DAS PROJEKT POW!
UND SEINE VISUALISIERUNGS-KOMPONENTEN

Dissertation zur Erlangung des akademischen Grades Doktor der Technischen Wissenschaften

eingereicht bei

o. Univ. Prof. Dr. Jörg R. Mühlbacher
Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM)
Johannes Kepler Universität Linz

von

Dipl.Ing. Peter René Dietmüller
Matr.-Nr. 87 55 422

Linz, im Juni 2000

Dank

Zum Gelingen dieser Arbeit haben viele Menschen beigetragen, denen ich auf diesen Weg herzlich danken möchte. An erster Stelle steht natürlich die Familie, die nicht müde wurde, mich immer wieder anzutreiben. Dafür gebührt ihnen ein besonderer Dank, denn manchmal stellte ich ihre Geduld arg auf die Probe.

Doch auch die Geduld von Prof. Mühlbacher wurde manchmal stark strapaziert. Dennoch hat er mich immer wieder angespornt und gerade zum Schluß mit dem nötigen Nachdruck ermuntert, die Arbeit fertigzustellen. Außerdem hat er in unzähligen Diskussionen immer wieder Ideen und neue Denkansätze eingebracht, für die ich ihm sehr dankbar bin. Ebenso möchte ich meinem Kollegen Rudolf Hörmanseder für viele interessante Diskussionen und neue Blickwinkel danken.

Auf keinen Fall dürfen all jene unerwähnt bleiben, die am Pow!-Projekt gearbeitet haben. Allen voran sind Bernhard Leisch und Ulrich Kreuzeder zu nennen, die von Anfang an dabei waren. Unsere Gespräche waren von großem Nutzen und manchmal auch nicht ganz ernsthaft, was der Stimmung im Team sehr förderlich war.

Zum Schluß möchte ich auch Markus Jöbstl und Wolfgang Zwicknagl danken, die im Visualisierungsprojekt unschätzbare Arbeit leisteten und ebenfalls viel zur guten Stimmung im Team beitrugen.

Hall of Fame

Das Pow!-Projekt erstreckte sich über mehr als sieben Jahre, in denen viele verschiedene Personen mitgearbeitet haben. Beta Tester waren involviert, Prototypen wurden entwickelt sowie Erweiterungsvorschläge und Anregungen eingebracht. Die folgende Liste ist ein Versuch, allen Mitwirkenden für ihre Bemühungen am Pow!-Projekt zu danken.

Aigner Christian
Angerer Ronald
Beitelmair Richard
Bogner Michael
Boninsegna Werner
Brian Kirk
Bürger Martin
De Moliner Richard
Demel Michael
Dickbauer Klemens
Dicklberger Christoph
Erdpresser Martin
Fellmayr Rainer
Fellner Paul
Fencl Karl
Gams Erich
Graf Alexander
Greifender Bernhard
Gruber Markus
Gschnell Christian
Hable Richard
Hanner Kurt
Helml Thomas
Höfler Wolfgang
Höld Robert
Holzleitner Gernot
Hörmanseder Rudolf
Imlinger Oliver
Irle Markus
Jöbstl Markus
Jungwirth Josef
Käferböck Reinhard
Kogler Manfred
Köttstorfer Marco
Kreuzeder Ulrich
Kurka Gerhard
Lang Peter
Leisch Bernhard
Mayrbäurl Max
Mühlbacher Jörg R.
Narzt Wolfgang
Nedbal Manuel
Öhler Klaus
Pfeifer Bernhard Erich
Pfeiffer Michael
Rathmayr Andreas
Schakerl Christian
Schickermüller Klaus
Schiller Andreas
Schinnerl Dietmar
Schneider Wolfgang
Schram Christian
Schumacher Thomas
Skarke Rainer
Steiner Thomas
Steinkogler Matthias
Stierschneider Odo
Trauner Michael
Ungerhofer Stefan
Willroider Walter
Winter Peter
Wögerbauer Norbert
Zarda Gerald
Zwicknagl Wolfgang

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Kurzfassung

Die Arbeit enthält zwei Teile: (I) die Beschreibung des Pow!-Projektes und (II) das Visualisierungswerkzeug, welches (I) und dessen Quellcode als Voraussetzung hat.

Pow! ist eine kostenlose Softwareentwicklungsoberfläche, die vor allem für Studenten und Schüler entwickelt wurde. Sie ist einfach zu bedienen und verfügt neben einer Projektverwaltung über die Möglichkeit, verschiedene Programmiersprachen und Compiler einzubinden. Zurzeit unterstützt Pow! die Programmiersprachen Oberon-2, Java, C und C++. Mit Pow! wurde ein Visualisierungswerkzeug implementiert, mit dem man objektorientierte Programme zur Laufzeit visualisieren kann.

Der Programmfluß in objektorientierten Programmen ist viel schwerer zu durchschauen als in herkömmlichen Programmen. Der zweite Teil dieser Arbeit beschreibt die Ergebnisse eines Projektes zur Visualisierung von Objekten in gleichzeitig laufenden Programmen. Ein Ziel der Forschungsarbeit war die Machbarkeit anhand eines Prototypen unter Beweis zu stellen, der mit Hilfe der Entwicklungsoberfläche Pow! in der Programmiersprache Oberon-2 implementiert wurde.

Zur Visualisierung von Objekten wird der dynamische Typ der beobachteten Objekte zur Laufzeit ermittelt und anhand des dynamischen Typs eine passende Darstellung ausgewählt. Neben einfachen Objekten können auch komplexe Datenstrukturen wie zum Beispiel Bäume und Graphen visualisiert werden. Außerdem ist es auf einfache Weise möglich, neue Darstellungen durch Ableitung vorhandener Visualisierungsklassen zu erstellen. Dadurch ist die Visualisierung sehr leicht erweiterbar.

Abstract

This thesis consists of two interleaved parts: (I) the description of the Pow! project and (II) the detailed presentation of the visualisation tool, which is based upon (I) and its source code.

Pow! is a free software development environment for students. It is easy to use and provides a project management and the ability to use different programming languages and compilers. Currently Pow! supports the programming languages Oberon-2, Java, C and C++. Using Pow! a visualisation tool was implemented which visualises object oriented programs during runtime.

The control flow of object oriented programs is sometimes hard to predict. The second part of this thesis presents results of a research project for visualisation of objects in running programs. One goal of the research was to implement a prototype which shows that the concept works. The prototype was implemented with the software development environment Pow! and the programming language Oberon-2.

To visualise objects the dynamic type of the inspected objects are determined and according to the dynamic type a proper visualisation is chosen. Besides simple objects complex data structures like trees or graphs can be visualised. It is also possible to extend the visualisation in an easy way, because the visualisation is based on a class library. Therefore new visualisation classes can inherit functionality from existing classes.

Inhaltsverzeichnis

Teil II: Das Visualisierungswerkzeug

1 Einleitung

Die folgende Arbeit beschreibt die Softwareentwicklungsoberfläche Pow! (Programmers Open Workbench) und Ergebnisse eines Projektes zur Visualisierung von Objekten in gleichzeitig kollateral laufenden Programmen[1]. Ziel dieser Forschungsarbeit war aufzuzeigen, wie ein solches Visualisierungswerkzeug funktionieren sollte. Dazu wurde ein Konzept für ein Visualisierungswerkzeug entwickelt und ein Prototyp mittels Pow! implementiert.

Die Arbeit ist im wesentlichen in zwei Teile geteilt. Der erste Teil umfaßt die Kapitel zwei bis einschließlich sechs und behandelt den grundsätzlichen Aufbau von Pow!. Die Kapitel sieben bis neun bilden den zweiten Teil und beschreiben das Visualisierungswerkzeug.

Das zweite Kapitel beschreibt die Entwicklung von Pow! und versucht, die Motivation und Zielsetzung von Pow! zu verdeutlichen. Das dritte Kapitel beschreibt den generellen Aufbau von Pow! und führt in die einzelnen Komponenten des Systems ein. Die Kapitel vier bis sechs widmen sich schließlich jeweils einer wichtigen Komponente des Systems, nämlich dem Compiler, dem Editor und den Werkzeugen, die in das System aufgenommen werden können.

Der zweite Teil beginnt mit Kapitel sieben, das die Motivation und Zielsetzung des Visualisierungsprojektes, sowie dessen grundsätzliches Konzept erläutert. In Kapitel acht werden dann einzelne Komponenten dieses Systems näher erläutert und vorgestellt. Das neunte Kapitel beschäftigt sich dann mit einigen praktischen Anwendungsfällen der Visualisierung.

[1] Dieses Projekt wurde aus Mitteln des Fonds zur Förderung der wissenschaftlichen Forschung (Projekt P11823-TEC: „Transport semantischer Informationen in der Programmvisualisierung) gefördert.

Teil I:

Das Pow-Projekt

2 Das Projekt Pow!

2.1 Motivation und Zielsetzung

Ende 1991 formierte sich am Forschungsinstitut für Mikroprozessortechnik (FIM) ein kleines Team bestehend aus Bernhard Leisch und Ulrich Kreuzeder unter der Leitung von Prof. Mühlbacher, das es sich zum Ziel setzte, eine Entwicklungsoberfläche zu schaffen, die im Programmierunterricht an der Johannes Kepler Universität Linz eingesetzt werden kann.

Zu dieser Zeit war die Programmiersprache Oberon gerade stark im Kommen. Da in der Programmierausbildung der Studienrichtung Informatik an der Universität Linz schon lange die Programmiersprache Modula-2 eingesetzt wurde, war es nur natürlich, daß danach der Nachfolger von Modula-2, nämlich Oberon bzw. Oberon-2 zum Einsatz kommen sollte.

Was manche allerdings von Oberon-2 zurückschreckte, war die Tatsache, daß die Programmiersprache Oberon-2 alleine nicht verwendet werden konnte. Sie konnte nur mit dem Oberon-System benutzt werden. Dabei handelt sich um ein vollständiges Betriebssystem mit graphischer Oberfläche. Die Gegner der Programmiersprache Oberon-2 führten ins Feld, daß sich die Bedienung des Oberon-Systems in sehr vielen Punkten von den bekannten Bedienungsparadigmen der Windows-Welt unterscheide, was den Einstieg für einen Studienanfänger unnötig erschwere. Daraus entstand die Idee eine Programmierumgebung zu schaffen, die ähnlich einfach zu bedienen ist, wie das damals sehr populäre Turbo-Pascal der Firma Borland[2], und die unter Windows läuft und die direkt unter Windows lauffähige Programme erzeugt. Damit sollte den Studenten der Studienrichtung Informatik ein einfach zu bedienendes Werkzeug für die Programmierausbildung zur Verfügung gestellt werden.

Im Laufe der Entwicklung wandelte sich das Bild. Oberon-2 verlor mehr und mehr an Bedeutung und wurde auch in der Ausbildung durch Java abgelöst. Entsprechend änderte sich auch die Ausrichtung von Pow!, was sich auch im Namen niederschlug. Stand Pow! am Anfang noch für Portable Oberon Workbench, so wurde der Name auf Programmers Open Workbench geändert. Pow! verlor also im Laufe der Zeit die Einschränkung nur mit einer Programmiersprache, nämlich Oberon-2, arbeiten zu können, und verwandelte sich in eine universell verwendbare Ent-

[2] Die Firma Borland änderte ihren Namen und ist heute unter dem Namen Inprise bekannt.

wicklungsumgebung. Trotz dieses Schwenk ist die Programmiersprache Oberon-2 noch immer ein wesentlicher Teil des Paketes.

Somit haben sich einige Motive und Zielsetzungen im Laufe der Zeit gewandelt. Welche davon aus damaliger Sicht entscheidend waren und heute vielleicht nicht mehr passend sind bzw. welche aus heutiger Sicht damals nicht von Bedeutung waren soll im folgenden diskutiert werden.

2.1.1 Wahl der Programmiersprache

Die Wahl der Programmiersprache für die Ausbildung der Studenten ist wahrscheinlich an jeder Universität eine heiß geführte Diskussion. Dabei läßt sich der eine oder andere auch schon einmal zu emotionellen Argumenten hinreißen, weil ihm eine Sprache besser gefällt oder weil er damit mehr Erfahrung hat. Daneben gibt es natürlich auch sehr gute sachliche Argumente für die Wahl einer bestimmten Programmiersprache. Die Wahl für die Programmiersprache Oberon-2 war von folgenden Argumenten geprägt.

Die Programmiersprache Oberon und das Oberon-System wurden von Prof. Wirth und Prof. Gutknecht an der ETH Zürich entwickelt (vgl. [WG89] und [WG92]). Oberon-2 ist eine Erweiterung der Programmiersprache Oberon (vgl. [MW91a] und [MGW91b]). Prof. Wirth war auch der Schöpfer eines Vorgängers von Oberon-2, nämlich Modula-2 (vgl. [WIR77a], [WIR77b] und [WIR77c]). Modula-2 wurde an der Universität Linz schon längere Zeit erfolgreich in der Programmierausbildung der Studienanfänger eingesetzt und daher war es naheliegend, auch den Nachfolger Oberon-2 zu verwenden.

Außerdem hatte Oberon-2 den Vorteil, über Erweiterungen zur objektorientierten Programmierung zu verfügen. Auch wenn die Schöpfer von Oberon-2 nicht explizit von objektorientierter Programmierung sprechen, sind erweiterbare Records und typgebundene Prozeduren mit Klassen und Methoden anderer objektorientierter Programmiersprachen vergleichbar. Nachdem kaum jemand mehr den Nutzen objektorientierter Programmierung bestreitet, sollte eine Programmiersprache, die in der Ausbildung verwendet wird, auch über objektorientierte Konzepte verfügen. Der Nachteil, daß nicht alle objektorientierten Feinheiten, wie zum Beispiel Konstruktoren und Destruktoren, von Oberon-2 zur Verfügung gestellt wird, wurde nicht als schwerwiegend betrachtet. Ebenso wurde die fehlende Mehrfachvererbung von manchen sogar begrüßt denn als Nachteil gesehen.

Daneben bietet die Programmiersprache Oberon-2 einen unschätzbaren Vorteil durch ihre klare und einfache Syntax. Im Gegensatz zu C und C++, die durch ihre Syntax viele Sachverhalte kürzer und prägnanter anschreiben läßt, verleitet Oberon-2 nicht zu einem zu knappen Programmierstil. Dies stellt sich gerade in der Programmierausbildung als unschätzbarer Vorteil heraus.

Weiters ist die gebotene Typsicherheit nicht zu unterschätzen. Typkonvertierungen sind sehr genau definiert und werden bis auf wenige wohl definierte Ausnahmen nicht automatisch durchgeführt. Die Wahrscheinlichkeit, daß ein ohne Fehlermeldung übersetztes Programm auch tatsächlich funktioniert, ist bei Oberon-2 um einiges höher als in manchen anderen Programmiersprachen, wie zum Beispiel C oder C++. Dies führt in der Regel dazu, daß Studienanfänger nicht so viel Zeit damit verbringen müssen, irgendwelchen unbeabsichtigten Fehlern nachzulaufen, sondern können mehr Zeit darauf verwenden, korrekte Programme zu schreiben.

Dies waren die wesentlichen Argumente für die Wahl der Programmiersprache Oberon-2 aus damaliger Sicht. Pow! unterstützte zu Beginn auch nur die Programmiersprache Oberon-2. Heute stellt sich die Situation ein wenig anders dar.

Mit Java entstand eine völlig neue Programmiersprache, die viele interessanten Eigenschaften in sich vereinigt. Obwohl sie eine ähnliche Syntax wie C bzw. C++ besitzt, sind viele Kritikpunkte, die bei C bzw. C++ gegriffen haben, weggefallen. Die Typsicherheit hat zum Beispiel Einzug gehalten. Mit der einfachen Vererbung und der Einführung von Interfaces wurden manche Nachteile der Mehrfachvererbung vermieden. Mit Exceptions und der Unterstützung von Threads sind weitere Pluspunkte anzuführen. Somit wird in der Regel über die knappe Syntax hinweg-gesehen und Java gerne in der Ausbildung verwendet. Für Pow! bedeutet dies allerdings, daß es eine zweite Programmiersprache unterstützen muß. Als dies erkannt wurde, wurde Pow! konsequent auf die Unterstützung mehrerer Programmiersprachen umgestellt. Heute präsentiert es sich als offenes System, das sich verschiedener Module bedienen kann, um verschiedene Programmiersprachen in das System einbinden zu können.

2.1.2 Oberon-System vs. Pow!

Wie schon in der Einleitung ausgeführt wurde, war die Entscheidung für die Programmiersprache Oberon-2 einfach. Der größte Nachteil, daß die Programmiersprache nur mit dem Oberon-System gemeinsam verwendet werden kann, sollte durch Pow! aufgehoben werden.

Warum wurde das Oberon-System abgelehnt? Hauptkritikpunkt am Oberon-System war die völlig neue Bedienungsphilosophie. Es gab zum Beispiel keine überlappenden Fenster, sondern nur sogenannte Panes. Alle offenen Panes teilen sich dabei den Bildschirm und keine Pane kann eine andere überlagern. Natürlich kennt man heute auch in Windows nicht überlappende Fenster. Allerdings werden sie nicht ausschließlich eingesetzt. Für manche Fensterarten eignen sich Panes besser. Hierunter fällt zum Beispiel ein Meldungsfenster, das andere Fenster nicht überlappen sollte. Für die meisten Fensterarten haben sich allerdings überlappende Fenster durchgesetzt.

Ein wesentlicher Schwachpunkt der Bedienung des Oberon-System waren außerdem die Vielzahl verschiedener Mausklicks. Zur vernünftigen Bedienung brauchte man eine Drei-Tasten-Maus. Auf den in der PC-Welt weit verbreiteten Zwei-Tasten-Mäusen wurde das Drücken der mittleren Maustaste durch Drücken der beiden Tasten simuliert. Daneben gab es aber auch viele verschiedenen Maustastenkombinationen, die für ungeübte Benutzer schwer zu behalten und durchzuführen waren.

Pow! sollte sich hinsichtlich der Bedienung an die unter Windows üblichen Bedienungsphilosophien anlehnen. Es zeigte sich, daß viele Studienanfänger bereits mit Windows umgehen können. Diese hatten mit Pow! einen großen Vorteil, da sie die grundlegenden Bedienungsschritte bereits kannten.

Außerdem erzeugt Pow! eigenständige Windows-Programme. Das Oberon-System dagegen erzeugt quasi nur Objektdateien. Um solche Dateien ausführen zu können, benötigt man einen speziellen Lader, der natürlich im Oberon-System integriert ist. Diese Variante hat allerdings den Nachteil, daß man seine Programme nicht eigenständig weitergeben kann. Außerdem ist es nicht möglich, externe Module eines anderen Compilers zu seinem Programm zu binden.

2.1.3 Lizenzierung

Ein wesentlicher Antrieb für die Entwicklung von Pow! war neben der Minimierung des Ressourcenbedarfs die Frage der Lizenzkosten. Meist erreichen Lizenzkosten für Software enorme Höhen, die nur schwer in den Budgets öffentlicher Bildungseinrichtungen unterzubringen sind.

Ein Ausweg aus den hohen Lizenzkosten sind spezielle Schulversionen oder Campuslizenzen. Schulversionen werden von vielen Softwareherstellern angeboten. Sie kosten im Schnitt nur ein Drittel des vollen Preises und sind in der Nutzung beschränkt. Bei einer Schulversion eines Compilers wird zum Beispiel meist der Verkauf von mit der Schulversion des Compilers generierten Programmen untersagt. Dies ist aber meist kein großer Nachteil für die Schule. Vielmehr sind die Lizenzkosten noch immer ein Problem. Auch wenn sie auf ein Drittel verringert werden, ergeben sich bei Ausstattung aller Schulungs-PCs enorme Kosten.

Ein anderer Weg sind sogenannte Campuslizenzen. Dabei erlaubt der Softwarehersteller der Bildungseinrichtung, alle PCs mit seiner Software zu einem bestimmten fixen Preis auszustatten. Dies hat den Vorteil, daß bei neu hinzukommenden Rechnern nicht sofort wieder Lizenzkosten anfallen. Allerdings bieten nicht alle Softwarehersteller solch günstigen Pakete an. Die Bedingungen, zu denen solche Campuslizenzen angeboten werden, sind außerdem sehr stark vom Softwarehersteller abhängig.

Ein weiteres Problem ist, daß viele Studenten auch zuhause bereits über einen PC verfügen. Die Umfragen unter den Informatikstudenten der Universität Linz ergeben regelmäßig, daß nur ein sehr kleiner Teil der Studenten (unter 3 %) über keinen eigenen Rechner verfügen. Alle anderen können einen Rechner ihr Eigen nennen. In den meisten Fällen wird aber bei der Software nicht darauf geachtet, daß jedes verwendete Paket korrekt lizenziert wurde. Hauptgrund ist natürlich das sehr knappe Budget, mit dem die meisten Studenten auskommen müssen. Hierfür bieten auch die Schul- oder Campuslizenzen keinen Ausweg, da sie Studenten mit ihrem eigenen PC meist gar nicht berücksichtigen.

Ein wesentliches Ziel bei der Entwicklung von Pow! war daher eine Entwicklungsumgebung zu schaffen, die keine leistungsstarken Rechner benötigt und die ohne Kosten von Schulen und Studenten bzw. Schülern verwendet werden kann.

2.1.4 Verschiedene Programmiersprachen

Wie schon bei der Wahl der Programmiersprache angedeutet wurde, erwuchs mit der Programmiersprache Java mächtige Konkurrenz für Oberon-2. Es zeigte sich leider auch im Laufe der Zeit, daß sich Oberon-2 nicht so durchgesetzt hat, wie dies von manchen erhofft wurde. Sie ist zwar als Ausbildungssprache nach wie vor gut geeignet. Allerdings hat sie praktisch keine Relevanz in der Wirtschaft. Es sind keine nennenswerten größeren Projekte mit Oberon-2 bekannt geworden.

Java ist auf der anderen Seite auch als Ausbildungssprache geeignet. Die C++ ähnliche Syntax trägt zwar nicht viel zur guten Lesbarkeit bei, aber der rasanten Verbreitung der Sprache hat es sicherlich nicht geschadet. Daneben bietet Java aber erhebliche Vorteile mit Exceptions, Threads, Garbage Collection und Plattformunabhängigkeit.

Aus diesem Grund wurde bei der Entwicklung von Pow! sehr bald umgedacht. Es sollte nicht nur eine Programmiersprache sondern viele verschiedene unterstützt werden. Damals entstand die Idee für jede Programmiersprache ein eigenes Modul zur Verfügung zu stellen, das zwischen Pow! und dem Compiler oder Interpreter vermittelt. Damit war die Idee der "Compiler-Schnittstelle" geboren, die Funktionen für Pow! zur Verfügung stellt. Innerhalb der Funktionen werden der Compiler oder Interpreter aufgerufen. Inzwischen gibt es mehrere solcher Schnittstellen, zum Beispiel für Editoren und für Werkzeuge (siehe Abbildung 1).

Der Vorteil für Studenten in der Programmierausbildung liegt auf der Hand. Auch wenn sie mehrere Programmiersprachen lernen, müssen sie sich nur mit einer Programmieroberfläche auseinandersetzen, wenn diese alle benötigten Programmiersprachen unterstützt. Studenten können

sich somit auf das Wesentliche konzentrieren, nämlich das Erlernen einer Programmiersprache und müssen sich nur einmal auf eine Programmierumgebung einstellen.

Dem Vortragenden spart dies auch viel Zeit. In aufeinander aufbauenden Lehrveranstaltungsreihen kann ab dem zweiten Teil auf die Einführung in die Programmieroberfläche verzichtet werden, da sie vom ersten Teil bereits bekannt ist. Dadurch kann wertvolle Ausbildungszeit eingespart werden, die in der Regel meist zu knapp bemessen ist. In der gewonnen Zeit können daher weitere Beispiele durchgeführt werden.

Natürlich kann am meisten Zeit gespart werden, wenn in einer Programmierausbildung nur eine Sprache unterrichtet wird. Dies geht aber auf Kosten einer breiteren Ausbildung. Es ist schwer abzuwägen, welche die bessere Methode ist. Für Pow! stellen beide kein Problem dar. Durch die Unterstützung beliebiger Programmiersprachen, egal ob sie compiliert oder interpretiert werden, kann ein und dieselbe Programmieroberfläche für eine Ausbildung eingesetzt werden.

2.1.5 Zugriff auf Quellcode des Programms

Ein positiver Effekt, den die Entwicklung von Pow! mit sich brachte, war, daß wir als Entwickler natürlich Zugriff auf den gesamten Quellcode einer Programmieroberfläche hatten. Dies war nicht nur auf den Compiler beschränkt, sondern ging vom Kern des Pow!-Systems bis zu allen notwendigen Werkzeugen, wie dem Linker oder Debugger.

Damit konnte weiterführende Projekte oder Experimente gestartet werden, von denen eines das im zweiten Teil der Arbeit beschriebene Visualisierungsprojekt war.

2.2 Projektgeschichte

Die Entwicklung von Pow! begann Ende 1991. Ein erster Prototyp wurde 1992 in [KM92] auf der 2nd European Modula-2 Conference 1992 in Leicester (UK) vorgestellt.

Die Entscheidung, unter welchem Betriebssystem Pow! laufen sollte, war im wesentlichen von der anvisierten Zielgruppe bestimmt. Wie schon beschrieben, sollten vor allem Studenten ein kostengünstiges Werkzeug für ihre Programmierübungen und -projekte in die Hand bekommen. Aus regelmäßigen Umfragen in Programmierpraktika und Algorithmenübungen wußten wir, daß die meisten Studenten PCs mit Windows 3.0 oder 3.1 zuhause installiert hatten. Beide Versionen bezeichnet man auch gerne als Windows 3.x oder 16-Bit Windows. Dazu gehört auch noch die Version 3.11.

Danach hat Microsoft einen radikalen Schnitt gemacht. Windows 95 arbeitet nicht mehr mit einem 16-Bit Speichermodell wie Windows 3.x sondern mit einem 32-Bit Speichermodell. Dies gilt im übrigen auch für Windows 98 und Windows NT. Daher werden diese drei Windows-Versionen im folgenden unter dem Begriff 32-Bit Windows zusammengefaßt werden. Die meisten Programme, die für Windows 3.x gemacht wurden, laufen zwar weiterhin unter 32-Bit Windows. Sie können aber nicht die zusätzlichen Fähigkeiten, wie zum Beispiel Multithreading, dieser Betriebssysteme nutzen.

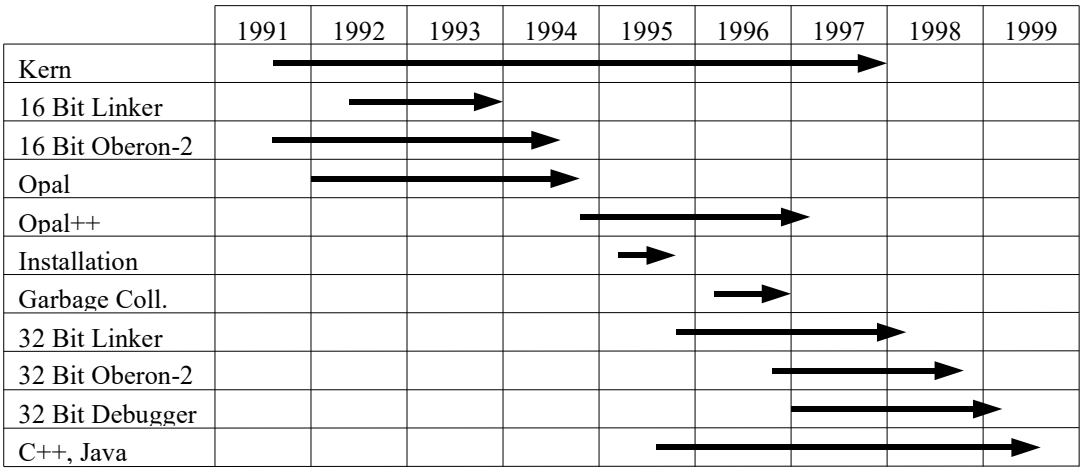


Abb. 2: Zeittafel für das Pow!-Projekt

Aus damaliger Sicht war somit die Entscheidung für 16-Bit Windows klar. Es ist auch leicht nachzuvollziehen, daß Pow! dann auf 32-Bit Windows umgestellt wurde. Aus heutiger Sicht, wäre die Entscheidung nicht mehr so einfach. Im Gegensatz zu damals verwenden viele Studenten heute das Betriebssystem Linux und eine Entscheidung für ein Betriebssystem wäre sehr schwer. Wegen der unterschiedlichen APIs und Benutzerschnittstellen ist eine simultane Entwicklung für zum Beispiel Windows und Linux aus Kapazitätsgründen nicht machbar.

2.2.1 16-Bit Version

Eine der ersten Entscheidungen war, welche Programmiersprache und welche Programm- oder Klassenbibliothek zum Einsatz kommen sollte. Mit der Programmiersprache C war sehr schnell ein passender Kandidat gefunden. Unter Windows gab es damals fast keine Alternativen zu dieser Programmiersprache.

Bei der Klassenbibliothek konnte man sich allerdings nicht entscheiden. Nachdem schlechte Erfahrungen in vorangegangenen Projekten mit den Klassenbibliotheken Glockenspiel, MFC und OWL gemacht wurden (siehe [WIE93] oder [GSC96]), konnte man sich auch nicht durchringen,

eine der beiden Klassenbibliotheken von Microsoft oder Borland zu verwenden. Die Microsoft Foundation Class (MFC) und die Object Windows Library (OWL) von Borland waren beide noch sehr jung und es war damals nicht abzusehen, welche sich durchsetzen würde. Man entschied sich daher, nur API-Funktionen von Windows zu verwenden.

Die Entwicklung des Oberon-2 Compilers startete fast gleichzeitig mit der Entwicklung der Programmieroberfläche. Während die Oberfläche am FIM entwickelt wurde (vgl. [KRE99]), wurde der Compiler von der Firma Robinson Associates, die in England beheimatet ist, entwickelt. Sie war Kooperationspartner des FIM in diesem Projekt und engagierte als freien Mitarbeiter für die Compilerentwicklung Richard De Moliner aus Zürich.

Dank des vorhandenen Oberon-2 Compilers der ETH Zürich mußte nicht von vorne begonnen werden. Da der Compiler bereits in ein Frontend, das für die Syntaxüberprüfung und den Aufbau des Syntaxbaumes zuständig ist, und in ein Backend, das für die Codegenerierung zuständig ist, getrennt war, mußte für unsere Zwecke nur das Backend neu geschrieben werden, damit es Objektcode für Windows generiert.

Das Pow!-Projekt wurde das erste Mal im Jahre 1992 in [KM92] beschrieben.

Nachdem der erste lauffähige Compiler von Richard de Moliner geliefert wurde, wurde mit der Entwicklung der Oberon Portable Application Library (OPAL) begonnen. Das Ziel der OPAL-Entwicklung war eine Bibliothek, mit der man schnell und einfach Oberon-2 Programme implementieren kann, ohne auf die API-Funktionen von Windows zurückgreifen zu müssen.

Im Programmierunterricht beginnt man gerne mit einfachen Aufgabenstellungen. Solche Programme stellen in der Regel keine großen Anforderungen an die Benutzerschnittstelle. Die Möglichkeit zur Eingabe von Zahlen und Texten und die Ausgabe derselben genügt meist. Für diesen Anwendungsbereich sollte die OPAL Unterstützung bieten und die Arbeit erleichtern. Im Laufe der Zeit kamen natürlich weitere Funktionen hinzu. Die Bibliothek eignet sich aber heute noch hervorragend für einfache Aufgabenstellungen. Die OPAL wurde zum ersten Mal in [LEI93] veröffentlicht.

Die ersten Versionen von Pow! benutzten den Linker von Microsoft. Da die Absicht bestand, Pow! ohne Lizenzgebühren für Studenten und Schüler weiterzugeben, konnte nicht ein Teil eines kommerziellen Paketes in Pow! benutzt werden. Es entstand also die Notwendigkeit einen eigenen Linker zu implementieren, der so wie der Linker von Microsoft Objektdateien zu einer lauffähigen Datei bindet. Dabei stellte sich heraus, daß das Hauptproblem die fehlende oder unvollständige Dokumentation seitens Microsoft war. Der Aufbau einer EXE-Datei ist zwar dokumentiert, aber im Detail steckten immer wieder einige Tücken. Insbesondere kostete es viel Zeit, die Bedeutung

einiger undokumentierter Bits zu ermitteln. Manche hatten so große Auswirkung, daß die erzeugte EXE-Datei gar nicht gestartet werden konnte. Solche Fehler konnten meist nur durch tagelanges Probieren und Vergleichen behoben werden.

Im Jahre 1994 wurde die Version 1.3 fertiggestellt. Sie bekam deshalb so viel Aufmerksamkeit, weil sie die erste Version war, die in der Ausbildung verwendet wurde. Alle Versionen davor wurden nur von einem kleinen Benutzerkreis verwendet. Es war die erste harte Bewährungsprobe, die Pow! bestehen mußte. Natürlich ging dies nicht ohne Blessuren aus. Am Anfang stöhnten die Studenten auch einigermaßen. Sie mußten eine neues Entwicklungswerkzeug kennenlernen und mußten sich gleichzeitig mit einigen Fehlern darin herumschlagen. Dadurch, daß Pow! quasi neu auf dem Markt waren und die Studenten es nicht kannten, waren sie sich nicht sicher, ob sie auf einen Fehler gestoßen sind oder ob sie Pow! falsch bedient haben. Nachdem einige lästige Fehler behoben waren, legte sich die Aufregung und Pow! wurde mit der Zeit immer beliebter.

Für das Kern-Entwicklungsteam (Kreuzeder, Leisch und Dietmüller) zeigte die Version 1.3 allerdings neue Seiten der Softwareentwicklung auf. Uns wurde bald klar, daß wir immer mehr Zeit mit der Wartung verbrachten und viele neue Ideen nur mehr schleppend verwirklicht werden konnten. Das Jahr 1995 war geprägt durch viele kleinere Wartungsarbeiten. Insbesondere durch den erstmaligen Einsatz in der Lehre kamen viele kleine Fehler zutage, die vorher unentdeckt blieben. Zu diesem Zeitpunkt stieß ich zum Pow!-Projekt hinzu. Damals begann ich mit einer einfachen Aufgabe, nämlich der Entwicklung eines Installationsprogrammes.

Leisch hatte ein einfaches Installationsprogramm für Pow! in Oberon-2 mit Hilfe der OPAL entwickelt. Es hatte allerdings den Nachteil, daß das Programm jedes Mal überarbeitet werden mußte, wenn eine neue Komponente oder Datei zum Pow! hinzukam. Daher entschied man sich, die Installationsroutine zu erneuern. Eine Eigenentwicklung kam nicht mehr in Frage, da dies zu viel Aufwand bedeutet. Es sollte daher ein passendes Werkzeug gefunden werden. Damals gab es noch nicht so viele gute Installationswerkzeuge wie dies heute der Fall ist. Die Wahl fiel auf ein einfaches Werkzeug von Microsoft, mit dem eine einfach und doch ganz ansprechende Installation gemacht wurde. Der Vorteil war, daß bei einer Änderung der Installationsroutine nur mehr ein Skript geändert werden mußte und nicht mehr das ganze Programm.

Nachdem die Installationsroutine gut lief, übernahm ich Wartung des Laufzeitsystems, das von Leisch bis zu diesem Zeitpunkt betreut wurde. Neben der Wartung sollte ich einen Garbage Collector implementieren. Im Pow! hatten wir im Gegensatz zum Oberon-System noch keinen Garbage Collector implementiert. Damit war die Übernahme von Programmen, die für das Oberon-System entwickelt wurden, ein wenig erschwert. Ein fehlender Garbage Collector hatte nämlich zur Folge, daß alle von einem Oberon-2 Programm angelegten Speicherbereiche auch

wieder frei gegeben werden mußten. Dieser Nachteil sollte durch einen Garbage Collector verschwinden. Außerdem hat ein Garbage Collector bzw. die Nichtnotwendigkeit von "Free" modifizierte Algorithmen bei Listen- und Baumverwaltungsprogrammen zur Folge, die Standardbeispiele in Lehrveranstaltungen wie ‚Algorithmen und Datenstrukturen' sind.

Die eigentliche Implementierung des Garbage Collectors war nicht besonders schwierig. Da es sehr viel Literatur zu diesem Thema gibt, war es einfach, einen passenden Algorithmus zu finden. Eine gute Übersicht über Algorithmen zum Thema Garbage Collection bietet [JL96] oder [WIL94]. Die Wahl fiel auf den Algorithmus ‚Mark & Sweep'. Dabei werden alle Speicherbereiche, die gerade benutzt werden, markiert und danach alle nicht markierten Speicherbereiche freigegeben. Ob ein Speicherbereich benutzt wird, wird daran erkannt, daß ein Zeiger auf ihn zeigt.

Zusätzlich mußte aber der Fall berücksichtigt werden, daß ein Speicherbereich von einer Windows-Funktion benutzt wird aber gleichzeitig kein Zeiger mehr auf diesen Speicherbereich zeigt. Normalerweise würde ein solcher Speicherbereich vom Garbage Collector als unbenutzt erkannt und freigegeben werden. Um dies zu verhindern, kann man Speicherbereiche speziell vor dem Zugriff des Garbage Collectors schützen. Auch an dieser Stelle geht die (verdeckte) Plattformabhängigkeit zum Betriebssystem Windows stark ein.

Pow! hatte zu diesem Zeitpunkt eine gewisse Stabilität erreicht, die es problemlos erlaubte, Pow! im Unterricht und auch in größeren Projekten einzusetzen. Immer mehr Studierende hatten Interesse an einer Mitarbeit und so entstanden einige sehr interessante Projekte (siehe [BON97], [DIC99], [GSC96], [HEL99], [HOL98], [JOE99], [PFE97a], [PFE97b], [RAT99] und [SCH97]). Dadurch erkannten wir auch, daß Pow! zu wenig modularisiert war und Erweiterungen nur schwer durchführbar waren. Es wurde ein neues Konzept überlegt, das es erlauben sollte, verschiedene Komponenten des Systems auszutauschen.

Als Ergebnis dieser Überlegungen entstand eine neue Spezifikation und die heute im Einsatz befindliche Implementierung. Pow! selbst besteht nur mehr aus einem Kern, an dem verschiedene Komponenten angedockt sind. Zwei dieser Komponenten sind der Editor und der Compiler. Alle Komponenten bekamen dadurch eine klare Schnittstelle, die auch dokumentiert wurde. Somit war es ab diesem Zeitpunkt jedem möglich, einen eigenen Compiler einzubinden oder auch seinen eigenen Editor.

In dieser Form wurde Pow! erstmals in der Version 3.0 ausgegeben. Dabei wurde auch die Benutzerschnittstelle überarbeitet, der fertige Garbage Collector integriert und Templates (siehe Kapitel 3.1.6) eingeführt. Es hatte sich gezeigt, daß manche Benutzer mit der Projektverwaltung von Pow! Schwierigkeiten hatten. Insbesondere war nicht immer ganz klar, welche Dateien in neues Projekt aufgenommen werden müssen. Um dem entgegenzuwirken, überlegten wir uns das

Konzept der Templates. Templates sind Vorlagen für Projekte. Um ein neues Projekt anzulegen, braucht der Benutzer nur mehr ein Template auswählen und das Zielverzeichnis für das neue Projekt angeben und alle notwendigen Dateien werden in das Zielverzeichnis kopiert und eine passende Projektdatei erstellt.

2.2.2 32-Bit Version

Die Version 3.0 lief von Anfang an sehr stabil. Trotzdem waren wir nicht zufrieden. Windows 95 und Windows NT wurden immer populärer. Die Version 3.0 von Pow! lief natürlich auch unter Windows 95 und Windows NT. Doch die neuen Fähigkeiten dieser Betriebssysteme konnten nicht ausgenutzt werden. Außerdem konnten auch keine echten 32-Bit-Programme erzeugt werden, was vor allem bei der Möglichkeit, große Speicherblöcke in einem Stück anzulegen, negativ auffiel.

Wir waren uns einig, daß die Zeit für Windows 3.x abgelaufen war und wir unsere Kräfte auf eine Portierung von Pow! konzentrieren sollten. Die Entscheidung war nicht leicht, da Pow! bereits aus einer Menge von Komponenten bestand, deren Portierung einige Zeit verschlingen werde.

Als erstes mußten der Compiler und der Linker portiert werden. Diesmal nahm sich Leisch um den Compiler an. Er konnte auf den Quellcode für die 16-Bit Version zurückgreifen, mußte allerdings die Codegenerierung neu implementieren, da unter 32-Bit Windows ein anderes Objektformat, nämlich COFF, verwendet wird. Außerdem mußten die Aufrufkonventionen geändert werden, da die Funktionen des Windows-API nicht mehr mit PASCAL-Aufrufkonventionen arbeiten.

Noch komplexer war die Situation beim Linker. Es konnten praktisch keine Teile des 16-Bit Linkers übernommen werden, da sich sowohl das Format, in dem die Objektdateien vorliegen, als auch das Format der EXE-Dateien geändert hatte. Der Linker wurde vollkommen neu im Rahmen einer Diplomarbeit entwickelt (siehe [SCH97]). Wie schon bei der Entwicklung des 16-Bit Linkers stellte sich die mangelnde Dokumentation als Hauptproblem heraus. Es gab zwar zu allen notwendigen Teilen des Linkers eine Dokumentation. Sie war allerdings meist nicht vollständig. Meist fehlten nur die Beschreibung einiger weniger Bits. Die Bedeutung dieser durch Probieren und Vergleichen mit dem Microsoft Linker herauszufinden, war eine zeitraubende und fehleranfällige Angelegenheit. Trotz dieser Schwierigkeiten konnten wir bald mit dem Linker zu arbeiten beginnen.

Die Portierung des Pow!-Systems selbst gestaltete sich nicht so schwierig. Nachdem der Kern vorwiegend in C geschrieben war und nur Funktionen des Windows-API verwendet wurden, mußten nur jene Teile geändert werden, in denen sich die Schnittstelle des Windows-API geändert hatte.

Nach der Portierung auf 32-Bit war es erstmals möglich, mehrere Programmiersprachen zu unterstützen. Vom Konzept und von den Schnittstellen war Pow! an sich zwar schon vorher in der Lage, mehrere Programmiersprachen zu unterstützen. Aber die dazu verwendeten Compiler, wie zum Beispiel der Java Compiler von SUN und der GNU C++ Compiler, gab es für 16-Bit Windows nicht sondern nur für 32-Bit. Daher wurde in der 16-Bit Version außer zu Versuchszwecken nie eine andere Programmiersprache integriert als Oberon-2. Mit der Portierung auf 32-Bit wurde erstmals auch die Unterstützung anderer Programmiersprachen, wie Java und C++, implementiert.

Das Jahr 1997 war geprägt durch die Portierung des Kerns von Pow! auf 32-Bit und durch viele kleine Detailverbesserungen an der 16-Bit Version. Letztere erfreute sich stetig steigender Beliebtheit. Ende 1998 arbeiteten wir mit Hochdruck an der ersten vollständigen 32-Bit Fassung von Pow! und konnten mit September 1998 die 32-Bit Version auf unserer Homepage zur Verfügung stellen.

