwendig, sich um die Aktivierung dieser Koroutine zu kümmern, da dies dann in der Verantwortung des Schedulers liegt.

Verbunden mit dem Leben einer Task sind folgende Grundoperationen, die jeweils zu Zustandsübergängen führen, die in Abbildung 7.4 dargestellt sind:

Tasks.Create kreiert eine Task, die zunächst mit der gegebenen Koroutine assoziiert ist, deren Aktivierung von nun an in der Verantwortung des Schedulers liegt. Ihr Zustand ist zunächst *ready*, d.h. sie ist rechenbereit, aber zur Zeit noch nicht aktiv.

Tasks.Dispatch wird von dem Scheduler aufgerufen, um eine Task zu aktivieren – sie ist anschließend im Zustand *running*.

Tasks.WaitFor erlaubt dann das Suspendieren einer Task, bis eine Bedingung wahr wird. Alternativ können auch *Tasks.WaitForOneOf* oder *Tasks.Select* verwendet werden. Eine Task ist dann im Zustand *waiting*.

[5] Realisiert durch die Module *Schedulers* und *RoundRobin*.
[6] Typischerweise indirekt durch Module wie *Iterators* oder *CrPipes*.

Conditions.Test ist die wichtigste Operation um später festzustellen, ob eine Task wieder aufgeweckt werden kann, da ihre Wartebedingung erfüllt ist.

Tasks.Suspend ermöglicht einen Zustandsübergang von *running* zu *ready*. Dies wird nur in Fällen rechenintensiver Tasks verwendet, um etwas Fairneß den anderen Tasks gegenüber zu üben.

Tasks.Terminate terminiert die aufrufende Task und führt sie in den Zustand *terminated*. Wichtig ist hier, daß eine Task nur sich selbst terminieren kann – eine zu dem UNIX-Systemaufruf *kill()* vergleichbare Operation ist nicht vorgesehen, da damit die Nachweisbarkeit eines korrekten Programmablaufs unglaublich erschwert wird.

7.3.4 Ein einfaches Beispiel mit globalen Bedingungen

Das folgende Programm[7] zeigt wie einfach es ist, eine Koroutine auf ein globales Ereignis warten zu lassen. Als globales Ereignis tritt hier das Eintreffen einer bestimmten Uhrzeit auf. Das Programm liest von der Eingabe eine Reihe von Zeiten (die relativ zur aktuellen Zeit in Sekunden angegeben werden) und zugehörige Texte, kreiert für jedes dieser Paare eine Koroutine, die dann darauf wartet, daß diese Zeit eintritt, um dann den entsprechenden Text auszugeben.

Durch die Eingabe eines Buchstaben wird die Einleseschleife und damit die "Haupt-Koroutine" beendet. Damit endet aber noch nicht das Programm, da hierfür erst alle dem Scheduler bekannten Koroutinen terminieren müssen.

```
MODULE Sleep;

   IMPORT Clocks, Conditions, Coroutines, Read, Streams, SYS := SYSTEM,
      Tasks, TimeConditions, Times, Write;

   PROCEDURE PrintMessage(VAR cr: Coroutines.Coroutine;
                          seconds: INTEGER;
                          message: ARRAY OF CHAR);
      (* create a coroutine which prints 'message' after 'seconds' seconds *)
      VAR
         timeCond: Conditions.Condition;
         time: Times.Time;
   BEGIN
      SYS.CRSPAWN(cr);
      Times.CreateAndSet(time, Times.relative, 0, seconds, 0);
      TimeConditions.Create(timeCond, Clocks.system, time);
      Tasks.WaitFor(timeCond);
      Write.LineS(Streams.stderr, message);
      Tasks.Terminate;
   END PrintMessage;

   PROCEDURE Cmds;
      VAR
         seconds: INTEGER;
```

[7] Im Web zu finden unter http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Sleep/

```
        message: ARRAY 80 OF CHAR;
        cr: Coroutines.Coroutine; task: Tasks.Task;
  BEGIN
     LOOP
        Write.String(": ");
        Read.Int(seconds);
        IF Streams.stdin.count = 0 THEN EXIT END;
        Read.Line(message);
        PrintMessage(cr, seconds, message); Tasks.Create(task, cr);
     END;
     Write.Line("Main loop terminates...");
  END Cmds;

BEGIN
  Cmds;
END Sleep.
```

So könnte dann ein beispielhafter Verlauf aussehen:

```
belinda$  tsleep
:  20 Hi Adi
:  10 etwas schneller
:  25 Schluss!
:  x
Main loop terminates...
 etwas schneller
 Hi Adi
 Schluss!
belinda$
```

7.3.5 Rendezvous

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. 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.

