Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 79 (1988)

Heft: 17

Artikel: Das Oberon-System

Autor: Gutknecht, J. / Wirth, N.

DOI: https://doi.org/10.5169/seals-904073

Das Oberon-System

J. Gutknecht und N. Wirth

Das Betriebssystem Oberon wurde von den Autoren an der ETH Zürich im Laufe der vergangenen zweieinhalb Jahre konzipiert und verwirklicht. Es handelt sich dabei um ein kompaktes Einprozess-Multitasking-System, welches die Einrichtungen und die Leistungsfähigkeit moderner Arbeitsplatzstationen soweit wie möglich ausschöpft. Das System ist in einer Sprache programmiert, die als eine Art objektorientierte Weiterentwicklung von Modula-2 aufgefasst werden kann.

Le système d'exploitation Oberon a été conçu et réalisé a l'Institut d'informatique de l'EPFZ par les auteurs au cours des deux années et demi passées. C'est un système compact de traitement multitâche-monoprocessus qui épuise le plus possible les équipements et la capacité des stations de postes de travail modernes. Le système est programmé dans un langage que l'on peut considérer comme un perfectionnement orienté objet de Modula-2.

Adresse der Autoren
Prof. Dr. *Jürg Gutknecht* und
Prof. Dr. *Niklaus Wirth*,
Institut für Informatik, ETH-Zentrum,
8092 Zürich.

Ende 1985 starteten die Autoren dieses Aufsatzes ein Projekt, dessen Zielsetzung die Entwicklung eines speziell auf persönliche Arbeitsplatzrechner zugeschnittenen Betriebssystems und einer Sprache zu seiner Programmierung war. Nach 30 Monaten intensiver Arbeit liegt das Resultat nun in Form eines äusserst flexiblen und zuverlässigen Werkzeuges, genannt *Oberon*, vor.

Das System ist auf dem vom zweitgenannten Autor und *H. Eberle* entwickelten Rechner *Ceres*[1] implementiert. Der Kern von Ceres ist ein NS-32032-Mikroprozessor. Als Peripheriegeräte dienen eine Tastatur, eine Maus und ein hochauflösender Bildschirm. Zusätzlich lässt sich ein Farbmonitor gleicher Auflösung anschliessen. Ceres-Stationen sind normalerweise in ein lokales Netz einbezogen und haben Zugang zu Dienstleistungen wie zentraler Massenspeicher, Laserdrucker und Elektronische Post.