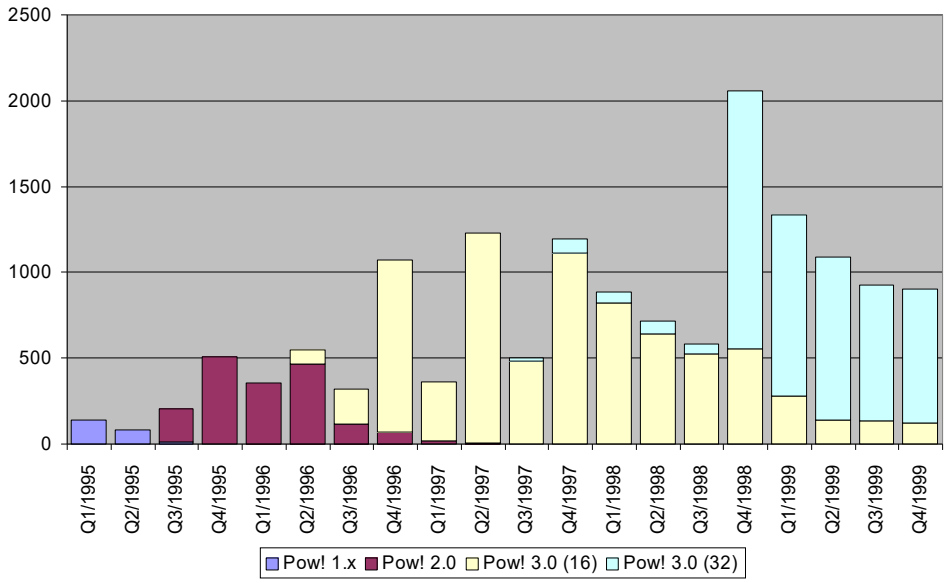


Abb. 3: Anzahl der Pow!-Downloads von 1995 - 1999

Abbildung 2 zeigt die zeitliche Verteilung der Downloads von Pow!, wobei in dieser Graphik nur die Downloads von unserem FTP- und Web-Server berücksichtigt sind und die Downloads von anderen FTP-Servern[3] nicht enthalten sind. Es ist zu erkennen, daß mit jeder neuen Version die Anzahl der Downloads angestiegen ist. Insbesondere sticht die 32-Bit Version heraus, die im 4. Quartal 1998 zu einem Höhepunkt von über 2000 Downloads führte. In Summe wurde Pow! seit 1995 über 15.000 Mal von unserem FTP- oder Web-Server heruntergeladen.

[3] Pow! kann auch von anderen FTP- bzw. Web-Servern heruntergeladen werden, nämlich Simtel (http://www.simtel.net) und Winsite (http://www.winsite.com).

2.3 Aktueller Stand

Als diese Arbeit entstand, gab es zwei aktuelle Versionen von Pow!, nämlich die 16-Bit und die 32-Bit Version. Beide tragen die Versionsnummer 3.0. Die 16-Bit Version wird nicht mehr weiterentwickelt, sondern nur mehr gewartet. An der 32-Bit wird laufend weitergearbeitet.

Wie das folgende Kapitel erklären wird, besteht Pow! aus einer Vielzahl von Einzelkomponenten und Hilfsprogrammen, an denen viele verschiedene Personen gearbeitet haben. Jeden einzelnen Beitrag im Detail anzugeben, würde den Rahmen dieser Arbeit sprengen. Daher sind alle, die am Pow! mitgearbeitet haben, in der "Hall Of Fame" eingetragen. Die folgende Tabelle gibt daher nur einen Überblick über die Kernkomponenten und ihre Autoren.

Pow!-Komponente	16-Bit Pow!	32-Bit Pow!
Pow!-Kern	Ulrich Kreuzeder	Ulrich Kreuzeder
Oberon-2 Compiler	Richard De Moliner (Robinson Associates)	Bernhard Leisch
Linker	Ulrich Kreuzeder	Christian Schackerl
OPAL	Bernhard Leisch	Bernhard Leisch
OPAL++	Bernhard Leisch	Bernhard Leisch
Installationsroutine	Peter René Dietmüller	Peter René Dietmüller
Laufzeitsystem für Oberon-2	Richard De Moliner, Bernhard Leisch, Peter René Dietmüller	Peter René Dietmüller
Garbage Collector	Peter René Dietmüller	Peter René Dietmüller
Editor PowEdit	Ulrich Kreuzeder	Ulrich Kreuzeder
Editor Boosted	Studenten u. Bernhard Leisch	Bernhard Leisch

3 Aufbau des Pow!-Systems

Wie schon im vorigen Kapitel erläutert wurde, sahen wir als Zielgruppe vor allem Schüler, Studenten und Lehrer. Sie sollten eine einfache und kostenlose Entwicklungsumgebung in die Hand bekommen, mit denen kleine bis mittlere Programme und Projekte durchgeführt werden konnten. Pow! sollte möglichst einfach zu bedienen sein, sodaß der Aufwand für das Erlernen und das Bedienen der Entwicklungsoberfläche möglichst gering ist und die Studenten die meiste Zeit mit der Lösung ihrer Probleme verbringen können und nicht viel Zeit für die korrekte Bedienung des Werkzeuges investieren müssen.

Aus diesen Überlegungen heraus entstand natürlich der Wunsch nach einem automatischen *Make* (siehe Kapitel 3.1.5). Ziel war es, daß nur mehr die geschriebenen Quelldateien in ein Projekt eingetragen werden müssen und sich danach Pow! um die korrekte Reihenfolge und Übersetzung der Quelldateien kümmert. Um die Erstellung von Projekten noch einmal zu vereinfachen, wurden Projektvorlagen entwickelt, die Templates genannt wurden. Erzeugt man ein neues Projekt aus einem Template, dann werden alle Quelldateien, die im Template eingetragen sind, automatisch in das Projekt übernommen. Dabei werden allfällig vorkommende Projektnamen automatisch in den Quelldateien angepaßt. Außerdem werden auch alle anderen Dateien, wie zum Beispiel Bibliotheken, in das neue Projekt aufgenommen (siehe Kapitel 3.1.6).

Eine Versionsverwaltung empfanden wir dagegen, sei nicht notwendig für diese Art und Größe von Programmen.

Pow! war zu Beginn als Entwicklungsumgebung für Schüler und Studenten gedacht und entwickelte sich zusehends zu einem offenen, modularen System, indem viele Teile nach Belieben ausgetauscht werden können. Trotz des modularen Aufbaus sollte der Benutzer aber immer nur mit einer konsistenten Benutzeroberfläche arbeiten, damit er nicht ständig umlernen muß. Man darf nicht übersehen, daß bei der Programmierausbildung das Umlernen auf unterschiedliche Editor- und Benutzeroberflächen ein nicht vernachlässigbarer Kostenfaktor (Personal, Zeit) ist.

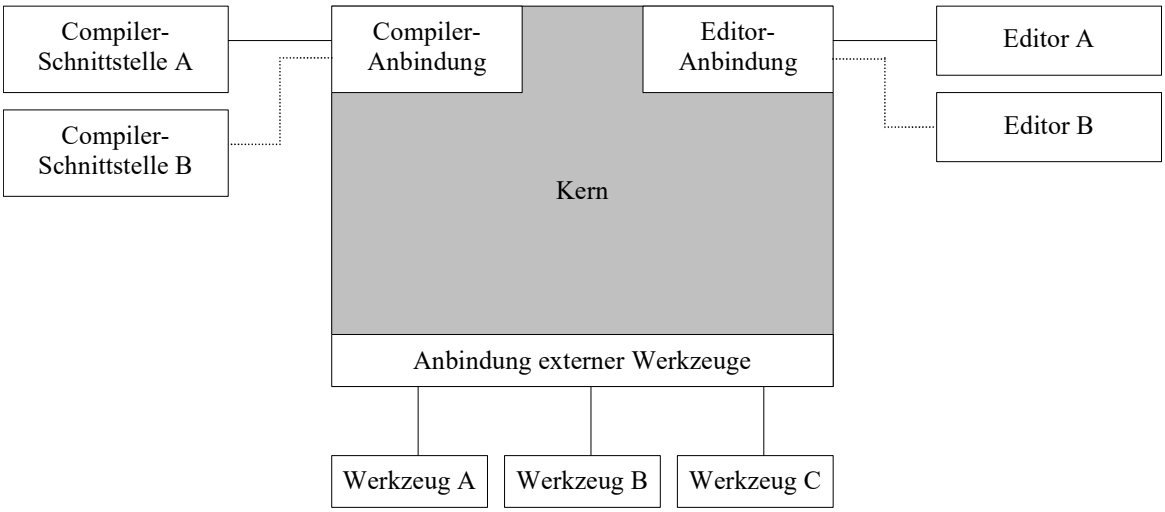


Abb. 4: Zusammenspiel der Pow!-Komponenten

Pow! besteht im wesentlichen aus vier Teilen (siehe Abbildung 5). Der wichtigste und gleichzeitig zentrale Teil ist der Kern des Systems, der das gesamte System steuert und kontrolliert. An ihn "docken" sich Compiler, Editoren und Werkzeuge an, die jeweils über eine klar definierte Schnittstelle mit dem Kern kommunizieren. Die einzelnen Teile werden im folgenden erläutert.

3.1 Kern

Der Kern ist die zentrale Komponente von Pow!. Er kontrolliert und steuert die gesamte Anwendung und ist für folgende Aufgaben bzw. Bausteine zuständig:

- Benutzerschnittstelle (Menüs, Symbolleiste, Statusleiste, MDI-Fenster)
- Steuerung des Compilers inkl. Make, Build und Compile
- Steuerung des Editors inkl. Suchen und Ersetzen, Drucken
- Steuerung der Werkzeuge
- Online-Hilfe
- Projektverwaltung (inkl. Default Projects)
- Templates

- Meldungsfenster
- Konfiguration
- DDE-Server

3.1.1 Benutzerschnittstelle

Eine wesentliche Aufgabe des Kerns von Pow! ist die Interaktion mit dem Benutzer. Wie es unter Windows üblich ist, verwendet Pow! das Multiple Document Interface (MDI). Das bedeutet, daß die Applikation aus einem Hauptfenster (Frame Window) besteht, in dem beliebig viele Fenster (Child Windows) eingebettet sein können. Ebenso wird eine Symbolleiste und eine Statuszeile verwendet. Dazu werden aber keine Standardkomponenten von Windows verwendet, weil es diese zum Zeitpunkt der Entwicklung von Pow! noch nicht gab und weil man Probleme mit dem Copyright von vorn herein vermeiden wollte.

Beim Start von Pow! öffnet der Kern das Hauptfenster und bettet darin eine Symbolleiste und eine Statuszeile ein. Danach wird das Client Window angelegt. Dabei handelt es sich um ein unsichtbares Fenster innerhalb des Frame Window, das zur Verwaltung der Child Windows dient. Das Anlegen und Öffnen der Child Windows wird dem jeweiligen Editor überlassen.

Befehle des Benutzers werden generell an das Frame Window gesendet. Die Ereignisprozedur dieses Fenster entscheidet, was mit dem Befehl zu geschehen hat. Handelt es sich zum Beispiel um den Befehl 'Speichern', dann wird dieser Befehl an den zurzeit aktiven Editor weitergereicht. Handelt es sich dagegen um einen Befehl zum Übersetzen des aktuellen Quellcodes, wird dieser Befehl an die aktive Compiler-Schnittstelle weitergereicht.

Der Kern kontrolliert also nur die Hauptkomponenten von Pow!. Alle anderen Komponenten werden vom jeweiligen Interface kontrolliert.

3.1.2 Steuerung des Compiler

Wie schon zu Beginn dieses Kapitels erläutert übernimmt der Kern nicht alle Aufgaben zum Übersetzen und Binden eines Programmes. Teile davon sind in ein Compilerinterface ausgelagert. Dieses kann selbst einen Compiler integriert haben oder einen externen Compiler aufrufen. Die Schnittstelle zwischen Kern und Compilerinterface wird im Detail in einem eigenen Abschnitt beschrieben.

Die Idee war, daß der Benutzer umstellen kann, welchen Compiler er verwenden will. Dabei macht es keinen Unterschied, ob der Benutzer nur zwischen verschiedenen Compiler einer Programmier-

sprache wählen kann oder auch zwischen verschiedenen Programmiersprachen. Im Menü Options/Preferences gibt es eine Combobox, die alle verfügbaren Compilerinterfaces anzeigt. Eines davon kann der Benutzer auswählen. Diese ist dann aktiv und wird für die Erledigung der Befehle Compile, Make, Build und Link im Menü Compile verwendet. Die verfügbaren Compilerinterfaces ermittelt der Kern von Pow!, indem er im Verzeichnis, aus dem Pow! aufgerufen wurde, alle Dateien mit der Endung *cll* sucht. Die Dateinamen ohne dieser Endung werden dann in die Combobox angezeigt (siehe Abbildung 6).

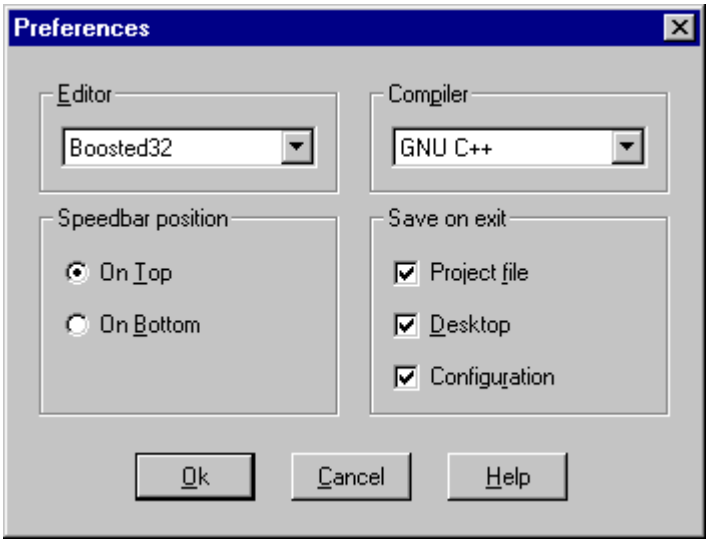


Abb. 7: Pow!-Einstellungen unter Options/PreferencesDas Compilerinterface kümmert sich um das Übersetzten von Quelldateien und Binden von Programmen. Beim Entwurf der Schnittstelle wurde davon ausgegangen, daß zwei Schritte zum Erzeugen eines Programmes notwendig sind. Im ersten Schritt werden alle Quelldateien von einem Compiler in Objektdateien übersetzt und im zweiten Schritt werden alle Objektdateien zu einer ausführbaren Datei gebunden.

Außerdem wurden Module beim Entwurf der Schnittstelle berücksichtigt. Es wurde davon ausgegangen, daß jede Quelldatei ein Modul implementiert und andere Module importieren kann. Das Importieren eines Moduls bedeutet in der Regel, daß das importierte Modul vor dem importierenden Modul übersetzt werden muß. Um dies berücksichtigen zu können, müssen solche Abhängigkeiten erkannt werden können. Dies ist ebenfalls Aufgabe des Compilerinterface, da das Modulkonzept nicht in allen Programmiersprachen gleich funktioniert.

Dem Benutzer stehen folgende Befehle für das Übersetzen und Linken seiner Programme zur Verfügung: Compile, Make, Build und Link. Bei Compile wird im wesentlichen nur der Quellcode übersetzt, der sich gerade im aktiven Fenster befindet. Ist kein Fenster geöffnet, kann diese Funktion nicht ausgeführt werden.

Die anderen Funktionen beziehen sich dagegen alle auf das aktuelle Projekt. Bei Make werden alle Dateien übersetzt, die sich seit der letzten Übersetzung geändert haben. Bei Build werden alle Dateien unabhängig vom Änderungsstatus übersetzt. Die Funktion Link bindet das Projekt zu einer ausführbaren Datei zusammen. Die letzte Funktion wird auch bei den Funktionen Make und Build durchgeführt. Dabei teilen sich Kern und Compilerinterface die Aufgaben. Das Compilerinterface stellt im wesentlichen Funktionen zum Übersetzen einer Quelldatei in eine Objektdatei, zum Binden eines Projektes, zur Ermittlung, ob eine Quelldatei von einer anderen abhängt, und zur Ermittlung, ob eine Quelldatei jünger ist als eine Objektdatei zur Verfügung.

Um den Rest kümmert sich der Kern. Insbesondere ist der Kern für das Make und Build zuständig. Dabei werden die dem Projekt zugeordneten und in einer Projektdatei eingetragenen Quelldateien analysiert. In einem ersten Schritt wird ermittelt, welche Abhängigkeiten zwischen den einzelnen Quelldateien besteht. Um zu dieser Information zu gelangen, bedient sich der Kern der erwähnten Funktionen des Compilerinterface. Anhand dieser Abhängigkeiten entscheidet der Kern, in welcher Reihenfolge die Quelldateien übersetzt werden müssen. Natürlich werden zuerst jene Quelldateien übersetzt, die von keiner anderen Quelldatei abhängen.

Nachdem die Reihenfolge geklärt ist, ruft der Kern die Funktion zum Übersetzen aus dem Compilerinterface auf. Der wesentliche Unterschied zwischen Make und Build liegt nur darin, daß die Funktion Make vorher prüft, ob die Übersetzung wirklich notwendig ist. Hat sich nämlich seit der letzten Übersetzung nichts an der Quelldatei geändert, dann muß die Übersetzung nicht durchgeführt werden. Dies kann durch den Vergleich der Daten der Quelldatei und ihrer zugehörigen Objektdatei ermittelt werden. Ist die Objektdatei jünger als die Quelldatei hat sich seit der letzten Übersetzung nichts an der Quelldatei geändert. In einem solchen Fall unterdrückt die Funktion Make das Übersetzen der Quelldatei und geht sofort zur nächsten Datei weiter. Die Funktion Build nimmt darauf keine Rücksicht.

Zusätzlich zu den eben beschriebenen Funktionen, gibt es noch die Möglichkeit, Optionen einzustellen. Dazu stellt der Kern drei Einträge im Menü Options zur Verfügung. Diese sind Compiler, Linker und Directories. Wie der Name schon sagt, kann man mit Compiler die Optionen des Compilers einstellen, mit Linker die Optionen für das Binden des Programmes und mit Directories kann man Verzeichnisse einstellen. Abbildung 8 zeigt als Beispiel den Einstellungs-dialog des Oberon-2 Compilers.

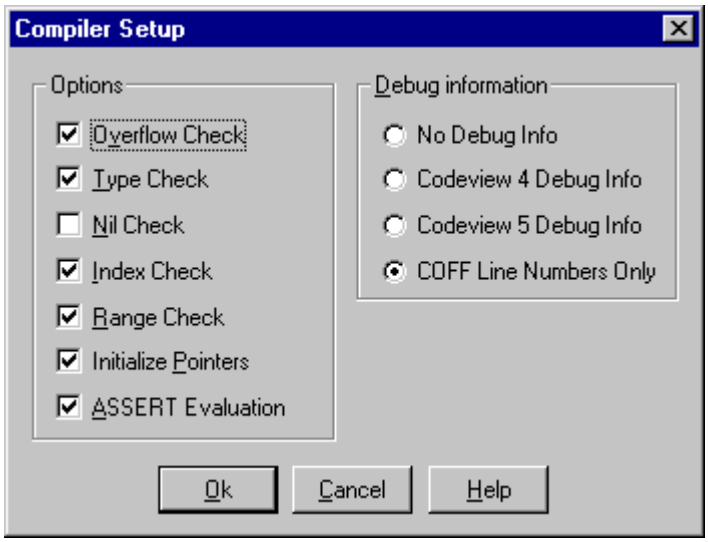


Abb. 9: Einstellungen des Oberon-2 CompilersNachdem sich diese Einstellungen in der Regel sehr stark zwischen den Compilern unterscheiden, überläßt der Kern die gesamte Arbeit dem Compilerinterface. Für jeden dieser drei Menüpunkte Compiler, Linker und Directories gibt es genau eine Funktion im Compilerinterface, die vom Kern aufgerufen wird. Das Compilerinterface muß dann dafür sorgen, daß eine Dialogbox mit den aktuellen Einstellungen angezeigt wird. Es ist überdies dafür zuständig, die vom Benutzer durchgeführten Einstellungen zu speichern.

3.1.3 Steuerung des Editor

Die Wahl eines passenden Editors für das Eintippen von Quellcode ist oft eine emotionale Entscheidung. Viele Programmierer verwenden einen ihnen lang bekannten Editor, weil sie ihn gewohnt sind und dadurch flott damit arbeiten können. Einem Programmierer einen Editor wegzunehmen, kann zu einer Krise führen. Daher ist es nur konsequent, daß es auch im Pow! die Möglichkeit gibt, sich seinen passenden Editor auszusuchen. Über das Menü Options/Preferences kann sich der Benutzer einen Editor aussuchen, der ab diesen Zeitpunkt aktiv ist. Alle Editorbefehle werden dann an diesen Editor gesendet.

Ähnlich dem Compilerinterface ermittelt der Kern die vorhandenen Editoren anhand der Dateien, die es im Pow!-Verzeichnis gibt. Alle Dateien mit der Endung *ell* werden in die Liste der Editoren aufgenommen, wobei jeweils nur der Name der Datei ohne Endung aufgenommen wird.

Wie beim Compilerinterface kommuniziert der Kern mit dem Editor über eine definierte Schnittstelle, der Editor-Schnittstelle. Diese ist im Anhang erläutert. An dieser Stelle soll nur die Idee hinter dieser Schnittstelle aufgezeigt werden.

Die Arbeitsteilung zwischen Kern und Editor ist einfacher als beim Compiler. Der Kern erwartet, daß der Editor selbständig ein Fenster als Kind des Client Window öffnet und darin den Quellcode anzeigt. Alle Befehle, die der Benutzer über das Menü oder der Symbolleiste eingibt, gehen direkt an den Kern. Die meisten davon werden unverändert an den Editor weitergereicht. Bei einigen Funktionen, wie zum Beispiel dem Öffnen einer Datei oder dem Suchen eines Textes, übernimmt der Kern noch einige Vorarbeiten. Für diese Funktionen zeigt er die entsprechenden Dialoge an und läßt dem Benutzer die gewünschten Eingaben tätigen. Danach schickt er den Befehl samt eingegebener Daten an den Editor.

Wie beim Compilerinterface gibt es auch für den Editor die Möglichkeit, benutzerdefinierte Einstellungen vorzunehmen. Im Menü Options findet sich der Menüpunkt Editor, mit dem der Benutzer einen Dialog zum Einstellen von Editoreinstellungen anzeigen lassen kann (siehe Abbildung 10).

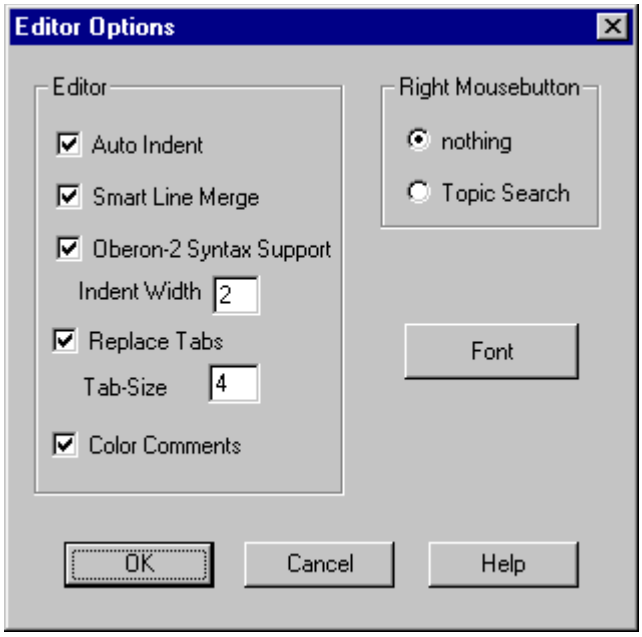


Abb. 11: EditoreinstellungenWie beim Compilerinterface wird auch in diesem Fall nur eine Funktion des Editors aufgerufen, die dafür zu sorgen hat, daß eine Dialogbox erscheint und daß die geänderten Einstellungen auch wieder abgespeichert werden.

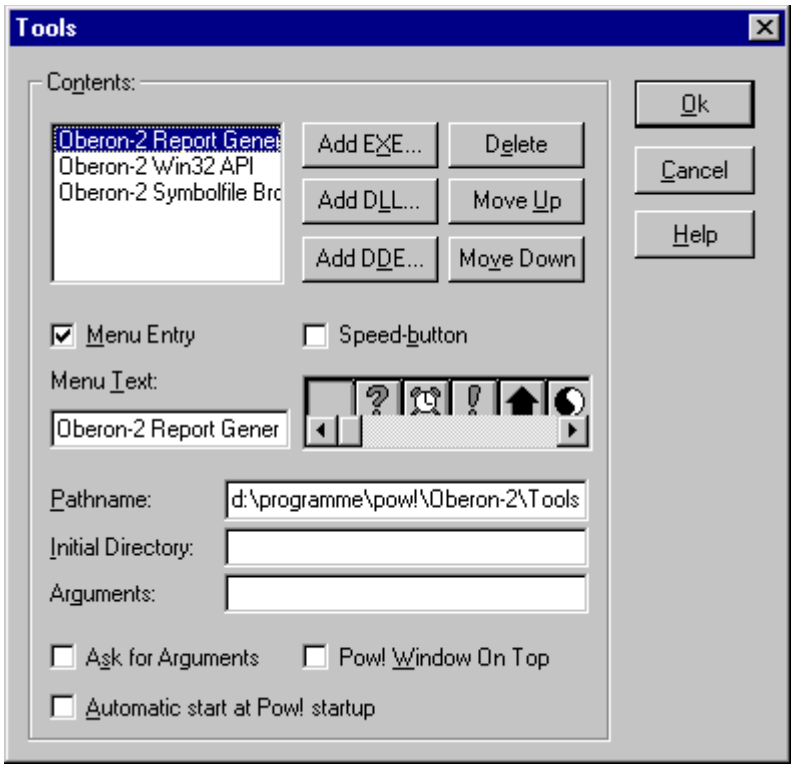
3.1.4 Steuerung der Werkzeuge

Unter Werkzeugen versteht Pow! externe Programme oder Dynamic Link Libraries (DLLs), die von der Entwicklungsumgebung aus aufgerufen werden können. So können zum Beispiel Werkzeuge eingebunden werden, die in Pow! nicht zur Verfügung stehen.

Im Gegensatz zu Compilern und Editoren sind Werkzeuge nicht so eng mit Pow! verbunden. Sie benötigen keine eigene Schnittstelle, sondern werden mit Mitteln des Betriebssystems aktiviert.

Grundsätzlich unterscheidet Pow! drei Arten der Interaktion mit einem externen Werkzeug. Es kann ein Programm aufgerufen werden, eine Funktion einer Dynamic Link Library aufgerufen werden oder ein DDE-Befehl[4] an ein Programm abgesetzt werden. Zusätzlich gibt es aber auch die Möglichkeit, Ergebnisse des Werkzeuges in einem Fenster von Pow! anzuzeigen.

Die eingebundenen Werkzeuge können über das Menü Tools/Options konfiguriert werden. Ruft man diese Funktion auf, erscheint die in Abbildung 12 dargestellte Dialogbox. Alle Einstellungen betreffend der Werkzeuge werden in der Konfigurationsdatei von Pow! gespeichert.



[4] DDE steht für Dynamic Data Exchange und ist ein in Windows gängiges Verfahren zum Datenaustausch.

3.1.4.1 Abb. 13: WerkzeugeinstellungenKommunikation mit Werkzeugen

Pow! unterstützt bei der Kommunikation mit externen Werkzeugen beide Richtungen der Kommunikation. Es ist also möglich, sowohl Informationen von Pow! an das Werkzeug zu senden als auch vom Werkzeug an Pow!.

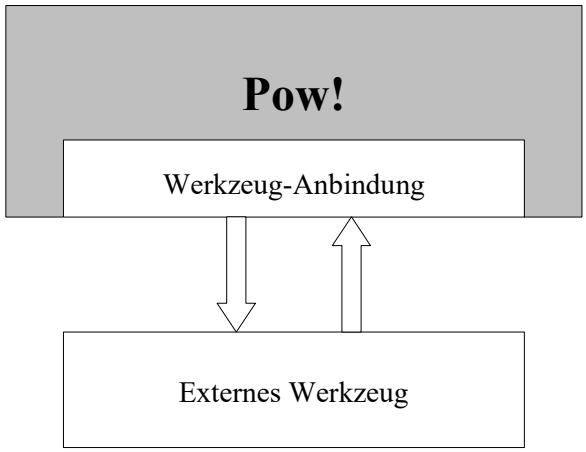


Abb. 14: Kommunikation mit externen Werkzeugen

Für die Kommunikation von Pow! zum Werkzeug unterstützt Pow! zwei Kommunikations-mechanismen. Als erste und einfachste Variante wird die Parameterübergabe unterstützt. Die zweite Variante ist das Versenden von DDE-Befehlen.

Welche der beiden Varianten eingesetzt werden kann, hängt in der Regel davon ab, welche Methode das Werkzeug unterstützt. Die meisten Werkzeuge unterstützen die Parameterübergabe, sodaß diese auch am häufigsten eingesetzt wird. Eine DDE-Kommunikation ist dagegen eher selten vorzufinden.

Wie dies beim Aufruf eines Programmes ebenfalls üblich ist, werden die Parameter in einer Zeichenkette an das Werkzeug übergeben. Die einzelnen Parameter stehen in dieser Zeichenkette hintereinander und sind durch Leerzeichen getrennt.

Entscheidet sich der Benutzer beim Einrichten eines Werkzeuges Parameter zu übergeben, kann er zwischen fixen und variablen Parameter wählen. Fixe Parameter sind Werte, die unverändert an das Programm übergeben werden. Variable Parameter sind spezielle Parameter von Pow!, die von Pow! vor dem Aufruf des Programmes durch aktuelle konkrete Werte ersetzt werden. Die folgende Tabelle führt alle möglichen variablen Parameter und ihre Bedeutung auf.

Parameter	wird ersetzt
%a	durch den Handle des aktiven Edit-Fensters (hex 4-stellig)
%d	durch den Pfad zu Pow!
%f	durch den Dateinamen des aktiven Edit-Fensters

Parameter	wird ersetzt
%i	durch den Pfad zu Microsoft Windows,
%n	durch den Dateinamen des aktiven Edit-Fensters ohne Pfad und Extension
%o	durch den Pfad zum aktuellen Projekt
%p	durch den aktuellen Projektnamen (ohne Verzeichnis und Extension)
%r	durch den vollständigen aktuellen Projektnamen
%s	durch den Pfad zur aktuellen Datei im Edit-Fenster (ohne Dateiname und Extension)
%w	durch den Handle des Hauptfensters (hex 4-stellig)
%x	durch den Namen der zu erzeugenden Applikation
%1	durch den ersten Programmparameter von Pow!
%2	durch den zweiten Programmparameter
%/	ersetzt alle Backslashes (\) im Text durch Schrägstriche (/)
%%	durch ein einzelnes Prozentzeichen

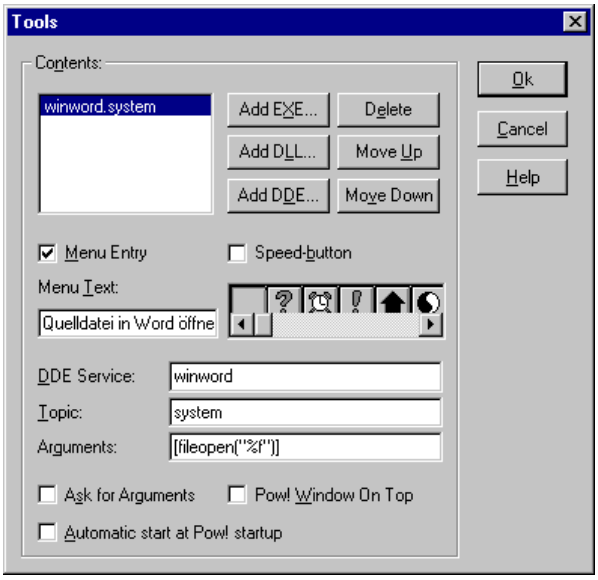
Auch ein Werkzeug kann mit Pow! in Verbindung treten. Dies wird ebenfalls über DDE abgewickelt. In Pow! wurde zu diesem Zweck ein DDE-Server implementiert, sodaß Pow! Befehle von außen entgegennehmen kann. Somit ist es zum Beispiel leicht möglich, daß die Ausgabe eines Programmes direkt in einem Pow!-Fenster angezeigt wird. Als Beispiel kann hierfür der später beschriebene Symbolfile Browser dienen. Dabei handelt es sich um ein Programm, das Symboldateien analysiert und den Inhalt in lesbarer Form anzeigt. Dieses Werkzeug ist in der Lage über DDE eine Verbindung zu Pow! aufzubauen, ein neues Fenster zu öffnen und das Ergebnis der Analyse in das Fenster zu übertragen.

3.1.4.2 Aufruf externer Werkzeuge

Pow! unterstützt drei Varianten für den Aufruf externer Werkzeuge. Welche Variante zum Zug kommt hängt vom Werkzeug ab. Je nachdem, in welcher Form das Werkzeug vorliegt, ist eine der folgenden Varianten zu verwenden.

- Das Werkzeug liegt als gewöhnliches Programm (eine Datei mit der Endung .exe) vor. Dann wird das Werkzeug wie jedes andere Programm von Pow! als Kindprozeß gestartet. Dabei übergibt Pow! die vom Benutzer eingegebenen Parameter als Programmargument.
- Das Werkzeug liegt als Dynamic Link Library (DLL) vor. Dann kann eine Funktion der Dynamic Link Library aufgerufen werden. Die vom Benutzer eingegebenen Parameter werden als erster Parameter in Form einer Zeichenkette der Funktion übergeben.
- Das Werkzeug läuft als eigenständiger Prozeß. Dann kann über eine DDE-Kommunikation mit dem Werkzeug Kontakt aufgenommen werden und über diesen Kanal Befehle an das

Programm gesendet werden. Als Beispiel kann hier Microsoft Word dienen. Über diesen Mechanismus könnte man zum Beispiel Word veranlassen, die gerade im Editor geladene Quelldatei zu laden. Dazu muß man lediglich eine DDE-Kommunikation mit dem DDE-Service "Winword", dem Topic "System" und dem Argument "[fileopen("%f")]" einrichten (siehe Abbildung 15). Klickt man auf den entsprechenden Menüeintrag im Pow! wird das eingegebene DDE-Kommando an Word geschickt und ausgeführt. Das Ergebnis ist dann, daß Word die in Pow! geöffnete Quelldatei anzeigt.



3.1.5 Abb. 16: DDE-Kommunikation mit Microsoft WordProjektverwaltung

Programme, die in einer einzigen Quelldatei implementiert werden können, sind eher die Ausnahme denn die Regel. Programmiersprachen wie Oberon-2 oder Modula-2 fördern die Aufteilung eines Programmes in mehrere Module. Das bedeutet aber, daß der Entwickler in der Regel mit mehreren Quelldateien für ein einziges Programm konfrontiert ist.

Damit der Programmierer nicht bei jeder Übersetzung jede Quelldatei einzeln übersetzen muß, gibt es verschiedenartige Unterstützungen. Die einfachste Variante ist eine Batch-Datei, in der alle Quelldateien übersetzt werden. Dies hat den Nachteil, daß immer alle Quelldateien übersetzt werden, unabhängig davon, ob sie sich geändert haben. Eine andere Möglichkeit ist ein Makefile. Dabei muß der Programmierer genau angeben, welche Quelldateien wie übersetzt werden und welche Abhängigkeiten zwischen den Dateien herrschen. Dies kann manchmal eine mühsame Angelegenheit werden. Viele Entwicklungsumgebungen bieten daher die Möglichkeit an, Projekte zu definieren. In ein Projekt kann man alle notwendigen Quelldateien aufnehmen und die

Entwicklungsumgebung entscheidet selbständig wie und in welcher Reihenfolge die Quelldateien übersetzt werden.

Diese Art der Projektverwaltung bietet auch Pow! an. Der größte Teil davon ist im Kern implementiert. Über das Menü Projects kann man ein Projekt öffnen, speichern und ändern. Im Änderungsdialog (siehe Abbildung 17) kann der Benutzer seine Quelldateien und eventuell zusätzlich notwendige Bibliotheken dem Projekt hinzufügen. Welche Dateiarten ausgewählt werden können, entscheidet übrigens das Compilerinterface. Es liefert in einer Funktion alle ihm bekannten Arten von Quelldateien zurück. Dies ist deshalb sinnvoll, weil es ja das Compilerinterface ist, daß bei einem Make oder Build entscheiden muß, wie eine Quelldatei übersetzt werden kann. Daher macht es Sinn, daß nur jene Arten von Quelldateien in ein Projekt eingetragen werden können, die dem Compilerinterface auch bekannt sind.

Die Namen der zu einem Projekt gehörenden Dateien werden in einer Projektdatei gespeichert. Dies wird vom Kern erledigt. Dabei werden aber nicht alle Dateien mit vollem Pfad abgespeichert. Es wird versucht, möglichst viele Dateien mit einem relativen Pfad zu speichern. Was dies bedeutet, kann am besten anhand eines kleinen Beispiels demonstriert werden.

In diesem Beispiel soll es um ein Projekt gehen, das in der Datei c:\test\test.prj gespeichert ist. Zu diesem Projekt gehören zwei Quelldateien, nämlich test1.mod und test2.mod. Beide sind im Verzeichnis c:\test\source gespeichert. Wenn dieses Projekt vom Kern gespeichert wird, dann wird von den beiden Quelldateien nicht der gesamte Pfad gespeichert, sondern nur *source\test1.mod* und *source\test2.mod*. In beiden Fällen wird zusätzlich ein Hinweis gespeichert, daß die Pfadangaben relativ zum Projektverzeichnis zu verstehen sind. Wird das Projekt später wieder geöffnet, wird das Verzeichnis, in dem sich die Projektdatei befindet, vor den Pfadangaben der Quelldateien eingefügt. Nachdem sich die Projektdatei im Verzeichnis c:\test befindet, wird der Eintrag *source\test1.mod* zu *c:\test\source\test1.mod* erweitert. Dasselbe passiert mit der zweiten Datei.

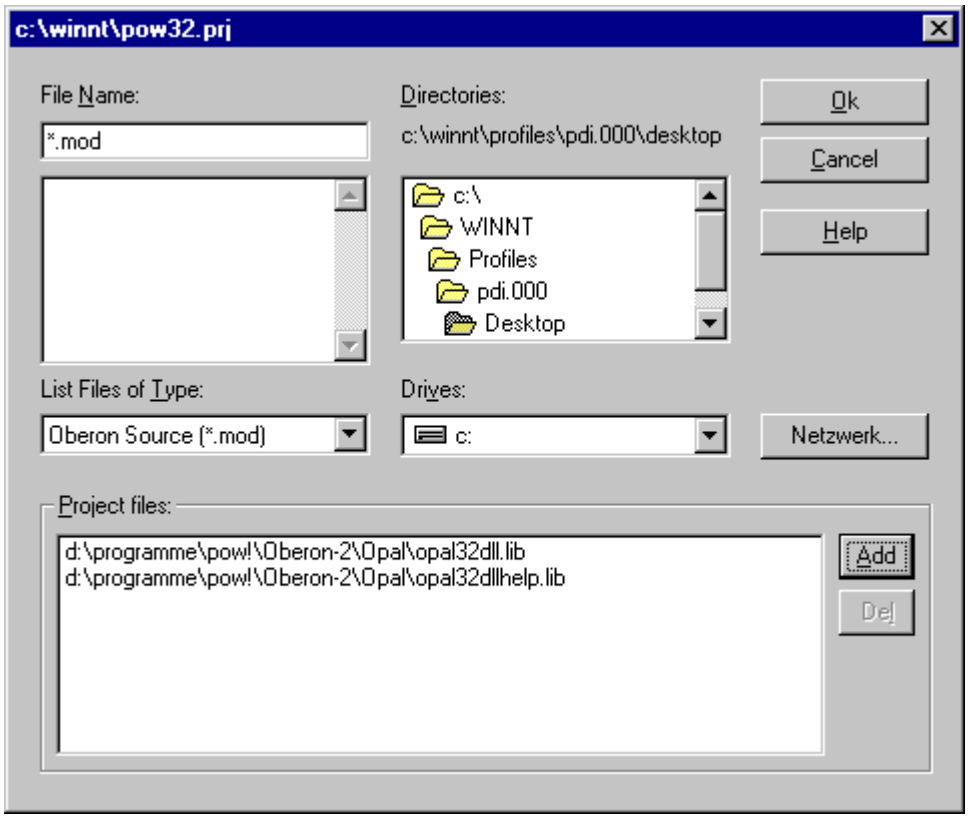


Abb. 18: Dialog Project/EditEin Vorteil dieser Technik liegt auf der Hand. Es wird weniger Platz zum Speichern der Dateinamen benötigt. Dies ist aber nicht der entscheidende Vorteil. Viel wichtiger ist die Tatsache, daß diese Projekte ohne Probleme verschoben werden können, solange sich die Struktur der Unterverzeichnisse nicht ändert.

Entscheidet sich der Benutzer zum Beispiel dieses Projekt in das Verzeichnis c:\projekte\mueller zu verschieben, dann können trotzdem die im Projekt befindlichen Quelldateien gefunden werden, weil beim Öffnen des Projekts vor dem Pfad der Quelldatei das Verzeichnis der Projektdatei eingefügt wird. Das bedeutet, daß beim Öffnen des Projektes im Verzeichnis c:\projekte\mueller die Quelldateien test1.mod und test2.mod im Verzeichnis c:\projekte\mueller\source gesucht werden.

Voraussetzung dafür ist allerdings, daß das Projekt immer mit allen seinen Unterverzeichnissen verschoben wird. Dies ist aber der Regelfall, da viele Programmierer Dateien eines Projektes in einem Verzeichnis zusammenfassen.

Neben dem Projektverzeichnis gibt es noch zwei andere Verzeichnisse, die als Referenz verwendet werden, nämlich das Verzeichnis, in dem Pow! installiert ist, und das Verzeichnis, in dem Windows installiert ist. Damit kann man ein Projekt, das zum Beispiel eine Bibliothek aus dem

Pow!-Verzeichnis benötigt, problemlos auf einen anderen Rechner kopieren. Sollte dort Pow! in einem anderen Verzeichnis als auf dem ursprünglichen Rechner installiert sein, spielt dies keine Rolle, weil beim Öffnen einer Projektdatei immer das aktuelle Pow!-Verzeichnis herangezogen wird.

Nachdem der Kern alle notwendigen Daten in die Projektdatei geschrieben hat, ruft er noch eine spezielle Funktion des Compilerinterface auf, damit das Compilerinterface eigene Daten in die Projektdatei speichern kann. Insbesondere muß das Compilerinterface seine eigenen Einstellungen aus den Einstellungsdialogen speichern.

Der Benutzer kann immer nur mit einem Projekt arbeiten. Dieses Projekt wird herangezogen, wenn der Benutzer die Befehle Make, Build oder Link ausführt. Einige der Rückmeldungen der ersten Benutzer betrafen die Projektverwaltung. Es zeigte sich, daß Einsteigern, die noch nie mit Projekten gearbeitet hatten, die Projektverwaltung unklar ist. In den ersten Turbo-Pascal-Versionen war es möglich, ein Programm zu erstellen, ohne ein Projekt anzulegen. Pow! kann aber ohne Projekt nicht arbeiten. Um den Einstieg in Pow! zu vereinfachen, wurde das Default Project eingeführt. Wenn kein Projekt im Pow! geöffnet ist, dann ist in Pow! das Default Project aktiv. Die zugehörige Projektdatei liegt im Windows-Verzeichnis und trägt den Namen pow32.prj.