Wie der Name bereits nahelegt, findet ein Rendezvous immer zwischen genau zwei Tasks statt. 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.

Zusätzlich 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 demonstrieren:[8]

```
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.

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:[9]

[8] Dieses Beispiel wurde dem Ada-Standard entnommen.

[9] Das vollständige Beispiel ist im Web unter http://www.mathematik.uni-ulm.de/sai/ws97/oolib/zb/Rendezvous/ zu finden.

```
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.[10]

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:

[10] 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.

```
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 =
      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:[11]

```
PROCEDURE Create(VAR ring: RingBuffer);
BEGIN
   NEW(ring); Rendezvous.Initiate(Buffering, ring.task);
END Create;
```

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:

[11] Ein Ausweg ergibt sich in Ada, wenn man auf in rekursive Prozeduren eingebettete Tasks zurückgreift.

```
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);
   VAR
      ok: BOOLEAN;
BEGIN
   ok := Disciplines.Seek(Tasks.Current(), taskdID, taskd); ASSERT(ok);
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;
      ok: BOOLEAN;
BEGIN
   ok := Disciplines.Seek(task, taskdID, taskd); ASSERT(ok);
   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:

```
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;
      taskd: TaskDiscipline;
      entry: Entry;
BEGIN
   CallTaskBody(cr, body);
   Tasks.Create(task, cr);
   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;
```

7.4 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.[12]

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.[13] In der Ulmer Oberon-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.

[12] 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).

[13] 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.

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.[14] 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.

[14] So würden alle gängigen UNIX-Shells sofort auf Ende der Eingabe erkennen und die Sitzung beenden.

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.

7.5 Netzwerkdämonen

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.

Auf dem Internet könnte eine Sitzung dann etwa so ablaufen:

`oberon$ telnet belinda 4711`	*Kontaktaufnahme*
`Trying 134.60.66.21...`	
`Connected to belinda.`	
`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.`	
`oberon$`	

Die Schnittstelle eines entsprechenden Moduls könnte folgendermaßen aussehen:

```
DEFINITION ChatDaemons;

   IMPORT Networks, RelatedEvents, Services;

   TYPE
      ChatDaemon = POINTER TO ChatDaemonRec;
      ChatDaemonRec = RECORD (Services.ObjectRec) END;

   PROCEDURE Create(VAR daemon: ChatDaemon; port: Networks.Address;
                    errors: RelatedEvents.Object) : BOOLEAN;
   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.

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, die im Rahmen eines Konferenz-Dämons erzeugt werden, interessieren sich für die Terminierung des Objekts, das den Dämonen repräsentiert (*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
         (Services.ObjectRec)
         broadcast: Events.EventType;
         termination: Events.EventType;
      END;
VAR
   type: Services.Type;

TYPE
   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.";
      shutdown = "Der Daemon 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 *)
      lineterm: StreamDisciplines.LineTerminator; (* salami *)

   PROCEDURE Terminate;
      (* clean up and terminate *)
   BEGIN
      EventConditions.Drop(broadcastMsgAvailable);
      EventConditions.Drop(terminationMsgAvailable);
      Streams.Release(s);
      Tasks.Terminate;
   END Terminate;

BEGIN
   SYSTEM.CRSPAWN(cr);
   lineterm[0] := ASCII.cr; lineterm[1] := ASCII.nl; lineterm[2] := 0X;
   StreamDisciplines.SetLineTerm(s, lineterm);
```

```
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) &
         (event(Resources.Event).change = Resources.terminated) THEN
      Write.LineS(s, shutdown);
      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 *Tasks* übergeben wird:

```
PROCEDURE ConnectionManager(VAR cr: Coroutines.Coroutine;
                            daemon: ChatDaemon; port: Networks.Address;
                            errors: RelatedEvents.Object;
                            VAR ok: BOOLEAN);
   VAR
      newConnection: Conditions.Condition;
      terminationMsgAvailable: Conditions.Condition;
      conditions: Conditions.ConditionSet;
      socket: Networks.Socket; (* registered port *)
      s: Streams.Stream; (* new communication channel *)
      event: Events.Event;
      dialogCR: Coroutines.Coroutine;
      task: Tasks.Task;
BEGIN
   (* register port address so that clients may connect to *)
   IF ~Networks.Listen(socket, port, errors) THEN
      ok := FALSE; RETURN
   END;

   SYSTEM.CRSPAWN(cr);

   (* 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) &
            (event(Resources.Event).change = Resources.terminated) THEN
         EXIT
      END;
      WHILE Conditions.Test(newConnection, errors) &
            ~RelatedEvents.EventsPending(errors) &
            Networks.Accept(socket, s, Streams.linebuf) DO
         Dialog(dialogCR, daemon, s); Tasks.Create(task, dialogCR);
         RelatedEvents.Forward(s, daemon);
      END;
   END;
   EventConditions.Drop(terminationMsgAvailable);
```

```
   Networks.Release(socket);
   Tasks.Terminate;
END ConnectionManager;
```

Create braucht dann nur *ConnectionManager* zu starten:

```
PROCEDURE Create(VAR daemon: ChatDaemon; port: Networks.Address;
                 errors: RelatedEvents.Object) : BOOLEAN;
   VAR
      cm: Coroutines.Coroutine;
      task: Tasks.Task;
      ok: BOOLEAN;
BEGIN
   NEW(daemon);
   Services.Init(daemon, type);
   Events.Define(daemon.broadcast); Events.Ignore(daemon.broadcast);
   Resources.TakeInterest(daemon, daemon.termination);
   ConnectionManager(cm, daemon, port, errors, ok);
   IF ok THEN
      Tasks.Create(task, cm);
      RelatedEvents.QueueEvents(daemon);
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END Create;
```

Terminate über *Resources* das den Dämonen repräsentierende Objekt. Da alle zugehörigen Tasks auf dieses Ereignis warten und bei Eintreffen sofort *Tasks.Terminate* ausführen, kommt es zur Terminierung aller für diesen Dämonen kreierten Tasks.

```
PROCEDURE Terminate(daemon: ChatDaemon);
BEGIN
   Resources.Notify(daemon, Resources.terminated);
END Terminate;
```

7.6 Zusammenfassung

Je umfangreicher Software-Systeme werden, desto entscheidender ist die Notwendigkeit, sie als Netzwerk parallel agierender Agenten modellieren zu können. In einem objekt-orientierten Umfeld sollte dabei insbesondere die Erzeugung konkurrierender Einheiten von der Synchronisierung unabhängig sein. Konsequenterweise sollte auch die Synchronisierung nicht nur innerhalb eines Adreßraumes möglich sein, sondern auch im Bereich verteilter Systeme. Dies läßt sich in vollem Umfang nur durch entsprechende Bibliotheken erreichen, da insbesondere die Synchronisierung mit externen Parteien offen und flexibel regelbar sein muß.

Dabei lohnt es sich wiederum, neue Mechanismen zunächst nur in einer denkbar einfachen und elementaren Form einzuführen. Auf diesen Grundbausteinen lassen sich dann die in der Praxis relevanten Techniken realisieren ohne daß damit eine Beschränkung auf einzelne Techniken verbunden ist. Gleichzeitig sollte auch der Programmieraufwand bei der Benutzung minimiert werden.

Diese Ziele wurden bei den Bedingungen in der Ulmer Oberon-Bibliothek angestrebt. Sie sind vollkommen offen, da jederzeit neue hinzu definiert werden können. Die zur Verfügungen stehen Operationen sind minimal sowohl in der Anzahl der Operationen als auch der Länge der Parameterlisten. Durch eine ausgefeilte Schnittstelle bei *Conditions* ist es möglich, sie mit größtmöglicher Effizienz auszuwerten und den eigenen Prozeß ohne Busy-Loop zu suspendieren, wenn nur auf das Eintreten externer Ereignisse gewartet wird. Diese Konzepte sind auch im Prinzip in jede andere objekt-orientierte Systembibliothek übertragbar.