Design-Prinzipien

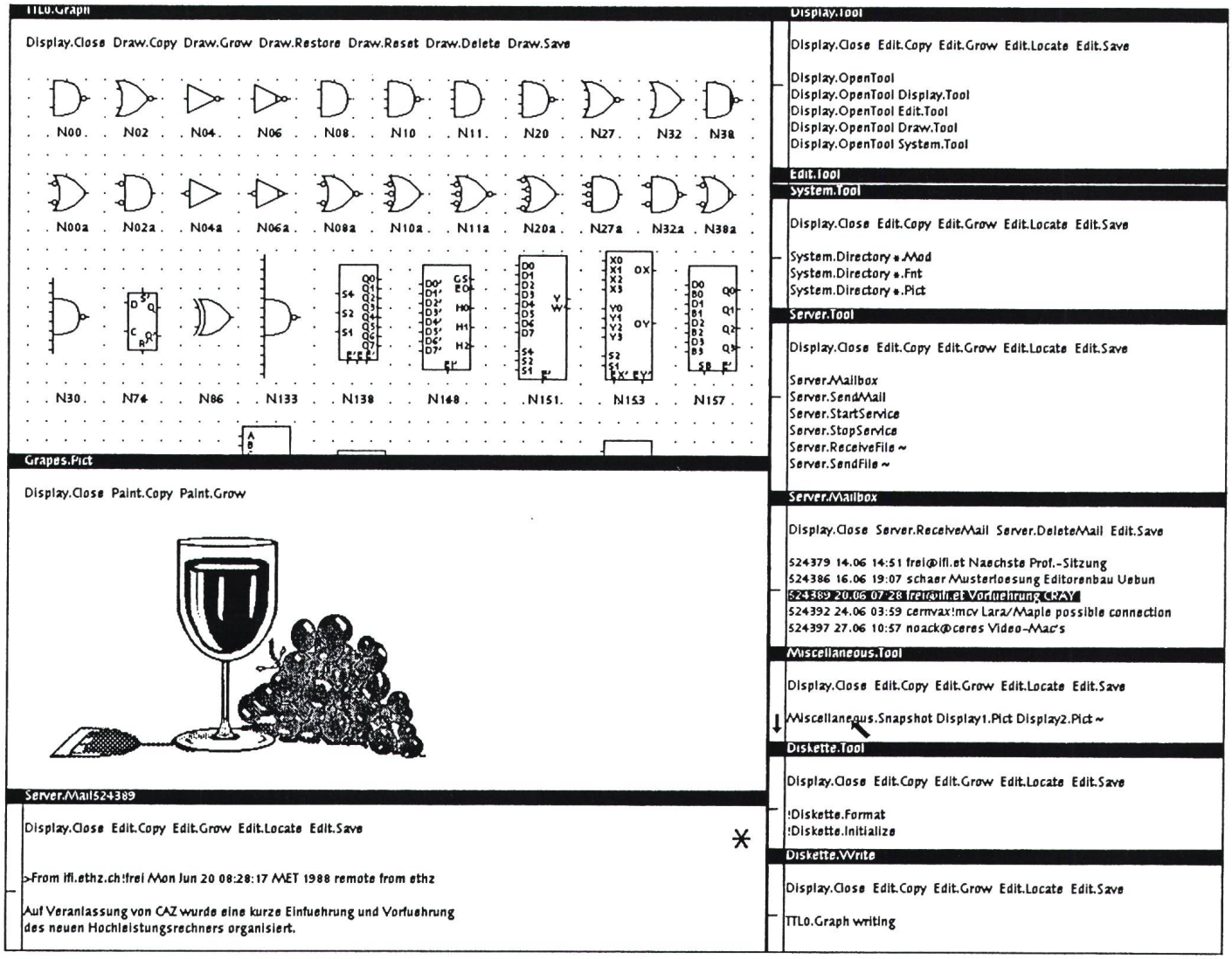
Zentrales Prinzip beim Design der Hardware und auch der Software war das Streben nach Klarheit und Einfachheit. Dies war nötig im Hinblick auf das winzige Team und den abgesteckten engen Zeitrahmen. Klarheit und Einfachheit sind aber ohnehin unerlässlich, wenn ein Produkt dem Anspruch nach Zuverlässigkeit genügen soll. Sie wird am besten durch eine reguläre und zweckgerichtete Struktur erreicht. Eine solche Struktur ist aber nur möglich, wenn das zugrundeliegende Modell gut verstanden, genügend einfach und frei von widersprüchlichen Vorgaben ist.

Modell für Oberon ist ein einzelner Prozess[1], der von einem einzigen Benutzer durch eine Folge von *Befehlen* gesteuert wird. Der Benutzer arbeitet typischerweise an mehreren *Aufgaben* (Tasks) gleichzeitig. Die Aufgaben manifestieren sich gewöhnlich in Form von Dokumenten, welche in *Bildschirmfenstern* dargestellt sind (Fig. 1). Weil sich aufeinanderfolgende Befehle auf verschiedene Aufgaben beziehen können, lässt sich Oberon als *Einprozess-Multitasking*-System kennzeichnen.

Dieses Modell steht im Gegensatz zu demjenigen konventioneller Multitasking-Systeme, in denen jede Aufgabe von einem einzigen Prozess getragen und ausgeführt wird und in denen die Steuerung prinzipiell an jeder beliebigen Programmstelle vom laufenden Prozess auf irgendeinen anderen umgeschaltet werden kann. Die Granularität (bezüglich der Länge der atomaren Ablaufseinheiten) ist in Oberon also viel gröber als in konventionellen Multitasking-Systemen, und die möglichen Umschaltestellen zwischen Aufgaben sind im Zeitpunkt der Programmierung wohlbekannt. Dies alles trägt entscheidend zur strukturellen Einfachheit des Systems bei. Aufwendige Mechanismen zur Speicherung und Wiederherstellung des Prozesszustandes sind ebenso unnötig wie Blockiermassnahmen zur vorübergehenden Reservation von Ressourcen durch einen einzelnen Prozess.