Das Besondere an diesem Projekt ist aber, daß zusätzlich zu den Dateien, die im Projekt eingetragen sind, automatisch immer die gerade geöffnete Quelldatei in das Projekt aufgenommen wird. Sind mehrere Quelldateien geöffnet, dann wird die Quelldatei des aktiven Fensters ausgewählt. Damit ist der gleiche Bedienungskomfort hergestellt wie beim diesbezüglichen Vorbild Turbo Pascal.

Für den Benutzer hat das folgende Auswirkung: Der Benutzer arbeitet zum Beispiel gerade an der Datei hello1.mod. Wenn er kein Projekt geöffnet hat und den Befehl Make ausführt, dann wird zum Default Project die Datei hello1.mod temporär hinzugefügt und danach der Befehl Make mit dem Default Project durchgeführt. Dies hat zur Folge, daß die Quelldatei hello1.mod übersetzt wird und ein Programm erzeugt wird. Der Programmierer kann also ohne ein Projekt anzulegen kleinere Programme mit dem Default Project übersetzen, binden und ausführen.

Dies geht allerdings nur solange gut, als die Einstellungen des Default Project für das Programm passen und das Programm nicht aus mehreren Quelldateien bestehen. Die Einstellungen des Default Project können geändert werden. Man muß dazu nur die Datei pow32.prj öffnen und die gewünschten Änderungen in der Entwicklungsumgebung vornehmen. Schließt man danach das Projekt, gelten die neuen Einstellungen für alle folgenden Übersetzungsläufe. Dies wird in der Regel dann notwendig sein, wenn man standardmäßig einen anderen Compiler verwenden will. Die Beschränkung, daß das Programm nur aus einer Quelldatei bestehen darf, kann nur durch das

Anlegen eines eigenen Projektes umgangen werden. Für solche Programme empfiehlt es sich ein Projekt anzulegen, um den Überblick über das Projekt zu behalten. Das Default Project hilft somit vor allem Einsteigern bei ihren ersten Programmierversuchen.

3.1.5.1 Aufbau einer Projektdatei

In einer Projektdatei werden alle Dateien, die zu einem Projekt gehören, sowie alle Einstellungen eines Projektes gespeichert. Die übliche Endung für Projektdateien ist *prj*.

Eine Besonderheit der Projektdateien muß näher beleuchtet werden, bevor der Aufbau von Projektdateien im Detail besprochen werden kann. Wie schon erwähnt, speichert Pow! die Dateinamen nicht immer mit vollen Pfad ab, sondern versucht so oft wie möglich einen relativen Pfad zu verwenden. Zu einem relativen Pfad muß es einen Ausgangspunkt geben. Dafür kann entweder das Verzeichnis, in dem sich die Projektdatei befindet, oder das Verzeichnis, in dem Pow! installiert ist, verwendet werden. Wenn Pow! einen Dateinamen relativ abspeichern kann, dann wird an der ersten Stelle des Dateinamens ein 1-Byte Code gespeichert, der den Ausgangspunkt angibt. Die folgende Tabelle zeigt die möglichen Codes und nennt ihre Bedeutung. Dahinter folgen dann eventuell vorhandene Unterverzeichnisse und der Dateiname.

Code	Bedeutung
1	Die Datei liegt im selben Verzeichnis wie die Projektdatei oder in einem Unterverzeichnis davon.
2	Die Datei liegt im selben Pow!-Verzeichnis oder in einem Unterverzeichnis davon.
3	Der Dateiname ist mit einem absoluten Pfad abgespeichert.
4	Die Datei liegt in einem 'Bruder'-Verzeichnis des Projektverzeichnisses.

Der Dateiname *c:\projekte\mueller\test1.mod* würde in der Projektdatei *c:\projekte\mueller\test1.prj* folgendermaßen abgespeichert werden: ☺*test1.mod*. Das Zeichen '☺' steht hierbei für den Wert 1.

Zeichenketten werden in der Regel mit Nullbyte am Ende und mit einer 2-Byte großen Längenangabe am Beginn gespeichert. Der Platzbedarf für eine Zeichenkette ist nicht fix, sondern es werden so viele Bytes benötigt, wie es der Länge der Zeichenkette entspricht.

Eine Projektdatei besteht im wesentlichen aus vier Teilen, die hintereinander gespeichert werden. Beim ersten Teil handelt es sich um einen Header, der generelle Informationen über das Projekt enthält. Daran schließt sich ein Teil an, der Informationen zu allen geöffneten Fenstern enthält.

Abhängig von der Anzahl der geöffneten Fenster variiert dieser Teil in der Größe und kann sogar fast leer sein. Danach kommen alle zum Projekt gehörenden Dateien.

	Bytes	Bedeutung
Header	6	Version der Projektdatei: "Prj10" oder "Prj20" oder "Prj21" (für aktuelle Projekte)
	2	Change Flag
	12	Leer (für zukünftige Zwecke reserviert)
	2	Länge des Namens der Compiler-DLL inkl. Nullbyte am Ende
		Name der Compiler-DLL (nullterminiert)
	2	Länge der Argumente für das Starten des Programmes
		Argumente für das Starten des Programmes (nullterminiert)
Fenster	4	Anzahl der offenen Fenster; für jedes offene Fenster werden folgende Informationen gespeichert:
	2	Länge des Dateinamens inkl. Nullbyte
		Dateiname (nullterminiert)
	4	Fensterzustand (WS_ICONIC, WS_MAXIMIZE)
	jew. 4	Koordinaten des Fensters relativ zum Vaterfenster (left, top, right, bottom)
	4	Zeile, in der die Textmarke steht
	4	Spalte, in der die Textmarke steht
Datei	2	Anzahl der Projektdateien; für jede Projektdatei werden folgende Informationen gespeichert.
	256	Name jeder Projektdatei (fixe Länge!)
Compiler-Int.	4	Interner Handle des Compilerinterface. Mit diesem Wert kann man nichts anfangen, sondern nur entscheiden, ob danach noch Daten des Compiler Interface kommen oder nicht. 0 bedeutet nein, jeder andere Wert bedeutet ja.
		Daten des Compiler Interface

3.1.6 Templates

Um das Anlegen von Projekten zu vereinfachen, sind Templates eingeführt worden. Dabei handelt es sich um spezielle Projekte, die man als Vorlage für neue Projekte verwenden kann.

Um in Pow! ein neues Projekt anzulegen, kann man zwei Wege wählen. Man kann über das Menü File/New/Project oder über das Menü Project/New einsteigen. In beiden Fällen erscheinen Untermenüs, aus denen man ein Template auswählen kann. Danach erscheint eine Dialogbox, in der man angeben kann, in welchem Verzeichnis das neue Projekt abgelegt werden soll und wie es

heißen soll. Nach einem Druck auf den OK-Button, wird das neue Projekt in dem angegebenen Verzeichnis angelegt. Existiert das Verzeichnis noch nicht, wird es angelegt.

Das Besondere an den Templates ist aber die Art und Weise wie das neue Projekt angelegt wird. Jedes Template besitzt eine zugehörige Template-Datei, die meist die Endung *tpl* hat. Dabei handelt es sich im wesentlichen um eine normale Projektdatei. Das heißt in dieser Datei können bereits Quelldateien und Bibliotheken eingetragen sein. Außerdem sind darin auch schon die Standardeinstellungen für den Compiler und Linker enthalten.

Wird nun ein neues Projekt mit Hilfe eines Templates angelegt, dann kopiert der Kern zuerst einmal die Template-Datei in das Projektverzeichnis und nennt sie auf den Namen des Projekts um. Dabei wird natürlich auch die Endung *tpl* in *prj* geändert. Außerdem werden alle Quelldateien, die in der Template-Datei eingetragen sind, in das Projektverzeichnis kopiert. Gleichzeitig werden alle Quelldateien, die denselben Namen wie die Template-Datei haben, in den neuen Projektnamen umbenannt. Heißt beispielsweise die Template-Datei *display.tpl* und enthält sie eine Quelldatei namens *display.mod*, dann wird die Template-Datei in *test.prj* umbenannt, wenn der Benutzer als Projektnamen test angegeben hat. Außerdem wird die Quelldatei *display.mod* in *test.mod* umbenannt. Gäbe es beispielsweise auch eine Quelldatei *display.html* dann würde diese nach dem Kopieren auch in *test.html* umbenannt.

Da in manchen Programmiersprachen der Modulname mit dem Dateinamen korrespondieren muß, kann es durch die Umbenennung von Quelldateien passieren, daß sie nicht mehr fehlerfrei übersetzt werden können. Daher wird nach dem Umbenennen einer Quelldatei noch ein weiterer Schritt eingelegt. Dabei wird die Quelldatei selbst verändert. Es wird versucht, den Modulnamen im Programmcode umzuändern. Da aber der Kern nichts über die Syntax einer Programmiersprache weiß, wird diese Aufgabe dem Compilerinterface übertragen, das in der Template-Datei eingetragen ist. Es muß eine Funktion zur Verfügung stellen, die in einer Quelldatei einen Modulnamen in einen anderen ändern kann.

Durch diesen Mechanismus ist es somit möglich, daß ein Template bereits Quellcode enthalten kann. Dieser wird beim Anlegen des Projektes kopiert und an den Namen des Projektes angepaßt. Dadurch kann man in einem Template ein fertiges Rahmenprogramm zur Verfügung stellen, das der Benutzer nur noch an seine Bedürfnisse anpassen braucht. Ein spezieller Satz von Templates kann daher in Verbindung mit einer Klassenbibliothek als Application Framework verstanden werden.

Auch das Erstellen eines Templates ist nicht besonders kompliziert. Aus jedem Projekt kann man ein Template erstellen. Dazu muß man nur die Funktion ‚Save as Template ...' im Menü Project aufrufen. Der Kern macht dann genau dasselbe wie beim Anlegen eines Projektes. Die Projektdatei

wird kopiert und umbenannt. Ebenso werden die Quelldateien kopiert und umbenannt, wenn sie mit dem Namen der Projektdatei übereinstimmen.

Am Ende der Beschreibung der Templates bleibt noch zu erwähnen, wie die Untermenüs des Menü Project/New aufgebaut werden. Das Untermenü in File/New/Project ist übrigens mit dem erstgenannten ident. Beide werden während der Initialisierungsphase des Kerns aufgebaut. Diese beiden Untermenüs werden also dynamisch aufgebaut und nicht aus den Ressourcen ausgelesen.

3.1.7 Konfiguration

Wie viele andere Applikationen muß auch Pow! gewisse Einstellungen abspeichern, um sie nach dem Beenden und erneuten Starten von Pow! wieder zur Verfügung zu haben. Welche Informationen beim Beenden von Pow! automatisch abgespeichert werden, kann der Benutzer im Dialog Options, Preferences einstellen (siehe Abbildung 19). Beim Starten von Pow! müssen diese Informationen wieder eingelesen werden.

Pow! speichert nicht alle Informationen in einer Datei, sondern verteilt die Konfigurationsdaten auf mehrere Dateien, um das Gesamtsystem flexibler gestalten zu können. Die wichtigste Datei ist die Konfigurationsdatei. In ihr werden zum Beispiel die Elemente der Symbolleiste und der aktuelle Editor gespeichert. Konfigurationsdateien kann es mehrere geben. So kann zum Beispiel jeder Benutzer seine eigene Konfiguration verwenden. Die Standardkonfigurationsdatei heißt pow.cfg und ist im Windows-Verzeichnis abgelegt.

Daneben gibt es noch die Desktop-Datei, in der der Zustand der Oberfläche beim Verlassen von Pow! gespeichert wird. Darunter fällt zum Beispiel die Größe des Fensters von Pow!. Sie liegt ebenfalls im Windows-Verzeichnis und heißt pow.dsk.

Zuletzt sind dann noch die Projektdateien. Wie schon beschrieben, enthalten auch Projektdateien Konfigurationsinformationen des in ihnen gespeicherten Projektes. Sie sind im Detail schon früher beschrieben worden.

3.1.7.1 Aufbau der Konfigurationsdateien

Konfigurationsdateien bestehen aus zwei Teilen, dem Kopf und den Informationen für die Symbolleiste.

	Bytes	Bedeutung
Header	4	Version
	24	unbenutzt (aus Kompatibilitätsgründen)
	40	Name der Compiler-DLL (nullterminiert)
	40	Name der Editor-DLL (nullterminiert)
	14	unbenutzt (aus Kompatibilitätsgründen)
	4	Projektdateien beim Beenden speichern? 0 bedeutet nein, jeder andere Wert bedeutet ja.
	4	Desktop beim Beenden speichern? 0 bedeutet nein, jeder andere Wert bedeutet ja.
	4	Konfiguration beim Beenden speichern? 0 bedeutet nein, jeder andere Wert bedeutet ja.
	4	Symbolleiste oben/unten? 0 bedeutet oben, jeder andere Wert bedeutet unten.
	4	Warnungen beim Fehler suchen ignorieren ? 0 bedeutet, daß Warnungen nicht ignoriert werden, jeder andere Wert bedeutet, daß Warnungen ignoriert werden.
	20	unbenutzt (für zukünftige Verwendung)
	4	Dieses Feld ist optional. Wenn es vorhanden ist, dann enthält es den Wert 9999 und bedeutet, daß die nachfolgenden Informationen über die Symbolleiste im neuen Format abgespeichert ist. Das neue Format ist eine Erweiterung des alten Formates.
	4	Anzahl der Elemente in der Symbolleiste
altes Format	4	Soll das Werkzeug im Menü erscheinen?
	4	Soll das Werkzeug in der Symbolleiste erscheinen?
	4	Soll beim Aufruf des Werkzeuges nach Parametern gefragt werden?
	4	Soll das Pow!-Fenster als oberstes Fenster angezeigt werden?
	4	ID des Knopfes
	80	Pfad und Dateiname des Werkzeuges
	80	Aktuelles Verzeichnis
	80	Menütext
	100	Argumente
	1	Typ: 'X' .. EXE, 'L' .. DLL, 'E' .. DDE
neues F.	4	Soll das Werkzeug beim Starten von Pow! gestartet werden?
	4	Ist das Werkzeug über DDE hinzugefügt worden?
	96	Reserviert

3.1.7.2 Aufbau der Desktop-Datei

Die Desktop-Datei ist die zweite Konfigurationsdatei von Pow! und enthält im wesentlichen Informationen über den Zustand der Fenster beim letzten Schließen von Pow!.

	Bytes	Bedeutung
Desktop-Datei	2	Länge des Dateinamens der benutzerspezifischen Konfigurationsdatei
		benutzerspezifische Konfigurationsdatei
	2	Länge des Dateinamens der aktuellen Projektdatei
		aktuelle Projektdatei
	4	Status des Pow!-Fensters (WS_ICONIC, WS_MAXIMIZE, ..)
	16	Koordinaten des Pow!-Fensters (left, right, top, bottom)
	20	unbenutzt (für zukünftige Verwendung)

3.1.8 DDE-Server

Dynamic Data Exchange (DDE) ist ein Mechanismus zum Austausch von Daten zwischen Programmen innerhalb eines Rechners. Diesen Mechanismus gibt es beinahe schon seit den Anfängen von Windows. In der heutigen Zeit greifen die Entwickler lieber zu COM, weil es flexibler und über Rechnergrenzen hinweg funktioniert.

DDE arbeitet nach einem Client-Server-Modell. Der DDE-Server stellt Funktionen oder Daten zur Verfügung, die ein DDE-Client benutzen kann. Pow! kann sowohl als DDE-Server als auch als -Client arbeiten. Im ersten Fall stellt es Funktionen zur Verfügung, mit den Pow! von externen Programmen gesteuert werden kann. Im zweiten Fall bedient es sich DDE, um externen Werkzeugen Daten oder Parameter zu übergeben.

Der DDE-Server ist Teil des Kern und wird bei Programmstart automatisch gestartet. Er stellt folgende Befehle zur Verfügung.

Befehl	Bedeutung
OpenFile name	Pow! veranlaßt den Editor, die angegebene Datei zu öffnen
NewFile name	Pow! veranlaßt den Editor, eine neue Datei anzulegen und sie unter dem angegebenen Namen zu speichern.
SaveFile name	Pow! veranlaßt den Editor, den Inhalt des aktiven Editorfensters unter dem angegebenen Namen zu speichern.
Activate name	Pow! holt jenes Fenster in den Vordergrund, in dem die Datei mit dem übergebenen Dateinamen angezeigt wird.
AppendFile name	Pow! hängt den Inhalt der angegebenen Datei an den Inhalt des gerade aktiven Editorfensters an.
AppendText buf	Pow! hängt den übergebenen Text an den Inhalt des gerade aktiven Editorfensters an.
InsertFile name	Pow! fügt den Inhalt der angegebenen Datei an der aktuellen Position des gerade aktiven Editorfensters ein.
InsertText buf	Pow! fügt den übergebenen Text an der aktuellen Position des gerade aktiven Editorfensters ein.

Neben diesen Befehlen unterstützt der DDE-Server auch die Möglichkeit, bestimmte Informationen über Pow! abzufragen. Im Konkreten sind dies:

Befehl	Bedeutung
ActiveFile	Pow! gibt den Namen der Datei zurück, die im gerade aktiven Editorfenster angezeigt wird.
EditBuffer	Pow! gibt den Inhalt des gerade aktiven Editorfensters zurück.

3.1.9 Starten von Pow!

Beim Start von Pow! öffnet der Kern das Hauptfenster und bettet darin eine Symbolleiste und eine Statuszeile ein. Danach wird das Client Window angelegt. Der größte Teil des Menüs wird aus den Ressourcen geladen. Allerdings gibt es auch variable Teile, die vom Kern während des Startens von Pow! aufgebaut werden. Dazu gehören zum Beispiel Teile der Symbolleiste. In der Symbolleiste gibt es sowohl fixe Buttons als auch solche, die vom Benutzer nach eigenen Vorlieben konfiguriert werden können. Diese zusätzlichen Einträge erscheinen sowohl im Menü Tools als auch in der Symbolleiste. Der Kern ist dafür verantwortlich, diese vom Benutzer konfigurierten Buttons in der Symbolleiste und im Menü hinzuzufügen.

Neben dem Tools-Menü gibt es noch einen weiteren Menüteil, der nicht fix ist. Es handelt sich dabei um die Untermenüs File/New/Project und Project/New. Der Kern baut diese Menüteile beim Start auf. Dabei schaut er nach, welche Templates im Pow!-Verzeichnis vorhanden sind und fügt diese an passender Position ein. Eine genaue Beschreibung dieser Arbeit befindet sich im Abschnitt über die Templates in diesem Kapitel.

Während des Starten muß Pow! seine Konfigurationsdaten einlesen. Dazu geht es folgendermaßen vor.

1. Pow! öffnet die Standardkonfigurationsdatei pow.cfg im Windows-Verzeichnis und lädt die darin gespeicherten Einstellungen.
2. Pow! lädt die Einstellungen der Desktop-Datei pow.dsk im Windows-Verzeichnis.
3. Wenn in der Desktop-Datei eine benutzerspezifische Konfigurationsdatei eingetragen ist, dann lädt Pow! alle Einstellungen aus dieser Datei.
4. Wenn in der Desktop-Datei eine Projektdatei eingetragen ist, dann wird sie geladen und zum aktuellen Projekt gemacht.

4 Compiler

Pow! ist seinem Namen nach ein offenes System. Zu einem offenen System gehört auch die Unterstützung verschiedener Programmiersprachen und Compiler. Um die Integration von Programmiersprachen möglichst flexibel zu gestalten, gibt es in Pow! sogenannte Compiler-Schnittstellen-Module.

Sie dienen als Schnittstelle zwischen Pow! und einem Compiler und sind in Form von Dynamic Link Libraries implementiert. Damit kann der Pow!-Benutzer jederzeit seine Programmiersprache wechseln, ohne die Entwicklungsumgebung wechseln zu müssen. Pow! verwendet einfach die vom Benutzer eingestellte Compiler-Schnittstelle.

Diese Compiler-Schnittstellen-Module müssen bestimmte Funktionen zur Verfügung stellen. Pow! ruft diese Funktionen auf und erwartet, daß dadurch bestimmte Aufgaben erledigt werden. Wie diese Aufgaben erledigt werden sollen, wird nicht bestimmt. Dies ist dem Programmierer des Schnittstellenmoduls überlassen und hängt insbesondere auch davon ab, wie der unterstützte Compiler arbeitet.

Die Möglichkeit verschiedene Compiler benutzen zu können, wurde zu Beginn in Pow! nicht ausgeschöpft, weil für die 16-Bit Version nur ein Compiler zur Verfügung stand. Die meisten frei erhältlichen Compiler wurden für 32-Bit Windows implementiert und konnten mit vertretbarem Aufwand nicht unter 16-Bit Windows verwendet werden.

Zu Beginn stand nur ein Oberon-2 Compiler zur Verfügung. Nach Umstieg auf 32-Bit Windows wurde dieser Compiler in eine 32-Bit Version portiert. Dazu kamen noch Compiler-Schnittstellen für Java und C++.

Die C++ Schnittstelle wurde in mehreren Schritten entwickelt. Die erste Version entwickelte Bernhard Pfeifer (siehe [PFE97a]). In dieser Version wurde eine von der Firma Cygnus für Windows portierte Version des GNU C/C++ Compilers benutzt. Diese Version lief zufriedenstellend, hatte aber den Nachteil, daß sie relativ viel Ballast durch die Portierung mit sich schleppte und mit Ressourcendateien nur mäßig zurecht kam. Daher wurde in der zweiten Version eine andere Version des GNU C++ Compilers namens MingW32 [MIN01] integriert. Diese war von jeglichem Unix-Ballast befreit und konnte auch mit Ressourcen vernünftig umgehen.

Parallel dazu wurde auch eine Compiler-Schnittstelle für Java entwickelt. Darin wird der Compiler der Firma SUN verwendet, der im kostenlosen Paket namens Java Development Kit enthalten ist.

Zurzeit sind drei Compiler-Schnittstellen im Einsatz, eine für die Programmiersprache Oberon-2, eine für Java und eine für die Programmiersprachen C bzw. C++.

Die Compiler-Schnittstelle für Oberon-2 verwendet einen vom FIM entwickelten Oberon-2 Compiler und Linker. Beide zusammen generieren 'echte' 32-Bit Windows Programme. Folgende Dateien gehören zur Oberon-2 Compiler-Schnittstelle:

- Compiler-Schnittstelle: Oberon-2 (32-bit).cll
- Compiler: Obrn32.dll
- Linker: Link32.dll
- Hilfe: Oberon.hlp

Die Compiler-Schnittstelle für die Programmiersprache Java verwendet den Compiler und Interpreter (Java Virtual Machine) des Java Development Kit (JDK) der Firma Sun und besteht aus folgenden Dateien:

- Compiler-Schnittstelle: Sun JDK.cll
- Compiler des JDK: Javac.exe
- Interpreter (Java Virtual Machine) des JDK: Java.exe
- Hilfe: _java.hlp

Die Compiler-Schnittstelle für die Programmiersprache C++ verwendet den Compiler MingW32 (siehe [MIN01]). Dabei handelt es sich um eine Portierung des GNU C/C++ Compilers für Windows 95 bzw. Windows NT. Ursprünglich wurde diese Portierung von Cygnus [CYG01] durchgeführt. Auf dieser Arbeit baut der Compiler MingW32 auf und wurde um den Unix-Balast der ersten Portierung befreit, sodaß mit MingW32 ein kleiner und flotter C/C++ Compiler für 32-Bit Windows zur Verfügung steht.

Die Compiler-Schnittstelle besteht aus folgenden Dateien:

- Compiler-Schnittstelle: GNU C++.cll
- Compiler und Linker von MingW32: gcc.exe
- Hilfe: GNU C++.hlp

4.1 Oberon-2

Der erste Compiler, der in Pow! zur Verfügung stand, war ein Compiler für die Programmiersprache Oberon-2, der ausführbare Dateien für 16-Bit Windows erzeugte. Für die Entwicklung dieses Compilers konnte man auf den Quellcode des Oberon-2 Compilers der ETH Zürich zurückgreifen. Dieser Compiler war in Frontend und Backend getrennt. Das Frontend analysiert den Quellcode (Parsing) und baut einen Syntaxbaum auf. Es ist in der Regel unabhängig von der Zielmaschine, auf der das vom Compiler generierte Programm laufen soll. Das Backend ist dafür zuständig, aus dem Syntaxbaum ein lauffähiges Programm zu generieren. Durch diese Trennung ist es möglich, das Frontend weitestgehend weiter zu verwenden und das Backend an Windows anzupassen. Das bedeutet, daß im wesentlichen die Codeerzeugung für Windows entwickelt werden mußte. Dabei traten einige Probleme vor allem im Zusammenhang mit Dynamic Link Libraries auf, die es unter Oberon-System nicht gibt. Genauere Details dazu und zum gesamten Compiler sind in [LEI00] beschrieben.

Danach wurde dieser Compiler auf 32-Bit Windows portiert. Dabei mußte die gesamte Code-generierung geändert werden, weil die Objektdateien in 32-Bit Windows ein anderes Format haben. Allerdings wurden auch einige andere nützliche Funktionen eingebaut, wie zum Beispiel VAR-Parameter, die nicht verändert werden können, oder spezielle Techniken für den Aufruf von Windows-API-Funktionen.

Ähnlich wie im Oberon-System verarbeitet der Oberon-2 Compiler von Pow! Quelldateien mit Oberon-2 Syntax und erzeugt daraus Objekt- und Symboldateien (siehe Abbildung 20). Im Gegensatz zum Oberon-System werden aber die Objektdateien nicht direkt geladen (vgl. [BCF95] und [MOE94]), sondern von einem Linker zu ausführbaren Dateien gebunden. Die Kombination aus Oberon-2 Compiler und Linker in Pow! kann daher direkt ausführbare Dateien für Windows erzeugen und zwar sowohl Programme als auch Dynamic Link Libraries. In Pow! wird daher kein eigener Lader für die Ausführung der Oberon-Programme benötigt. Diese Aufgabe kann dem Lader des Betriebssystems übertragen werden.

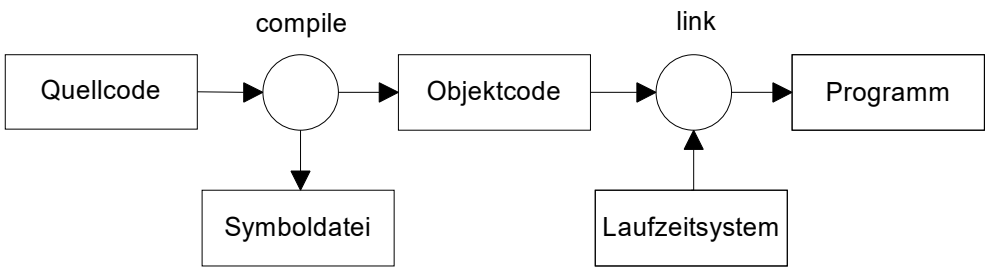


Abbildung 21: Übersetzen und Binden eines Oberon-2 Programmes

Die Symboldateien haben zur Folge, daß man die Module eines Oberon-2 Programmes in einer bestimmten Reihenfolge übersetzen muß. Dies kann man am besten anhand eines kleines Beispiels demonstrieren. In diesem Beispiel besteht das Programm Test.exe aus zwei Modulen, dem Modul 1 auf der linken und dem Modul 2 auf der rechten Seite. Modul 1 importiert dabei Funktionen des Moduls 2. Die folgende Grafik zeigt nun den Ablauf der Erzeugung des Programmes.

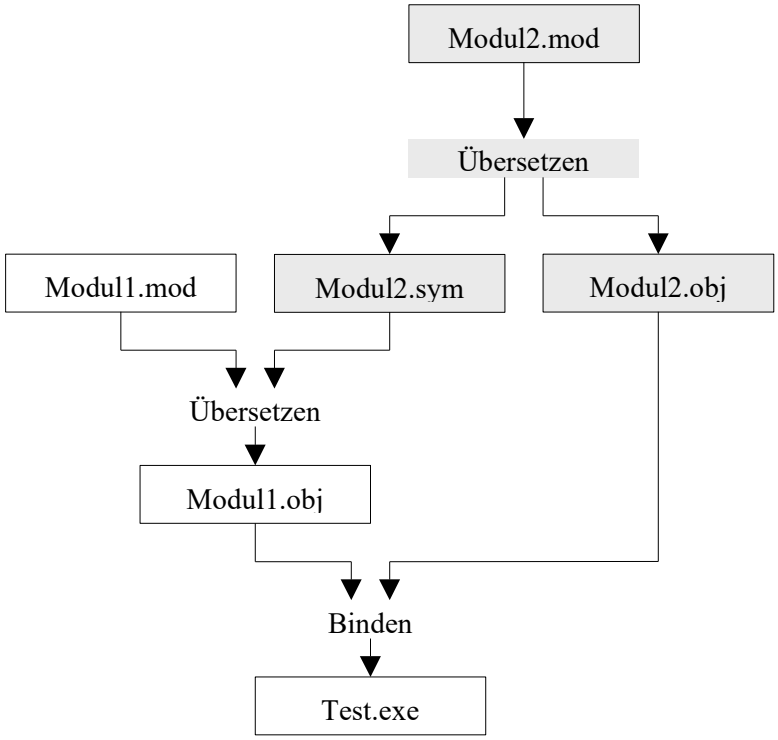


Abb. 22: Übersetzen und Binden zweier Oberon-2 Module

Anhand der Darstellung des Ablaufes kann man erkennen, daß Module in einer gewissen Reihenfolge übersetzt werden müssen. Als erstes muß das importierte Modul übersetzt werden und danach das importierende Modul. In diesem Beispiel muß also zuerst das Modul 2 und danach das Modul 1 übersetzt werden.

Die Compiler-Schnittstelle muß demnach die Reihenfolge bestimmen, in der die Module übersetzt werden müssen. Dazu muß sie wissen, welche Abhängigkeiten zwischen den Modulen bestehen, und muß die Module so reihen, daß jene Module, die keine anderen Module importieren, als erstes übersetzt werden. Danach kommen jene Module dran, die nur jene Module importieren, die im ersten Schritt übersetzt wurden. Das läßt sich insofern fortsetzen, daß in einem Schritt immer nur jene Module übersetzt werden, die nur solche Module importieren, die bisher schon übersetzt wurden. Als letztes wird das Hauptmodul übersetzt, weil es alle anderen Module benötigt.

Für die Ermittlung der Abhängigkeiten zwischen den Modulen gibt es eine Unterstützung durch den Oberon-2 Compiler. Mit speziellen Parametern aufgerufen, teilt er mit, welche Module von einem angegebenen Modul importiert werden. Die Compiler-Schnittstelle erhält also vom Compiler eine Liste der importierten Module. Um eine Übersicht über die Abhängigkeiten zwischen den Modulen zu bekommen, fragt sie den Compiler für jede im Projektdatei eingetragene Oberon-2 Quelldatei nach der Liste der importierten Module.

Neben dem Compiler sind noch weitere Komponenten für das Erzeugen lauffähiger Dateien notwendig. Diese werden im folgenden beschrieben.

4.1.1 Das Laufzeitsystem (Runtime System, RTS)

Eine der wichtigsten Komponenten neben einem Compiler ist das Laufzeitsystem. Es handelt sich dabei um eine Sammlung von Funktionen, die zu jedem Programm gebunden werden, ohne die es nicht lauffähig wäre. Diese Funktionen werden von jenem Code aufgerufen, den der Compiler generiert. Das bedeutet, daß der Compiler bei gewissen Funktionen nicht vollständigen Code generiert, sondern statt dessen nur Funktionen des Laufzeitsystems aufruft. Dies hat den Vorteil, daß die Funktionen des Laufzeitsystems leicht verändert werden können, ohne daß der Compiler geändert werden muß.

Typische Beispiele für Funktionen in einem Laufzeitsystem sind Funktionen zur Speicherverwaltung und zur Prüfung von Laufzeitfehlern (Runtime Checks) und Ausgabe von Fehlermeldungen. Dies wurde auch bei der Implementierung des Laufzeitsystems für Oberon-2 eingehalten. Es enthält im wesentlichen Funktionen zur Speicherverwaltung, zur Garbage Collection und zur Ausgabe von Fehlermeldungen.

Die erste Version des Laufzeitsystems war die 16-Bit Version. Nur wenige Teile des 16-Bit Laufzeitsystems konnten für die 32-Bit Version adaptiert werden. Wegen der unterschiedlichen Systemarchitektur (API, Scheduling, Protection) war ein Redesign und eine fast komplette Neuentwicklung erforderlich.

Das Laufzeitsystem steht in zwei Varianten zur Verfügung, nämlich als Dynamic Link Library und als statische Bibliothek. Der Vorteil der DLL-Variante liegt darin, daß das Laufzeitsystem nur einmal in den Speicher geladen werden muß, auch wenn sich mehrere Programme im Speicher befinden, die es verwenden. Von jeder der beiden Varianten gibt es noch jeweils zwei Untervarianten, nämlich eine mit Garbage Collector und eine ohne. Die folgende Tabelle listet alle Varianten mit den dazugehörigen Dateinamen auf.

	Dynamic Link Library	statische Bibliothek
mit GC	Rts32d.dll Rts32d.lib	Rts32s.lib
ohne GC	Rts32dgc.dll Rts32dgc.lib	Rts32sgc.lib

4.1.1.1 Schnittstelle zum Compiler

Der Compiler erwartet eine bestimmte Sammlung von Funktionen im Laufzeitsystem. Man kann sie in drei Kategorien einteilen:

- Initialisierungs- und Terminierung
- Speicherverwaltung
- Fehlermanagement

Die Funktionen der drei Kategorien werden im folgenden detailliert beschrieben:

4.1.1.2 Funktionen zur Initialisierung und Terminierung

Um diese Funktionen erklären zu können, muß kurz geschildert werden, wie die Initialisierung eines Moduls in Oberon-2 vor sich geht. Nähere Details zu diesem Thema kann man in [LEI00] finden.

Das Betriebssystem erwartet einen Eintrittspunkt in jedem Programm oder in jeder Dynamic Link Library (DLL). Dabei handelt es sich um die Position, an der das Programm gestartet werden soll, bzw. bei einer DLL um die Funktion, die zur Initialisierung aufgerufen werden soll. Die Funktion kann grundsätzlich jeden beliebigen Namen haben. Üblich sind die Namen WinMain für Programme und DllEntryPoint für DLLs. Der Oberon-2 Compiler erwartet diese beiden Namen im Hauptmodul eines Programmes oder einer DLL, weil er speziellen Initialisierungscode für diese beiden Funktionen generiert. Aus Kompatibilitätsgründen gibt es auch noch eine Funktion namens ProgMain, die vom Compiler ebenfalls als Startfunktion angesehen wird und die Generierung speziellen Initialisierungscodes nach sich zieht.

Jedes Modul in Oberon-2 kann einen Block enthalten, der durch die Schlüsselworte BEGIN und END eingeschlossen ist, in dem beliebige Befehle enthalten sein können (siehe folgendes Codestück aus dem Modul RTSOberon). Diese Befehle sollen aufgeführt werden, bevor irgendeine Funktion des Moduls aufgerufen wird. Sie dienen daher hauptsächlich der Initialisierung des

Moduls. Die Aufgabe des vom Compiler generierten Initialisierungscode liegt gerade in der Ausführung dieser Befehle.

```
MODULE RTSOberon;
VAR
    currentID:   LONGINT;  (* ID counter for unique ID *)
    gcActive:    BOOLEAN;
    gcMem:       LONGINT;  (* memory allocated since last gc *)
    logActive:   BOOLEAN;
    memBlockN:   LONGINT;
    moduleN-:    LONGINT;  (* number of modules loaded *)
...
BEGIN (* Modulinitialisierung *)
  currentID := 0;
  gcActive  := GC_ACTIVE;
  gcMem     := 0;
  logActive := LOG_ACTIVE;
  memBlockN := -1;
  moduleN   := 0;
END RTSOberon.
```

Der generierte Initialisierungscode ruft zu diesem Zweck die Initialisierung des obersten Moduls auf. Als oberstes Modul wird jenes Modul angesehen, das eine der speziellen Funktionen WinMain, DllEntryPoint oder ProgMain enthält. Eine weitere Bedingung für das oberste Modul ist, daß es von keinem anderen Modul importiert wird. Dagegen kann es seinerseits beliebig viele andere Module importieren.

Der Code für die Initialisierung des obersten Moduls besteht wie für alle anderen Module aus drei Teilen. Zuerst wird die Funktion *InitModule* aus dem Laufzeitsystem aufgerufen. Als zweiter Schritt wird die Modulinitialisierung aller importierten Module aufgerufen. Erst als letzter Schritt werden die Befehle der Modulinitialisierung des obersten Moduls ausgeführt.

Durch den zweiten Schritt wird sichergestellt, daß alle Module initialisiert werden. Im zweiten Schritt werden nämlich alle Module initialisiert, die das oberste Modul importiert. Da bei der Initialisierung dieser Module ebenfalls alle von ihnen importierten Module initialisiert werden, werden somit alle Module quasi rekursiv initialisiert. Zeichnet man die Modulabhängigkeiten als Baum auf, dann werden durch dieses Vorgehen zuerst die Blätter initialisiert und danach die Knoten.

Beinahe derselbe Ablauf wird ausgeführt, wenn eine DLL geladen wird. Allerdings wird eine weitere Funktion des Laufzeitsystems am Ende der Initialisierung - quasi als vierter Schritt - aufgerufen. Es handelt sich dabei um die Funktion *InitDLL*.

Im folgenden werden nun die beiden Initialisierungsfunktionen beschrieben, die vom Initialisierungscode des Compilers aufgerufen werden.

Funktion InitModule(mda)

Wie schon beschrieben wird diese Funktion von jeder Modulinitialisierung aufgerufen. Dies gibt dem Laufzeitsystem die Chance, alle Module zu registrieren. Und tatsächlich führt diese Funktion über alle Aufrufe Buch und vermerkt alle Module in einer internen Tabelle.

Als Parameter bekommt die Funktion die Adresse des Moduldeskriptors. Ein Moduldeskriptor ist ein Record, der das Modul beschreibt. Er ist einem Typdeskriptor sehr ähnlich und wird ebenfalls vom Compiler generiert und in den Objektcode eingefügt. Die folgende Typdeklaration zeigt den Aufbau des Moduldeskriptors.

```
ModuleDescriptorT = RECORD
  moduleName-:      POINTER TO ARRAY MAX_ID_LEN OF CHAR;
  globalDataSize-:  LONGINT;
  globalData-:      SYSTEM.PTR;
  typetagList-:     POINTER TO ARRAY OF SYSTEM.PTR;
  commandList-:     POINTER TO ARRAY OF SYSTEM.PTR;
END;
```

Im Feld *moduleName* findet man einen Zeiger auf den Modulnamen. Die nächsten beiden Felder *globalDataSize* und *globalData* geben die Größe und die Lage des globalen Datensegments an. Wie in [LEI00] zu erfahren ist, hat jedes Modul sein eigenes Datensegment, in dem die globalen Variablen des Moduls gespeichert sind. Das Feld *typetagList* ist interessant. Es enthält nämlich einen Zeiger auf eine Liste von Typetags des Moduls. Typetags sind Zeiger auf Typdeskriptoren. Somit kann man über dieses Feld alle Record-Typen des Moduls erfahren.

Das letzte Feld *commandList* wird eher selten benötigt. Es zeigt auf eine Liste von parameterlosen Funktionen. Diese sind vor allem im Oberon-System von Bedeutung. In Pow! haben sie praktisch keine Bedeutung.

Funktion InitDLL(mda, instanceDLL, reason, reserved)

Diese Funktion wird nur für Dynamic Link Libraries benötigt. Sie wird vom Initialisierungscode, der für die Funktion *DllEntryPoint* generiert wird, aufgerufen. Der Aufruf findet nach der

Initialisierung des Moduls statt. Der Parameter *mda* ist wie vorher die Adresse des Moduldeskriptors. Die anderen Parameter sind dieselben Parameter, die von Windows an die Funktion *DllEntryPoint* übergeben werden. Bei *instanceDLL* handelt es sich um den Instance Handle der DLL. Dies ist eine Zahl, die die DLL im System eindeutig kennzeichnet. *Reason* ist ein Code für den Grund des Aufrufs. Da die Funktion nur bei der Initialisierung aufgerufen wird, kann hier als Grund nur das Laden einer DLL übergeben werden. Der letzte Parameter namens *reserved* wird nicht benutzt.

```
Funktion LeavingWinMain()
```

Eine Funktion, die bisher noch nicht erwähnt wurde, ist *LeavingWinMain*. Sie wird aufgerufen, wenn die Funktion WinMain verlassen wird. Der Compiler generiert für die Funktion *WinMain* also nicht nur speziellen Eintrittscode sondern auch speziellen Austrittscode. Dazu gehört eben auch der Aufruf dieser Funktion.

4.1.1.3 Funktionen zur Speicherverwaltung

Die Speicherverwaltung ist der zweite und fast wichtigste Teil des Laufzeitsystems. Bei der Überlegung, welche Funktionen das Laufzeitsystem dafür zur Verfügung stellen soll, muß man sich anschauen, von welchen Befehlen die Speicherverwaltung benötigt wird. Im Oberon-System gibt es nur einen Befehl, nämlich NEW, mit dem man einen Speicherbereich reserviert. Die Freigabe nicht mehr benötigten Speichers geschieht vollständig ohne Zutun des Programmierers mit Hilfe des Garbage Collectors. In Pow! wurde allerdings der Befehl DISPOSE hinzugefügt, mit dem man Speicher freigeben kann, weil es zu Beginn der Entwicklung keinen Garbage Collector gab.

Geht man von diesen beiden Befehlen aus und schaut sich den vom Compiler generierten Code an, dann kommt man sehr schnell drauf, daß das Laufzeitsystem drei Funktionen zu exportieren hat. Zwei davon, nämlich die Funktionen New und *SysNew* sind sich sehr ähnlich. Sie werden für den Befehl NEW benötigt. Welche der beiden Funktionen verwendet wird, hängt von der angelegten Datenstruktur ab. SysNew wird für offene Arrays verwendet, während New für alle anderen Datenstrukturen verwendet wird. Die letzte Funktion des Laufzeitsystems ist *Dispose* und wird - wie leicht zu vermuten ist - für den Befehl DISPOSE verwendet.

```
Funktion New(typetag, size, adr)
```

Diese Funktion wird bei jedem Vorkommen des Befehls NEW außer für offene Arrays aufgerufen. Sie ist dafür zuständig einen Speicherbereich zu reservieren und die Adresse des reservierten Speicherbereichs zurückzugeben. Wie groß der reservierte Speicherbereich sein soll, erfährt die

Funktion durch den zweiten Parameter *size*. Er enthält die Anzahl der Bytes, die zu reservieren sind. Dieser Parameter kann auch negative Werte enthalten. Das macht auf den ersten Blick keinen Sinn. Es hat aber eine spezielle Bedeutung. Wird ein negativer Wert übergeben, dann sollen nämlich genau so viele Bytes reserviert werden, wie der absolute Wert von *size* angibt. Zusätzlich soll der reservierte Speicherblock mit Null initialisiert werden. Die Initialisierung des reservierten Speicherbereiches entfällt dagegen für positive Werte.