Während der Ausführung eines Befehls wird grundsätzlich kein Dialog mit dem Benutzer geführt. Befehle sind also *atomare* Aktionseinheiten in Oberon. Ihre Spezifikation erfolgt hauptsächlich durch Eingaben über die Maus oder die Tastatur. Besonders interessant ist die Strategie der Befehlsinterpretation. Jeder Befehl wird direkt demjenigen Bildschirmfenster zugeführt und zur individuellen Behandlung übergeben, in welchem sich der Mauszeiger *(Cursor)* oder die Einfügemarke *(Caret)* befindet.

[1] Sequenz von logisch zusammengehörigen Aktionen.



Figur 1 Beispiel einer Oberon-Bildschirmdarstellung

Die Benutzerspur (links) stellt typischerweise Dokumentenfenster, die Systemspur (rechts) vor allem Tools und Log-Fenster dar. So findet sich z.B. das Inhaltsverzeichnis der Mailbox in der Systemspur, während die abgerufene Meldung als Dokument in der Benutzerspur erscheint.

Vielleicht der wichtigste Effekt des geschilderten Ablaufschemas ist der, dass kein Befehl das Oberon-System in einem versteckten Zustand *(Modus)* hinterlässt. Würde die Ausführung eines Befehls einen Dialog zulassen, so würde eine Aufforderung wie z.B. «type file name>» das System in einen Modus versetzen, der dem Benutzer eine ganz bestimmte Aktion aufzwingt, nämlich Eintippen eines Filenamens. Wir glauben, dass Modi und die damit verbundene zustandsabhängige Behandlung von Eingaben wesentlich zur Benutzerunfreundlichkeit eines Systems beitragen.

Befehle übernehmen ihre Parameter aus dem globalen Zustand des Systems, meistens aus einem dargestellten Text. Vielleicht noch wichtiger ist aber die Tatsache, dass Befehle umgekehrt stets nichtflüchtige Ausgabedaten (typischerweise Texte, Grafiken oder Rasterbilder) in Form von Datenstrukturen erzeugen. Daraus lassen sich Bildschirmdarstellungen ableiten oder Kopien auf Disk herstellen. Allgemeiner kann eine erzeugte Datenstruktur als Eingabe für irgendeinen folgenden Befehl verwendet werden. Beispielsweise ist ein vom Befehl *System.Directory* erzeugter Auszug aus dem Inhaltsverzeichnis der Disk ein editierbarer Text und kann als Parameterliste für einen weiteren Befehl verwendet werden. Der Benutzer hat für die Vorbereitung der nächsten Tätigkeit also insbesondere immer die ganze Vielfalt der am Bildschirm dargestellten Daten zur Verfügung.

Die allgemeine Form eines Befehlsnamens ist M.P, wobei P der Name einer Prozedur und M der Name des P enthaltenden Moduls ist. Befehlsnamen können eingetippt oder, wenn sie schon auf dem Bildschirm sichtbar sind, einfach durch Mausklick angewählt werden. Nur einige wenige sehr häufig benutzte Befehle sind sozusagen eingebaut und werden nicht über ihren Namen erteilt, sondern durch Drücken geeigneter Tasten eingeleitet. Zum Beispiel bedeutet das Niederdrücken der mittleren Maustaste, dass

der durch den Cursor angezeigte Text als Befehlsname zu interpretieren ist. Diese Aktion dient also gewissermassen als Anker für die Eingabe eines allgemeinen, nicht fest eingebauten Befehls. Drücken und anschliessendes Loslassen der rechten Maustaste bewirkt, dass der vom Cursor überstrichene Text markiert und ausgewählt wird. Ein drittes Beispiel betrifft die Tastatur. Das Anschlagen einer Taste der Tastatur signalisiert, dass das betreffende Zeichen am Ort des Caret im Text eingefügt werden soll.

Es sei nochmals betont, dass die Interpretation einer Eingabe nicht ein für allemal vom Systemkern festgelegt ist, sondern individuell von dem mit dem betroffenen Bildschirmfenster verknüpften Befehlsinterpreter durchgeführt wird. Die im letzten Abschnitt besprochenen Einzelheiten der Interpretation beziehen sich auf die vordefinierte Klasse der sogenannten *Textfenster*. In *Grafikfenstern* könnte die Befehlseingabe beispielsweise über das Antippen von Symbolen erfolgen.