Als erster Parameter wird ein Typetag übergeben, also ein Zeiger auf einen Typdeskriptor. Nähere Informationen zu Typdeskriptoren findet man in [LEI00]. An dieser Stelle sei nur so viel erklärt, daß der Typdeskriptor den Datentyp eines Speicherbereiches bestimmt.

Die Übergabe des Typetag informiert die Funktion New darüber, welchen Typ der Speicherbereich nach der Reservierung bekommen soll. Dazu muß die Funktion New dafür sorgen, daß vor dem reservierten Speicherbereich die Adresse des Typdeskriptors eingetragen wird. Sie kopiert also die vier Byte lange Adresse vier Bytes vor den Beginn des Speicherbereichs. Genau an dieser Stelle erwartet der Compiler auch die Adresse des Typdeskriptors für seine Typprüfungen. Die Reservierung dieser zusätzlichen vier Bytes obliegt im übrigen völlig der Funktion New und ist nicht im Parameter *size* enthalten. Wenn die Funktion New also 16 Bytes laut Parameter *size* anlegen soll, dann werden in Wirklichkeit 20 Bytes angelegt. Wie bei der späteren Beschreibung der internen Abläufe der Speicherverwaltung noch zu sehen sein wird, werden in Wirklichkeit mehr Speicher angelegt.

Zuletzt bleibt noch zu erwähnen, daß die Adresse des reservierten Speicherbereichs im Parameter *adr* zurückgegeben wird. Die Adresse zeigt auf das erste freie Byte und nicht auf den Typetag. Dieser wird an der Position *adr* - 4 gespeichert.

Funktion SysNew(size, adr)

Die Funktion SysNew ist sehr ähnlich der Funktion New. Sie wird allerdings nur für offene Arrays verwendet. Daher hat sie auch eine leicht unterschiedliche Schnittstelle. Ihr fehlt der erste Parameter *typetag*. Die beiden anderen Parameter verhalten sich dagegen sehr ähnlich. *size* gibt die Größe des zu reservierenden Speicherbereichs an und in *adr* wird die Adresse des Speicherbereichs zurückgegeben.

Funktion Dispose(adr)

Die Funktion Dispose ist, wie bereits beschrieben, notwendig geworden, weil es zu Beginn der Entwicklung keinen Garbage Collector gab. Sie wird für jedes Vorkommen des Befehls DISPOSE verwendet und hat die Aufgabe einen vorher angelegten Speicherbereich freizugeben.

4.1.1.4 Funktionen für die Behandlung von Laufzeitfehlern

Die letzte Gruppe von Funktionen, die vom Laufzeitsystem erwartet werden, enthält nur eine Funktion, nämlich die Funktion Halt. Sie wird immer dann aufgerufen, wenn zur Laufzeit eine Situation eintritt, die nicht mehr vernünftig zu meistern ist und daher eine Fehlermeldung auszugeben ist. Ein gutes Beispiel dafür ist ein Indexüberlauf bei einem Arrayzugriff. Um solche Situationen erkennen zu können, kann der Compiler bei Bedarf speziellen Prüfcode generieren. Um den Indexüberlauf zu erkennen, generiert der Compiler zum Beispiel bei jedem Arrayzugriff Code, der den Index zur Laufzeit mit der Größe des Arrays vergleicht. Ist der Index größer als die Anzahl der Arrayelemente, dann wird die Funktion Halt aufgerufen.

Überdies wird die Funktion auch aufgerufen, wenn der Programmierer den Befehl HALT verwendet. In manchen Situationen kann dies sehr nützlich sein. Auch das Laufzeitsystem bedient sich der Funktion Halt. Die Speicherverwaltung ruft zum Beispiel HALT auf, wenn sie einen defekten Heap vorfindet.

```
Funktion Halt(mda, lineHaltCode)
```

Diese Funktion wird bei jedem Laufzeitfehler oder bei jedem HALT-Befehl aufgerufen. Der erste Parameter ist der Moduldeskriptor jenes Moduls, in dem der Laufzeitfehler aufgetreten ist oder in dem der Befehl HALT kodiert wurde. Die Struktur des Moduldeskriptors wurde im Abschnitt über die Initialisierungsfunktionen näher erläutert. Der zweite Parameter *lineHaltCode* ist ein 32-Bit Wert, der zwei 16-Bit Werte enthält. Im unteren 16-Bit Wort ist ein Code enthalten, im oberen 16-Bit Wort steht die Zeilennummer, in dem der Laufzeitfehler aufgetreten ist oder der HALT-Befehl kodiert wurde. Der Code gibt darüber Auskunft, aus welchem Grund die Funktion Halt aufgerufen wurde. Folgende Werte sind vordefiniert:

Code	Laufzeitfehler
-1	value out of range
-2	index out of range
-3	arithmetic overflow
-4	wrong dynamic type
-5	wrong module version
-6	wrong definition file (INIT in EXPORTS)
-7	WITH trap (no ELSE included)
-8	CASE trap (no ELSE included)
-9	no result returned
-10	ASSERT fault
-11	referenced pointer is nil (assignments can also cause pointer dereferencing because of type checks)

Weitere Codes sind innerhalb des Laufzeitsystems definiert und werden auch von diesem verwendet. Während Laufzeitfehler negative Codes haben, werden für Fehler des Laufzeitsystems positive Zahlen verwendet.

4.1.1.5 Speicherverwaltung

In den vorigen Unterkapiteln wurde die Schnittstelle zur Speicherverwaltung beschriebene. Insbesondere die vom Laufzeitsystem zur Verfügung gestellten Funktionen wurden erläutert. In diesem Abschnitt soll es nun darum gehen, wie diese Funktionen im Detail implementiert sind.

Zu Beginn muß natürlich erklärt, warum überhaupt eine eigene Speicherverwaltung notwendig ist. Wie in [SDK01] nachzulesen ist, stellt Windows einige Funktionen zur Speicherverwaltung zur Verfügung, die vom Laufzeitsystem direkt verwendet werden könnten. Ein wesentliches Problem mit diesen Funktionen ergibt sich erst im Zusammenhang mit dem Garbage Collector. Eine wesentliche Funktionalität für den Garbage Collector, ist die Möglichkeit, alle reservierten Speicherbereiche aufzulisten. Im 32-Bit Windows-API gibt es eine solche Funktion. Sie trägt den Namen *HeapWalk*. Sie hat allerdings den großen Nachteil, daß sie in Windows 95 nicht implementiert ist. Dies war ein wesentlicher Grund, eigene Routinen für die Speicherverwaltung zu schreiben. Außerdem haben einige Tests gezeigt, daß die eigene Speicherverwaltung für kleine Programme wesentlich schneller arbeitet als die Windows-Funktionen.

Eine wesentliche Aufgabe der Speicherverwaltung ist es also alle reservierten Speicherbereiche zu kennen. Dazu verwendet das Laufzeitsystem ein Array, in dem es alle reservierten Speicherbereiche einträgt. Dieses Array kann aber sehr schnell voll sein, da es gerade in objektorientierten Programmen üblich ist, viele kleine Speicherblöcke anzulegen. Um diesen Überlauf zu verhindern, unterscheidet das Laufzeitsystem große und kleine Speicherblöcke. Die Grenze, ab wann ein Speicherblock als groß angesehen wird, wird durch die Konstante MEMBLOCKSIZE festgelegt und ist im aktuellen Laufzeitsystem auf 65536 eingestellt. Diese Einstellung bedeutet, daß Speicheranforderungen unter 64 KB als klein betrachtet werden und alle darüber liegenden als groß angesehen werden. Der wesentliche Unterschied in der Verarbeitung großer und kleiner Speicherblöcke liegt darin, daß das Laufzeitsystem mehrere kleine Speicherblöcke zusammenfaßt und dafür nur einmal einen Speicherbereich vom Betriebssystem anfordert. Für die zusammengefaßten Speicherblöcke gibt es daher auch nur einen Eintrag im Array.

Bevor die Details der Zusammenfassung kleiner Speicherblöcke beschrieben werden, soll noch näher auf das Array eingegangen werden, in dem die Speicherblöcke eingetragen werden. Das Array hat eine maximale Größe, die durch die Konstante MAXMEMBLOCKS spezifiziert wird. Der Index des letzten benutzten Eintrages wird in der globalen Variablen memBlockN mitgeführt.

Das Array selbst ist in der globalen Variablen memBlocks untergebracht. Jeder Eintrag in diesem Array umfaßt mehrere Informationen. Eine mögliche Record-Definition kann folgende sein:

```
TYPE
  MemoryBlockT = RECORD
    size:   LONGINT;       (* size of the memory block           *)
    handle: W.HANDLE;      (* handle of the memory block         *)
    adr:    LONGINT;       (* address of the memory block        *)
    last:   LONGINT;       (* address of the last sub block      *)
    first:  LONGINT;       (* address of the first free sub block *)
  END;
```

Daraus ist zu erkennen, daß neben der Größe des Speicherbereichs, der Handle, die Adresse und zwei wichtige Zeiger in den Speicherbereich gespeichert werden. Die letzten beiden Felder *last* und *first* sind für die Zusammenfassung kleiner Speicherblöcke wichtig. Das Laufzeitsystem fordert nämlich vom Betriebssystem keine Speicherblöcke an, die kleiner als MEMBLOCKSIZE sind. Werden vom Oberon-Programm kleinere Speicherblöcke angefordert, dann wird ein Speicherblock der Größe MEMBLOCKSIZE unterteilt. Diese Unterteilung wird solange fortgesetzt, bis kein freier Speicher mehr zur Verfügung steht. Dann wird ein neuer Block der Größe MEMBLOCKSIZE angelegt. Werden größere Speicherblöcke vom Oberon-Programm angefordert, dann wird ein entsprechend großer Speicherblock vom Betriebssystem angefordert. Dies hat zur Folge, daß die relativ häufigen kleinen Speicheranforderungen sehr schnell vom Laufzeitsystem ohne Aufruf einer Windows-Funktion erledigt werden können.

Wie ein Speicherblock der Größe MEMBLOCKSIZE in kleinere Bereiche unterteilt wird, soll folgende Grafik zeigen. In dieser Grafik sind die benutzten Speicherbereiche grau eingefärbt, während die unbenutzten keinen Hintergrund haben und somit weiß sind. Die einzelnen Speicherbereiche in so einem Block werden übrigens *Subblock* genannt. Ihr Aufbau wird im folgenden noch näher erläutert. Um sprachlich zwischen dem *Subblock* und dem vom Betriebssystem angeforderten Speicherbereich unterscheiden zu können, soll ab hier der Begriff Speicherblock für *Subblock* und der Begriff Speicherbereich für den vom Betriebssystem angeforderten Speicher verwendet werden.

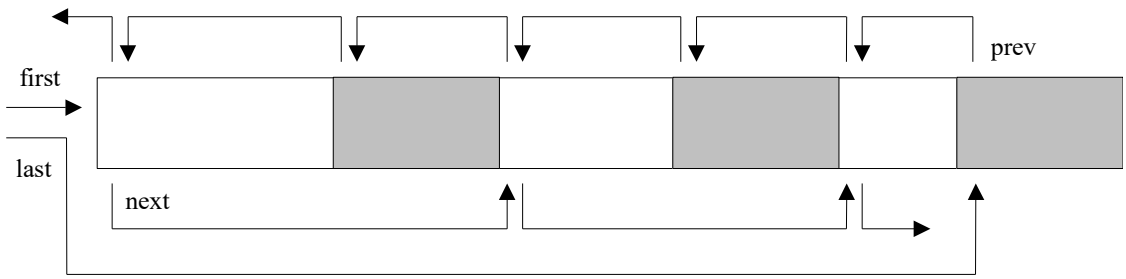


Abb. 23: Aufbau des Heap

Das Feld *last* zeigt dagegen auf den letzten Speicherblock. Dabei ist es unerheblich, ob der Speicherblock belegt oder unbenutzt ist. Innerhalb eines Speicherblockes gibt es in Analogie zum Zeiger *next* einen Zeiger *prev*, der auf den vorhergehenden Speicherblock zeigt. Wieder ist die Belegung des Speicherblocks dabei nicht von Bedeutung. Der Zeiger *last* und die Zeiger *prev* in den Speicherblöcken bilden somit eine einfach verkettete Liste aller Speicherblöcke in einem Speicherbereich.

Bisher wurde nur erwähnt, daß die beiden Zeiger *prev* und *next* in einem Speicherblock gespeichert sind. Darüber hinaus gibt es aber noch weitere Informationen, die in jedem Speicherblock abgelegt sind. Sie dienen vor allem dem Garbage Collector und sind dem Datenbereich, der vom Oberon-Programm benutzt werden kann, vorangestellt. Die folgende Typdeklaration zeigt alle diese Informationen.

```
SubBlockHeader- = RECORD
  prev:       LONGINT; (* previous sub block         *)
  next:       LONGINT; (* next free sub block        *)
  size:       LONGINT; (* size of sub block          *)
  unused:     CHAR;
  locked*:    BOOLEAN; (* is the block locked for GC *)
  used-:      BOOLEAN; (* is the block used          *)
  scanned-:   BOOLEAN; (* is the block scanned by the gc *)
  uniqueID-:  LONGINT; (* every memory block has a unique id for
                          serialization *)
  typeDesc-:  LONGINT; (* type descriptor *)
END;
```

Die beiden ersten Felder *prev* und *next* sind ja schon ausführlich beschrieben worden. Das nächste Feld gibt die Größe des Speicherblocks in Byte an. Darin ist auch der Platz, der für die hier beschriebenen Zusatzdaten benötigt wird, enthalten. Das Feld *locked* gibt an, ob dieser Speicherblock vom Garbage Collector "verschont" werden soll. Wie bei der Beschreibung des Garbage Collectors noch erklärt wird, gibt es die Möglichkeit, Speicherblöcke vom Freigeben durch den Garbage Collector auszuschließen. Diese Speicherblöcke haben den Wert TRUE in diesem Feld eingetragen. Die nächsten beiden Felder *used* und *scanned* werden vom Garbage Collector verwendet. Das Feld *uniqueID* enthält eine eindeutige Nummer. Diese wird unter anderem für die Serialisierung von objektorientierten Datenstrukturen in der Bibliothek Opal++ [LEI00] verwendet. Zu diesem Zweck zählt das Laufzeitsystem alle Speicheranforderungen in der globalen Variable *currentID* mit und trägt jeweils den aktuellen Wert der Variablen *currentID* in das Feld *uniqueID* ein. Das letzte Feld namens *typeDesc* enthält schließlich die Adresse des

Typdeskriptors. Wie schon früher beschrieben wurde, erwartet der Compiler die Adresse des Typdeskriptors direkt vor den Daten. Daher darf dieses Feld auf keinen Fall an eine andere Stelle verschoben werden. Es muß immer als letztes Feld kommen.

Nachdem nun der Aufbau eines Speicherblockes beschrieben wurde, soll nun erläutert werden, wie das Anlegen und Freigeben von Speicher vor sich geht. Dazu soll die folgende Grafik zeigen, welche Funktionen dabei im Spiel sind.

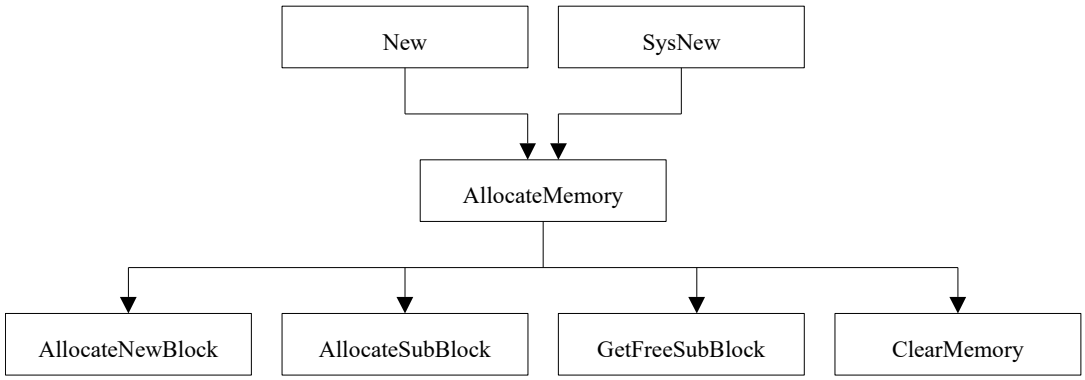


Abb. 24: Funktionen zum Reservieren eines Speicherblock

Die exportierten Funktionen *New* und *SysNew* rufen beide die Funktion *AllocateMemory* auf. Sie ist für das Anlegen eines Speicherbereichs zuständig. Dabei wird die vom Oberon-Programm angeforderte Größe um die Größe der Zusatzdaten in einem Speicherblock erhöht und auf die nächste durch vier teilbare Zahl aufgerundet. Übersteigt die Größe die in der Konstanten *MEMBLOCKSIZE* eingestellte Grenze, dann wird mit *AllocateNewBlock* ein Speicherbereich mit der ausgerechneten Größe vom Betriebssystem angefordert.

Die folgende Abbildung zeigt einen solchen Speicherbereich, der durch AllocateNewBlock angelegt wird. Die beiden Zeiger *first* und *last* zeigen jeweils auf den Beginn des Speicherbereichs. Der Speicherbereich enthält genau einen freien Speicherblock. Die Zeiger *prev* und *next* dieses Speicherblocks sind NIL, weil es keinen anderen Speicherblock gibt.

Abb. 25: Ein leerer Speicherbereich

Um innerhalb einer Speicherbereiches einen freien Speicherblock zu ermitteln, bedient sich die Funktion *GetFreeSubBlock* einer einfach verketteten Liste der freien Speicherblöcke, die mit dem Zeiger *first* beginnt. Im schlechtesten Fall muß die Funktion also alle im Array *memBlocks* eingetragenen Speicherbereiche und darin wiederum die Liste der freien Speicherblöcke durchgehen. Wie in Kürze noch genauer erläutert werden wird, tritt dieser Fall selten auf.

Findet *GetFreeSubBlock* einen passenden freien Speicherblock, dann reserviert die Funktion *AllocateSubBlock* diesen Speicherblock. Dabei prüft die Funktion, ob der freie Speicherblock groß genug ist, um geteilt werden zu können. Dies ist der Fall, wenn nach der Teilung des freien Speicherblocks im übrigbleibenden Teil noch genug Platz für Zusatzdaten und vier Byte Daten bleibt. Ist dies nicht der Fall, so macht eine Teilung keinen Sinn und der ganze freie Speicherblock wird reserviert.

Die folgende Abbildung zeigt das Ergebnis einer Unterteilung eines Speicherbereichs wie er in der vorigen Abbildung zu sehen war. Ausgangspunkt ist also ein Speicherbereich, der einen großen freien Speicherblock enthält. Dieser wird unterteilt, wobei der hintere Teil des Speicherblocks für die Speicheranforderung verwendet wird und der vordere Teil als freier Speicherblock übrigbleibt. Als Ergebnis zeigt der Zeiger *first* auf den Beginn des Speicherbereiches. Der Zeiger *next* des ersten freien Speicherblock ist NIL, da es keinen weiteren freien Speicherblock gibt. Der Zeiger *last* zeigt auf den Beginn des letzten Speicherblock, also den gerade angelegten Speicherblock. Der Zeiger *prev* zeigt auf den vorhergehenden Speicherblock, also dem freien Speicherblock.

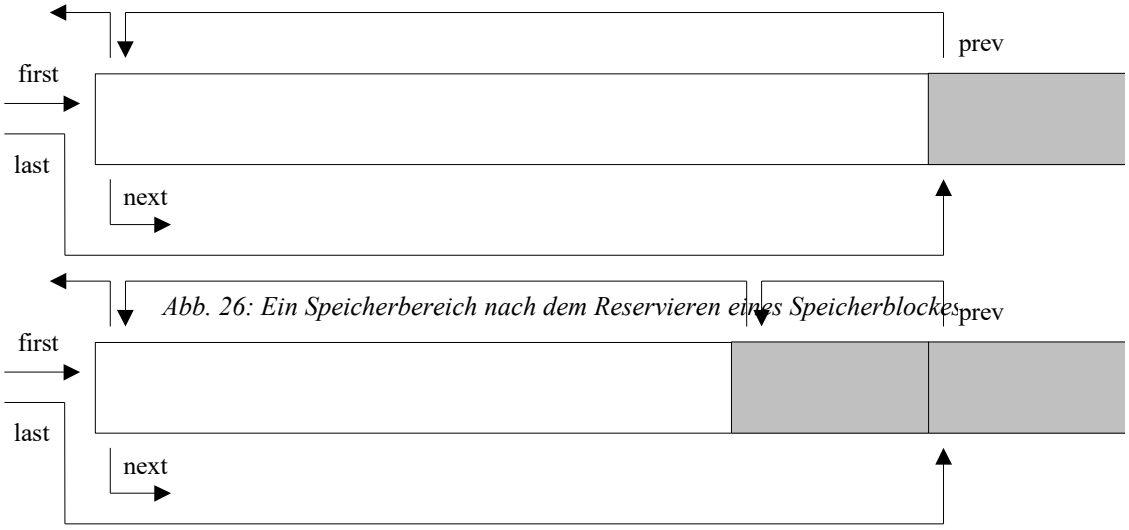


Abb. 26: Ein Speicherbereich nach dem Reservieren eines Speicherblockes

Abb. 27: Zwei reservierte Speicherblöcke in einem Speicherbereich

Aus diesen beiden Abbildungen kann man erkennen, daß das Suchen nach einem freien Speicherblock in der Regel sehr schnell geht. Durch zwei einfache Maßnahmen wird ein langwieriger Durchlauf aller Speicherbereiche vermieden.

Zunächst beginnt die Funktion immer beim zuletzt angelegten Speicherbereich zu suchen. Da die Wahrscheinlichkeit sehr hoch ist, daß dieser Speicherbereich noch genügend Platz bietet, muß in der Regel nur dieser Speicherbereich angeschaut werden. Außerdem wird durch das "First Fit"-Verfahren sichergestellt, daß im Regelfall gleich der erste freie Speicherblock benutzt wird. Im Normalfall wird also der erste freie Speicherblock im zuletzt angelegten Speicherbereich benutzt.

Der zuletzt angelegte Speicherbereich kann durch einen Zugriff auf den letzten Eintrag des Arrays memBlocks erreicht werden. Dies geschieht direkt über die globale Variable memBlockN. In diesem Speicherbereich kann über den Zeiger *first* direkt auf den ersten freien Speicherbereich zugegriffen werden. Somit sind im Regelfall für das Finden eines freien Speicherblocks nur ein Arrayzugriff und ein Zeigerzugriff notwendig. Erst wenn dieser Speicherblock zu klein wird, wird nach weiteren Möglichkeiten gesucht.

Falls *GetFreeSubBlock* allerdings keinen freien Speicherblock findet, dann wird mit der Funktion *AllocateNewBlock* ein neuer Speicherbereich vom Betriebssystem angefordert und darin ein Block für die Speicheranforderung reserviert.

Zum Schluß wird der reservierte Speicherbereich noch initialisiert, falls dies erwünscht ist. Zuständig dafür ist die Funktion *ClearMemory*.

Bis jetzt wurde nur das Anlegen von Speicher besprochen. Wie schon erwähnt gibt es im Oberon-2 von Pow! auch die Möglichkeit explizit Speicher freizugeben. Die dafür notwendigen Funktionen sind in Abbildung 28 dargestellt. Der Garbage Collector verwendet im übrigen dieselben Funktionen zum Freigeben von nicht mehr benötigten Speicher.

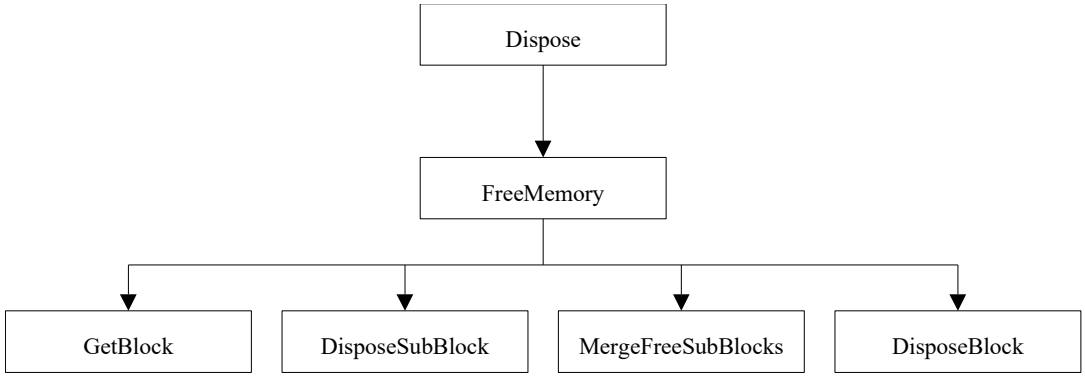


Abb. 29: Funktionen zum Freigeben eines Speicherblock

Die exportierte Funktion *Dispose* ruft direkt die Funktion *FreeMemory* auf. Sie ist im wesentlichen für das Freigeben eines Speicherblocks verantwortlich. Als erstes wird dazu der Speicherbereich gesucht, in dem der freizugebende Speicherblock liegt. Verantwortlich dafür ist die Funktion *GetBlock*, die alle im Array *memBlocks* eingetragenen Speicherbereiche durchgeht und überprüft,

ob die übergebene Adresse innerhalb des Speicherbereiches liegt. Findet die Funktion einen passenden Speicherbereich, wird der Index des Speicherbereichs in das Array *memBlocks* zurückgegeben.

Danach wird die Funktion *DisposeSubBlock* aufgerufen, die den Speicherblock tatsächlich freigibt. Dazu wird der Speicherblock als freier Block gekennzeichnet und in die verkettete Liste der freien Blöcke eingehängt.

Links und rechts neben dem gerade freigegebenen Block können unter Umständen weitere freie Blöcke existieren. Das Laufzeitsystem versucht an dieser Stelle mehrere nebeneinander liegende freie Blöcke zu einem großen freien Block zusammenzufassen. Daher wird als nächster Schritt geprüft, ob ein freier Block neben dem gerade freigegebenen Block existiert. Ist dies der Fall wird die Funktion *MergeFreeSubBlocks* aufgerufen, die drei nebeneinander liegende freie Blöcke zu einem großen freien Block zusammenfaßt. Mehr als drei nebeneinander liegende freie Blöcke können nicht auftreten, da das Zusammenfassen freier Blöcke nach jeder Speicherfreigabe durchgeführt wird. Es ist leicht zu zeigen, daß der schlimmste Fall dann auftritt, wenn ein Speicherblock freigegeben wird, der zwischen zwei freien Speicherblöcken liegt. In diesem Fall entstehen genau drei nebeneinander liegende freie Speicherblöcke.

Nachdem mehrere Speicherblöcke zu einem zusammengefaßt wurden, wird noch überprüft, ob der ganze Speicherbereich nun frei ist. In diesem Fall kann der ganze Speicherbereich dem Betriebssystem zurückgegeben werden. Dies erledigt die Funktion *DisposeBlock*.

4.1.1.6 Garbage Collector

In den letzten Jahrzehnten wurden bereits viele Artikel über das Thema Garbage Collection publiziert. Es ist daher nicht notwendig, umfassend über das Thema zu informieren. Dafür gibt es geeignete Literatur, wie zum Beispiel [JL96] und [WIL94]. Hier soll nur soweit darauf eingegangen werden als es für die Beschreibung und Erklärung der Implementierung notwendig ist.

Den Begriff "Garbage Collector" kann man wörtlich als "Müllsammler" übersetzen. Diese Übersetzung gibt schon einen Hinweis auf die Aufgaben des Garbage Collectors, nämlich Müll einzusammeln. Was aber mit Müll im Zusammenhang mit Programmen gemeint ist, beschreibt [JL96] folgendermaßen:

> *"Dynamically allocated storage may become unreachable. Objects that are not live, but are not free either, are called garbage. With explicit deallocation, garbage cannot be reused: its space has leaked away."*

Mit "Garbage" sind also Speicherblöcke gemeint, die von der Speicherverwaltung zwar als belegt angesehen werden, aber nicht mehr vom Programm benötigt und nicht mehr referenziert werden. Die Aufgabe des Garbage Collectors ist es nun, diese Speicherblöcke ausfindig zu machen und wieder frei zu geben.

Für diese Aufgabe gibt es viele verschiedene Algorithmen. Zu Beginn jeder Implementierung eines Garbage Collectors steht immer die Entscheidung, welcher davon verwendet werden soll.

Der vermutlich bekannteste Algorithmus ist das Reference Counting. Dabei bekommt jeder Speicherblock einen Zähler, der anzeigt, wie oft der Speicherblock gerade referenziert wird. Ein Wert von 5 bedeutet also, daß gerade fünf Zeiger im Programm auf diesen Speicherblock zeigen. Wenn der Wert 0 erreicht, dann weiß der Garbage Collector, daß der Speicherblock nicht mehr benötigt wird und kann ihn freigeben.

Von entscheidender Bedeutung für diesen Algorithmus ist es also, daß dieser Zähler immer den korrekten Wert enthält. Dies ist leicht für das Anlegen eines Speicherblocks zu bewerkstelligen. In diesem Fall bekommt der Zähler den Wert 1. Die Überlegung, wann sich dieser Wert verändern muß, läßt den Blick schnell auf die Zuweisung fallen. Weist man einem Zeiger einen neuen Wert zu, dann muß der Zähler des Speicherblocks verändert werden. Auf jeden Fall muß der Zähler jenes Speicherblock, dessen Adresse dem Zeiger zugewiesen wird, um eins erhöht wird. Allerdings darf auch nicht vergessen werden, daß der Zeiger vor der Zuweisung auf einen anderen Speicherblock gezeigt haben kann. In diesem Fall muß der Zähler jenes Speicherblocks, auf den der Zeiger vor der Zuweisung gezeigt hat, um eins erniedrigt werden. Diese Aufgabe kann man nicht dem Programmierer aufbürden. Daher wird diese Aufgabe meist dem Compiler überlassen. In der Regel generiert der Compiler speziellen Code für die Zuweisung von Zeigern, sodaß die beschriebene Zählermanipulierung vorgenommen wird.

Ein wesentlicher Nachteil dieses Verfahrens ist die Behandlung von zyklischen Referenzen. Wie an einem einfachen Beispiel in der folgenden Abbildung gezeigt werden kann, kann es passieren, daß Datenstrukturen nicht mehr freigegeben werden, obwohl sie nicht mehr vom Programm benötigt werden.

Abbildung 31 zeigt eine einfache zyklische Referenz. Ein Zeiger x zeigt auf ein Objekt namens y,

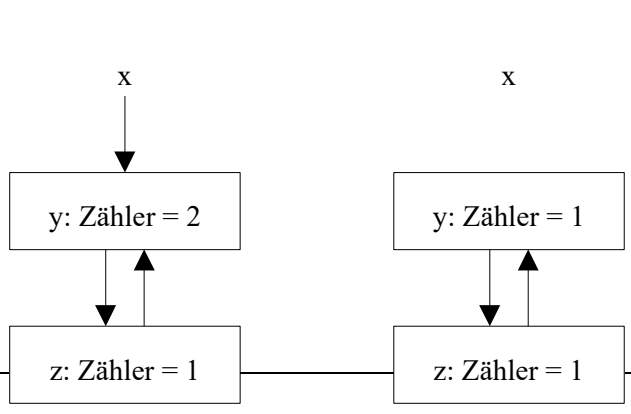


Abb. 30: Zyklische Referenz

Wird nun in dieser Situation dem Zeiger x ein neuer Wert oder der Wert NIL zugewiesen, dann wird der Zähler des Objektes y erniedrigt. Dadurch entsteht die Situation wie sie in der rechten Hälfte der Abbildung zu sehen ist. Im Objekt y und das Objekt z steht der Zähler jeweils auf 1 und beide Objekte können nicht freigegeben werden, obwohl kein Zeiger des Programmes mehr auf sie zeigt.

Ohne zusätzlichen Maßnahmen kann ein Reference Counting Algorithmus zyklisch referenzierte Objekte nicht mehr freigeben. Dies war ein Grund für die Entscheidung gegen diesen Algorithmus. Ein weiterer Grund liegt in der fehlenden Unterstützung durch den Compiler. Wie schon erläutert, muß der Compiler speziellen Code für die Zuweisung generieren. Nachdem diese Unterstützung nicht vorhanden war sondern, wie noch erläutert werden wird, die Unterstützung für ein anderes Verfahren eingebaut war, wurde das Reference Counting nicht in Erwägung gezogen.

Ein weiteres Verfahren für Garbage Collection ist "Mark and Sweep" [JL96]. Es handelt sich hierbei um einen zweistufigen Algorithmus. Im ersten Teil genannt "Mark" werden alle Speicherblöcke markiert, die vom Programm gerade benutzt werden. Alle nicht markierten Speicherblöcke werden danach im zweiten Teil des Algorithmus, genannt "Sweep", freigegeben. Was sich in diesen beiden Sätzen so einfach anhört, bedarf einiger Erklärung.

Eine entscheidende Frage bei diesem Algorithmus ist, wie man bestimmen kann, welche Speicherblöcke gerade von einem Programm benötigt werden und welche nicht. Die Antwort ist simpel, was man von der Realisierung nicht behaupten kann. Der Algorithmus muß alle im Programm verwendeten Zeiger ermitteln und jene Speicherblöcke markieren, auf die einer dieser Zeiger zeigt.

Wie aber kann man alle Zeiger eines Programmes ermitteln? Dazu muß man systematisch an das Problem herangehen und überlegen, wo Zeiger in einem Programm gespeichert werden können. Zeiger können wie alle anderen Datentypen in globalen und in lokalen Variablen gespeichert werden. Außerdem können sie in dynamisch angelegten Datenstrukturen vorkommen. Somit müssen für die Ermittlung der Zeiger die globalen, die lokalen und die dynamisch angelegten Daten untersucht werden. Dabei kann es sich um relativ große Datenbestände handeln, deren Untersuchung einige Zeit nach sich ziehen würde. Um die Analyse zu beschleunigen, baut man in der Regel in den Compiler spezielle Routinen ein, die dem Garbage Collector Zusatzinformationen zur Verfügung stellen, mit deren Hilfe er die Lage von Zeigern schneller finden kann.

Genau diese Routinen sind im Oberon-2 Compiler bereits vorhanden. Nachdem der Compiler auf dem Oberon-2 Compiler der ETH Zürich aufbaut und im Oberon-System "Mark and Sweep" als Garbage Collection Algorithmus eingesetzt wird (vgl. [PHT91] und [PCF95]), lag es nahe, in Pow! auch denselben Algorithmus zu verwenden.

Es gibt dann noch eine weitere Gruppe von Algorithmen, die eine gemeinsame Eigenschaft besitzen. Es handelt sich dabei um die große Gruppe der Algorithmen, die Speicherblöcke verschieben. Hierunter fallen zum Beispiel die "Mark-Compact Garbage Collection" und die "Copying Garbage Collection". Es gibt einen wichtigen Grund, warum diese Gruppe nicht zum Zug kommen konnte. Der Einsatz eines solchen Algorithmus hat nämlich zur Folge, daß Zeiger geändert werden müssen. Es ist klar, daß ein Zeiger verändert werden muß, wenn der Speicherblock, auf den er zeigt, an eine andere Stelle kopiert oder verschoben wird.

Dies setzt wiederum voraus, daß jeder Zeiger im Programm eindeutig als Zeiger erkannt wird. Es dürfen nicht zu wenige und nicht zu viele erkannt. Werden zu wenig Zeiger erkannt, dann kann es passieren, daß Speicher freigegeben wird, der noch benötigt wird. Dies ist für jeden Garbage Collector fatal. Werden zu viele Zeiger erkannt, dann macht es dem "Mark and Sweep" Algorithmus nichts aus. Die Konsequenz wäre lediglich, daß zu wenig Speicher freigegeben wird. Dies führt zwar nicht zum Ziel, daß möglichst viel Speicher freigegeben wird, aber das Programm wird in seiner Funktionalität nicht beeinträchtigt. In manchen Publikationen wird dieses Verfahren als konservative Garbage Collection bezeichnet (siehe [JL96]).

Bei Algorithmen, die Speicherblöcke verschieben, sieht die Lage anders aus. Da sie die Zeiger verändern müssen, ist es für einen solchen Algorithmus fatal, wenn er zu viele Zeiger als solche erkennt. In diesem Fall würde nämlich ein Wert, der gar kein Zeiger ist, aber vom Garbage Collector als Zeiger interpretiert wird, verändert werden. Dies kann und wird in der Regel das Programm beeinträchtigen und zu Fehlern führen, die nur schwer zu finden sind.

Wie noch erklärt wird, ist es dem Garbage Collector von Pow! nicht in allen Situationen möglich, einen Zeiger eindeutig zu identifizieren. Daher können Algorithmen, die Speicherblöcke verschieben, auf keinen Fall eingesetzt werden.

Nachdem nun die Entscheidung für einen Algorithmus erläutert wurde, soll nun beschrieben werden, wie der Algorithmus implementiert wurde. Als erstes wird der Frage nachgegangen, wie die Zeiger des aktuellen Programmes gefunden werden. An dieser Stelle muß allerdings darauf hingewiesen werden, daß das Laufzeitsystem mehrere Programme gleichzeitig bedienen kann. Wie schon erläutert wurde, kann das Laufzeitsystem als statische Bibliothek oder als dynamische Bibliothek gebunden werden. Im ersten Fall wird der Code des Laufzeitsystem zu jedem Programm gebunden und jedes Programm hat sein eigenes Laufzeitsystem. Jedes Laufzeitsystem verwaltet dann nur den dynamischen Speicher eines Programmes.

Wird das Laufzeitsystem dagegen als dynamische Bibliothek gebunden, dann wird das Laufzeitsystem nur einmal in Form einer Dynamic Link Library in den Speicher geladen und alle Programme teilen sich dieses Laufzeitsystem. Der Vorteil liegt klar auf der Hand. Der Code des

Laufzeitsystems muß nur einmal in den Speicher geladen werden. Der Nachteil für die Implementierung ist aber nicht sofort ersichtlich. Das Laufzeitsystem verwaltet nun den dynamischen Speicher mehrerer Programme. Für die Speicherverwaltung stellt dies kein Problem dar. Allerdings muß das Laufzeitsystem diese Tatsache beim Auffinden der aktuellen Zeiger berücksichtigen. Es muß nämlich alle laufenden Programme berücksichtigen und in allen laufenden Programmen Zeiger suchen.

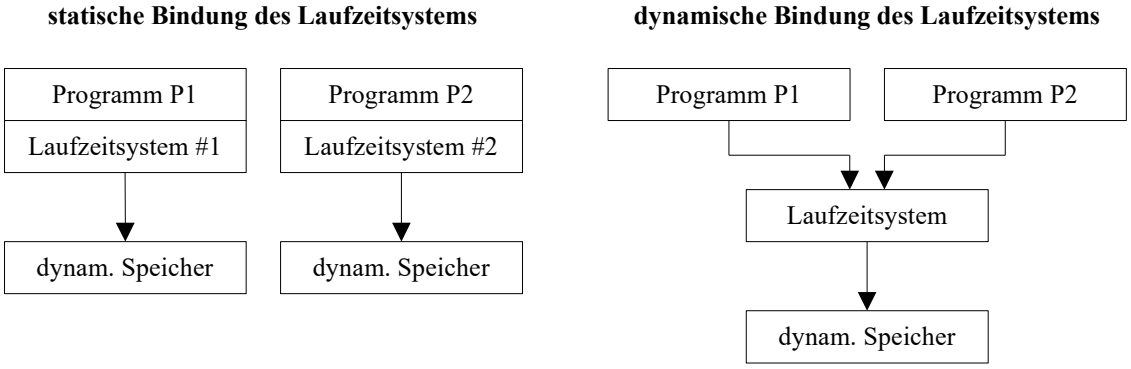


Abb. 32: Varianten des Laufzeitsystems

Um dieser Forderung nachkommen zu können, führt das Laufzeitsystem über alle laufenden Programme Buch. Dazu verwendet es die Funktion *InitModule*, die bereits in einem früheren Abschnitt beschrieben wurde. Sie wird aufgerufen, wenn ein Oberon-2 Modul initialisiert wird. Das Laufzeitsystem trägt nun die Adresse des Moduldeskriptors des aufrufenden Moduls in ein Array namens moduleList ein. Die maximale Größe dieses Arrays wird durch die Konstante *MAX-MODULES* festgelegt. Die globale Variable moduleN enthält die Anzahl der aktuell eingetragenen Module.

Das Laufzeitsystem muß also alle laufenden Programme bei der Ermittlung der Zeiger berücksichtigen. Es macht dabei keinen Unterschied zwischen statischer und dynamischer Bindung, sondern behandelt den Fall der statischen Bindung einfach als Sonderfall, bei dem nur ein Programm das Laufzeitsystem benützt.

Zur Ermittlung aller Zeiger ist, wie bereits beschrieben, die Untersuchung der globalen und lokalen Daten sowie der dynamisch angelegten Daten nötig. Im folgenden soll nun darauf eingegangen, wie die Zeiger in den drei Typen von Daten gefunden werden.

Die Analyse startet mit den globalen und lokalen Daten. Die darin enthaltenen Zeiger nennt man auch "root pointers", weil sie der Ausgangspunkt für den Garbage Collector sind und die Wurzel eines gerichteten Graphen sind, wenn man die dynamisch angelegten Objekte und ihre Zeiger

aufzeichnen würde. Man nennt daher die Menge der Zeiger in den globalen und lokalen Daten gerne auch "root set" (siehe auch [JL96]).

Für die globalen Daten ist dies relativ leicht, da der Compiler einige Unterstützung liefert. Wie schon erwähnt, liefert der Compiler einen Moduldeskriptor für jedes Modul, über den das Laufzeitsystem Buch führt. Im Moduldeskriptor sind die Adresse der globalen Daten des Moduls enthalten sowie Laufzeittypinformationen. Worum es bei den Laufzeittypinformationen geht, wird in Kürze beschrieben werden. Im wesentlichen handelt es sich um Zusatzinformationen, die dem Garbage Collector mitteilen, an welcher Stelle der globalen Daten Zeiger vorkommen. Damit kann der Garbage Collector also eindeutig jeden Zeiger in den globalen Daten feststellen.

Im ersten Schritt geht der Garbage Collector also alle gespeicherten Moduldeskriptoren durch und sucht alle Zeiger in den globalen Daten. In einem zweiten Schritt werden die lokalen Daten durchsucht. Die lokalen Daten werden auf dem Stack gespeichert. Leider generiert der vorgegebene Compiler hierfür keine Zusatzinformationen, die die Lage der Zeiger angeben würden. Daher muß der Garbage Collector eine andere Methode verwenden, die wir "heuristische Suche" genannt haben. Dabei wird der gesamte Stack Byte für Byte durchgegangen und jeder Wert als potentieller Zeiger auf einen dynamisch angelegten Speicherblock angesehen. Durch einen einfachen Vergleich des potentiellen Zeigers mit dem Adressbereich der dynamischen Speicherverwaltung kann eindeutig entschieden werden, ob der Wert auf jeden Fall verworfen werden soll. Ist der Wert nämlich außerhalb des Adressbereichs der dynamisch angelegten Speicherblöcke, dann wird er einfach verworfen. Liegt er innerhalb, wird vermutet, daß es sich um einen Zeiger handelt.