Das angesprochene allgemeine Schema der Befehlseingabe durch Anklicken eines Namens oder eines Symbols beeinflusst die Arbeitsweise mit dem Oberon-System tiefgreifend. So zeigt jedes Fenster ein *Menu*, d.h. eine Liste von Befehlen, welche sich automatisch auf das betreffende Fenster beziehen. Noch wichtiger aber ist, dass der Benutzer individuelle, kurze Texte oder Grafiken am Bildschirm zur Darstellung bringen kann, welche Listen von häufig benutzten Befehlen und Parametern enthalten (s. Fig. 1, rechte Spur). Wir nennen sie *Tools*. Man beachte, dass Tools normale editierbare Dokumente sind und jederzeit auf individuelle Bedürfnisse und Vorlieben abgestimmt werden können. Die geschilderte Arbeitsweise hat zur Folge, dass die meisten Aktivitäten ohne Eintippen von Text ausgeführt werden können.

Zusammenfassend lässt sich sagen, dass der Phantasie in der Ausschöpfung des dargelegten universellen Schemas zur dezentralen Befehlsinterpretation kaum Grenzen gesetzt sind.

Erweiterbarkeit

Ein anderes zentrales Thema im Design von Oberon waren Offenheit und Erweiterbarkeit. Konventionelle modulare Systeme [2] unterstützen zwar weitgehend die Erweiterung der *Funktionalität* des Systems, jedoch nicht die Erweiterung von Objekttypen. Vorausgesetzt es liege eine geeignete Modulbasis vor, kann beispielsweise jederzeit eine Kollektion neuer Editierfunktionen für Texte oder Grafiken in Form eines neuen Moduls hinzugefügt werden. Hingegen ist es nicht möglich, nachträglich neue Varianten bestehender Objekte einzuführen.

Angenommen, ein bestehendes Grafikpaket stelle einen Datentyp zur Beschreibung von Linien und Symbolen samt Funktionen zur Behandlung solcher Objekte zur Verfügung. Dann mag es wünschbar sein, dieses Paket als Basis für einen Editor zur Konstruktion elektronischer Schaltungen zu verwenden. Die neuen Objekte sind Verbindungen, Widerstände, Transistoren, Dioden und Tore, also Varianten von Linien und Symbolen mit einer zusätzlichen Semantik. Leider gibt es in konventionellen modularen Sprachen keine Möglichkeit, diese Varianten zu definieren.

Die Entwicklung der *Sprache* Oberon [3] war grösstenteils durch den Wunsch nach Erweiterbarkeit von Datentypen im skizzierten Sinne motiviert. Das gewählte Schema ist ähnlich zu dem der Klassen und Unterklassen in objektorientierten Sprachen. Ein ganz entscheidendes Merkmal von Oberon ist jedoch die Verbindung dieses Schemas mit einem rigorosen Typenprüfsystem. Wir werden auf die Bedeutung eines zuverlässigen Typensystems zurückkommen.

Dank der wirkungsvollen Erweiterbarkeit des Oberon-Systems und der umfassenden Sammlung von Schnittstellen zum Kernsystem ist die Grenze zwischen Benutzern und Programmierern unscharf. Tatsächlich werden Benutzer leicht zu Programmierern, sobald sie entdecken, dass eine zusätzliche Programmfunktion ihre Aufgabe wesentlich erleichtern würde. Wir sind überzeugt davon, dass nur durch eine solche Betrachtungsweise das Potential, welches inhärent im Konzept der Software steckt, vollständig ausgeschöpft werden kann.

Systemstruktur

Das Oberon-System ist eine hierarchische Sammlung von Modulen (Fig. 2). Im wesentlichen gibt es keinen Unterschied zwischen Betriebssystemmodulen und solchen, die ein Programmierer später hinzugefügt hat. Man versteht die Systemstruktur vielleicht am besten, wenn man den Ablauf nach dem Einschalten des Computers verfolgt.

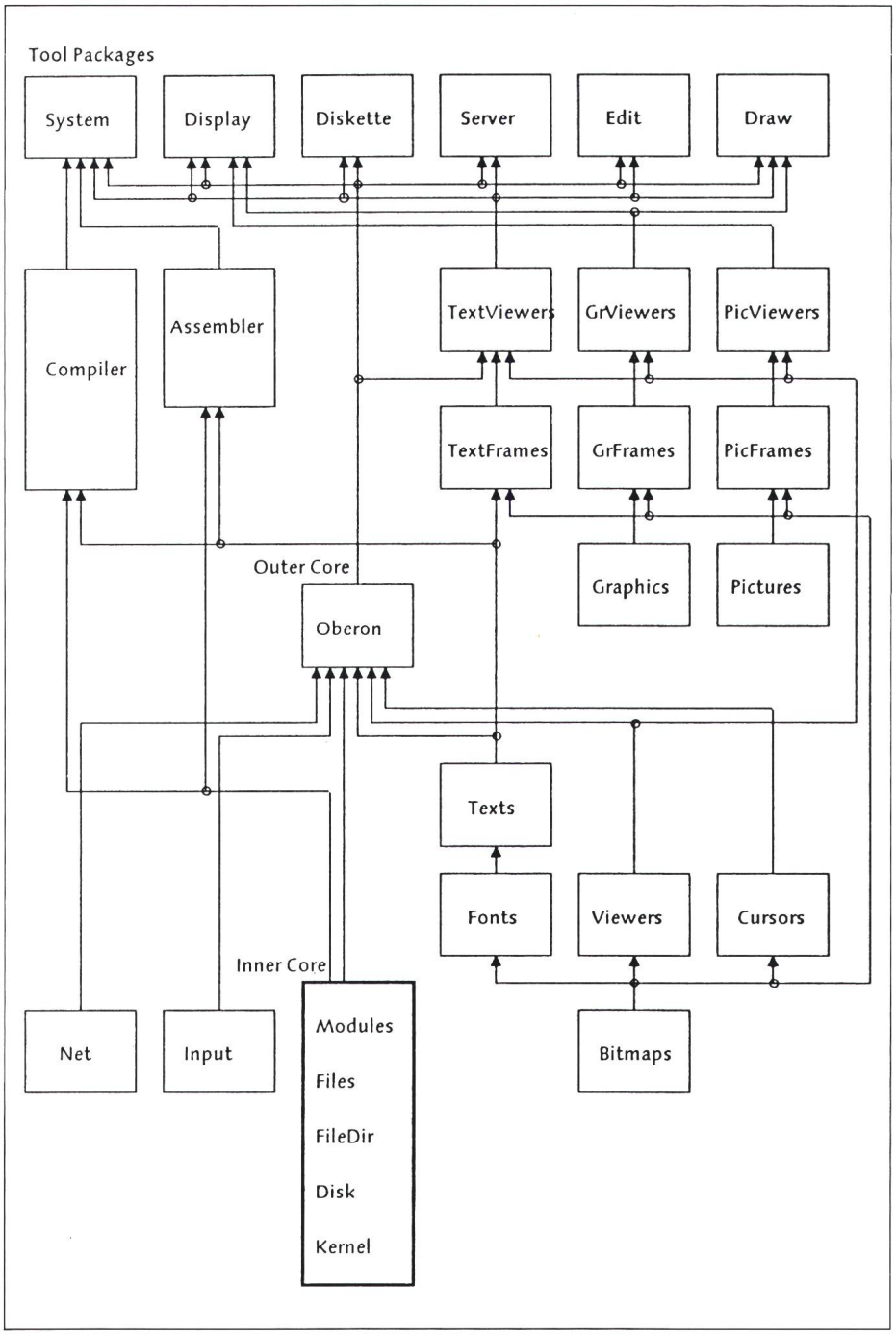
Zuerst wird die Steuerung an den *Boot Loader* übergeben. Dieses kurze Programm residiert im nichtflüchtigen Lesespeicher (ROM) des Computers. Es liest das *Boot File* von der eingebauten Hard-Disk. Das Boot File umfasst den *Inneren Kern* des Oberon-Systems, d.h. die Module *Kernel, Disk, FileDir, Files, Modules*. Kernel enthält alle Funktionen, welche entweder Gebrauch von privilegierten Instruktionen machen oder zu geschützten Daten zugreifen, nämlich Routinen für die Verwaltung des virtuellen und des realen Speichers und für die Reservation von Disksektoren. Disk ist der Driver für die Disk, FileDir verwaltet das Inhaltsverzeichnis der Disk und Files stellt Operationen auf Files zur Verfügung. Modules lädt Module in Objektform von der Disk in den Speicher und bringt sie zur Ausführung.

Anschliessend wird der *Äussere Kern* geladen, d.h. das Modul *Oberon* samt allen importierten Modulen. Schliesslich verlangt die Initialisierung des Moduls Oberon das Laden des Moduls *Display*, wiederum mit allen importierten Modulen. Dabei wird das *Display Tool*-Fenster geöffnet. Es enthält Einträge zur Eröffnung weiterer Tools und kann deshalb als Wurzel in der Hierarchie der Tool-Fenster betrachtet werden. Damit ist der Systemstart abgeschlossen und die Steuerung wird der *Zentralschleife* im Modul Oberon übergeben. Diese überwacht fortwährend alle Eingabegeräte und leitet Eingaben direkt an die zuständigen Befehlsinterpreter weiter.