Um die Zeiger in den dynamisch angelegten Speicherblöcken finden zu können, kann der Garbage Collector wieder auf die Unterstützung durch den Compiler zurückgreifen. Wenn nämlich Records dynamisch angelegt werden, dann wird vor dem Record die Adresse des Typdeskriptors gespeichert. Wie schon der Moduldeskriptor enthält der Typdeskriptor auch Zusatzinformationen, die dem Garbage Collector helfen, Zeiger innerhalb des Records finden. Allerdings gibt es diese Unterstützung für offene Arrays nicht. In diesem Fall setzt wieder die heuristische Suche wie bei der Suche nach lokalen Zeigern ein.

Bisher wurde immer nur erwähnt, daß der Modul- und Typdeskriptor praktische Zusatzinformationen für den Garbage Collector enthält. Im Detail gilt folgendes: Als Name für die Zusatzinformationen wurde "Run Time Type Information" (RTTI) gewählt. Der genaue Aufbau und eine exakte Beschreibung ist in [LEI00] zu finden. An dieser Stelle soll nur der grundlegende Aufbau beschrieben werden.

Die RTTI ist eine Erweiterung des Modul- bzw. Typdeskriptors, die im wesentlichen Informationen über die Namen und Typen von Feldern in einem Record bzw. von Variablen in den

globalen Daten enthält. Um an die RTTI heranzukommen, muß man lediglich wissen, wo der zugehörige Modul- bzw. Typdeskriptor liegt. Der Zugriff auf die Moduldeskriptoren ist leicht, da sich das Laufzeitsystem die Adresse der Moduldeskriptoren in einer Tabelle merkt. Ebenso ist das Auffinden der Typdeskriptoren nicht schwer. Die Adresse des Typdeskriptors wird, wie ebenfalls schon mehrfach erwähnt wurde, vor den Daten eines Records gespeichert. Somit ist geklärt, wie man zu den jeweiligen Deskriptoren kommt. Der Zugriff auf die RTTI ist dann simpel: Man muß lediglich den Aufbau der Deskriptoren kennen. Die RTTI ist nämlich ein Teil der Deskriptoren. Der Aufbau der Modul- und Typdeskriptoren ist zwar nicht vergleichbar, aber die RTTI wird in ähnlicher Weise eingebettet. In beiden Fällen wird die RTTI dem Deskriptor vorangestellt. Im folgenden wird daher die RTTI nur anhand des Moduldeskriptors beschrieben. Eine Beschreibung des Typdeskriptors mit RTTI kann in [LEI00] gefunden werden.

Bei den Deskriptoren gibt es auf den ersten Blick eine Eigenheit. Die Adresse des Modul- bzw. Typdeskriptors zeigt nicht auf den Beginn der Datenstruktur, sondern mitten in die Datenstruktur hinein. Die folgende Tabelle, die den Aufbau eines Moduldeskriptors zeigt, hat daher auch negative Werte in der Spalte Offset. Diese bedeuten, daß die entsprechenden Daten vor dem Modul-deskriptor liegen.

	Offset	Größe (in Bytes)	Bedeutung
RTTI	-(6+nc*2+nn)	*nn*	Namenstabelle (wird durch eine leere Zeichenkette abgeschlossen)
	-(6+nc*2)	*nc**2	Tabelle der Datentypen
	-6	2	Größe nc der Tabelle der Datentypen in Worten (16-Bit Werte)
	-4	4	Reserviert.
Moduldeskriptor	0	45	Modulname
	45	4	Größe der globalen Daten in Bytes
	49	4	Adresse der globalen Daten
	53	4	Adresse eines offenen Arrays von Adressen aller Typdeskriptoren des Moduls
	57	4	Adresse eines offenen Arrays von Adressen aller parameterlosen Funktionen des Moduls

Die Tabelle zeigt also, daß die Adresse des Moduldeskriptors genau auf den Namen des Moduls zeigt. Davor liegt ein Feld, das für den Garbage Collector reserviert wurde und zurzeit noch nicht gebraucht wird. Vor diesem Feld befindet sich die eigentliche RTTI. Sie besteht im wesentlichen aus zwei Teilen, nämlich einer Tabelle der Datentypen und einer Tabelle der Namen. Beide Informationen beziehen sich bei einem Moduldeskriptor auf die globalen Variablen, während sie sich bei einem Typdeskriptor auf den zugehörigen Record beziehen.

Beide Tabellen in der RTTI sind quasi verkehrt herum gespeichert. Üblicherweise speichert man eine Tabelle so ab, daß an der niedrigsten Adresse das erste Element und an der höchsten Adresse das letzte Element steht. Bei der RTTI wird dies genau umgekehrt gemacht. Das erste Element ist an der höchsten Adresse und das letzte Element an der niedrigsten Adresse abgelegt. Das bedeutet zum Beispiel, daß der erste Eintrag der Datentypentabelle am Offset -8 steht. Der letzte Eintrag steht dann an der Adresse - 6 - nc * 2.

Außerdem können beide Tabellen in der RTTI beliebig lang werden. Die Größe der Tabelle der Datentypen ist in einem eigenen Feld abgespeichert, das den Namen *nc* bekommen hat. Die Größe der Namenstabelle wird durch den letzten Eintrag, einer leeren Zeichenkette bestimmt. Der Wert im Feld *nc* kann auch negativ werden. Das bedeutet, daß der Compiler eine zu große Tabelle generiert hat, die nicht mehr vollständig abgespeichert werden konnte. In diesem Fall gibt der absolute Wert von nc die Größe der Datentypentabelle an. Dieser Fall sollte in der Praxis nicht eintreten. Man kann allerdings Quelldateien bauen, die zu diesem Ergebnis führen.

Für den Garbage Collector ist nur die Datentypentabelle von Interesse. Daher wird der Aufbau der Namenstabelle im folgenden nicht mehr berücksichtigt. Die Datentypentabelle besteht aus einer Folge sogenannter Token, die jede globale Variable eines Moduls beschreibt. Die Reihenfolge entspricht dabei genau der Reihenfolge der Deklaration und der Reihenfolge, wie die Variablen in den globalen Daten abgelegt werden.

Zur Identifizierung eines Tokens werden 16-Bit Werte verwendet, die in der folgenden Tabelle zusammengestellt sind.

Datentyp	**Symbolischer Name im Quellcode**	**Token-Wert**
Interner Fehler	TYP_UNDEF	0
SYSTEM.BYTE	TYP_BYTE	1
BOOLEAN	TYP_BOOL	2
CHAR	TYP_CHAR	3
SHORTINT	TYP_SHORTINT	4
INT	TYP_INT	5
LONGINT	TYP_LONGINT	6
REAL	TYP_REAL	7
LONGREAL	TYP_LONGREAL	8
SET	TYP_SET	9
POINTER	TYP_POINTER	0DH
POINTER ("unsichtbar" [5])	TYP_HDPOINTER	F0H
PROCEDURE …	TYP_PROCTYP	0EH
ARRAY (fixe Größe)	TYP_ARRAY	100H

[5] Ein Zeiger kann unsichtbar werden, wenn er nicht exportiert wird und wenn er in einem anderen Modul als Teil einer Datenstruktur verwendet wird, das nicht mit jenem ident ist, in dem er deklariert wurde.

ARRAY (beliebige Größe)	TYP_DYNARRAY	200H
RECORD	TYP_RECORD	400H
end of RECORD	TYP_ENDRECORD	800H

Zu jedem Token kann es eine oder mehrere Token-Attribute geben, die das Token näher beschreiben und direkt im Anschluß an den Token-Wert gespeichert sind. Im Gegensatz zu den Token-Werten sind die Token-Attribute alle 32-Bit lang. Die folgende Tabelle beschreibt die Attribute aller Token.

Token	**Token-Attribute**
TYP_UNDEF	-
TYP_BYTE	Offset [6]
TYP_BOOL	Offset
TYP_CHAR	Offset
TYP_SHORTINT	Offset
TYP_INT	Offset
TYP_LONGINT	Offset
TYP_REAL	Offset
TYP_LONGREAL	Offset
TYP_SET	Offset
TYP_POINTER	Offset Token [7]
TYP_HDPOINTER	Offset
TYP_PROCTYP	Offset
TYP_ARRAY	Offset Elementgröße [8] Elementanzahl
TYP_DYNARRAY	Offset Elementgröße
TYP_RECORD	Offset Tokenliste Token"TYP_ENDRECORD"
TYP_ENDRECORD	-

Um das ganze an einem einfachen Beispiel zu veranschaulichen, soll davon ausgegangen werden, daß in einem Modul folgende beiden globalen Variablen deklariert sind.

```
flag:  BOOLEAN;
field: ARRAY 32 OF CHAR;
```

Dann würde die RTTI des zugehörigen Moduldeskriptors folgendermaßen ausschauen:

[6] Relativ zum Beginn der globalen Daten oder zum Beginn eines umgebenden Records oder Array

[7] Dieses Token beschreibt den Datentyp auf den der Zeiger zeigt. Im Falle eines Tokens TYP_RECORD, kommt hinter dem Token TYP_RECORD kein Offset, keine Tokenliste und kein Token TYP_ENDRECORD. Folgt einem Token TYP_POINTER das Token TYP_UNDEF, dann wird damit der Datentyp SYSTEM.PTR repräsentiert.

[8] Größe eines Arrayelementes inklusive Füllbytes für Alignment

Offset	Werte (Hexadezimal)	Werte	Bedeutung
-32	00 00 00 00	0	Offset eines Arrayelementes innerhalb des Arrays
-28	03 00	TYP_CHAR	Typ eines Arrayelementes
-26	20 00 00 00	32	Anzahl der Arrayelemente
-22	01 00 00 00	1	Größe eines Arrayelementes
-18	01 00 00 00	1	Offset des Arrays "field" innerhalb der globalen Daten
-14	00 01	TYP_ARRAY	Token
-12	00 00 00 00	0	Offset der Variablen "flag" innerhalb der globalen Daten
-8	02 00	TYP_BOOL	Token

Für den Garbage Collector ist hauptsächlich die Information über die Positionen aller Zeiger von Bedeutung. Alle anderen Informationen werden nicht berücksichtigt. Dementsprechend analysiert der Garbage Collector die RTTI, um die Position aller Zeiger zu finden. Dazu werden alle Vorkommen des Tokens TYP_POINTER und TYP_HDPOINTER speziell unter die Lupe genommen. Es bleibt aber auch nicht aus, quasi den Inhalt von Arrays und Records zu untersuchen, da auch darin Zeiger enthalten sein können.

Nachdem nun geklärt ist, auf welche Art und Weise die Position von Zeigern gefunden wird, kann man jetzt dazu übergehen, die Arbeitsweise des Garbage Collectors zu beleuchten. Die folgende Grafik zeigt die Funktionen, die im Laufzeitsystem am Garbage Collector beteiligt sind.

Jede Funktion einzeln zu beschreiben, würde den Rahmen dieser Arbeit sprengen. Trotzdem soll die wesentliche Arbeitsweise erläutert werden. Wi afik gut zu entnehmen ist, ist die Wurzel der Aufrufhierarchie die Funktion GC. In ihr ist der Garbage Collector implementiert.

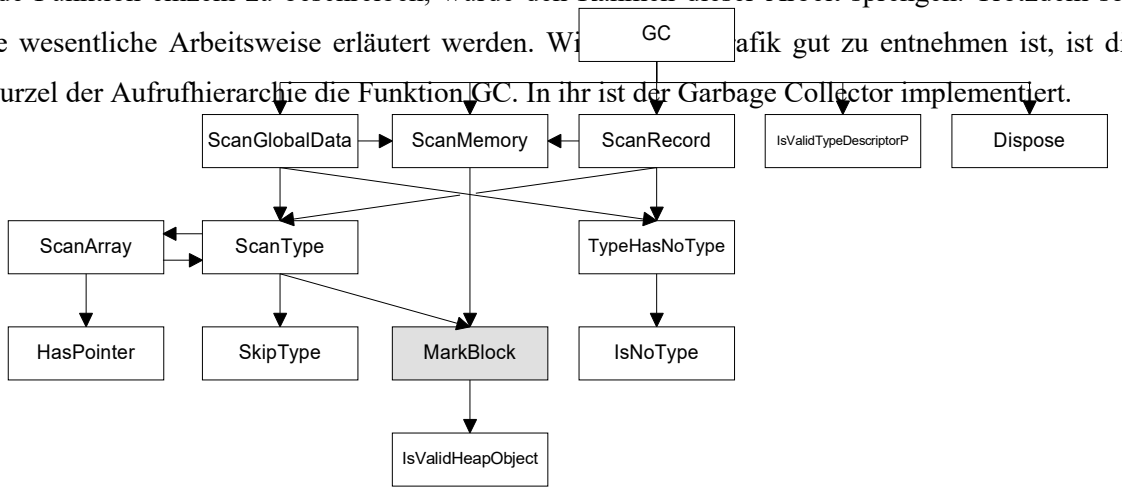


Abb. 33: Funktionen und Aufrufhierarchie des Garbage Collector

Bei der Implementierung des Garbage Collector war auf ein Problem zu achten. Es ist leicht möglich, daß es beim Durchlaufen der Zeiger und Objekte zu einem Stack-Überlauf kommt. Die

soll an folgendem abstrakt gehaltenem Codefragment gezeigt werden, das aus [JL96] entnommen ist.

```
Mark_Sweep =
  for R in Roots do Mark(R)
  Sweep()

Mark(N) =
  if mark_bit(N) == unmarked then begin
    mark_bit(N) = marked
    for M in Children(N) do Mark(M)
  end
```

Dabei wird für jeden Zeiger aus dem Root Set die Funktion Mark aufgerufen. Die Funktion Mark prüft, ob der referenzierte Speicherblock bereits referenziert wurde. Ist er noch nicht markiert, dann wird er markiert und alle Zeiger in diesem Speicherblock ermittelt. Für jeden ermittelten Zeiger wird rekursiv die Funktion Mark aufgerufen.

Insbesondere bei verketteten Listen kann dieser Algorithmus dazu führen, daß für jedes Element der verketteten Liste rekursiv die Funktion Mark aufgerufen wird. Das hat für verkettete Liste mit vielen Elementen sehr leicht zur Folge, daß der Stack für die rekursiven Aufrufe nicht mehr ausreicht und das Programm terminiert wird.

Eine solche Situation ist natürlich unter allen Umständen zu vermeiden. Daher arbeitet der Garbage Collector in mehreren Schritten und vermeidet bis auf wenige Ausnahmen rekursive Aufrufe. Die einzige Ausnahme bildet die Analyse der RTTI. Hier können rekursive Aufrufe vorkommen, wenn zum Beispiel ein Array aus Record-Elementen besteht. Die Rekursionstiefe ist dabei abhängig wie stark Array und Records ineinander verschachtelt sind. Nachdem die Verschachtelung hier nicht endlos sein kann, ist somit die Rekursionstiefe auch begrenzt. In der Praxis sind Rekursionstiefen größer als drei schon sehr unüblich.

Im ersten Schritt werden natürlich erst einmal alle belegten Speicherblöcke initialisiert. Der Garbage Collector benötigt die beiden Felder *used* und *scan*. Diese werden mit FALSE initialisiert.

Im nächsten Schritt werden alle Zeiger in den globalen Moduldaten ermittelt. Dazu werden alle Moduldeskriptoren des Arrays moduleList in einer Schleife durchgegangen. Für jedes Modul wird die Funktion *ScanGlobalData* aufgerufen. *ScanGlobalData* analysiert die RTTI des Moduldeskriptors und ruft für jeden Zeiger die Funktion *MarkBlock* auf. *MarkBlock* prüft, ob die übergebene Adresse ein gültige ist. Manchmal kommt es ja vor, daß ein Zeiger nicht initialisiert

wurde oder daß ein Zeiger NIL ist. In diesen Fällen darf die Funktion *MarkBlock* keinen Speicherblock markieren. Sonst setzt sie das Feld *used* des referenzierten Speicherblocks auf TRUE.

Als dritter Schritt werden alle Zeiger in den lokalen Daten ermittelt. Dazu wird der Stack der laufenden Programme ermittelt. Wie schon früher erwähnt, liefert der Compiler ja keine Informationen über die Parameter und Variablen am Stack. Daher muß der gesamte Stack nach Zeigern durchsucht werden, was "heuristische Suche" genannt wurde. Um diese aufwendige Suche zu vermeiden, bedient sich der Garbage Collector eines kleinen Tricks und nutzt dazu das Wissen aus, wie der Stack aufgebaut ist.

Jedesmal wenn der Compiler eine Funktion übersetzt, dann generiert er am Beginn und am Ende der Funktion speziellen Code für die Verwaltung des Stack.

```
push    ebp
mov     ebp, esp
sub     esp, <Anzahl der Bytes für die lokalen Variablen>

... (Code der Funktion)

mov     esp, ebp
pop     ebp
ret
```

Schlüsselt man die Assembler-Instruktionen auf, dann wird im wesentlichen am Beginn der Funktion das Register ebp am Stack gerettet und mit dem Inhalt des Registers esp gefüllt. Das Register esp wird dann um die Anzahl der Bytes, die für die lokalen Daten der Funktion benötigt werden, erniedrigt. Am Ende der Funktion wird das Register esp wieder mit dem Inhalt des Registers ebp gefüllt und der zwischengespeicherte Wert von ebp restauriert. Schaut man sich an, welche Struktur dadurch am Stack entsteht, wenn mehrere Funktionen aufgerufen werden, dann kommt zur Abbildung 34.

Sie zeigt, daß das Register ebp auf den Beginn der lokalen Variablen der zuletzt aufgerufenen Funktion zeigt. Schaut man sich den gesicherten Wert des Registers ebp, der an der Adresse ebp - 4 steht und in der Grafik grau hinterlegt ist, dann verweist dieser Wert auf den Beginn der lokalen Variablen der vorher aufgerufenen Funktion. Diese Kette läßt sich beliebig fortsetzen, bis man zur ersten aufgerufenen Funktion stößt. Dann trifft man auf einen Wert 0 für das Register ebp.

Diese Struktur macht sich der Garbage Collector zu Nutze und untersucht nur die Bereiche der lokalen Variablen und der Parameter. Dazu wird in einer Schleife die Funktion *ScanMemory* aufgerufen, die im Bereich der lokalen Variablen und Parameter nach Zeigern sucht. Dazu

vergleicht sie jeden Wert in diesem Bereich mit den Adressen der Speicherblöcke und ruft die Funktion *MarkBlock* auf, falls sie einen möglichen Zeiger gefunden hat. *MarkBlock* setzt dann wieder das Feld *used* des Speicherblocks auf TRUE.

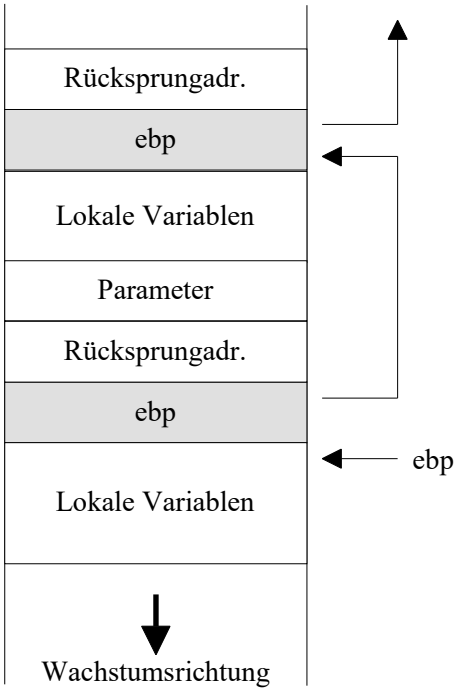


Abb. 35: Dynamischer Aufbau des Stacks

Bis hierher sind alle Speicherblöcke, die von einem Zeiger des "root set" direkt referenziert werden, markiert worden. Nun müssen aber noch jene Speicherblöcke markiert werden, die indirekt von einem Zeiger des "root set" referenziert werden. Dabei handelt es sich um jene Speicherblöcke, die von einem anderem Speicherblock referenziert werden, der selbst wiederum direkt oder indirekt von einem Zeiger des "root set" referenziert wird. Im einfachen Algorithmus aus [JL96] wird dies durch einen rekursiven Aufruf der Funktion Mark erledigt. Auf den Garbage Collector von Pow! umgelegt, hieße das, daß die Funktion *MarkBlock* rekursiv aufgerufen werden müßte. Wegen der bereits beschriebenen Gründe, sollte dies unbedingt vermieden werden.

Diese Aufgabe wird iterativ erledigt. In der Literatur gibt es dazu einige Vorschläge, wie die Rekursion vermieden werden kann. In [JL96] kann man einen Algorithmus nachlesen, der eine eigene Datenstruktur Stack verwendet. Außerdem gibt es auch zwei unabhängig voneinander entwickelte Algorithmen, die durch geschickte Manipulation der Zeiger selbst den Durchlauf durch den Graphen in linearer Zeit schaffen (siehe [SW67] und [KNU73]).

In Pow! wird dazu ein einfacher Algorithmus verwendet. Dazu wird das zweite Feld des Speicherblocks verwendet. Das Feld *scanned* gibt an, ob ein Speicherblock schon nach Zeigern durchsucht

wurde. Zu Beginn wird dieses Feld wie beschrieben mit FALSE initialisiert. Danach werden wie beschrieben die direkt referenzierten Speicherblöcke markiert. Jetzt gibt es einige Speicherblöcke, bei denen das Feld *used* auf TRUE steht und bei denen das Feld *scanned* nach wie vor auf FALSE steht.

Genau diese Speicherblöcke werden jetzt nach Zeigern durchsucht und die von diesen Zeigern referenzierten Speicherblöcke markiert. Für die durchsuchten Speicherblöcke wird das Feld *scanned* auf TRUE gesetzt. Dieser Vorgang wird solange wiederholt, bis in einem Durchlauf kein weiterer Speicherblock mehr markiert wurde. Durch eine geschickte Wahl der Durchlaufreihenfolge kann die Anzahl der Durchläufe sehr niedrig gehalten werden. Im Durchschnitt sind nicht mehr als drei Durchläufe notwendig.

Für das Durchsuchen eines Speicherblocks nach Zeigern sind die Funktionen *ScanRecord* und *ScanMemory* zuständig. *ScanRecord* wird aufgerufen, wenn der Speicherblock einen gültigen Typdeskriptor hat. Dies ist nicht der Fall, wenn der Speicherblock ein offenes Array enthält. Dann wird die Funktion *ScanMemory* aufgerufen, die wie bei den lokalen Daten den gesamten übergeben Speicherbereich nach Zeigern durchsucht. *ScanRecord* nutzt dagegen die Informationen des Typdeskriptors aus und analysiert die darin enthaltene RTTI. Wird ein Zeiger gefunden, wird wieder die Funktion *MarkBlock* aufgerufen.

Jetzt sind wir an dem Punkt angelangt, an dem alle noch "lebenden" also direkt oder indirekt referenzierten Speicherblöcke markiert sind. Alle anderen Speicherblöcke, die nicht markiert sind, werden jetzt freigegeben. Dazu wird einfach die Funktion Dispose aufgerufen.

Zum Schluß der Beschreibung des Garbage Collectors bleiben nur noch zwei Details über. Erstens muß erwähnt werden, daß es einen speziellen Mechanismus gibt, mit dem man Speicherblöcke quasi sperren kann. Solche Speicherblöcke gibt der Garbage Collector niemals frei, auch wenn sie nicht referenziert werden. Dies ist vor allem wegen des Betriebssystems Windows notwendig. Bei manchen Betriebssystemfunktionen muß man Speicherbereiche an Windows übergeben, mit denen dann das Betriebssystem selbständig arbeitet. Es ist vorstellbar, daß innerhalb des Oberon-Programmes kein Zeiger mehr auf diesen Speicherblock mehr zeigt, aber daß das Betriebssystem nach wie vor damit arbeitet. Würde in diesem Moment der Garbage Collector einsetzen, würde er den Speicherblock freigeben, weil er vom Oberon-2 Programm aus nicht mehr angesprochen wird. Das hätte aber zur Folge, daß die Betriebssystemfunktion nicht korrekt arbeiten kann. Daher wurde die Möglichkeit geschaffen, Speicherblöcke vom Garbage Collector auszuschließen.

Jeder Speicherblock hat dazu ein Feld *lock*. Ist dieser Wert TRUE, dann ist der Speicherblock für den Garbage Collector tabu. Mit den beiden Funktionen *Lock* und *Unlock* des Laufzeitsystems kann der Oberon-Programmierer explizit einen Speicherblock sperren oder freigeben.

Zuletzt bleibt noch die Frage zu klären, wann der Garbage Collector eigentlich aktiv ist. Der Garbage Collector ist zurzeit nicht als eigener Thread ausgelegt, sondern wird vor dem Anlegen eines Speicherblocks aufgerufen. Allerdings wird dies nicht bei jedem Speicherblock gemacht, sondern nur wenn ein bestimmtes Maß an Speicher angelegt wurde. In der aktuellen Version des Laufzeitsystem wird der Garbage Collector immer dann aufgerufen, wenn seit dem letzten Aufruf mehr als 64 KB Speicher allokiert wurden. Die Variable *gcMem* merkt sich dabei, wieviel Speicher angelegt wurde und die Konstante GC_*MEM* gibt die Grenze an, bei der der Garbage Collector wieder aktiviert werden soll.

Der Programmierer kann auf Wunsch den Garbage Collector auch ausschalten. Dazu exportiert das Laufzeitsystem die beiden Funktionen *GCEnable* und *GCDisable*. Die erste schaltet ihn ein und die zweite schaltet ihn aus.

4.1.2 Startup

Fast jede Programmierumgebung verwendet Startup-Module. Es handelt sich dabei um Module, die zu jedem Programm gebunden werden. In der Regel liegt der Eintrittspunkt in einem solchen Startup-Modul. Dadurch wird das Startup-Modul als erstes ausgeführt, wenn das Programm aufgerufen wird, und es hat damit die Möglichkeit, gewisse Initialisierungsaufgaben zu erfüllen. Danach wird meist das Hauptprogramm der jeweiligen Programmiersprache aufgerufen. In einem C-Programm wäre dies zum Beispiel die Funktion main. In Oberon-2 ist dies die Funktion *WinMain*.

Genauso verhalten sich auch die Startup-Module von Oberon-2. Allerdings gibt es gleich zwei verschiedene Varianten davon, nämlich für sogenannten Konsolenanwendungen und für GUI-Applikationen. Mit Konsolenanwendungen sind Programme gemeint, die ihre Ausgabe in ein Textfenster ähnlich dem DOS-Fenster schreiben. Man spricht in diesem Zusammenhang gerne auch von Textmodus-Applikationen, weil sie nur Text ausgeben können. Im Gegensatz stehen GUI-Applikationen. Sie arbeiten mit Fenstern als Ausgabemedium und können neben Text auch Grafiken ausgeben. Die folgende Tabelle schlüsselt die beiden Varianten auf.

Modul	Verwendung
obcon.mod	Konsolenanwendung
obgui.mod	GUI-Applikation

Der Programmierer muß die beiden Startup-Module nicht in sein Projekt aufnehmen. Die Compiler-Schnittstelle für Oberon-2 sorgt selbst dafür, daß das richtige Startup-Modul zu einem Programm gebunden wird. Welche der beiden Varianten verwendet wird, hängt von der Einstellung im Feld Target des Dialog "Linker Setup" (siehe Abbildung 36) ab. Ist dort als Target Console

ausgewählt, dann wird das Modul obcon zum Programm gebunden. Bei der Auswahl des Targets Windows GUI wird das Modul obgui verwendet und bei Auswahl des Targets DLL wird gar kein Modul gebunden.

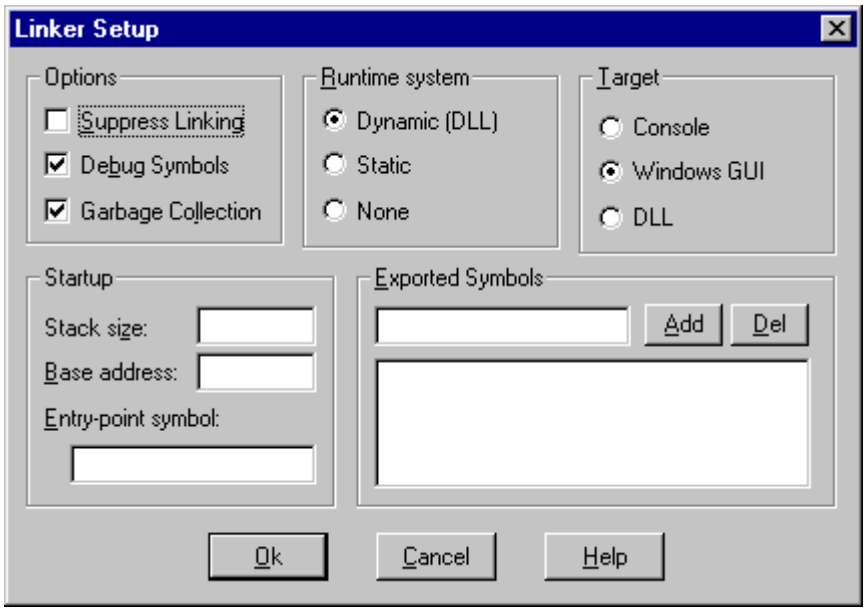


Abb. 37: Linker-Einstellungen für Oberon-2Beide Module exportieren eine parameterlose Funktion namens ExeEntryPoint. Dabei handelt es sich um den Standardnamen des Eintrittspunktes in ein Programm. Die Compiler-Schnittstelle sorgt dafür, daß der Linker diese Funktion als Eintrittspunkt behandelt.

Das Modul obcon öffnet als erstes ein Konsolenfenster und ermittelt danach die an das Programm übergebene Befehlszeile. Danach wird die Funktion *WinMain* aufgerufen. Nach der Funktion *WinMain* wird das Konsolenfenster geschlossen und das Programm beendet. Ähnlich verläuft die Arbeit auch beim Modul obgui. Der entscheidende Unterschied ist natürlich, daß obgui kein Konsolenfenster öffnen muß.

4.1.3 Linker

Der Linker hat die Aufgabe, alle Objektdateien eines Projektes, sowie alle notwendigen Bibliotheken, das Laufzeitsystem und das Startup-Modul zu einem Programm zu binden. Die Compiler-Schnittstelle verwendet dafür schon seit Beginn des Projektes ein eigenständiges Programm. Sie kümmert sich lediglich darum, alle notwendigen Informationen für den Linker zusammenzutragen und den Linker korrekt aufzurufen.

Den Linker gibt es einerseits als eigenständiges Programm und andererseits als Dynamic Link Library. Die Compiler-Schnittstelle verwendet die DLL-Version. Im wesentlichen exportiert der Linker eine Funktion namens Link32 mit einer langen Liste von Parametern. Eine der Hauptaufgaben der Compiler-Schnittstelle ist es diese große Anzahl von Parametern mit korrekten Werten zu füllen.

Einer der Parameter erwartet zum Beispiel alle zu bindenden Objektdateien. Um diese zu ermitteln, sucht sich die Compiler-Schnittstelle unter anderem alle im Projekt eingetragenen Module zusammen und ermittelt die Namen der dazugehörigen Objektdateien durch Ändern der Dateinamensendung von *mod* auf *obj*. Zusätzlich wird auch die benötigte Startup-Datei in die Liste aufgenommen. Welche der beiden genommen wird, hängt von den Linker-Einstellungen ab. Außerdem werden auch alle Dateien mit der Endung *obj* unverändert in die Liste aufgenommen. Das bedeutet, daß Quelldateien aus anderen Programmiersprachen problemlos in Pow! eingebunden werden können.

Ähnlich verhält es sich mit den Ressourcendateien. Sind im Projekt Dateien mit der Endung *rc* eingetragen, dann werden diese in die Liste der Ressourcendateien aufgenommen, allerdings nicht ohne vorher die Endung auf *res* zu ändern. Dateien mit der Endung *res* werden dagegen unverändert in diese Liste übernommen. Das bedeutet, daß übersetzte Ressourcendateien von anderen Werkzeugen in Pow! mit eingebunden werden können.

Schließlich muß die Compiler-Schnittstelle noch die benötigten Bibliotheken ermitteln. Standardmäßig fügt die Compiler-Schnittstelle einige Bibliotheken des Windows API automatisch hinzu. Dabei handelt es sich um die Bibliotheken kernel32.lib, user32.lib, gdi32.lib, comdlg32.lib und win32.lib. Außerdem muß die richtige Bibliothek für das Laufzeitsystem aufgenommen werden. Wie schon in einem früheren Abschnitt beschrieben wurde, gibt es vier Varianten des Laufzeitsystem. Welche davon verwendet wird, hängt wieder von den Linker-Einstellungen des Projektes ab. Entscheidend dafür sind die beiden Felder "Runtime System" und Garbage Collection im in der Abbildung 38 dargestellten Dialog. Die folgende Entscheidungstabelle zeigt welche Bibliothek in welcher Situation verwendet wird.

Runtime System	Garbage Collection	Bibliothek d. Laufzeitsystems
Dynamic (DLL)	nein	rts32d.lib
Static	nein	rts32s.lib
None	nein	-
Dynamic (DLL)	ja	rts32dgc.lib
Static	ja	rts32sgc.lib
None	ja	-

Ein weiterer wichtiger Punkt ist die Aufbereitung der Exportsymbole. Es ist zum Beispiel üblich, daß Dynamic Link Libraries mehrere Funktionen exportieren wollen. Damit dies funktioniert, muß der Linker beim Binden der DLL die Namen der exportierten Funktionen kennen. Der Programmierer kann sie im Dialog Linker-Einstellungen eingeben. Die Compiler-Schnittstelle sorgt dann dafür, daß alle Einträge korrekt an den Linker weitergeleitet werden.

Allerdings kann es auch vorkommen, daß die Compiler-Schnittstelle selbständig Symbole generiert. Wenn man nämlich eine DLL in Oberon-2 implementiert, müssen für jedes Modul zwei Symbole exportiert werden, damit die Initialisierung der in der DLL enthaltenen Module funktioniert. Es handelt sich dabei um die Symbole <mod>_$INIT und <mod>_$GLOBALDATA, wobei <mod> durch den Namen des jeweiligen Moduls zu ersetzen ist. Als Erleichterung für den Programmierer generiert die Compiler-Schnittstelle diese Symbole für eine DLL automatisch, indem es die Liste der im Projekt eingetragenen Module durchgeht und für jedes Modul die beiden Symbole in die Exportliste hinzufügt.

Darüber hinaus gibt es noch einige weitere Parameter, deren Werte aber meist direkt aus dem Dialog Linker-Einstellungen übernommen werden. Dazu gehört zum Beispiel die Stack-Größe oder die Basisadresse.

Als letztes bleibt noch zu klären, wie der Linker Fehler oder Erfolg an die Compiler-Schnittstelle meldet. Wichtigstes Instrument dafür ist der Rückgabewert der Funktion Link32. An diesem erkennt die Compiler-Schnittstelle, ob überhaupt eine ausführbare Datei entstanden ist. Daneben möchte aber auch der Programmierer erfahren, was passiert ist. Dazu gibt die Compiler-Schnittstelle die Adresse einer Funktion an den Linker als Parameter mit. Es handelt sich dabei um eine Funktion, die eine Zeichenkette als Parameter erwartet und diese Zeichenkette in das Ausgabefenster von Pow! schreibt.

4.1.4 Die Bibliotheken Opal und Opal++

Gerade für kleine Projekte ist es von großem Vorteil, wenn es einfache Bibliotheken gibt, die zum Beispiel die Ein- und Ausgabe von Texten oder Zahlen oder Dateioperationen regeln. Genau für diesen Zweck war die Bibliothek Opal gedacht. Die Bezeichnung "Opal" ist als Abkürzung von "Oberon Portable Applications Library" entstanden.

Die Entwicklung der Opal für 16-Bit Pow! begann sehr früh. Mit Oberon-2 alleine sind Windows-Programme nämlich nur sehr umständlich zu programmieren. Es war sehr bald klar, daß eine Bibliothek entstehen sollte, die viele einfache Routinen zur Verfügung stellen sollte.

Beim Entwurf der Opal versuchte man auch auf die Schnittstellen bereits bestehender Bibliotheken anderer Oberon-Systeme Rücksicht zu nehmen [KIR95], sodaß Programme, die die Opal benutzen, möglichst unverändert auch auf anderen System laufen.

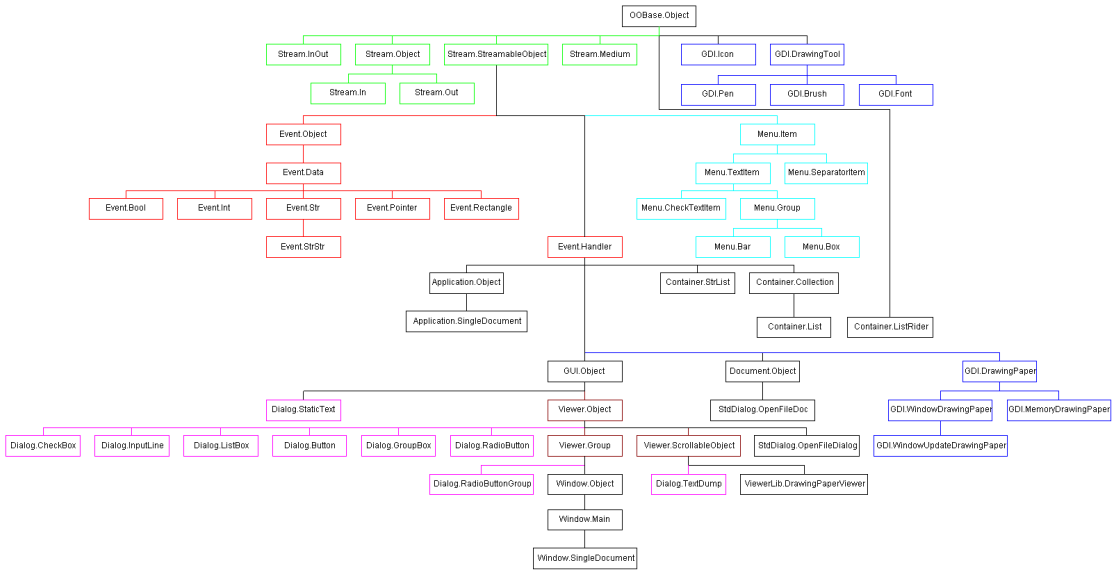
Bei der Portierung des Pow!-Systems war auch eine Portierung der Opal auf 32-Bit notwendig. Da die Schnittstelle der beiden Versionen sehr ähnlich ist, wird im folgenden nur mehr die 32-Bit Version beschrieben. Betreffend Details zu den einzelnen Modulen und Funktionen sei auf [LEI00] verwiesen. Im folgenden soll daher lediglich ein Überblick über die Möglichkeiten der Bibliothek gegeben werden. Dazu dient die folgende Tabelle:

	Module	**Kurzbeschreibung**
Basismodule	Strings	Verarbeitung von Zeichenketten und Konvertierung von Zeichenketten in Zahlen und umgekehrt
	Float	Mathematische Funktionen wie zum Beispiel Sinus, Cosinus, Tangens für Gleitkommazahlen und Funktionen zur Konvertierung von Gleitkommazahlen in Zeichenketten und umgekehrt.
	Utils	Einige Hilfsfunktionen, wie zum Beispiel Bitoperationen und Extrahieren von High- und Low-Byte. Ähnlich dem Modul SYSTEM deutet dieses Modul darauf hin, daß das Programm, das dieses Modul verwendet, nicht portabel ist.
	OOBase	Basisfunktionen für das Arbeiten mit Objekten und das Ermitteln von Laufzeittypinformationen von Objekten
	Param	Ermittlung der Kommandozeile, die an das Programm übergeben wurde.
	Process	Funktionen zur Kontrolle des Programmes
Benutzerschnittstelle	Display	Einfache Texteingabe- und Textausgabefunktionen in einem Fenster, das 25 Zeilen und 80 Zeichen Platz bietet. Unter anderem kann ein Textcursor an eine bestimmte Position plaziert werden und die Farbe der Schrift sowie des Hintergrundes eingestellt werden.
	ColorPlane	Einfache Grafikfunktionen in einem Fenster, das 800 * 650 Pixel groß ist. Es stehen einfache Textfunktionen mit einem plazierbaren Cursor sowie einige einfache Zeichenfunktionen zur Verfügung.
Dateisystem	File	Funktionen zum Anlegen, Öffnen, Lesen, Schreiben und Schließen von Dateien.
	Volume	Funktionen zum Anlegen und Löschen von Verzeichnissen sowie zum Ermitteln der Dateien in einem Verzeichnis.
	Print	Funktionen zum Drucken von Text
	In	Funktionen für eine stream-orientierte Eingabe. Die Daten können dabei von der Tastatur oder einer Datei kommen.
	Out	Funktionen für eine einfache sequentielle Ausgabe von Text. Der Text wird in einem Fenster mit Scrollbar

	Module	Kurzbeschreibung
Oakwood Guidelines		ausgegeben, sodaß immer auf die gesamte Ausgabe zugegriffen werden kann.
	XYplane	Funktionen für einfache einfärbige Grafiken

Neben der Opal gibt es auch die Opal++ [LEI00]. Die zwei zusätzlichen Pluszeichen sollen andeuten, daß es sich bei Opal++ um eine Klassenbibliothek handelt und nicht um eine klassische Funktionsbibliothek. Dementsprechend ist die Opal++ vollkommen anders aufgebaut. Von Bedeutung sind nicht mehr die Module sondern die Klassen, die die Bibliothek zur Verfügung stellt.

Wie bei der Opal soll auch über die Opal++ nur ein Überblick gegeben werden. Eine detaillierte Beschreibung kann in [LEI00] nachgelesen werden. Die folgende Grafik soll anhand der Klassenhierarchie einen ersten Eindruck vom Aufbau der Opal++ geben.



4.2 Abb. 39: Klassenhierarchie der Opal++Java

Die Geschichte von Java ist eine bemerkenswerte Erfolgsstory. Im Sog des einsetzenden Internet-Booms zu Beginn und Mitte der 90er Jahre konnte die sehr junge Programmiersprache viele Fans im Sturm erobern und war in kürzester Zeit in aller Munde.

Zu Beginn war Java hauptsächlich dadurch beliebt, daß sich sogenannte Java Applets leicht in HTML-Seiten integrieren ließen. Damit war es möglich, bewegte Grafiken auf einer Webseite darzustellen. Dies spiegelt auch folgendes Zitat wider, das vor einigen Jahren am Webserver von Sun zu lesen war.

> *It is commonly thought of as a way to make Web pages sexy -- incorporating stock tickers, sound or video into Web pages. It has evolved into much more. It is becoming known as a computing platform -- the base upon which software developers can build applications. Developers can build a variety of applications using Java -- traditional spreadsheets and word processors in addition to mission critical applications used by the biggest companies: accounting, asset management, databases, human resources and sales.*
>
> *(Quelle: Sun Microsystems, http://java.sun.com/nav/whatis/index.html)*

Doch dies war nicht der einzige Grund für die rasante Verbreitung von Java. Wie auch das Zitat schon mitzuteilen versucht, kann man mit Java nicht nur hübsche Applets programmieren. Die Programmiersprache bringt auch einige sehr interessante Konzepte mit sich. Viele dieser Konzepte sind zwar nicht neu – man kann sie in anderen Systemen unter anderem auch der Programmiersprache Oberon-2 finden –, aber noch niemals wurden sie in eine homogene Einheit gegossen. Es zahlt sich durchaus aus, diese Ideen kurz zu beleuchten.