Verfolgen wir nun die Ereignisse, nachdem eine Eingabe, z.B. ein Klick der mittleren Maustaste, festgestellt wurde. Zunächst wird das Fensterverwaltungsmodul *Viewers* zwecks Identifikation des durch die Maus angezeigten Fensters aufgerufen. Dann wird die Eingabe dem diesem Fenster zugehörigen Befehlsinterpreter gemeldet. Es ist erwähnenswert, dass die Meldung einer Eingabe an den Befehlsinterpreter eines Fensters sich von einem gewöhnlichen Prozeduraufruf geringfügig unterscheidet. Der Hauptunterschied besteht darin, dass die aufgerufene Prozedur als *Variable* im Fensterobjekt installiert und somit ihre genaue Identität (Implementation) dem Aufrufenden nicht bekannt ist. Damit erhält der Aufruf den Charakter «Reagiere auf die als Parameter übergebene Meldung in irgendeiner sinnvollen Art und Weise». Dies ist genau das Paradigma der *objektorientierten* Programmierung.

Wenn wir nun annehmen, dass das angesprochene Fenster ein Textfenster ist, so wird also der Befehlsinterpreter der Textfenster aktiviert. Er erhält die Meldung «mittlere Maustaste gedrückt» und reagiert, indem er mit Hilfe der Basismodule *Texts* und *TextFrames* das Wort bei der Cursorposition identifiziert, dieses als Befehlsname M.P interpretiert, das Modul M lädt (falls es nicht schon geladen ist) und die Prozedur P aufruft. Nach deren Beendigung fällt die Steuerung wieder an die Zentralschleife zurück, und zwar unabhängig davon, ob P normal oder abnormal *(Exception)* beendet wurde.

In Figur 2 lassen sich drei ähnliche Triplets von Modulen erkennen: (Texts, TextFrames, TextViewers), (Graphics, GraphicFrames, GraphicViewers) und (Pictures, PictureFrames, Picture Viewers). Sie zeigen eine typische modulare Aufteilung eines thematischen Komplexes. Die Grundmodule *Texts, Graphics* und *Pictures* verwalten die Texte, Grafiken und Rasterbilder in Form von Datenstrukturen und stellen die Grundoperationen für diese Objektklassen zur Verfügung. Die *Frame*-Module behandeln die Darstellung der Objekte in rechteckigen Feldern (Frames) in Bildschirmfenstern. Gewöhnlich zerfällt ein Fenster in genau zwei Frames, ein Menuframe und ein Hauptframe. Die *Viewer*-Module schliesslich steuern den zur Fensterklasse gehörigen Befehlsinterpreter bei. Aus programmtechnischer Sicht sind sowohl die Objekttypen Viewer als auch TextFrame, GraphicFrame und PictureFrame erweiterte Varianten des Grundtyps Frame. Dies ist ein gutes Beispiel für die Anwendung des Typenerweiterungsmechanismus der Sprache Oberon.



Figur 2 Struktur des Oberon-Systems

Man beachte die Schalenstruktur, die durch den inneren und äusseren Kern sowie durch die Tool Packages gebildet wird. Das Modul Oberon enthält die Zentralschlaufe zur dynamischen Steuerung des Ablaufs sowie die Parameter der Systemkonfiguration.

Ressourcenverwaltung

Wir wissen bereits, dass Oberon den Programmbegriff im konventionellen Sinne nicht kennt. Damit fehlen diejenigen Instanzen im System, welche üblicherweise verantwortlich sind für die Anforderung und Rückgabe von Speicherressourcen, die für die Erfüllung eines bestimmten Auftrages benötigt werden. Die Befehlsprozeduren können diese Rolle natürlich nicht übernehmen, da, wie wir festgestellt haben, Hauptspeicherdaten als wesentliche Verbindung zwischen aufeinanderfolgenden Befehlen wirken.