- Portabilität: Die Portabilität – also die Möglichkeit Programme auf vielen verschiedenen Plattformen laufen zu lassen – ist keine umwerfende Neuerung. Viele Systeme haben in der Vergangenheit von sich behauptet, portabel zu sein. Im Gegensatz zu anderen Systemen sind Java-Programme nicht nur auf Quellcodeebene portabel sondern auch im übersetzten Zustand. Ein übersetztes Java Programm kann daher ohne neuerliche Übersetzung auf vielen verschiedenen Plattformen laufen. Diese Eigenschaft war von großer Bedeutung für die Verbreitung von Java-Applets im Internet.
- Sicherheit: Gerade im Zusammenhang mit dem Internet ist die Sicherheit ein entscheidendes Thema. Wenn sich ein Benutzer ein Applet aus dem Internet lädt, muß er dem Applet vertrauen können, daß es auf seinem Rechner nichts anstellt. Nachdem der Ursprung vieler Applets im Internet nicht bekannt ist, muß es dafür technische Mechanismen geben, die einen Zugriff auf Ressourcen des lokalen Rechners verbieten. Genau solche Sicherheitsmechanismen wurden in Java integriert, sodaß ein Benutzer mit gutem Gefühl beliebige Applets aus dem Internet verwenden kann, weil das Applet keinen Zugriff auf seinen Rechner hat.

- Einfache Vererbung: Die Vermutung liegt nahe, daß man aus den schlechten Erfahrungen der Mehrfachvererbung von C++ gelernt hat und Java nur mehr eine einfache Vererbungsstruktur spendiert hat. Die Mehrfachvererbung kann nämlich zu mehrdeutigen Situationen führen und birgt für den Programmierer erhebliche Möglichkeiten zum Fehler machen.
- Interfaces: Um den Nachteil der fehlenden Mehrfachvererbung ein wenig zu lindern, entsann man das Konzept der Interfaces. Es handelt sich dabei um eine Definition einer Schnittstelle. Man könnte ein Interface auch als Klasse beschreiben, die nur die Deklaration von Funktionen und Konstanten enthält, aber keinen Code. Klassen können eine oder mehrere Interfaces implementieren. Dazu müssen sie die in den Interfaces definierten Funktionen implementieren. Interfaces können auch als Datentypen verwendet werden. Man kann dann ein Objekt jeder Klasse zuweisen, die das Interface implementiert.
- Zeiger: Die Programmiersprache Java kommt zur Gänze ohne Zeiger aus. Damit sind viele Fehlermöglichkeiten, die durch die Zeigerarithmetik in C oder C++ entstehen, von vorn herein ausgeschlossen.
- Typsicherheit: Die Programmiersprache Java ist sehr typsicher gestaltet. Das bedeutet, daß es dem Programmierer nicht möglich ist, Werte zuzuweisen, deren Datentypen nicht kompatibel sind. Die Typsicherheit wird aber nicht nur im Quellcode geprüft, sondern auch im laufenden Programm. Ein spezieller Verifier prüft vor dem Start eines Java-Programmes die Korrektheit des Programmes.
- Garbage Collection: Gerade im Zusammenhang mit der dynamischen Speicherverwaltung kommt es immer wieder zu Programmfehlern. Ein sehr bekannter Vertreter dieser Gattung ist der "Dangling Pointer"; also ein Zeiger, der noch auf einen Speicherbereich zeigt, der gar nicht mehr verwendet wird oder bereits für etwas anderes verwendet wird. Ein Garbage Collector kann hier Abhilfe schaffen, da er sich um das Freigeben von Speicher kümmert. Nachdem der Garbage Collector prüft, ob ein Speicherbereich noch von einem Zeiger referenziert wird, gibt er keinen benutzten Speicher frei.
- Threads: Die Programmiersprache Java bringt eine Unterstützung für Threads mit. Somit ist es dem Java Programmierer möglich, in seinen Programmen Threads einzusetzen, ohne die jeweiligen Funktionen des Betriebssystems aufrufen zu müssen.
- Exceptions: Exceptions sind ein gutes Mittel um das Auftreten eines Fehler mitzuteilen und um die Fehlersituationen zu behandeln. In Java wurden die Exceptions so gut eingebaut, daß sie auch in der Schnittstelle einer Funktion auftauchen. Wenn eine Funktion eine Exception auswirft, dann muß diese Exception in der Funktionsschnittstelle angegeben werden.

Nachdem der Compiler dies sogar prüft, weiß der Programmierer immer, welche Exceptions in einer Funktion auftreten können.

- Syntax: Der letzte Punkt ist eher aus psychologischen Gründen ein Vorteil von Java und nicht so sehr ein technischer. Java verwendet eine Syntax, die stark an C++ erinnert. Viele Konstrukte sind sogar vollkommen ident. Dies wahr sicherlich ein wesentlicher Grund für die Verbreitung von Java, weil alle C und C++ Programmierer ohne große Probleme Java-Programme lesen können und auch mit wenig Aufwand erlernen können.

Das Pow!-Team wollte sich dieser neuen Programmiersprache nicht verschließen und hat diese neue Programmiersprache in Pow! integriert. Zu Beginn dieser Versuche scheiterte man an einem fehlenden Java-Compiler für 16-Bit Windows. Für 32-Bit Windows gab es damals schon eine Reihe interessanter Produkte. Außerdem war die Portierung auf 32-Bit bereits im Gange. Somit machte es keinen Sinn für die 16-Bit Version noch eine Compiler-Schnittstelle für Java zu entwickeln. Daher wurde gleich eine 32-Bit Version entwickelt.

Um die Compiler-Schnittstelle im Detail verstehen zu können, muß man vorher erläutern, wie Java übersetzt und ausgeführt wird. Die folgende Abbildung zeigt den typischen Ablauf. Der Quellcode von Java-Programmen wird in der Regel in Dateien mit der Endung *java* gespeichert. Durch einen Java-Compiler wird der Quellcode in Java-Bytecode übersetzt, der normalerweise in Dateien mit der Endung *class* abgelegt wird. Beim Java-Bytecode handelt es sich nicht um einen Maschinencode für einen speziellen Prozessor bzw. für einen bestimmten Prozessor, sondern um den Code einer eigens für Java entworfenen Maschine [LY96]. Um den Bytecode auf einer realen Plattform ablaufen zu lassen, braucht man die Java Virtual Machine, einen Interpreter für Java-Bytecode. Der Bytecode wird also interpretiert.

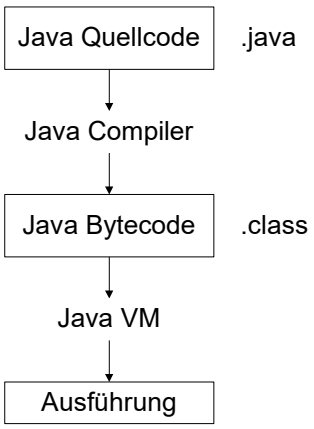


Abb. 40: Übersetzung und Ausführung eines Java-Programmes

4.2.1 Java Development Kit (JDK)

Für 32-Bit Windows gibt es heute viele verschiedene interessante und gute Java-Compiler. Auch beim Beginn der Entwicklung der Compiler-Schnittstelle war die Auswahl schon sehr groß. Daher machte es keinen Sinn, einen eigenen Compiler zu entwickeln. Die Compiler-Schnittstelle wurde also so implementiert, daß sie einen externen Compiler und Interpreter aufruft.

Bei der Suche nach einem geeigneten Compiler und Interpreter war die entscheidende Bedingung, daß sie kostenlos verfügbar sein sollen. Es wäre ja nicht sehr sinnvoll, auf der einen Seite Pow! kostenlos zur Verfügung zu stellen und auf der anderen Seite einen kostenpflichtigen Java-Compiler zu verwenden.

Ein Produkt, auf das diese Bedingung zutraf, war das Java Development Kit (JDK) von Sun. Neben der kostenlosen Verfügbarkeit sprach auch die Tatsache, daß es von Sun, dem Erfinder von Java, selbst entwickelt wird und damit die Wahrscheinlichkeit sehr hoch ist, daß es immer auf dem neuesten Stand ist.

Das JDK ist eine kostenlose Sammlung mehrerer Werkzeuge für die Entwicklung von Java-Programmen. Darunter fallen natürlich auch Compiler und Interpreter. Die Werkzeuge sind allerdings sehr spartanisch ausgelegt. Es handelt sich durchwegs um Werkzeuge ohne graphische Oberfläche. Außerdem fehlt in diesem Paket eine Entwicklungsumgebung. Somit waren Pow! und das JDK eigentlich optimale Partner. Pow! stellt eine graphische Entwicklungsumgebung zur Verfügung und das JDK den Compiler und den Interpreter.

Von den im JDK enthaltenen Programmen nutzt Pow! den Compiler javac.exe, den Interpreter java.exe, den Appletviewer appletviewer.exe und natürlich die Standardklassenbibliotheken und das Laufzeitsystem. [JDK01]

4.2.2 Integration des JDK in Pow!

Da der Übersetzungsablauf für Java-Programme denen von Oberon-Programmen sehr ähnlich ist, ist die Integration des JDK vom Konzept her nicht schwierig. Wenn die Compiler-Schnittstelle aufgefordert wird, eine Quelldatei zu übersetzen, dann wird das Programm javac.exe mit den richtigen Parametern aufgerufen. Der Vorgang des Bindens eines Programmes fehlt dagegen völlig. In diesem Fall gibt die Compiler-Schnittstelle eine Fehlermeldung aus, falls der Programmierer explizit diesen Vorgang über das Menü "Compile/Link" startet.

Die Ausführung eines Programmes unterscheidet sich jedoch ein wenig. Im Gegensatz zu Oberon-2, wo lediglich die erzeugte EXE-Datei ausgeführt wird, ist eine Class-Datei nicht direkt ausführbar. Es muß der Interpreter aufgerufen werden, dem als Parameter die Klasse mitgeteilt wird, die auszuführen ist.

Allerdings gibt es zwei Varianten von Java-Programmen. Bei Java-Programmen kann es sich um eigenständige Applikationen oder um Java Applets handeln. Eigenständige Applikationen erkennt man an der Funktion main. Sie werden mit dem Programm java.exe ausgeführt. Java Applets sind dagegen nicht eigenständig lauffähig. Sie müssen in eine HTML-Seite eingebettet werden und werden dann in einem Webbrowser ausgeführt, wenn der Browser die HTML-Seite mit dem eingebetteten Java-Applet anzeigt. Eine einfachere Variante ist der Appletviewer, der im JDK mitgeliefert wird. Mit ihm kann man ebenfalls Applets laufen lassen, ohne daß ein aufwendiger Browser-Start notwendig ist. Diese Variante nutzt auch die Java-Compiler-Schnittstelle und ruft für Java-Applets das Programm appletviewer.exe auf. Allerdings kann der Programmierer auch einen Browser verwenden, wenn er will. Die notwendigen Änderungen werden in Kürze erläutert.

Das bedeutet, daß die Compiler-Schnittstelle beim Ausführen von Java-Programmen zwischen zwei Varianten unterscheidet, nämlich eigenständigen Applikationen und Applets. Um welche der beiden Varianten es sich bei einem Projekt handelt, erkennt die Compiler-Schnittstelle nicht selbständig. Der Programmierer muß dies korrekt konfigurieren. Er kann dies wie in der folgenden Abbildung zu sehen ist, im Dialog "Compiler options" tun.

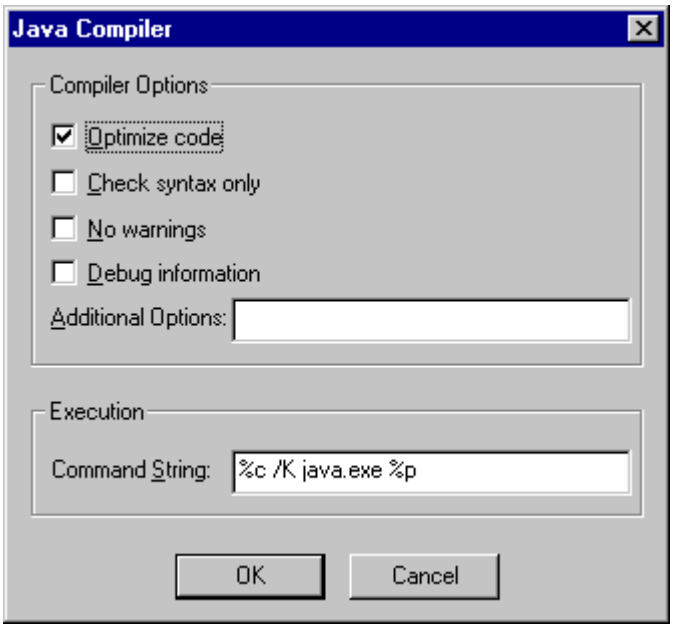


Abb. 41: Einstellungen der Java-Compiler-SchnittstelleBisher wurde nur am Rande erwähnt, daß die Compiler-Schnittstelle die entsprechenden Werkzeuge des JDK aufruft. In diesen Aufrufen steckt aber ein bißchen mehr. Vor allem der Aufruf des Compilers ist eine genauere Betrachtung wert, da der Compiler Meldungen ausgibt, wenn er Fehler im Quellcode findet. Diese Meldungen müssen auf geeignete Weise in das Pow!-System einfließen, damit sie auch korrekt angezeigt werden. Insbesondere soll dem Pow! bekannt gemacht werden, in welcher Zeile und Spalte der Fehler im Quellcode aufgetreten ist, damit der Textcursor gleich an die Stelle des Fehlers plaziert werden kann.

Nimmt man das Programm java.exe unter die Lupe, dann kommt man schnell drauf, wie es kommuniziert. Informationen an das Programm übergibt man über die Befehlszeile. Fehlermeldungen gibt das Programm über die Konsole aus. Dabei verwendet es einen sequentiellen Ausgabestrom, der jeder Applikation zur Verfügung steht. Es handelt sich um den sogenannten "standard out". Der Vorteil dieses Ausgabestromes ist, daß man ihn umlenken kann. Man kennt dieses Umlenken von der DOS-Befehlszeile, wo man mit dem Zeichen ">" die Ausgabe eines Programmes von der Konsole in eine Datei umlenken konnte.

Diese Fähigkeit nutzt auch die Compiler-Schnittstelle aus. Sie leitet "standard out" des Programmes java.exe auf eine temporäre Datei um und analysiert nach Beenden des Programmes den Inhalt der Datei. Insbesondere wird nach Fehlermeldungen gesucht und analysiert.

Beim Übersetzen sollte noch ein Faktum beachtet werden. Der Java-Compiler hat eine Fähigkeit, die der Oberon-2 Compiler nicht hat. Der Java-Compiler kann nämlich feststellen, welche Klassen

importiert werden. Das ist an sich nichts Erstaunliches. Allerdings kann der Compiler auch feststellen, ob die importierten Klassendateien auch existieren und wenn sie existieren, ob sie jünger als die Quelldateien sind. Wenn eine benötigte Klassendatei nicht existiert oder älter als die Quelldatei ist, dann übersetzt der Java-Compiler automatisch diese Datei auch.

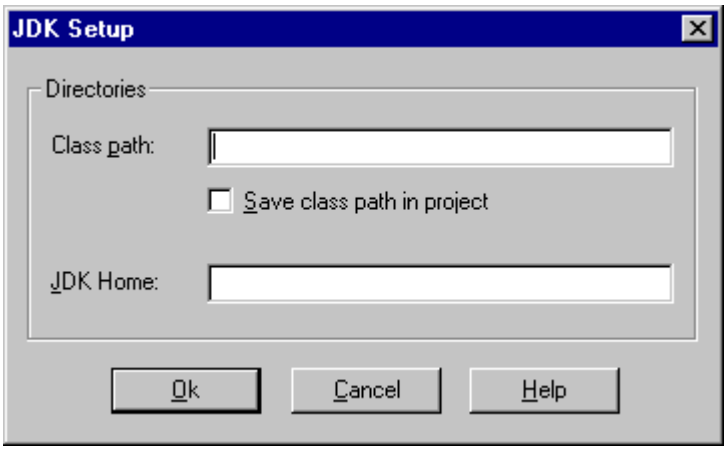
Das bedeutet, der Java-Compiler kann selbständig feststellen, ob importierte Klassen bereits übersetzt wurden und ob sie in der aktuellsten Version vorliegen, und kann sie übersetzen, falls dies nicht der Fall ist.

Was bedeutet dies für die Compiler-Schnittstelle? Auf den ersten Blick scheint dies eine Vereinfachung zu sein, weil die Compiler-Schnittstelle nur die Hauptquelldatei – also jene Quelldatei, die selbst nicht mehr importiert wird und alle anderen benötigten Klassen importiert – übersetzen lassen muß. Dabei werden, wie gerade beschrieben wurde, alle importierten Klassen automatisch übersetzt.

Das ist aber nur auf den ersten Blick eine Vereinfachung. Die Compiler-Schnittstelle muß nämlich herausfinden, welche Datei die Hauptquelldatei ist. Dazu wurde ein einfacher Parser für Java-Syntax geschrieben, der im wesentlichen nach Importbefehlen im Quellcode sucht. Anhand des Importbefehles kann der Parser feststellen, welche Klassen importiert werden. Wenn er diese Analyse auf alle Quelldateien eines Projektes anwendet, kann er feststellen, wie die Quelldateien untereinander abhängen und kann die Hauptquelldatei bestimmen. Dabei müssen Importbefehle für Klassen des JDK nicht berücksichtigt werden.

Es muß natürlich zugegeben werden, daß diese Methode bei voll qualifizierten Bezeichnern versagt. In Java ist es nämlich auch möglich, Klassen ohne Angabe eines Importbefehles zu verwenden. Dann müssen sie allerdings voll qualifiziert werden. Für diesen Fall reicht die Intelligenz des einfachen Parsers nicht aus. Wie die Praxis zeigt, spielt dies in der Regel keine große Rolle. Voll qualifizierte Bezeichner werden nämlich eher selten verwendet und wenn sie verwendet werden, werden sie gerne für Klassen des JDK verwendet. Somit hat sich diese Variante als funktionierender Kompromiß herausgestellt.

Neben den bereits beschriebenen Compilereinstellungen gibt es noch einen interessanten Dialog. Es handelt sich um den Dialog "Directory options". Dort kann man einstellen, in welchem Verzeichnis das JDK installiert ist. Außerdem kann man den Classpath einstellen. Der Classpath gibt an, in welchen Verzeichnissen nach Klassendateien gesucht werden soll. Vor allem sind darin die Verzeichnisse enthalten, in denen die Klassen der JDK abgelegt sind.



4.3 Abb. 42: Verzeichniseinstellungen der Java-Compiler-SchnittstelleC++

Aus vielen Statistiken erfährt man, daß die beiden Programmiersprachen C und C++ jene Programmiersprachen sind, die in der PC-Welt am häufigsten eingesetzt werden. Daher verwundert es kaum, daß auch in Pow! eine Unterstützung für C bzw. C++ integriert werden sollte. Kurze Zeit gab es sogar eine eigene Version von Pow!, die den Namen BlueGnu trug und nur die Programmiersprache C++ unterstützte.

Bevor man sich die Compiler-Schnittstelle für C++ genauer ansieht, muß man analysieren, wie C++ Programme übersetzt werden. Vom Ablauf her gibt es kaum Unterschiede zu Oberon-2. Im Details stecken allerdings kleine Änderungen. Die folgende Grafik zeigt ein Programm, das aus zwei Modulen besteht, links das Modul 1 und rechts das Modul 2, wobei das Modul 1 das Modul 2 importiert. In Oberon-2 wäre dazu lediglich ein Importbefehl notwendig. C und C++ kennen das Konzept der Module eigentlich nicht. Statt dessen wird die Schnittstelle von Modulen in sogenannten H-Files gespeichert und mit Hilfe des Include-Befehls in den Quellcode inkludiert.

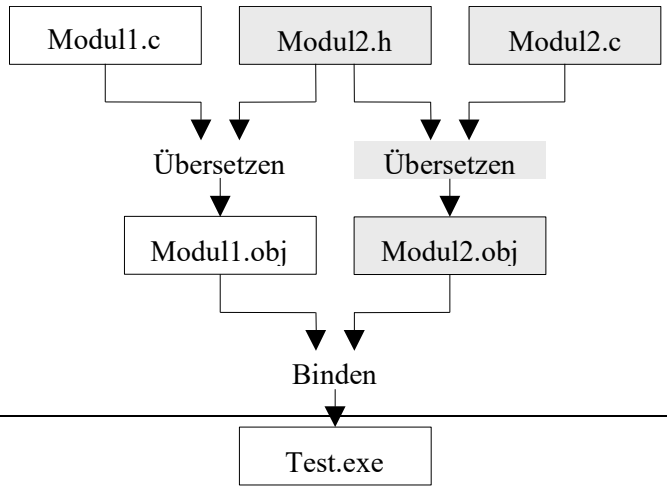


Abb. 43: Übersetzen eines C-Programmes

Der Compiler erzeugt wie der Oberon-2 Compiler beim Übersetzen einer Quelldatei eine Objektdatei. Alle Objektdateien werden dann mit Hilfe des Linkers zu einem Programm gebunden.

Überlegt man sich die Reihenfolge, in der die Quelldateien übersetzt werden müssen, dann kommt man drauf, daß die Reihenfolge völlig egal ist. In diesem Beispiel ist es egal, ob zuerst die Quelldatei Modul1.c oder die Modul2.c übersetzt wird, obwohl das Modul 2 von Modul 1 importiert wird.

Dies bringt natürlich eine erhebliche Vereinfachung der Compiler-Schnittstelle mit sich. Sie muß nämlich nicht die richtige Reihenfolge für die Übersetzung der Quelldateien ermitteln, weil es keine Reihenfolge gibt.

4.3.1 GNU C++

Die Auswahl eines passenden C++ Compilers, der in vernünftigem Aufwand Anwendungen für Windows erzeugt, war nicht leicht. Die meisten frei verfügbaren C++ Compiler waren für Unix-Systeme entwickelt worden. Ein sehr bekannter Vertreter dieser Gattung ist der GNU-Compiler [GNU01]. Es handelt sich dabei um einen kostenlosen C und C++ Compiler der Free Software Foundation.

Dieser Compiler wurde in den letzten Jahren auf verschiedene Plattformen portiert. Darunter befinden sich auch einige Portierungen für Windows. Die bekannteste scheint die Portierung der Firma Cygnus zu sein. Wie alle anderen GNU-Werkzeuge wird auch diese Portierung unter der GNU General Public License vertrieben, was zur Folge hat, daß die Software lizenzfrei genutzt werden kann. Man findet sie unter der Webadresse [CYG01].

Wie schon bei der Auswahl des Java-Compilers war die kostenlose Nutzung des C/C++ Compilers von entscheidender Bedeutung. Daher wurde in der ersten Version der Compiler-Schnittstelle der Compiler von Cygnus benutzt. Es handelte sich um die Version B18.

Einige Zeit später wurde die Compiler-Schnittstelle überarbeitet. Dabei wurde darüber nachgedacht, ob der Cygnus Compiler wirklich die richtige Wahl war. Zu diesem Zeitpunkt war die Version B20 aktuell. Da die neue Version eine vollkommen andere Verzeichnisstruktur hatte als die Version B18, mußte die Compiler-Schnittstelle sowieso gründlich überarbeitet werden.

Um nachvollziehen zu können, wie es zu dieser Überlegung kam, muß erklärt werden, welche Nachteile beim Cygnus Compiler auffielen. Dabei stach ein Nachteil besonders heraus, der durch folgendes Zitat belegt wird.

> *„The Cygwin tools are ports of the popular GNU development tools for Windows NT, 95, and 98. They run thanks to the Cygwin library which provides the UNIX system calls and environment these programs expect.*
>
> *With these tools installed, it is possible to write Win32 console or GUI applications that make use of the standard Microsoft Win32 API and/or the Cygwin API. As a result, it is possible to easily port many significant Unix programs without the need for extensive changes to the source code. This includes configuring and building most of the available GNU software (including the packages included with the Cygwin development tools themselves). Even if the development tools are of little to no use to you, you may have interest in the many standard Unix utilities provided with the package. They can be used both from the bash shell (provided) or from the standard Windows command shell.“ [CYG03]*

Aus diesem Zitat erkennt man die Ziele der Entwickler des Cygnus Compilers sehr gut. Der Compiler sollte vor allem gute Dienste für die Portierung von Unix-Programmen auf Windows leisten. Dazu hat die Firma Cygnus eine eigene Bibliothek entwickelt, die Unix Betriebssystem-funktionen unter Windows zur Verfügung stellt.

Der Ansatz und die Intention von Pow! war aber ein ganz anderer. Mit Pow! sollte es einfach und schnell möglich sein, kleine bis mittelgroße Programme für Windows zu schreiben. Dem stand die Größe des Paketes im Wege. Zum Lieferumfang des Paketes gehörten sehr viele Unix-Utilities, unter anderem auch eine eigene Kommandozeilenoberfläche. Dies macht das Paket sehr groß. Der Benutzer muß circa 14 MB herunterladen. Auf der Festplatte installiert benötigt es ungefähr 30 MB.

Dies war aber nicht der Hauptgrund für die ablehnende Haltung gegenüber dem Cygnus Compiler. Der Hauptgrund lag darin, daß der Compiler auf einer Unix-Emulation aufbaut. Das bewirkt, daß jedes Programm, das mit diesem Compiler erzeugt wird, die Unix-Emulation laden muß. Die Unix-Emulation liegt zwar in Form einer DLL namens CYGWIN1.DLL vor, aber es ist nicht möglich, die DLL nicht zu verwenden. Nachdem die Unix-Emulation nicht benötigt wird, wäre es schön gewesen, sie einfach ausschalten zu können.

Ein weiterer Nachteil war die Behandlung von Ressourcendateien. Die mitgelieferten Werkzeuge verwenden ein eigenes Ressourcenformat, das durch ein zusätzliches Werkzeug in das Ressourcen-format von Windows konvertiert werden muß. Dabei traten manchmal Fehler auf, die teilweise nur sehr schwer nachzuvollziehen waren.

Aus diesen Gründen wurde nach Alternativen gesucht. Die erste Alternative wurde in Form des Projektes MingW32 gefunden. MingW32 steht für Minimalist GNU Win32 und besteht aus einer Sammlung von Bibliotheken und Header-Dateien. Einen eigenen Compiler gibt es in diesem Projekt nicht. Es wird nämlich der Cygnus Compiler verwendet. Mit Hilfe der Bibliotheken und Header-Dateien sowie eines speziellen Schalters im Compiler ("-mno-cygwin") werden die Objektdateien mit den Dateien CRTDLL.DLL und KERNEL32.DLL gelinkt. Dabei handelt es sich um Standarddateien des Betriebssystems Windows. In CRTDLL.DLL ist das Laufzeitsystem für C-Programme enthalten und KERNEL32.DLL enthält einen Teil des Windows-API. Die Unix-Emulation CYGWIN1.DLL wird dann nicht mehr benötigt.

Bei den ersten größeren Tests gab es allerdings einen herben Rückschlag. Folgende Fehlermeldung tauchte zum Beispiel auf, deren Ursache zunächst nicht klar war:

```
C:\cygnus\...:iostream.cc: undefined reference to `_ctype_'
```

Nach einer Analyse des Problems und Studium einiger FAQs (siehe [CYG02]) war das Problem eingegrenzt. Ursache war unter anderem das Modul iostreams. Durch den oben beschriebenen Schalter "-mno-cygwin" wurden nämlich die Bibliotheken des Projektes MingW32 zum Programm gebunden. Diese Bibliotheken unterstützen aber nur die Sprache C und nicht die Sprache C++. Somit konnten keine Module der C++ Bibliothek verwendet werden, was auch für das Modul iostreams gilt. Mit dieser Lösung konnten also C Programme aber keine C++ Programme geschrieben werden.

Parallel dazu fanden wir noch einen Compiler namens LCC32 [LCC01]. Es handelte sich um einen schnellen, leistungsfähigen und kostenlosen Compiler für 32-Bit Windows. Leider mußten wir feststellen, daß der Compiler nur die Programmiersprache C unterstützt.

Diese Suche nach einem geeigneten Compiler war letztlich doch noch erfolgreich. Es gab nämlich auch schon andere Personen, die sich mit dem Problem herumgeschlagen hatten, daß der GNU Compiler soviel Ballast mit sich herumschleppt. Als Antwort darauf haben sie das Projekt "MingW: Minimalist GNU for Windows" [MIN02] gegründet.

Das Ergebnis dieses Projektes war eine spezielle Zusammenstellung des GNU Compilers für das Betriebssystem Windows, bei der der Compiler ohne den zusätzlichen Parameter "-mno-cygwin" auskommt und die auch Unterstützung für C++ bot. Der gesamte Ballast der Unix-Entwicklung war in diesem Paket entfernt worden.

Die Distribution besteht aus folgenden Paketen, die einzeln oder als gesamtes Paket heruntergeladen werden können. Damit Pow! funktioniert, sollte man alle Pakete installieren.

- GCC 2.8.1 (ca. 2.9 MB): GNU C, C++ und Objective C Compiler. Der C++ Compiler wurde stark verbessert. Insbesondere unterstützt er spezielle Optimierungen für Pentium und Pentium Pro. Das Laufzeitsystem des Objective C Compilers ist als DLL verfügbar.
- Binutils (ca. 1.9 MB): Einige GNU Werkzeuge sowie der neue Ressourcen-Compiler "windres".
- Mingw32 (ca. 186 KB): Header-Dateien und Importbibliotheken für die Programmiersprache C.
- Windows32api (400 KB): Header-Dateien und Importbibliotheken für 32-Bit Windows
- libstdc++ 2.8.1 (360 KB): Standard C++ Bibliotheken und Header-Dateien.

Zählt man die Größe der einzelnen Pakete zusammen, dann erkennt man, daß man mit weniger als 6 MB auskommt. Diese Distribution fand schlußendlich bei allen Gefallen und die neue Version der Compiler-Schnittstelle wurde darauf ausgerichtet.

4.3.2 Integration des GNU C++ Compilers

Die Integration der Programmiersprache C++ mit Hilfe des gefundenen Compilers war nicht mehr allzu schwierig. Vom Übersetzungsablauf gab es kaum Schwierigkeiten, da C++ Programme fast wie Oberon-2 Programme übersetzt werden. Im Gegenteil: Die Übersetzung von C++ Programmen ist sogar ein wenig einfacher, da keine zwingende Reihenfolge bei der Übersetzung von C++ Quellcode besteht.

Wenn eine Quellcodedatei übersetzt werden soll, dann startet die Compiler-Schnittstelle das Programm gcc mit dem Parameter "-c" und dem Namen der Quellcodedatei. Der Parameter bewirkt übrigens, daß nur die Quellcodedatei übersetzt wird. Darüber hinaus übergibt die Compiler-Schnittstelle noch weitere Parameter, wie zum Beispiel die Optimierungsstufe. Der Programmierer kann diese Parameter durch die Einstellungen im Dialog Options/Compiler beeinflussen.

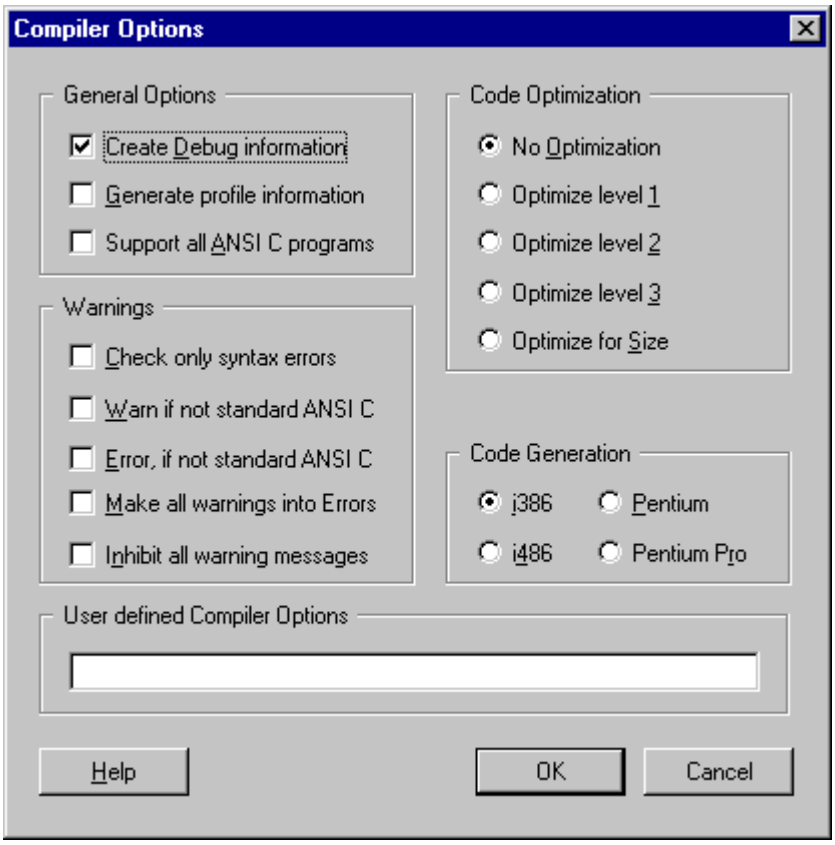


Abb. 44: Compilereinstellungen der C++ Compiler-SchnittstelleLäßt man den Parameter "-c" weg, dann verhält sich das Programm gcc wie ein Linker. Es bindet dann alle angegebenen Objektdateien zu einem Programm. Diese Tatsache nutzt die Compiler-Schnittstelle für das Binden eines Programmes aus.

Dabei muß die Compiler-Schnittstelle allerdings unterscheiden, ob ein Programm oder eine Dynamic Link Library entstehen soll. Welche der beiden Dateien erzeugt werden soll, kann der Benutzer im Linker-Dialog festlegen.

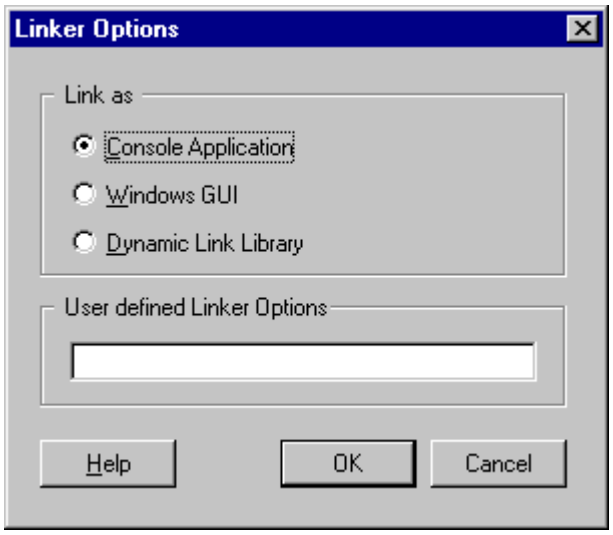


Abb. 45: Linker-Einstellungen der C++ Compiler-SchnittstelleFür die Ermittlung der Fehlermeldungen der externen Programme wird dieselbe Technik wie in der Java-Compiler-Schnittstelle eingesetzt. Die externen Programme werden in einem eigenen Prozeß gestartet, der parallel zu Pow! ausgeführt wird. Da die Meldungen der Programme auf dem Standardausgabestrom ausgegeben wird, wird dieser beim Starten des Prozesses in eine temporäre Datei umgeleitet. Dadurch werden die Ausgaben der externen Programme nicht am Bildschirm ausgegeben, sondern in eine temporäre Datei geschrieben.

Pow! wartet dann bis zur Beendigung des Prozesses und beginnt danach, die temporäre Datei zu analysieren. Falls Fehlermeldungen in der Datei enthalten sind, werden sie mit Zeilen- und Spaltennummer an Pow! weitergeleitet.

Zum Schluß sollte nicht unerwähnt bleiben, daß die Header-Dateien zwar bei der Reihenfolge der Übersetzung helfen, aber mehr Probleme bereiten, beim Ermitteln welche Dateien neu übersetzt werden müssen. Header-Dateien werden nämlich in der Regel nicht in die Liste der Quelldateien aufgenommen. Selbst wenn man sie aufnehmen würde, würde dies an folgendem Problem nichts ändern.

Es kann vorkommen, daß eine Header-Datei geändert wird. Dann müssen alle Quellcodedateien neu übersetzt werden, die diese Header-Datei inkludieren. Die Compiler-Schnittstelle weiß aber nicht, welche Header-Dateien von welchen Modulen importiert werden. Um diese Information heraus zu bekommen, wurde ein kleiner Parser entwickelt, der die im Projekt eingetragenen Quelldateien analysiert. Dabei sucht er im wesentlichen nur die Include-Befehle. Die darin vermerkten Header-Dateien merkt sich die Compiler-Schnittstelle und überprüft, ob eine dieser

Dateien seit dem letzten Übersetzungslauf geändert wurden. Wenn ja, wird die gerade analysierte Quelldatei übersetzt.

Mit Hilfe dieses einfachen Parsers kann die Compiler-Schnittstelle somit dynamisch feststellen, welche Header-Dateien von einer Quellcodedatei inkludiert werden.

5 Editoren

Eine der ersten Erweiterungsmöglichkeiten, die für Pow! entwickelt wurden, war die Editor-Schnittstelle. Trotzdem dauerte es relativ lang, bis ein zweiter Editor zur Verfügung stand. Es zeigte sich, daß die Implementierung eines vernünftigen Editors einige Zeit beansprucht.

Beide Editoren sind so implementiert, daß die Editor-Schnittstelle und der Editor selbst in einer Einheit implementiert sind. Im Gegensatz zur Compiler-Schnittstelle werden also keine externen Programme aufgerufen.

Zu Beginn der Entwicklung von Pow! stand nur ein einfacher Editor zur Verfügung. Er trug den Namen PowEdit und verfügte über keine herausragenden Eigenschaften. Ein zweiter Editor namens Boosted wurde später implementiert, um dem Programmierer bei seiner Arbeit mehr Unterstützung zu bieten.

Mit dem Einstieg in die 32-Bit Welt mußten natürlich auch die Editoren auf 32-Bit portiert werden. Die Portierung von PowEdit war wegen seiner Einfachheit leicht durchzuführen. Die Portierung des Boosted zog ein wenig mehr Arbeit nach sich. Sie konnte aber auch in relativ kurzer Zeit erledigt werden.

5.1 PowEdit

Das Konzept des PowEdit ist schnell erklärt. Im wesentlichen verläßt sich der Editor auf die Edit-Control von Windows. Dies tut im übrigen auch das Programm Notepad, das mit Windows ausgeliefert wird und mit dem man ebenfalls Texte editieren kann.

Bei der Edit-Control handelt es sich um eine bestimmte Fensterklasse in Windows, die normalerweise für einzeilige Eingabefelder verwendet wird. Es ist aber auch möglich und laut Dokumentation erlaubt, sie für mehrzeilige Texte einzusetzen. Die entsprechende Stelle in der Dokumentation von Windows 3.1 lautet:

> *"An edit control is a rectangular child window in which the user can type and edit text. Edit controls have a variety of features, such as multiline editing and scrolling." [MIC92, Seite 182]*

Liest man an dieser Stelle weiter, erfährt man, daß die Edit-Control alle notwendigen Fähigkeiten für das Editieren eines längeren Textes mit sich bringt. Außerdem wird das Anzeigen des Textes und das Positionieren des Cursors im Text vollständig durch die Edit-Control vorgenommen.

PowEdit hat dann eigentlich nur mehr für die Kommunikation zwischen Pow! und der Edit-Control zu sorgen. Dazu werden die Funktionsaufrufe der Editor-Schnittstelle in entsprechende Nachrichten umgesetzt, die an die Edit-Control gesandt werden. Umgekehrt werden gewisse Nachrichten der Edit-Control in Meldungen an Pow! umgesetzt.

Beim Öffnen eines Fensters muß PowEdit natürlich dafür sorgen, daß eine Edit-Control als Kindfenster in das Fenster eingefügt wird. Die Edit-Control wird so plaziert, daß sie den gesamten Fensterinhalt ausfüllt. Bei jeder Verkleinerung oder Vergrößerung des Fensters wird die Größe der Edit-Control wieder angepaßt.

Ein Nachteil der Edit-Control soll aber nicht verschwiegen werden. Die Textlänge, die man mit einer Edit-Control editieren kann, ist auf 64 KB begrenzt. In der Praxis können die 64 KB selten ausgenutzt werden. Es zeigte sich, daß meist zwischen 50 und 55 KB Schluß war. Der Grund für diese Beschränkung liegt in der Implementierung der Edit-Control, die erst in Windows NT geändert wurde. In diesem Betriebssystem können mit der Edit-Control Texte beliebiger Länge editiert werden.

5.2 Boosted

Einer der Gründe für die Entwicklung eines neuen Editors war die Beschränkung auf 64 KB. Bei den meisten Modulen spielte dies zwar keine große Rolle, weil sie in der Regel viel kleiner waren. Es gab aber gerade im Bereich des Laufzeitsystem ein Modul, das regelmäßig an die Grenzen stieß.

Boosted steht für "Basic Operative Oberon Source Text Editor". Daraus erkennt man auch ein wenig die Zielrichtung dieser Entwicklung. Es sollte ein einfacher funktionierender Editor sein, der gewisse Erleichterungen für das Editieren von Quelltexten mit sich bringt. Einige Funktionen davon sind speziell für die Programmiersprache Oberon-2 vorgesehen.

Im Gegensatz zum Editor PowEdit kümmert sich Boosted um alles selbst. Er ist dafür zuständig, den Text im Hauptspeicher zu verwalten, ihn in einem Fenster anzuzeigen und auch den Cursor korrekt zu plazieren. Nähere Details über die Implementierung findet man in [LEI00]. An dieser Stelle sollen nur einige Erleichterungen aus der Sicht des Programmierers präsentiert werden.

- Auto Indent: Diese Funktion fügt in eine neue Zeile genau so viele Leerzeichen am Beginn ein, wie Leerzeichen am Beginn jener Zeile sind, wenn die Taste Enter gedrückt wird.

Gerade in blockstrukturierten Programmiersprachen wie Oberon-2 oder C kennzeichnet man unterschiedliche Blocktiefen durch unterschiedliche Anzahl von Leerzeichen. Daher ist das automatische Einrücken am Beginn einer neuen Zeile sehr hilfreich, weil es dem Programmierer die Arbeit abnimmt, selbständig immer die richtige Anzahl von Leerzeichen einzufügen. Dies kann, wie sicherlich jeder bestätigen wird, bei einer hohen Schachtelungstiefe sehr lästig werden.

- Smart Line Merge: Diese Funktion ist in Zusammenhang mit obiger Funktion zu sehen. Wenn am Beginn einer Zeile in der Regel mehrere Leerzeichen stehen und zwei Zeilen zu einer Zeile zusammengefügt werden, dann bleiben an dem Punkt, wo die zwei Zeilen zusammengefügt wurden, mehrere Leerzeichen übrig. Die Funktion "Smart Line Merge" verhindert dies und entfernt beim Verschmelzen zweier Zeilen alle bis aus ein Leerzeichen.
- Oberon-2 Syntax Support: Wie schon angesprochen wurde, gibt es eine spezielle Unterstützung für Oberon-2 Programmierer. Der Editor Boosted kann nämlich gewisse Konstrukte selbständig ergänzen. Dazu sucht der Editor nach Oberon-2 Schlüsselworten, wenn die Taste Enter gedrückt wird. Findet der Editor das Schlüsselwort "PROCEDURE", dann fügt er automatisch Zeilen mit "VAR", "BEGIN", "END name" hinzu, wobei "name" für den aktuellen Prozedurnamen steht. Damit bekommt der Oberon-2 Programmierer nach dem Eintippen einer PROCEDURE-Anweisung einen vollständigen Prozedurrumpf. Er braucht fast "nur" mehr die Prozedur ausfüllen. Wird dagegen eines der Schlüsselworte "IF", "WHILE", "REPEAT", "CONST", "VAR" oder "TYPE" gefunden, dann wird für die nächste Zeile die Einrückungstiefe um eins erhöht. Umgekehrt wird beim Schlüsselwort "END" die Einrückungstiefe um eins erniedrigt. Die Anzahl der Leerzeichen, die pro Einrückungsstufe hinzukommen, kann man im Dialog "Editor Options" (siehe Abbildung 46) einstellen. Meistens verwendet man zwei Leerzeichen pro Stufe.
- Replace Tabs: Wenn eine Quelldatei in den Hauptspeicher geladen wird, dann werden alle Tabulatoren durch Leerzeichen ersetzt. Wie viele Leerzeichen pro Tabulator verwendet werden, wird im Feld "Tab-Size" des Dialogs "Editor Options" festgelegt.
- Color Comments: Kommentare können in einer anderen Farbe als der übrige Quelltext dargestellt werden. Dies kann die Übersichtlichkeit des Quelltextes erhöhen.

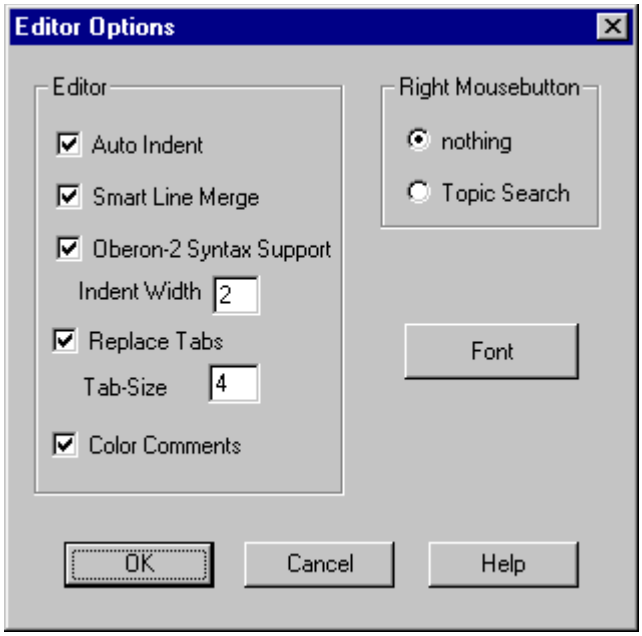


Abb. 47: Editoreinstellungen des BOOSTED Editors

- Topic Search: Klickt man mit der rechten Maustaste auf ein Wort im Quelltext, dann wird die aktuelle Compiler-DLL dazu veranlaßt, einen Hilfetext zu dem unter dem Cursor stehenden Wort anzuzeigen. Klickt man zum Beispiel ein Schlüsselwort der Programmiersprache an, dann sollte die Compiler-DLL ohne weiteren Eingriff des Programmierers eine Hilfe zu diesem Schlüsselwort anzeigen.
- Schriftarten: Der Editor Boosted ist nicht auf eine Schriftart und -größe beschränkt. Man kann aus den drei Schriftarten Courier, Courier New und Fixedsys wählen. Die Schriftgröße kann man von 6 bis 72 Punkte einstellen. Außerdem unterscheidet der Editor zwischen Schriftarten für das Darstellen am Bildschirm und beim Ausdrucken. Man kann also verschiedene Größen für die beiden Ausgabegeräte einstellen.

6 Werkzeuge

In Kapitel 3 wurde erläutert, daß Pow! eine eigene Schnittstelle besitzt, mit der Werkzeuge integriert werden können. Werkzeuge können dabei sehr lose an Pow! gekoppelt sein. Dies ist zum Beispiel der Fall, wenn die Kommunikation zwischen Pow! und dem Werkzeug lediglich darin besteht, daß Pow! das Programm aufruft und danach das Programm ohne Zutun von Pow! arbeitet.

Werkzeuge können aber auch enger an Pow! gebunden sein. Dies ist zum Beispiel der Fall, wenn das Programm seine Ausgabe in einem Fenster von Pow! anzeigt. Darüber hinaus könnte ein Werkzeug Pow! über die DDE-Schnittstelle fernsteuern.

Im folgenden sollen stellvertretend für alle Werkzeuge, die mit Pow! zusammenarbeiten bzw. die für Pow! geschrieben wurden, drei Werkzeuge näher erläutert und beschrieben werden. Es handelt sich dabei um dem Symbolfile Browser, einem Werkzeug der zweiten Kategorie, und dem Documentation Generator, der lose an Pow! gekoppelt ist, sowie um den Debugger. Dieser läuft als eigenständiger Prozeß und ist in [BON97] dokumentiert.

6.1 Symbolfile Browser

Symboldateien werden vom Oberon-2 Compiler erzeugt und enthalten im wesentlichen die Schnittstelle eines Moduls in binärer Form. Den Aufbau einer Symboldatei hier zu beschreiben, würde den Rahmen dieser Arbeit sprengen. Eine detaillierte Darstellung ist in [LEI00] zu finden. Der Symbolfile Browser dient dazu, Symboldateien zu analysieren und in einer für den Menschen lesbaren Form darzustellen. Als Ausgabeform wurde eine Oberon-2 ähnliche Syntax gewählt.

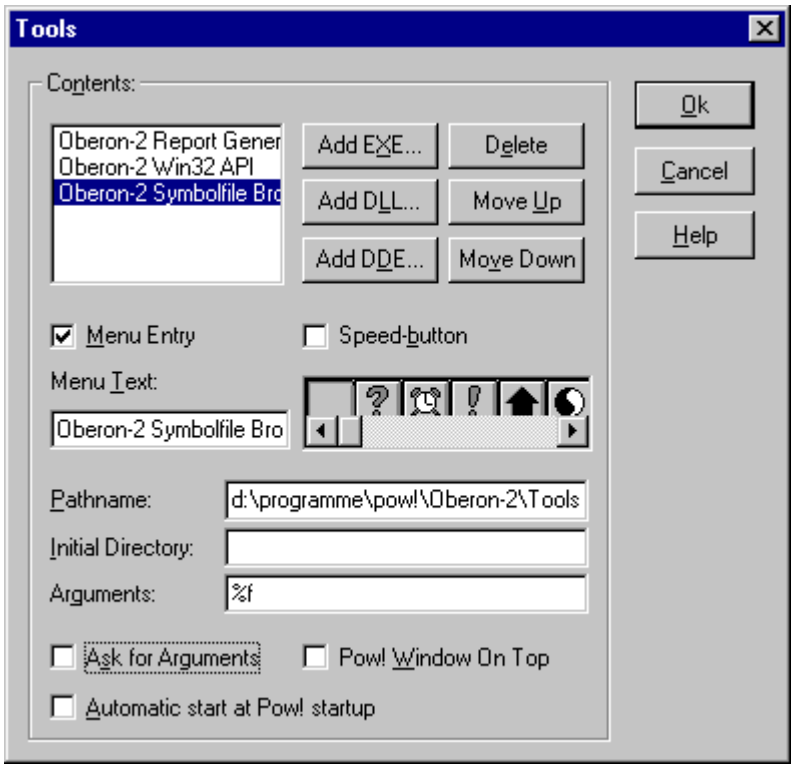


Abb. 48: WerkzeugeinstellungenDer Symbolfile Browser ist ein hilfreiches Werkzeug für Oberon-2 Programmierer. Er kann in das Menü Werkzeuge integriert werden. Als Parameter kann man dem Programm den Namen der aktuellen Datei übergeben. Dazu trägt man in Pow! als Argument %f ein (siehe Abbildung 49). Der Symbolfile Browser bekommt dadurch den Pfad und Namen der gerade aktiven Datei übergeben. Er wandelt den übergebenen Namen in den Namen der entsprechenden Symboldatei um, indem die Erweiterung des Dateinamens auf *sym* geändert wird. Danach liest er die Datei ein und analysiert sie.

Wird dem Symbolfile Browser kein Dateiname übergeben, dann läßt er dem Benutzer eine Symboldatei auswählen. Dazu öffnet er den Standarddialog zum Öffnen einer Datei.

Die durch die Analyse der Symboldatei gewonnen Daten werden in einem Fenster von Pow! ausgegeben. Dazu baut der Symbolfile Browser eine DDE-Verbindung zu Pow! auf und öffnet ein leeres Fenster mit Hilfe des DDE-Befehls "NewFile". Danach wird der Text mit Hilfe des DDE-Befehls "AppendText" über dieselbe DDE-Verbindung an Pow! gesendet, das diesen Text im gerade geöffneten Fenster anzeigt.

Somit hinterläßt dieses Programm gegenüber dem Programmierer den Eindruck, als wäre es Bestandteil von Pow!. Es kann direkt aus der Oberfläche heraus gestartet werden und zeigt die Ausgabe auch in einem Fenster von Pow! an.

6.2 Documentation Generator

Der Documentation Generator ist ein nützliches Werkzeug für jeden Programmierer. Er hilft aus dem Quelltext automatisch eine Dokumentation zu erstellen. Im Gegensatz zum Symbolfile Browser wird die Ausgabe nicht in einem Fenster von Pow! dargestellt, sondern in eine Datei geschrieben, die danach weiterverarbeitet werden kann. Ziel des Programmes ist das Erzeugen einer Hilfedatei, die zumindest die Modulschnittstellen beschreibt.

Der Documentation Generator wird in [LEI00] ausführlich beschrieben. Als erstes Einsatzgebiet für den Documentation Generator fungierte die Opal. Dabei zeigte sich, daß das Einfügen der zusätzlich notwendigen Kommentare zwar einige Zeit kostete, aber die Dokumentation der Opal immer am letzten Stand war.

Der Documentation Generator kann Oberon-2 Quellcode analysieren und die Schnittstelle der Module in verschiedenen Formen exportieren. Zur Schnittstelle gehören alle Teile des Quelltextes, der von einem Modul exportiert wird, also alle exportierten Prozeduren, Datentypen und Konstanten.

Zusätzlich zu den Oberon-2 Elementen kann der Documentation Generator auch Kommentare mit einbeziehen. Normalerweise übernimmt er nur Kommentare, die mit der Zeichenfolge "(**" beginnen. Wie man aber in Abbildung 50 sehen kann, gibt es aber auch eine Einstellung, die den Documentation Generator dazu bringt, alle Kommentare zu verwenden.

Die Kommentare werden nach einem ausgeklügelten Verfahren den jeweiligen Oberon-2 Elementen zugeordnet und als Beschreibung des zugeordneten Elementes verwendet. Will man beispielsweise zu einer exportierten Prozedur eine Beschreibung haben, dann sollte man den Kommentar in der Zeile nach der Prozedurdeklaration beginnen. Der Generator ordnet nämlich den Kommentar nach einer Prozedurdeklaration der Prozedur zu und verwendet den Inhalt des Kommentars als Beschreibung der Prozedur.

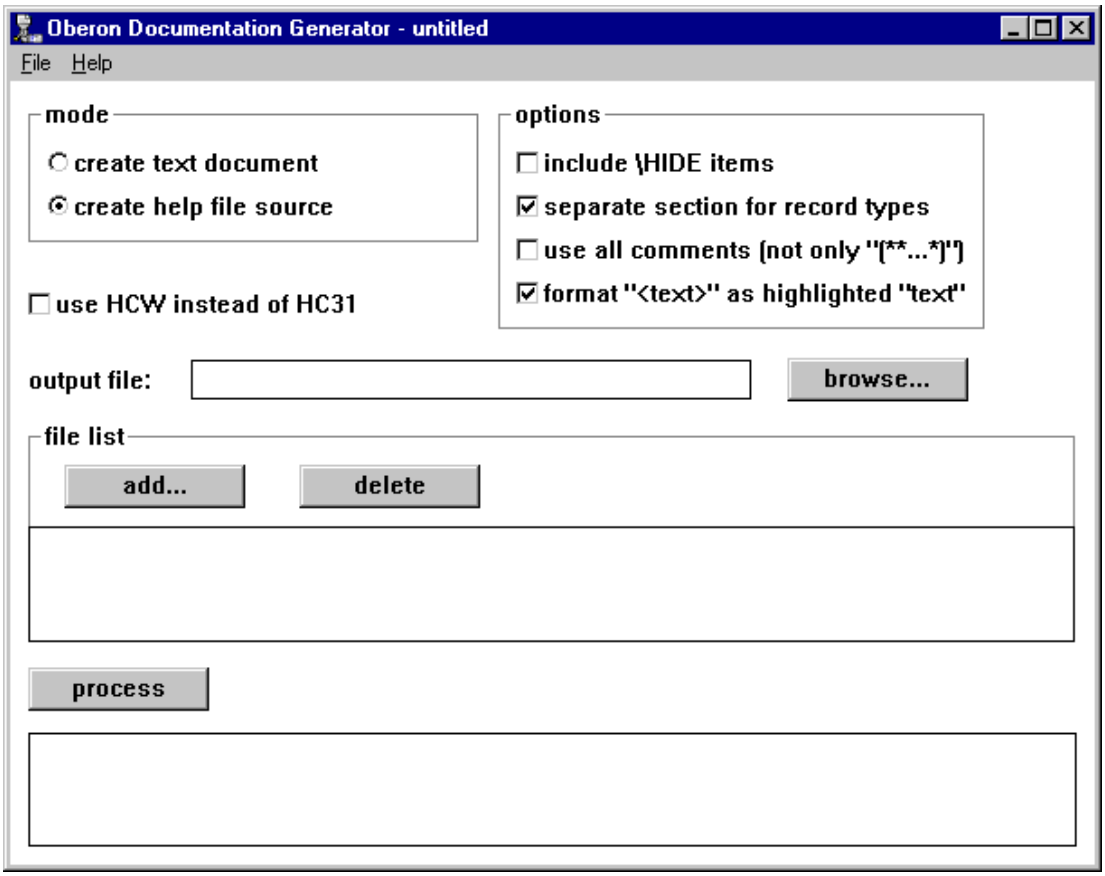
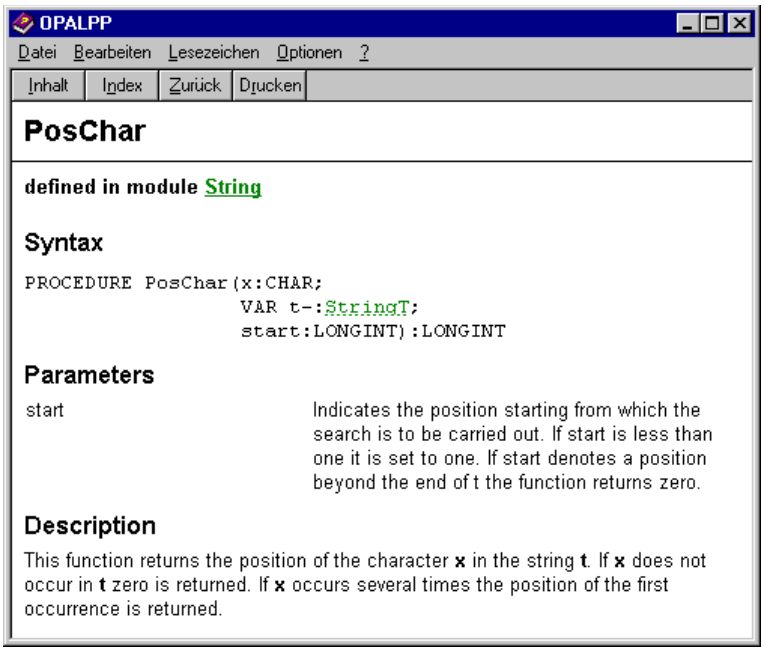


Abb. 51: Documentation GeneratorWie ebenfalls aus der Abbildung 52 zu ersehen ist, kann der Documentation Generator verschiedene Ausgabeformate erzeugen. Das für den Programm interessanteste Format ist sicherlich die Erzeugung von Hilfedateien. Dabei erzeugt der Documentation Generator eigentlich Dateien im sogenannten Rich-Text-Format, die danach vom Hilfecompiler in Hilfedateien übersetzt werden. Alle für diesen Prozeß notwendigen Dateien werden vom Documentation Generator erzeugt. Außerdem ruft er auch den Hilfecompiler auf, sodaß letztlich eine fertige Hilfedatei zur Verfügung steht.

Am Beispiel des Moduls *String* der Opal++ soll nun die Funktionsweise des Documentation Generators demonstriert werden. Der folgende Quellcode zeigt jenen Teil der Funktion *PosChar* des Moduls *String*, der für den Documentation Generator entscheidend ist.

```
PROCEDURE PosChar* (x: CHAR;
                    VAR t-: StringT;
                    start:  LONGINT (** Indicates the position
                                       starting from which the search
                                       is to be carried out. If start
                                       is less than one it is set to
                                       one. If start denotes a
                                       position beyond the end of t
                                       the function returns zero. *)
                   ): LONGINT;
(** This function returns the position of the character <x> in the
    string <t>. If <x> does not occur in <t> zero is returned. If
    <x> occurs several times the position of the first occurrence is
    returned. *)
...
BEGIN
...
END PosChar;
```

Abbildung 53 zeigt die dazu passende Seite der Hilfedatei an, die vom Documentation Generator generiert wurde. Wie leicht zu erkennen ist, übernimmt der Documentation Generator automatisch die Schnittstelle der Funktion, sowie eventuell vorhandene Kommentare von Parametern und die Beschreibung der Funktion selbst.



6.3 Abb. 54: Hilfeseite der Funktion PosCharDebugger

Das letzte hier beschriebene Werkzeug ist der Debugger. Eine detaillierte Beschreibung findet sich in [BON97].

Dieser Debugger ist eine eigenständige Applikation, die nicht nur Programme, die mit Pow! erzeugt wurden, sondern jedes beliebige Programm debuggen kann. Dazu stellt der Debugger übliche Funktionen zum Laden, zum Starten und zum Stoppen eines Programmes zur Verfügung. Außerdem kann man sich auch Variablen oder Datenstrukturen anzeigen lassen.

Darüber hinaus kann der Debugger auch die symbolischen Informationen in den ausführbaren Dateien verarbeiten. Mit symbolischen Informationen sind jene Informationen gemeint, die der Compiler generiert, damit der Debugger zum Beispiel weiß, wo eine Prozedur in der ausführbaren Datei beginnt. Nur mit diesen symbolischen Informationen ist es möglich, daß man sich im Debugger eine Variable anschauen kann oder einen Breakpoint auf eine Prozedur legen kann.

Der Debugger stellt zwei Möglichkeiten der Benutzung zur Verfügung. Ohne spezielle Vorkehrungen stellt der Debugger eine textorientierte Oberfläche zur Verfügung, in der man Befehle eingeben kann. Der Debugger ist aber auch schon auf die Zusammenarbeit mit anderen Programmen vorbereitet. Er kann nämlich alle Befehle auch über DDE erhalten und verarbeiten. In diesem Fall verhält sich der Debugger wie ein DDE-Server.

Teil II:

Das Visualisierungs-werkzeug

7 Erweiterung für Visualisierung

Im Jahre 1997 begann ein Forschungsprojekt mit dem Namen "Transport Semantischer Informationen in der Programmvisualisierung". Ziel dieses Projektes war die Entwicklung eines neuen Systems zur Visualisierung von Programmen. Um einen Prototyp realisieren zu können, sollte Pow! eingesetzt werden. Da zu Beginn des Projektes die Entwicklung der 32-Bit Version von Pow! gerade erst angefangen hatte, wurde der Prototyp mit Hilfe der 16-Bit Version entwickelt. Wie später noch erläutert werden wird, ist das kein Nachteil. Man kann zeigen, daß mit kleinen Änderungen die Portierung auf ein 32-Bit System möglich ist. Im folgenden wird daher der 16-Bit Prototyp dargestellt.

Bevor auf die technischen Details eingegangen wird, sollte vorerst einmal geklärt werden, welche Ziele sich das Projekt gesetzt hatte und wie es sich gegenüber anderen Visualisierungssystemen unterscheidet. Dazu sollten einige Begriffe erläutert werden. Den Anfang macht der Begriff Visualisierung, der in [BRO94, Seite 380] folgendermaßen definiert wird:

> *"Bezeichnung für bildliche Formulierung und Kommunikation, das heißt für Aufbereitung von Information mit vor allem bildlichen Mitteln wie auch für visuelle Wahrnehmung. Daneben wird der Begriff im heutigen, auch wissenschaftlichen Sprachgebrauch oft unscharf verwendet: Verschiedene Interpretationen, Traditionen und Anwendungsbereiche lassen eine Eindeutigkeit nicht zu. Zum Teil wird Visualisierung sogar über den Bereich des Sehens hinaus verwendet, zum Beispiel für die Bereicherung der Sprache durch Lautmalerei oder mit Geräuschen, die beim Zuhörer innere Bilder entstehen lassen, oder für das Phänomen des 'Visual transfer', bei dem in der Werbung die optischen Eindrücke eines Fernsehspots gezielt durch die lautliche Entsprechung nachfolgender Radiospots für dasselbe Produkt wieder in Erinnerung gerufen werden."*

Aus diesem Zitat kann man erkennen, daß trotz des unscharfen Gebrauchs des Wortes Visualisierung eine wesentliche Kernaussage übrig bleibt. Mit Visualisierung möchte man Informationen vor allem mit bildlichen Mitteln formulieren und weiterleiten. Darüber hinaus möchte man aber auch Informationen sichtbar machen, die a priori gar nicht sichtbar sind.

Genau darin besteht auch die Zielsetzung des Projektes. Es soll Informationen über Programme in leicht verständliche Bilder verpacken, sodaß der Betrachter das dargestellte Programm leichter

verstehen oder überblicken kann. Wie aber auch schon die Definition andeutet, soll die Visualisierung nicht nur auf bildliche Darstellungen beschränkt, sondern soll auch andere Medien, wie zum Beispiel Akustik, verwenden können.

In der Literatur findet man viele Arbeiten zum Thema Visualisierung, eigentlich genauer Software-Visualisierung. Einen guter Überblick über das Thema liefert unter anderem [PBS93]. Darin wird Software-Visualisierung als "Verwendung von Schrift, Grafik, Animation und Film in Mensch-Computer-Systemen zum besseren Verstehen und Benutzen von Software" verstanden.

Man kann sie nach der Art der Informationsgewinnung in zwei Kategorien teilen. Die erste Kategorie gewinnt ihre Informationen aus einer statischen Programmanalyse. Das bedeutet, daß der Quellcode oder der Objektcode analysiert wird. Das Programm muß dazu niemals ausgeführt werden. Die Analyse geschieht nur auf Basis des Programmcodes. Dabei entstehen Maßzahlen, die eine gewisse Eigenschaft des Programmes beschreiben.

Einfache Maßzahlen sind zum Beispiel die Schachtelungstiefe eines Quellcodes oder die Anzahl der Programmzeilen eines Moduls. Läßt man sich diese Maßzahlen für alle Module eines Programmsystems erstellen, kann man sie in Grafiken darstellen. Mit solchen Grafiken kann man zum Beispiel leicht feststellen, welche Module besonders umfangreich sind. Diese Information könnte man zum Beispiel für die Auswahl der Testverfahren berücksichtigen. Besonders umfangreiche Module könnte man mehr Zeit für das Testen zur Verfügung stellen.

Der Nachteil statischer Programmanalyse liegt auf der Hand. Es läßt sich nur sehr wenig darüber aussagen, wie sich das Programm zur Laufzeit verhält. Natürlich kann man anhand des Quellcodes Überlegungen über das Laufzeitverhalten eines Programmes anstellen. Man kann aber nicht genau sagen, wie Objekte zueinander in Beziehung stehen. Um zum Beispiel herauszufinden, wie oft im Durchschnitt eine Methode eine andere aufruft, muß man das Programm laufen lassen.

Damit kommt man zur zweiten Kategorie der Programmvisualisierung, die sogenannte dynamische Programmanalyse. Im Gegensatz zur statischen Analyse wird die Information über das Programm nicht aus dem Quellcode alleine sondern aus dem Verhalten des Programmes zur Laufzeit gewonnen. Die dynamische Programmanalyse beobachtet also das Programm und registriert wichtige Ereignisse oder Veränderungen.

Das Ergebnis dieser Beobachtungen kann man im wesentlichen wieder auf zwei Arten darstellen. Entweder interessiert einem nur der augenblickliche Zustand des Programmes. Dann wird man eine Darstellung bevorzugen, die den Zustand möglichst gut beschreibt. Ist man zum Beispiel am Wert einer numerischen Variablen interessiert, dann wäre eine einfache Visualisierung die Anzeige des

augenblicklichen Wertes mit einem Balken. Verändert sich der Wert, dann wird auch der Balken entsprechend angepaßt.

Es kann aber auch vorkommen, daß die Entwicklung eines Wertes von Interesse ist. Dann wird man eine Darstellung wählen, die den aktuellen Wert und einige oder alle vorherigen Werte anzeigt. Dabei kommt auch die Zeit ins Spiel. Es ist in der Regel von Interesse, wann sich ein Wert geändert hat. Eine einfache Darstellungsform dafür ist ein xy-Diagramm. Dabei werden wie bei einer mathematischen Funktion auf der y-Achse die Werte aufgetragen und auf der x-Achse die Zeit.

Für die dynamische Analyse kann man also zwei Darstellungsarten unterscheiden. Die erste zeigt den aktuellen Zustand an während die zweite den zeitlichen Verlauf eines Zustandes darstellt. Nachteil der zweiten Darstellungsart ist, daß nicht sehr viele Zustände gleichzeitig angezeigt werden können, weil die visuelle Darstellung unübersichtlich wird. Das ist genau der Vorteil der anderen Darstellung. Sie kann viel mehr Objekte gleichzeitig darstellen. Was aber noch viel wichtiger ist: Man kann auch die Beziehungen zwischen Objekten darstellen. Dafür geht natürlich die Information über die Vergangenheit ab.

Führt man die Idee der Darstellung eines Algorithmus über einen längeren Zeitraum fort, dann gelangt man unweigerlich zu Systemen der Algorithmenanimation. Dazu gibt es schon einige interessante Arbeiten, z.B. das System Zeus von Marc Brown [BRO92] oder XTANGO von John Stanko [STA92]. Ergebnisse über die Wirkung von Visualisierungen im Unterricht sind in [FOR93] abgehandelt.

Diese Systemen haben eines gemeinsam: Ihr Ziel ist es einen Algorithmus möglichst professionell und fließend zu animieren. Wenn also zum Beispiel ein Element aus einer Datenstruktur entfernt wird, dann wird nicht einfach das Element in der visuellen Darstellung von einem Moment auf den anderen entfernt. Statt dessen versuchen die Systeme einen mehr oder weniger fließenden Übergang zu generieren. Im Extremfall gewinnt der Benutzer den Eindruck, vor einem Film zu sitzen.

Dazu ist einiger technischer Aufwand notwendig, vor allem softwaretechnischer Natur. Die meisten Systeme können nämlich nicht einfach ein bestehendes Programm visualisieren. Bevor ein Algorithmus animiert werden kann, muß der Quellcode massiv an das System angepaßt werden. Bei manchen Systemen muß der Quellcode sogar in eine eigene Klassenbibliothek integriert werden. Dies bedeutet viel Vorarbeit, bis man ein Ergebnis erhält. Die Arbeit lohnt sich zwar, weil man eine Animation erhält, aber es dauert eben einige Zeit, bis man etwas sieht.

In der vorliegenden Arbeit war es nicht das Ziel, optisch hervorstechende Animationen zu erstellen. Hauptziel des Projektes war es, ein Visualisierungssystem zu schaffen, das jedes Programm möglichst ohne Änderung des Quelltextes sofort visualisieren kann. Dadurch, daß keine Änderung des Quellcodes notwendig ist, könnten mit diesem System auch Programme visualisiert werden, zu denen man keinen Quellcode besitzt.

Ein weiteres wichtiges Ziel des Visualisierungssystems war die Berücksichtigung der objektorientierten Programmierung. Im Gegensatz zur klassischen Programmierung ist der Ablauf objektorientierter Programme nicht so leicht durch das Studium des Quelltextes zu ermitteln. Viele Aufrufe von Methoden werden erst zur Laufzeit durch polymorphe Objekte festgelegt (siehe Yo-Yo-Effekt in [TGP89]). Beim Betrachten eines Methodenaufrufes kann man daher nicht immer entscheiden, welche konkrete Methode zur Laufzeit aufgerufen wird. Erst durch den dynamischen Typ des Objektes wird festgelegt, welche Methode aufgerufen wird.

Um solche Zusammenhänge besser überblicken zu können, ist es also wichtig zu wissen, welche Objekte im System existieren und wie sie untereinander zusammenhängen. Objekte können auf unterschiedliche Weise zusammenhängen. So kann ein Objekt einfach einen Zeiger auf ein anderes Objekt haben oder zwei Objekte können in derselben Datenstruktur enthalten sein. Im letzteren Fall werden die Objekte dann von einem anderen Objekt referenziert oder die Objekte einer Datenstruktur sind untereinander verkettet. In allen Fällen hängen Objekte aber immer durch Zeiger zusammen.

Gerade das Wissen dieser Zusammenhänge ist für das Verständnis eines objektorientierten Programmes noch viel wichtiger als für ein konventionelles Programm. Das in Kapitel 8 beschriebene Visualisierungssystem versucht diesem Umstand Rechnung zu tragen und stellt diese Zusammenhänge dar. Als Darstellungsform wird meist ein Pfeil benutzt, der von jenem Objekt ausgeht, das den Zeiger beinhaltet, und der zu jenem Objekt zeigt, auf den der Zeiger verweist.

Stellt man auf diese Weise mehrere Objekte dar, kann der Benutzer sehr schnell die Zusammenhänge zwischen den Objekten erkennen. Insbesondere für generische Datenstrukturen kann dies von Vorteil sein, wenn der Benutzer auf einem Blick erkennen kann, welche Objekte enthalten sind. Dabei kann man sich auch Darstellungsvarianten vorstellen, die die Pfeile weglassen und nur mehr die Objekte anzeigen, die zu einer Datenstruktur gehören. Je mehr Objekte dargestellt werden, umso unübersichtlicher wird die Darstellung mit Pfeilen. In der Regel kann man dann keine Anordnung der Objekte mehr finden, in der sich die Pfeile nicht kreuzen. Dann wird es für den Benutzer sogar schwieriger den Überblick zu behalten.

Für solche Fälle muß das Visualisierungssystem andere Möglichkeiten zur Verfügung stellen. Ein Ansatz dazu ist in [SB94] dargestellt. Dabei wird ein Graph nicht gleichmäßig dargestellt, sondern

verzehrt dargestellt. Die Mitte des Graphen wird besonders hervorgehoben, indem die Abstände zwischen den Objekten sehr groß gemacht wird. Dadurch ist in diesem Bereich jedes Detail gut erkennbar. Am Rand rücken die Objekte dafür zusammen und verschmelzen teilweise. Der Betrachter hat den Eindruck, als würde er durch ein Fischauge – ein spezielles Objektiv für Fotoapparate – auf den Graphen schauen. Jener Teil, der in der Mitte des Fischauges liegt, wird dabei hervorgehoben. Es muß natürlich möglich sein, das Fischauge zu verschieben, sodaß auch andere Teile des Graphen genauer betrachtet werden können.

Diese Ansicht eines Graphen hilft spezielle Details eines größeren Graphen zu erarbeiten. Für die Software-Visualisierung dürfte aber ein anderer Ansatz interessanter sein, der auch im Visualisierungssystem implementiert wurde. Es soll die Möglichkeit geben, auf verschiedenen Abstraktionsstufen zu visualisieren. Das bedeutet, man soll das Visualisierungssystem dazu veranlassen können, mehrere Objekte zu einem Objekt zusammenzufassen und als ein Objekt darzustellen. Dabei sollen aber nur die Objekte selbst zusammengefaßt werden. Die Referenzen zu Objekten, die nicht zusammengefaßt wurden, sollen weiterhin gezeichnet werden.

Diese Variante ist vor allem für größere Datenstrukturen von Interesse. Geht man beispielsweise von einer verketteten Liste aus, die mehrere Elemente enthält, dann kann es in manchen Situationen von Interesse sein, jedes Element einzeln zu sehen. Manchmal kann es aber auch genau umgekehrt sein und man will nur wissen, wieviele Elemente in der Datenstruktur enthalten sind und welche anderen Objekte von ihnen referenziert werden. Dann soll der Benutzer die einzelnen Objekte der Datenstruktur markieren können und zu einem Objekt zusammenfassen können (Operation "fold"). Die zusammengefaßten Objekten werden dann nicht mehr einzeln gezeichnet, sondern nur mehr durch ein einziges graphisches Objekt dargestellt. Die Pfeile, die von den einzelnen Objekten ausgegangen sind, sollen dann alle von diesem einzigen Objekt ausgehen. Eine Umkehroperation gibt es natürlich auch, bei der ein Objekt wieder auseinandergefaltet wird. Es soll auch möglich sein, das Zusammenfalten in mehreren Schritten zu wiederholen, sodaß quasi in mehreren Abstraktionsstufen immer mehr Objekte zusammengefaßt werden.

7.1 Visualisierung und Semantik

Das letzte und wohl wichtigste Ziel des Visualisierungsprojektes ist bereits im Titel des Forschungsprojektes enthalten, nämlich der Transport semantischer Informationen. Von entscheidender Bedeutung ist hier der Begriff Semantik. Er stammt vom griechischen Wort σημαινειν ('bezeichnen') ab und wird in [BRO93, Seite 114] folgendermaßen definiert:

> *"Im Unterschied zur allgemeinen Zeichentheorie (Semiotik) Disziplinbezeichnung für alle Untersuchungen der Bedeutung sprachlicher Ausdrücke."*

Die Semantik untersucht also die Bedeutung sprachlicher Ausdrücke. Setzt man diese Definition auf den Titel des Forschungsprojektes um, dann zielt das Projekt darauf ab, daß die Visualisierung die Bedeutung eines Programmes oder seiner agierenden Objekte verdeutlichen soll. Was man sich darunter vorstellen soll, kann man anhand eines kleinen Beispiels verdeutlichen.

Es soll ein Programm geben, daß Autos in irgendeiner Form verwaltet. Bei der Visualisierung des Objektes Auto kann man sich nun verschiedene Darstellungsformen vorstellen. Man könnte das Auto durch ein Foto darstellen, das den gleichen Autotyp mit der gleichen Farbe zeigt, wie es gerade im Programm verwendet wird. Man könnte das Auto aber auch als Rechteck symbolisieren.

Beide Darstellungsformen können neben der eigentlichen Information, daß es sich um ein Auto handelt, noch weitere Informationen transportieren. Sie können also dem visualisierten Objekt weitere Bedeutung verleihen. Im ersten Fall könnte die Bedeutung darin liegen, um welchen Autotyp es sich handelt und welche Farbe das Auto hat. Versierte Autokenner können vielleicht sogar den Jahrgang des Autos erkennen. Im zweiten Fall könnte die Bedeutung vielleicht darin liegen, die Existenz eines Autos anzuzeigen. Es spielt dabei keine Rolle, welche Farbe das Auto hat und von welchem Typ es ist.

Bettet man diese beiden Darstellungsvarianten in konkrete Anwendungen ein, dann kann man den Sinn der beiden Beispiele besser erkennen. Die Visualisierung eines Autos mit Hilfe eines Fotos würde zum Beispiel gut in ein Programm passen, mit dem man gebrauchte Autos verwalten will. Dabei sind die beschriebenen Details von Interesse und ein Foto kann diese Details sicherlich adäquat ausdrücken.

Die zweite Variante wird man dagegen in einer Verkehrssimulation benutzen. Dort sind die Details eines Autos nicht von Interesse. Vielmehr interessiert hier, wie sich die Verkehrsteilnehmer verhalten und ob sich zum Beispiel vor einer Ampel oder Baustelle lange Schlangen bilden.

In beiden Fällen wird durch unterschiedliche Darstellungsformen unterschiedliche Informationen zum Betrachter transportiert. Welche Informationen für ihn relevant – also von Bedeutung – sind, entscheidet letztlich der Betrachter. Wichtig ist aber, daß die Visualisierung möglichst viel bedeutsame Information transportiert. Das ist auch das Ziel des Visualisierungssystem. Die Darstellungsformen soll so gestaltet werden, daß der Benutzer möglichst viel Information, die für ihn von Bedeutung ist, bekommt.

Dieses Ziel zu erreichen ist nicht sehr leicht. Man erkennt nämlich an diesen beiden einfachen Beispielen, daß die Bedeutung, die eine Darstellung dem Betrachter übermittelt, von der jeweiligen Anwendung abhängt. Im Fall des Gebrauchtwagenhändlers übermittelt die Darstellung des Autos in Form eines Rechteckes keine für die Auswahl eines Gebrauchtwagens relevante Information. Dagegen kann sie für die Verkehrssimulation genau die richtige Information sein.

Man darf natürlich nicht verschweigen, daß die geeignete Darstellungsform auch vom Benutzer abhängt. Der Betrachter einer Visualisierung ist jene Instanz, die entscheidet, was für ihn wichtig ist oder nicht. Wie die Psychologie lehrt, filtert jeder Mensch seine Sinneswahrnehmungen nach gewissen ihm eigenen Regeln.

Faßt man diese Erkenntnisse zusammen, dann benötigt man ein System, in dem man die Visualisierung eines Systems an die jeweilige Anwendung und den jeweiligen Anwender anpassen kann. Das bedeutet, das Visualisierungssystem muß äußerst flexibel auf Änderungen in den Darstellungsformen reagieren können.

Das Endprodukt dieser Überlegungen ist ein Visualisierungssystem, dessen Visualisierungsteil sehr leicht gegen neue Teile ausgetauscht werden kann und auch erweitert werden kann. Konkret wird die Technik später noch vorgestellt. Die wesentliche Idee der Umsetzung liegt darin, die Visualisierung völlig abzukoppeln und in eine eigene Klassenbibliothek zu verlagern. Jede Klasse kann eine bestimmte Klasse von konkreten Programmobjekten visualisieren. Der Anwender kann durch einfache Erweiterung dieser Klassen neue Visualisierungen für bestehende oder auch neue Klassen implementieren. Häufig benutzte Datenstrukturen wie 'lineare Listen' oder 'binäre Bäume' sind als Beispiele vorgefertigt und in der Klassenbibliothek enthalten.

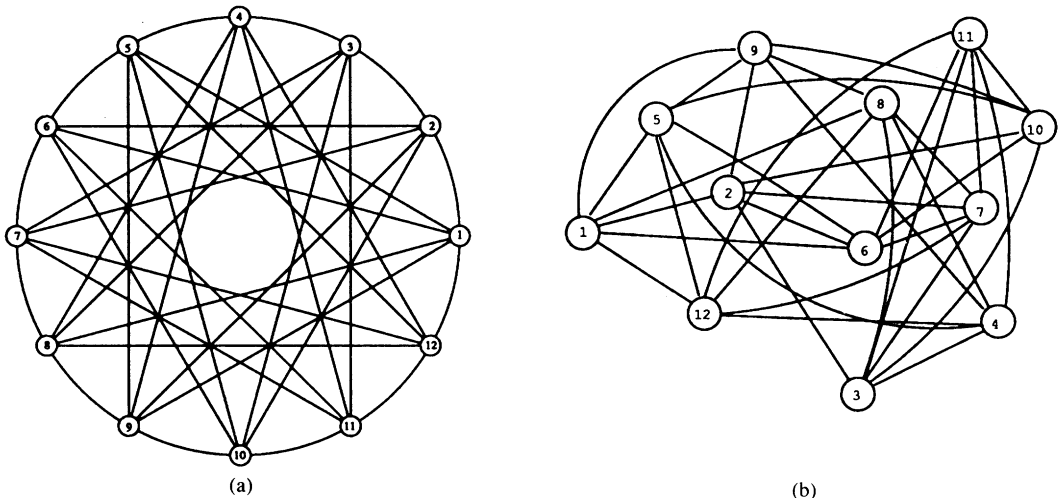
7.1.1 Graphische und inhärente Semantik

Wenn man sich mit der Semantik von graphischen Darstellungen beschäftigt, muß man sich auch fragen, ob Grafiken selbst semantische Information transportieren. Die Frage ist klar mit ja zu beantworten. Das bekannte Sprichwort „Ein Bild sagt mehr als tausend Worte“ deutet sogar an, daß graphische Darstellungen besonders viel Informationen transportieren können. Der Vorteil einer Grafik gegenüber einer Beschreibung in Worten liegt in der Regel darin, daß Grafiken für dieselbe Menge an Information weniger Raum als geschriebener Text benötigen und daß Zusammenhänge viel leichter ersichtlich sind und damit das Gesamtsystem einfacher zu begreifen sind. Die semantische Information einer Grafik resultiert aus der Summe der Semantik der verwendeten Symbole und der Semantik der Anordnung dieser Symbole.

Die Semantik von Symbolen ist ein zweischneidiges Schwert. Sind Symbole standardisiert, dann besitzen sie eine hohe Aussagekraft, die auch über Landes- und Kulturgrenzen hinweghelfen kann. Elektroniker haben zum Beispiel eine Reihe von Symbolen für verschiedene Bauelemente in einer Schaltung. Da diese Symbole standardisiert sind, können Schaltpläne von jedem Elektroniker der Welt gelesen werden. Kulturelle oder sprachliche Hürden beeinträchtigen den Informationsaustausch nicht. Fehlt dagegen eine Standardisierung, dann ist die Wahrscheinlichkeit eines Mißverständnisses sehr hoch. Es kann passieren, daß ein Symbol in einem bestimmten Kulturkreis ein völlig andere Bedeutung hat als dies der Autor einer Grafik vor Augen hatte. Das bedeutet, daß die Semantik eines Symbols vom Betrachter abhängt.

Da es keine Standardisierung im Bereich der Visualisierung von Datenstrukturen bzw. Objektexemplaren (object instances) gibt, beschränkt sich das Visualisierungssystem auf für uns gängige Darstellungen bekannter Datenstrukturen. Durch die leichte Erweiterbarkeit der Visualisierung ist es aber jederzeit möglich, eigene Darstellungsformen zu implementieren, wodurch die Visualisierung auf die jeweilige Semantik des Betrachters angepaßt werden kann.

Neben der richtigen Auswahl der Symbole in einer Grafik spielt auch die Anordnung der Symbole eine große Rolle. Eine geschickte Anordnung der Symbole kann dem Betrachter helfen, die abgebildeten Strukturen leichter zu verstehen. Ein sehr gutes Beispiel dafür ist in Abbildung 55 dargestellt, die aus [DM90] übernommen wurde. Die linke Grafik läßt die Struktur des Graphen leicht erkennen. Man kann ohne große Mühe erfassen, daß jeder Knoten mit seinen beiden Nachbarn und vier weiteren Knoten verbunden ist. Genauer gesagt ist jeder Knoten *i* mit den Knoten (i - 1) mod 12, (i + 1) mod 12, (i + 4) mod 12, (i + 5) mod 12, (i + 7) mod 12 und (i + 8) mod 12 verbunden. Die rechte Grafik dagegen läßt dies nicht auf den ersten Blick erkennen.