Die Verwaltung des Hauptspeichers ist in Oberon einem systemweit agierenden Mechanismus, einem sogenannten *Garbage Collector*, übertragen. Er durchläuft periodisch den Speicher und markiert dabei sämtliche noch benötigten Blöcke[2]. Die nichtmarkierten Speicherblöcke gibt er anschliessend zur Wiederbenutzung frei. Die Korrektheit und Zuverlässigkeit dieses Mechanismus hängt entscheidend von der intakten Typeninformation in den einzelnen Speicherblöcken ab. Deshalb ist das rigorose Typensy-

[2] Nach dem Prinzip der Zugreifbarkeit, ausgehend von globalen Wurzeln.

stem der Sprache Oberon, das auch in der erschwerten Situation von Typenerweiterungen wirksam ist, nicht bloss Luxus, sondern Notwendigkeit.

Nicht der ganze Hauptspeicher wird durch den Garbage Collector verwaltet. Der Modulspeicher unterliegt einer eigenen Verwaltung. Module werden in Oberon *verzögert* geladen, d.h. wenn sie das erste Mal aufgerufen werden. Diese Methode steht im Gegensatz zum Ladevorgang in üblichen Systemen, bei welchem mit einem Modul alle direkt und indirekt importierten Module unmittelbar mitgeladen werden. Das verzögerte Laden ist dann besonders gewinnbringend, wenn von einem statisch grossen Programmpaket bei normaler Anwendung nur ein kleiner Teil der beteiligten Module tatsächlich benötigt wird. Also etwa wenn ein Dokumentenverarbeitungssystem, das normalerweise nur für das Editieren von gewöhnlichem Text verwendet wird, Teile zur Bearbeitung von Formeln und Tabellen enthält[3].

Neben dem Hauptspeicher ist der *Plattenspeicher* eine wichtige Ressource des Oberon-Systems. Er ist in *Files* gegliedert, die als Bytefolgen zu verstehen sind. Ein wichtiges Merkmal des Oberon-File-Systems ist die konzeptuelle Trennung von File und Zugriffsmechanismus. Jeder Schreib- oder Lesezugriff zu einem File wird von einem sogenannten *Rider* geleitet. Für ein gegebenes File können mehrere Riders gleichzeitig geöffnet sein, beispielsweise einer pro aktive Aufgabe. Files können *benannt* oder *anonym* sein. Anonyme Files existieren nur bis zum nächsten Systemstart, der deshalb als eine Art Garbage Collector auf Disk-Stufe betrachtet werden kann. Anonyme Files werden hauptsächlich zur vorübergehenden Speicherung von eingegebenem oder während der Ausführung von Befehlen automatisch erzeugtem Text verwendet.

[3] Zur Realisierung des verzögerten Ladens wird Gebrauch von der virtuellen Adressierung gemacht.

Schlusswort

Das Design sowohl der Sprache als auch des Betriebssystems Oberon war vom Bestreben begleitet, den Computerbenutzer von künstlichen Einschränkungen, wie sie in traditionellen Systemen häufig vorhanden sind, zu befreien. Dies erforderte Mut zum Experiment, der durch die überwältigende Leistungsfähigkeit moderner Hardware deutlich gestärkt wurde. Leider wird diese Leistungsfähigkeit bei der Konstruktion von konventionellen Systemen oft als Einladung zum unbeschränkten Vergrössern missverstanden. Wir glauben, dass sie tatsächlich viel eher eine Herausforderung zu innovativem Gebrauch darstellt.

Oberon unterscheidet sich in seinem Umfang um eine Grössenordnung von vergleichbaren Systemen. Samt Compiler, Text- und Grafikeditor, Hilfsmittel für die Benutzung von Disk und Diskette und Werkzeugen zum Zugriff auf Dienstleistungen wie Laserdrucker und Elektronische Post besteht das System aus etwa 15 000 Zeilen Quellencode. Die Grösse in Maschinenform beträgt ungefähr 150 Kilobytes, und die Übersetzung dauert 5 Minuten. Sowohl in Quellen- als auch in Maschinenform findet das System leicht auf einer einzigen 3,5 Zoll-Diskette Platz!

Literatur

[1] *H. Eberle:* Entwicklung eines 32-bit-Arbeitsplatzrechners. Bull. SEV/VSE 78(1987)21, S. 1328...1332.

[2] *J. Gutknecht:* Modulare Programmierung mit Modula-2. Bull. SEV/VSE 78(1987)1, S. 8...14.

[3] *N. Wirth:* The programming language Oberon. Software - Practice and Experience 18(1988)7, S. 671...690.