7.2 Abb. 56: Vergleich der Anordnung der Knoten in einem GraphenAllgemeine technische Voraussetzungen

Damit das im folgenden beschriebene Visualisierungssystem funktioniert, sind einige wenige technische Voraussetzungen zu erfüllen. Die meisten davon werden von modernen Programmiersprachen erfüllt. Dadurch ist die Übertragbarkeit auf andere Systeme gegeben.

7.2.1 Dynamischer Typ

Eine wichtige Voraussetzung für das Visualisierungssystem ist die Verfügbarkeit von dynamischen Typen. Es wird nämlich vom dynamischen Typ abgeleitet, welche Darstellungsform verwendet werden soll. Was mit dem Begriff "dynamischer Typ" gemeint ist, kann sehr gut anhand des folgenden Zitates aus [RP97, Seite 380] erläutert werden.

> *"Der Typ einer Variablen definiert die Menge ihrer möglichen Werte. Technisch gesehen ist jeder Wert durch eine Bitfolge codiert; der Typ definiert die Interpretation solcher Bitfolgen als ganze oder Gleitpunktzahlen, als boolesche Werte, Texte, Gruppen solcher Werte usw. Hängt die Interpretation und damit der Typ von der jeweils ausgeführten Operation ab, so heißt die Programmiersprache typfrei, sonst typisiert oder typgebunden. Typisierte Sprachen können in zwei Richtungen weiter klassifiziert werden: Besitzt jede Variable und jeder Wert einen festen unveränderlichen Typ, so heißt die Sprache statisch typisiert. Kann sich der Typ einer Variablen durch Zuweisung eines neuen Wertes ändern, heißt sie dynamisch typisiert."*

Der Datentyp einer Variablen definiert also, welche Werte sie annehmen kann. Wenn sich der Datentyp während des Programmablaufes nicht verändern kann, dann spricht man von einem statischen Datentyp. Solche Variablen behalten über ihre gesamte Lebensdauer den festgelegten Typ. Der Typ einer Variablen wird meist durch eine Deklaration am Beginn des Programmes festgelegt. Eine Beispiel dafür ist die folgende Variablendeklaration, bei der eine Variable namens *a* vom Typ INTEGER und eine Variable namens *c* vom Typ CHAR vereinbart werden.

```
VAR
  a: INTEGER;
  c: CHAR;
```

Entsprechend der Implementierung der Programmiersprache Oberon-2, in der die obige Deklaration geschrieben ist, bedeutet diese Deklaration folgendes: Die Variable *a* kann jeden ganzzahligen Wert aus dem Bereich von -32768 bis +32767 und *c* jedes der 256 Zeichen aus der ASCII-Tabelle annehmen.

Statische Typen sind für den Compiler leichter zu verarbeiten, da der Compiler genau die Menge der Werte kennt, die die Variable einnehmen kann. Dynamische Typen sind dagegen viel komplizierter zu bewältigen, weil der Compiler eben keine Annahmen über den Wertebereich einer Variablen treffen kann. Man spricht dann von dynamischer Typbindung, weil die Variable nicht für die gesamte Programmausführung an einen Typ gebunden ist.

Das bedeutet aber nicht, daß beliebige Typänderungen möglich sind. Meist sind diese Typänderungen nur nach gewissen Regeln möglich. In objektorientierten Programmiersprachen beispielsweise kann einer Variablen nur ein abgeleiteter Typ zugewiesen werden. Dies soll an folgendem Beispiel verdeutlicht werden.

```
TYPE
  BaseTypeDesc     = RECORD ... END;
  BaseType         = POINTER TO BaseTypeDesc;
  ExtendedTypeDesc = RECORD (BaseTypeDesc) ... END;
  ExtendedType     = POINTER TO ExtendedTypeDesc;
VAR
  b: BaseType;
  e: ExtendedType;
...
BEGIN
  NEW(b);
  NEW(e);
  b := e;    (* b hat nun den dynamischen Typ ExtendedType *)
  ...
END.
```

In diesem kleinen Beispiel werden zwei Record-Typen vereinbart, wobei der Inhalt der Records unbedeutend ist. Viel wichtiger ist die Tatsache, daß der Typ ExtendedTypeDesc ein erweiterter Record ist und von BaseTypeDesc abgeleitet wurde. Danach werden mit NEW zwei Objekte dieser Typen angelegt. Die Variable b ist dabei als Variable vom Typ BaseType deklariert und e als Variable vom Typ ExtendedType.

Durch die Zuweisung *b := e* bekommt die Variable einen neuen Typ zugewiesen, nämlich den Typ ExtendedType. Man spricht davon, daß sich der dynamische Typ der Variablen geändert hat. Er wurde durch die Zuweisung von BaseType auf ExtendedType geändert.

In objektorientierten Programmiersprachen können Variablen also zwei Typen haben, den statischen Typ und den dynamischen Typ. Der statische Typ einer Variablen wird durch die Deklaration der Variablen festgelegt und kann sich nicht ändern. Der dynamische Typ wird dagegen durch die Zuweisung eines konkreten Objektes festgelegt und kann sich laufend verändern.

In obigem Beispiel hat die Variable *b* also immer den statischen Typ BaseType. Dieser ist am Beginn des Programmes auch mit dem dynamischen Typ ident, weil durch die Anweisung *NEW(b)* der Variablen b ein Objekt des Typs BaseType zugewiesen wird. Erst durch die nachfolgende Zuweisung *b := e* laufen der statische und der dynamische Typ auseinander. Der statische Typ ändert sich dadurch nicht, aber der dynamische Typ der Variablen ändert sich auf ExtendedType.

7.2.2 Typinformationen zur Objektvisualisierung

Es genügt aber nicht nur, daß dynamische Typbindung verfügbar ist. Das Visualisierungssystem versucht anhand des dynamischen Typs einer Variablen die passende Darstellung auszuwählen. Daher muß es auch möglich sein, den dynamischen Typ einer Variablen zur Laufzeit zu ermitteln.

Dies ist aber nur möglich, wenn der Compiler spezielle Informationen in das Programm einbettet, die es erlauben, den dynamischen Typ einer Variablen festzustellen. Nachdem fast alle Programmiersprachen solche Typtests auch als Teil der Sprache zur Verfügung stellen, ist es eigentlich der Regelfall, daß diese Informationen zur Verfügung stehen.

Wie diese Informationen aussehen, unterscheidet sich stark und ist von der Programmiersprache und unter Umständen auch vom Compiler abhängig. Wie sie unter Pow!-Oberon-2 aussehen, wurde bereits ausführlich beschrieben. Es handelt sich um die Laufzeittypinformation (RTTI), die auch der Garbage Collector intensiv nutzt.

In Oberon-2 gibt es für jeden Record-Datentyp solche Typinformationen. Sie enthalten neben einer Beschreibung der einzelnen Felder zum Beispiel auch eine Methodentabelle. Damit man weiß, welches Objekt welchen Typ hat, wird vor dem Datenbereich des Objektes ein Zeiger auf die Typinformationen abgelegt.

Jedes Objekt hat also, wie in der Abbildung 58 skizziert wird, einen Zeiger auf seine Typinformationen, wob[illegible]ormalerw[illegible] verborgen bleibt. Über

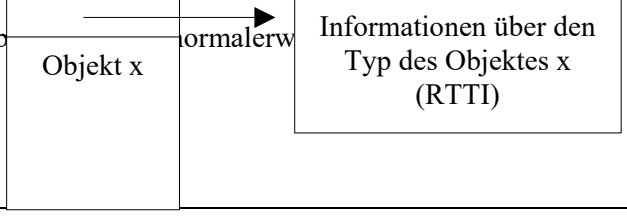


Abb. 57: Typdeskriptor

diesen Zeiger läßt sich natürlich auch der dynamische Typ bestimmen, der ja gerade durch die RTTI beschrieben wird.

In anderen Programmiersprachen ist dies ähnlich implementiert. Allen haben auf jeden Fall eine Gemeinsamkeit. Sie können den dynamischen Typ einer Variablen zur Laufzeit feststellen.

7.2.3 Heap Browser

Der 'Heap Browser' ist eine Komponente des Visualisierungssystem. Es geht dabei um die Möglichkeit, alle dynamisch angelegten Objekte zu ermitteln. Genauer gesagt muß das Visualisierungssystem in der Lage sein, alle dynamisch angelegten Speicherbereiche abzufragen und den dynamischen Typ der darin enthaltenen Objekte herauszufinden.

Daß diese Funktionalität keine Selbstverständlichkeit ist, zeigt ein kleiner Ausflug in das Windows-API. Dort gibt es zwar eine Funktion namens *HeapWalk*, mit der man alle benutzten Speicherbereiche eines Programmes ermitteln kann. (Allerdings ist diese Funktion in Windows 95 nicht implementiert. Das bedeutet, daß diese Funktion nur von Windows NT und neuerdings auch von Windows 98 zur Verfügung gestellt werden.) *HeapWalk* betrachtet aber nur dynamisch angelegte Speicherbereiche und liefert keine Information über den dynamischen Typ der darin liegenden Objekte. Da das Visualisierungssystem Objekte visualisieren soll, muß es alle erzeugten Objekte und ihren dynamischen Typ ermitteln können.

Die Bezeichnung "Heap Browser" ist übrigens an die Bezeichnung "Internet Browser" angelehnt. Ähnlich wie sich ein Benutzer mit einem Internet Browser durch die HTML-Seiten des Internets durchbewegen und sie sich anschauen kann, soll der Benutzer mit einem Heap Browser durch die Welt der Objekte eines Programmes navigieren können und sich einzelne Objekte anschauen können.

7.2.4 Paralleles Auslesen von Daten

Eine der wesentlichen technischen Voraussetzungen ist gleichzeitig auch die schwierigste. Moderne Betriebssysteme wie Windows 98 und Windows NT schützen nämlich den Speicher eines Programmes vor dem Zugriff anderer Programme. Daher ist es ohne weitere Vorkehrungen nicht möglich, auf Speicherbereiche anderer Programme zuzugreifen. Der Zugriff auf andere Programme ist aber für das Visualisierungssystem von entscheidender Bedeutung.

Es liegt auf der Hand, daß das Visualisierungssystem auf den Speicher jener Objekte zugreifen muß, die es visualisieren soll. Dies kann man auf zwei Arten erreichen. Entweder macht man das

Visualisierungssystem zu einem Teil des visualisierten Programmes oder man benutzt spezielle Mechanismen um auf andere Programme zuzugreifen.

Die erste Variante, bei dem das Visualisierungssystem zu einem Teil des visualisierten Programmes wird, ist in der Regel leichter zu implementieren. Die Visualisierung muß nämlich nicht auf ein anderes Programm zugreifen, weil es Teil jenes Programmes ist, das visualisiert werden soll. Daher können die oben erwähnten Einschränkungen moderner Betriebssystemen solchen Visualisierungssystemen keine Probleme bereiten.

Solche Visualisierungssysteme werden meist als Bibliothek implementiert, damit die Visualisierung leicht in bestehende Programme eingebaut werden kann. Um die Visualisierung anzustoßen, muß man an geeigneten Stellen im Quellcode des visualisierten Programmes Funktionen des Visualisierungssystem aufrufen. Natürlich kann das Visualisierungssystem auch als objektorientierte Klassenbibliothek implementiert sein. Das ändert aber nichts an der Tatsache, daß auch in diesem Fall der Quellcode des visualisierten Programmes geändert werden muß, um an geeigneten Stellen Methoden des Visualisierungssystem aufzurufen.

Damit ist der wesentliche Nachteil dieser Variante herausgearbeitet. Sobald ein Programm visualisiert werden soll, muß der Quellcode des Programmes geändert werden. Das bedeutet, daß der Quellcode des visualisierten Programmes verfügbar sein muß. Dies muß aber nicht immer der Fall sein. Dann kann das Programm mit dieser Methode nicht visualisiert werden.

Außerdem ist eine Veränderung des visualisierten Programmes nicht ganz unkritisch. Wer kann mit Sicherheit feststellen, daß die hinzugefügten Aufrufe des Visualisierungssystem die eigentliche Funktionalität des Programmes nicht beeinflussen. Wie leicht sich Fehler in ein Programm einschleichen kann, ist jedem Entwickler bewußt. Genauso leicht können auch Fehler in der Visualisierung entstehen.

Natürlich ist jeder Fehler unerwünscht. Aber in diesem Fall gibt es eine Klasse von Fehlern, die sogar zu einem anderen Verhalten des visualisierten Programmes führen können. Man kann sich leicht einen Fehler vorstellen, der dazu führt, daß eine Speicherzelle geändert wird, die eigentlich nicht verändert werden sollte. Im günstigsten Fall wird eine Speicherzelle außerhalb des Adreßraumes angesprochen. Dann wird das Programm abgebrochen, weil das Betriebssystem einen illegalen Speicherzugriff erkennt. Das Programm funktioniert dann zwar nicht, aber der Benutzer hat keine Mühe, den Fehler zu erkennen und entsprechende Gegenmaßnahmen einzuleiten.

Dies ist aber nicht der Fall, wenn es sich um eine gültige Adresse handelt. Dann wird ein Wert verändert, der eigentlich nicht verändert werden sollte. Das Betriebssystem erkennt diesen Fehler in aller Regel nicht. Welcher Wert fälschlicherweise verändert wird, hängt davon ab, wie die Daten

im Speicher angeordnet sind. Da die Visualisierung Teil des visualisierten Programmes ist, ist es leicht denkbar, daß es sich bei der fälschlicherweise veränderten Speicherzelle um Daten des visualisierten Programmes handelt. Dann hat der Fehler im Visualisierungssystem einen Wert des visualisierten Programmes geändert, was ein anderes Verhalten des visualisierten Programmes nach sich ziehen kann. Das bedeutet, daß das visualisierte Programm mit der Visualisierung ein anderes Verhalten an den Tag legt als ohne Visualisierung. Dies kann auf keinen Fall das Ziel der Visualisierung sein, insbesondere wenn man bedenkt, daß solche Fehler für den Benutzer gar nicht sofort erkennbar sein müssen.

Das Visualisierungssystem kann das visualisierte Programm aber nicht nur durch mögliche Fehler beeinflussen. Dadurch, daß es Teil des visualisierten Programmes ist, beeinflußt es auch das Laufzeitverhalten des Programmes. Das Programm muß im Vergleich zur Situation ohne Visualisierung zusätzlich die Visualisierungsfunktionen ausführen, was durchaus einige Zeit kosten kann. Gerade die aufwendigen graphischen Berechnungen können eine große Menge an CPU-Zyklen verschlingen. Für die meisten Applikationen spielt dies sicherlich keine Rolle. Sie liefern dasselbe Ergebnis unabhängig davon, ob sie schnell oder langsam ablaufen. Handelt es sich dagegen um zeitgesteuerte Applikationen, dann kann diese Einflußnahme das Ergebnis verändern.

Um diese potentiellen Probleme von vornherein auszuschließen, wurde das Visualisierungssystem von Anfang an als eigenständige Applikation konzipiert. Das bedeutet, es ist nicht Teil des visualisierten Programmes, sondern es ist eine eigenes Programm, das parallel zum visualisierten Programm läuft. Dies schließt zum einen aus, daß ein Fehler im Visualisierungssystem das Verhalten des visualisierten Programmes beeinflußt, und zum anderen in gewissen Maße auch die Einflußnahme auf das Laufzeitverhalten des visualisierten Programmes. Als eigenständige Applikation werden die von der Visualisierung benötigten CPU-Zyklen nämlich nicht dem visualisierten Programm zugerechnet.

Zuletzt gibt es aber noch einen viel wichtigeren Vorteil. Der Quellcode des visualisierten Programmes muß mit dieser Technik nicht verändert werden. Daher muß er auch nicht verfügbar, wodurch jedes Programm – egal ob dessen Quellcode vorhanden ist oder nicht – visualisiert werden kann.

Dadurch, daß das Visualisierungssystem als eigenständige Applikation läuft, ergibt sich aber das Problem, daß die Visualisierung nicht mehr so leicht auf die Daten des visualisierten Programmes zugreifen kann. Wie schon erwähnt, wird dies durch die Schutzmaßnahmen moderner Betriebssysteme verhindert.

Es muß daher eine Möglichkeit geschaffen werden, daß das Visualisierungssystem auf parallel laufende Programme Zugriff hat. Um eine Lösung dafür zu finden, muß man sich einer anderen

Klasse von Anwendungen zuwenden, den Debuggern. Vergleicht man das Visualisierungssystem mit Debuggern, dann erkennt man, daß einiges gemeinsam haben. Debugger sind ebenfalls eigenständige Applikationen, die andere Programme steuern und deren Daten anzeigen. Folglich liegt es nahe, dieselben Techniken wie ein Debugger zu verwenden, um auf Speicher von fremden Programmen zuzugreifen. Ein Blick in die Dokumentation moderner Betriebssysteme enthüllt sehr schnell, daß Debugger zu diesem Zweck Funktionen des Betriebssystems benutzen.

Daß solche Funktionen vorhanden sein müssen, ist leicht zu zeigen. Wenn ein Betriebssystem den Zugriff auf den Speicher anderer Programme unterbindet, dann kann man ohne diese Funktionen keinen Debugger implementieren. Da ein Debugger aber ein sehr wichtiges Werkzeug ist, können es sich Entwickler von kommerziellen Betriebssystemen nicht leisten, solche Funktionen nicht zur Verfügung zu stellen.

7.3 Zweck eines graphischen Debugger

Im vorangegangenen Unterkapitel wurde bereits auf den Debugger eingegangen. Für Programmentwickler ist es ein unersetzliches Werkzeug. Bei größeren Projekten ohne Debugger auszukommen, ist nur sehr schwer möglich.

Debugger arbeiten meist als eigenständige Applikationen, die das untersuchte Programm starten und steuern. Nach dem Start zeigt der Debugger in der Regel die Befehle an, aus denen das Programm besteht. Abhängig von der Auswahl des Benutzers kann der Debugger die Befehle in der verwendeten Programmiersprache oder als Assembler-Befehle anzeigen lassen.

Der Debugger kann auf unterschiedliche Weise die Ausführung des Programmes steuern. Sehr beliebt ist zum Beispiel die Möglichkeit, das Programm an einer bestimmten Stelle anzuhalten. Dadurch kann der Programmierer beispielsweise prüfen, ob ein bestimmter Programmteil auch wirklich ausgeführt wird. Außerdem wird auch gerne die Möglichkeit genutzt, einzelne Befehle des Programmes auszuführen. Damit kann der Programmierer mitverfolgen, welche Programmteile ausgeführt werden.

Ist das Programm angehalten, dann kann der Debugger auch die Daten des Programmes anzeigen. Neben einfachen Variablen können auch Datenstrukturen angezeigt werden. Damit der Debugger die Daten korrekt interpretiert, nutzt er sogenannte Debug-Informationen. Dabei handelt es sich um Informationen, die der Compiler zusätzlich zum erzeugten Code generiert und abspeichert.

Datenstrukturen können ziemlich kompliziert und unübersichtlich werden. Mit einem Debugger kann eine Datenstruktur, wie zum Beispiel eine verkettete Liste nicht zur Gänze angezeigt werden.

Statt dessen muß man sich den Zeiger suchen, der auf das erste Element zeigt. Geht man den Zeiger des ersten Elementes nach, kommt man zum zweiten Element. Dies kann man für jedes Element wiederholen, bis man am Ende der Liste angekommen ist. Diese Tätigkeit kann zu einer zeitraubenden Arbeit werden, wenn man nach jedem Schritt kontrollieren will, ob die verkettete Liste noch in Ordnung ist. Außerdem sieht der Benutzer immer nur ein Element der Liste. Wenn er aber alle Elemente kontrollieren muß, hat er keine andere Chance als alle Elemente durchzugehen.

Für solche Fälle wäre ein graphischer Debugger von Nutzen, der im Gegensatz zu einem klassischen Debugger nicht nur einzelne Records oder Variablen anzeigen kann, sondern Datenstrukturen zur Gänze darstellen kann. Das heißt er muß die Datenstrukturen kennen und interpretieren können.

Bleibt man beim Beispiel der verketteten Liste, dann würde diese so dargestellt, daß jedes Element der Liste durch ein kleines Rechteck symbolisiert wird. Zeiger von einem Element auf das nächste kann man durch einen Pfeil darstellen. Der wesentliche Vorteil dieser Ansicht ist, daß man alle Elemente der Liste auf einmal sieht. Der Benutzer hat damit einen Überblick über die gesamte Datenstruktur. Interessiert er sich für ein spezielles Element, dann soll er zum Beispiel durch einen Doppelklick das Element anschauen können.

Dies könnte sicherlich bei der Fehlersuche von Vorteil sein. Der Benutzer müßte bei einer verketteten Liste nicht immer wieder alle Zeiger bis zum Ende der Liste verfolgen, sondern könnte schon anhand der Grafik sehen, ob alle Elemente der Liste vorhanden sind.

Der Zweck eines graphischen Debuggers ist es also Datenstrukturen zur Gänze anzuzeigen um so dem Benutzer möglichst viel Information zu geben, die ihm bei der Suche nach Fehlern helfen können.

8 Technische Details zur Visualisierung

Im letzten Unterkapitel wurden die technischen Voraussetzungen für die Visualisierung besprochen. In diesem Kapitel wird die gesamte technische Umsetzung des Visualisierungssystem erläutert. Nach einem Überblick über die einzelnen Komponenten des Systems wird auf jede Komponente im Detail eingegangen.

8.1 Das Visualisierungssystem und seine Komponenten

Wie schon im vorigen Kapitel angedeutet wurde, ist das Visualisierungssystem eine eigenständige Applikation. Die beobachteten Anwendungen laufen parallel zur Visualisierungsapplikation und haben keine Ahnung von der Existenz der Visualisierung. Sie sind also völlig unbeeinträchtigt von der Visualisierung. Abbildung 59 versucht diese Situation darzustellen. Links sind zwei Anwendungen eingezeichnet, die gerade von der Visualisierung beobachtet werden. Die beiden Pfeile von den beobachteten Anwendungen zur Visualisierung stellen den lesenden Zugriff der Visualisierung auf die Daten dieser Anwendungen dar. Im übrigen kann das Visualisierungssystem beliebig viele Programme gleichzeitig darstellen.

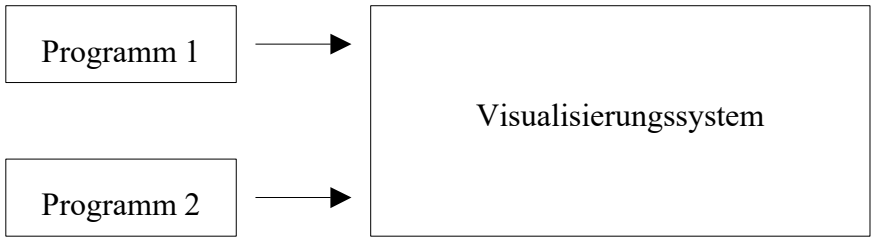


Abb. 60: Visualisierungssystem

Abbildung 61 zeigt die Komponenten des Visualisierungssystems und die Kommunikationskanäle zwischen den einzelnen Komponenten.

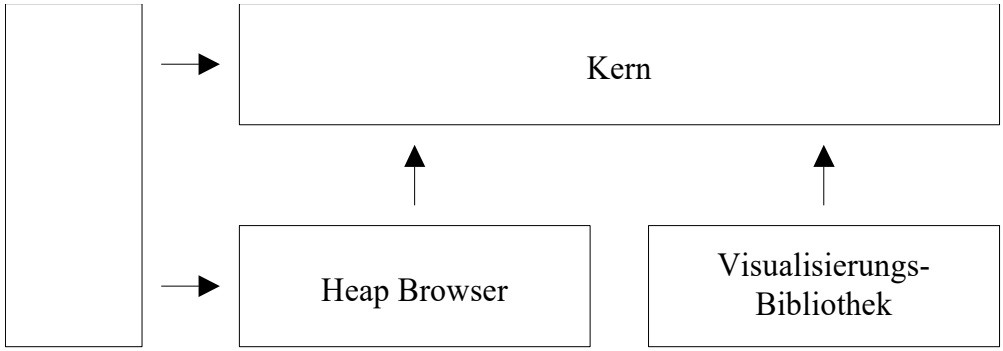


Abb. 62: Aufbau des Visualisierungssystem

sich hierbei um Funktionen handelt, die auch von Debuggern benutzt werden, ist der Name durchaus gerechtfertigt.

Die Aufgaben der Komponente Debugger liegen im Wesentlichen im Einlesen der Daten der beobachteten Programme. Darüber hinaus kümmert sie sich auch noch um die Steuerung der Programme.

Die über die beobachteten Programme gelieferten Informationen werden vor allem vom Heap Browser und dem Kern des Systems benötigt. Der Heap Browser hat die Aufgabe, alle Objekte der beobachteten Programme zu ermitteln und dem Benutzer in geeigneter Form zu präsentieren. Neben dem technisch aufwendigen Prozeß der Analyse der Laufzeitinformationen der beobachteten Programme stellt der Heap Browser auch eine Benutzerschnittstelle zur Verfügung, mit dessen Hilfe der Benutzer Objekte auswählen kann.

Hat der Benutzer ein Objekt ausgewählt, dann kommt die Visualisierungsbibliothek zum Tragen. Sie ist im Wesentlichen eine Sammlung von Visualisierungsobjekten, von denen jedes einzelne ein bestimmtes Objekt visualisieren kann. Im Visualisierungsobjekt steckt die eigentliche Visualisierung, die vom Kern nur mehr angestoßen werden muß.

Der Kern selbst verwaltet eigentlich nur alle aktiven Visualisierungsobjekte und stellt dem Benutzer eine Benutzeroberfläche zur Verfügung, in die alle Komponenten integriert sind.

8.2 Zusammenspiel der Komponenten

Im folgenden wird demonstriert, wie die beschriebenen Komponenten zusammenarbeiten. Alles beginnt damit, daß eine Anwendung aus dem Visualisierungssystem heraus gestartet wird. Dazu gibt es im Menü *File* den Menüeintrag *Start App*. Intern wird die Anwendung nicht einfach als normaler Prozeß gestartet, sondern mit Hilfe der Komponente 'Debugger'. Diese stellt eine Funktion zur Verfügung, die Applikationen in einer Form startet, daß sie vom Visualisierungssystem kontrolliert werden kann. Dazu benutzt sie die Debugging-Funktionen des Betriebssystems.

Wenn die Anwendung läuft, dann tritt der Heap Browser in Aktion, um dem Benutzer die Möglichkeit zu geben, die gewünschten Objekte auszuwählen. Dazu zeigt der Heap Browser alle laufenden Programme an. Über einen Navigationsmechanismus, der später noch beschrieben wird, kann sich der Benutzer nun eine oder mehrere Objekte auswählen. Damit der Heap Browser die Objekte eines Programmes anzeigen kann, bedient er sich der Funktionen der Komponente

'Debugger'. Sie stellt zu diesem Zweck Funktionen für das Lesen von Daten eines laufenden Programmes bereit.

Hat der Benutzer Objekte ausgewählt, dann werden diese Objekte dem Kern übermittelt. Der Kern erhält dabei lediglich die Adresse der ausgewählten Objekte. Über Funktionen des Heap Browsers stellt der Kern nun fest, welchen dynamischen Typ diese Objekte besitzen. Die Feststellung des dynamischen Typs wurde bereits im vorigen Kapitel ausführlich erläutert. Durch Zugriff auf die Laufzeittypinformationen kann der Heap Browser feststellen, zu welcher Klasse ein Objekt gehört.

Bis zu diesem Zeitpunkt ist noch keine Visualisierung passiert. Erst jetzt beginnt die Visualisierung, in dem für jedes ausgewählte Objekt ein Visualisierungsobjekt angelegt wird. Dies klingt im ersten Moment umständlich. Das hat aber gute Gründe, wie man im folgenden sehen wird.

Die Visualisierung findet also nicht im Kern oder im Heap Browser statt, sondern dadurch, daß ein Visualisierungsobjekt angelegt wird. Jedes Visualisierungsobjekt ist dabei so gestaltet, daß es genau ein Objekt visualisieren kann. Das bedeutet, wenn man zwei Objekte visualisieren will, dann benötigt man auch zwei Visualisierungsobjekte. Somit gibt es im Visualisierungssystem immer genau so viele Visualisierungsobjekte wie ausgewählte Objekte.

Verwaltet werden die Visualisierungsobjekte im Kern, der alle Visualisierungsobjekte in einer Adjazenzmatrix speichert. Wenn der Kern die ausgewählten Objekte darstellen will, dann geht er einfach alle aktiven Visualisierungsobjekte durch und teilt ihnen durch einen Methodenaufruf mit, daß sie ihr Objekt anzeigen sollen. Das Visualisierungsobjekt liest dann über den Heap Browser die aktuellen Daten jenes Objektes, für dessen Visualisierung es zuständig ist. Danach wird der Zustand des Objektes am Bildschirm angezeigt.

Die Frage nach der Herkunft der Visualisierungsobjekte ist leicht beantwortet. Sie stammen aus der Komponente Visualisierungsbibliothek. Wirft man einen Blick auf diese Komponente, dann erkennt man, daß es sich hierbei um eine Klassenbibliothek handelt. Die Klassen dieser Komponente nennen wir – analog zu den Visualisierungsobjekten – Visualisierungsklassen. Ein Visualisierungsobjekt entsteht nun dadurch, daß ein Objekt von einer der Visualisierungsklassen angelegt wird.

Eine Visualisierungsklasse kann aber nicht jedes beliebige Objekt visualisieren, sondern sie ist normalerweise nur für eine Klasse von Objekten zuständig. Eine Visualisierungsklasse, die zum Beispiel für die Darstellung von verketteten Listen zuständig ist, kann nicht binäre Bäume darstellen.

Damit auch dem Kern bekannt ist, welche Klasse wofür zuständig ist, gibt es in der Komponente Visualisierungsbibliothek noch eine einfache Visualisierungsdatenbank. Dabei handelt es sich um eine Tabelle, die jeder Visualisierungsklasse ihren Aufgabenbereich zuordnet. Genauer gesagt enthält sie Wertepaare. Der erste Wert eines Eintrags enthält den Namen einer Visualisierungsklasse und der zweite Wert enthält den Namen jener Klasse, für den die Visualisierungsklasse zuständig ist.

Sucht man nun zu einem bestimmten Objekt eine passende Visualisierungsklasse, dann muß man zwei Schritte durchführen. Man muß, wie bereits beschrieben, den dynamischen Typ des Objektes feststellen. Dadurch erhält man den Namen der Klasse, zu der das Objekt gehört. Nun muß man im zweiten Schritt nur mehr die Visualisierungsdatenbank durchgehen und jene Einträge auswählen, deren erster Wert mit dem eben ermittelten Klassennamen übereinstimmen. Dadurch erhält man alle Visualisierungsklassen, die das ausgewählte Objekt darstellen können.

Genauso geht auch der Kern vor. Er ermittelt zuerst den dynamischen Typ aller ausgewählten Objekte und ermittelt danach über den beschriebenen Weg die passenden Visualisierungsklassen. Gibt es bei einem Objekt mehr als eine Visualisierungsklasse, dann wird dem Benutzer die Entscheidung überlassen, welche Visualisierungsklasse ausgewählt werden soll. Steht eine Visualisierungsklasse fest, dann wird ein Objekt dieser Klasse angelegt. Dem angelegten Visualisierungsobjekt wird die Adresse des zu visualisierenden Objektes mitgeteilt, damit sich das Visualisierungsobjekt die Daten des Objektes holen kann. Danach wird das Visualisierungsobjekt aufgefordert, die Darstellung des Objektes am Bildschirm zu zeichnen.

8.3 Die Debugger-Komponente

Technisch gesehen ist diese Komponente die wichtigste, weil moderne Betriebssysteme den Zugriff auf Speicher anderer Programme schützen. Außerdem ist sie die einzige Komponente, für die es zwei Versionen geben muß, nämlich eine für 16-Bit Windows und eine für 32-Bit Windows. Der Grund liegt darin, daß der Zugriffsschutz unterschiedlich implementiert wurde. Durch einfache Maßnahmen konnte man diesen Schutz im 16-Bit System umgehen. Im 32-Bit System ist dies allerdings nicht mehr möglich. Hier muß man die Funktionen des Betriebssystems benutzen, um auf Speicherbereiche anderer Programme zugreifen zu können.

Nachdem das Visualisierungssystem mit 16-Bit Pow! entwickelt wurde, wird im folgenden vor allem die 16-Bit Version beschrieben. Im Anschluß wird eine 32-Bit Version skizziert. Davon existiert aber noch keine vollständige Implementierung.

8.3.1 16-Bit Version

Wie bereits kurz angedeutet, gibt es im 16-Bit Windows Möglichkeiten, den Zugriffsschutz zu umgehen. Um diese Technik zu erläutern, muß ein Detail von Dynamic Link Libraries (DLLs) genauer besprochen werden.

Eine DLL ist eine Sammlung von Funktionen, die von einem Programm aufgerufen werden können. Natürlich kann auch eine DLL Funktionen einer anderen DLL aufrufen. Trotzdem wird der Einfachheit halber im folgenden nur davon gesprochen, daß ein Programm eine DLL benutzt. Damit ist gemeint, daß sowohl ein Programm als auch eine DLL eine andere DLL benutzen kann.

Neben den lokalen Daten innerhalb der Funktionen, kann die DLL auch über globale Daten verfügen, auf die alle Funktionen der DLL zugreifen können. Auf diese Daten können sogar die Programme zugreifen, die die DLL benutzen. Im Gegensatz zu statischen Bibliotheken wird die DLL aber nicht beim Linken sondern erst beim Laden zum Programm gebunden. Daher ist es von entscheidender Bedeutung, wie mit den globalen Daten verfahren wird.

Generell gibt es zwei Möglichkeiten. Bei der ersten Variante werden die globalen Daten jedesmal neu angelegt, wenn ein neues Programm hinzukommt. Jedes Programm verwendet somit seine eigene Kopie der globalen Daten der DLL. In Abbildung 63 ist diese Situation auf der linken Seite dargestellt. Mehrere Programme namens P1, P2 bis Pn benutzen die DLL D1. Diese DLL verfügt über globale Daten. Folgerichtig gibt es mehrere Kopien der globalen Daten, die in der Abbildung mit G1, G2 bis Gn bezeichnet wurden.

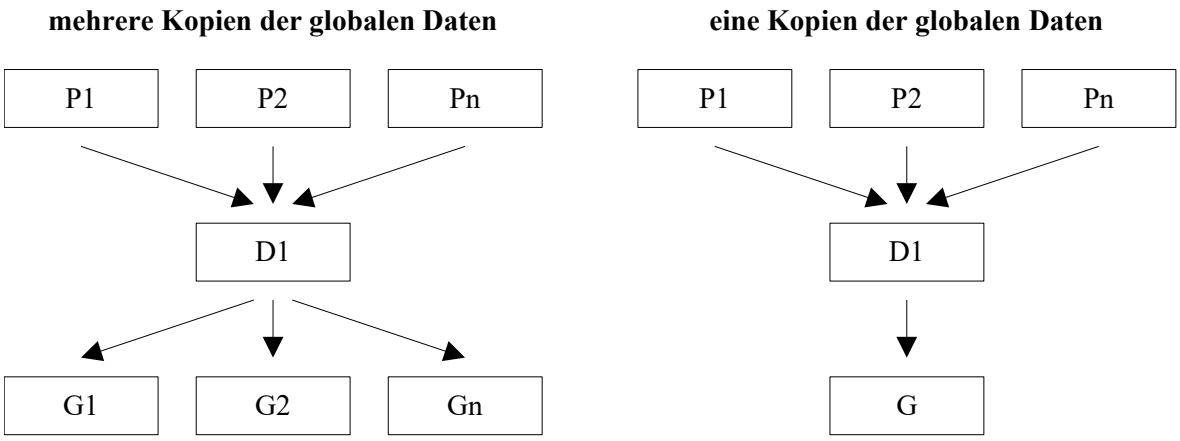


Abb. 64: Behandlung globaler Daten in Dynamic Link Libraries

Bei der zweiten Möglichkeit werden die globalen Daten nur einmal angelegt. Dies passiert, wenn die DLL in den Hauptspeicher geladen wird. Das hat aber zur Folge, daß ein zweites Programm auf

dieselben globalen Daten zugreift. In diesem Fall teilen sich also alle Programme, die eine DLL gerade benutzen, die globalen Daten. Alle Programme greifen auf dieselben Daten zu.

Über diesen Mechanismus kann man Daten zwischen verschiedenen Programmen austauschen. Das Betriebssystem blockt nämlich den Zugriff auf die globalen Daten nicht ab, sodaß alle Programme auf diesen Speicherbereich zugreifen können, die die DLL gerade benutzen.

In 16-Bit Windows ist die zweite Variante implementiert (vgl. [SAR96]). Das bedeutet, daß die globalen Daten einer DLL immer nur einmal existieren. Dabei ist nicht von Bedeutung, wieviele Programme gerade die DLL benutzen. Alle können auf die globalen Daten der DLL zugreifen.

Diesen Mechanismus macht sich auch das Visualisierungssystem zu Nutze. Mit diesem Mechanismus ist es nämlich sehr einfach möglich, auf Speicherbereiche anderer Programme zuzugreifen, ohne daß man komplizierte Debugging-Funktionen des Betriebssystems nutzen muß.

Konkret geht das Visualisierungssystem folgendermaßen vor. Es geht davon aus, daß alle untersuchten Programme mit der dynamischen Version des Laufzeitsystems gebunden wurden. Das hat zur Folge, daß alle Programme das Laufzeitsystem als DLL benutzen. Das Laufzeitsystem selbst ist so organisiert, daß alle dynamisch angelegten Objekte mit Hilfe globaler Daten verwaltet werden. Das bedeutet, daß jedes Programm auf die dynamisch angelegten Objekte eines anderen Programmes zugreifen kann. Da das Visualisierungssystem selbst auch die dynamische Version des Laufzeitsystems benutzt, kann das Visualisierungssystem sehr leicht auf die Objekte der untersuchten Programme zugreifen.

Abbildung 65 verdeutlicht diese Situation noch einmal. Die beobachteten Programme sind darin mit P1 und P2 bezeichnet, das Visualisierungssystem bekam das Kürzel V. Alle Programme benutzen das Laufzeitsystem als DLL und greifen daher auf die DLL RTS.DLL zu. Diese wiederum verwaltet die dynamischen Objekte aller Programme.

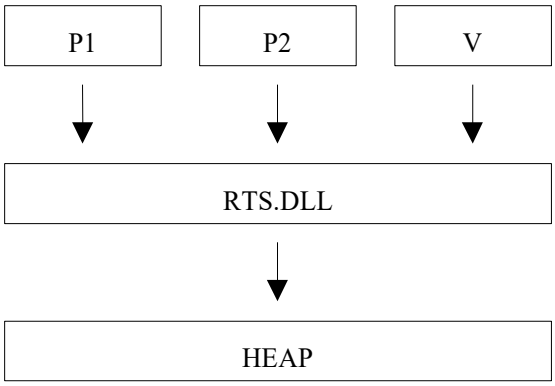


Abb. 66: Visualisierungs- und Laufzeitsystem

8.3.2 32-Bit Version

Durch den Trick, das Laufzeitsystem als Kommunikationsmedium zu nutzen, ist die Implementierung der Debugging-Komponente im 16-Bit Windows wesentlich einfacher. Im 32-Bit Windows ist dies leider nicht mehr möglich, da die globalen Daten einer DLL für jedes Programm extra angelegt werden. Das bedeutet, daß das Laufzeitsystem nicht mehr als Datenaustausch dienen kann. Natürlich gibt es auch im 32-Bit Windows die Möglichkeit, globale Daten so anzulegen, daß sie nur einmal angelegt werden. Allerdings würde das nicht zum Ziel führen, weil noch weitere Gründe dagegen sprechen.

Ein wichtiger Grund liegt darin, wie die beiden System Multitasking behandeln. 16-Bit Windows verfügt lediglich über kooperatives Multitasking während 32-Bit Windows über preemptive Multitasking verfügt. Kooperatives Multitasking bedeutet, daß die CPU nur dann an einen anderen Task abgegeben wird, wenn der gerade laufende Task die CPU freiwillig abgibt. Es ist also nicht möglich, daß ein Task mitten in seiner Verarbeitung unterbrochen wird. Bei preemptive Multitasking ist dies dagegen sehr wohl möglich.

Welche Auswirkungen hat dies auf das Visualisierungssystem? Im 16-Bit Windows hat dies keine Auswirkungen. Das Visualisierungssystem kann darauf vertrauen, daß sich das untersuchte Programm nicht verändert, solange das Visualisierungssystem gerade den Zustand des Programmes ermittelt. Dies ist im 32-Bit Windows nicht der Fall. Das Visualisierungssystem kann jederzeit unterbrochen werden und das untersuchte Programm kann die CPU erhalten. Dadurch kann das untersuchte Programm weiterlaufen, obwohl das Visualisierungssystem noch nicht den gesamten Zustand ermittelt hat. Würde das Visualisierungssystem danach wieder an die Reihe kommen, findet es das untersuchte eventuell in einem ganz anderen Zustand vor als vor der Unterbrechung.

Wenn in dieser Situation mit der Zustandsermittlung fortgefahren wird, würde das Visualisierungssystem den alten und neuen Zustand vermischen, ohne daß es davon etwas bemerkt.

Das würde dazu führen, daß der Zustand des untersuchten Programmes nicht korrekt bestimmt werden kann. Daher muß das Visualisierungssystem dafür sorgen, daß das untersuchte Programm während der Ermittlung seines Zustandes angehalten wird. Ein Blick in die Dokumentation des 32-Bit Windows [SDK01] enthüllt hierfür folgende Funktionen:

```
DWORD SuspendThread(HANDLE);
DWORD ResumeThread (HANDLE);
```

Mit *SuspendThread* kann man einen Thread anhalten und mit *ResumeThread* kann man einen angehaltenen Thread wieder weiterlaufen lassen. Mit diesen beiden Funktionen ist aber noch nicht das Auslangen gefunden. Man soll das Programm auch an einer bestimmten Stelle anhalten können. Dazu benötigt man Breakpoints.

Breakpoints sind bereits aus Debuggern bekannt. Mit ihnen kann man eine Stelle in einem Programm angeben, an der das Programm angehalten werden soll, wenn es diese Stelle erreicht. Neben dieser Art von Breakpoints gibt es auch noch die Möglichkeit auf Daten Breakpoints zu legen. Dann wird das Programm angehalten, wenn Daten verändert werden.

Die 32-Bit Version des Visualisierungssystem arbeitet auch mit Breakpoints. Neben den üblichen Breakpoints wurden noch weitere Arten definiert, die das Arbeiten mit dem Visualisierungssystem erleichtern. Insbesondere sind die Watchpoints zu erwähnen. Dabei handelt es sich um spezielle Breakpoints. Im Gegensatz zu normalen Breakpoints wird die Ausführung eines Programmes an einem Watchpoint nur eine bestimmte Zeit lang aufgehalten. Danach wird das Programm wieder fortgesetzt. Die Dauer der Unterbrechung hängt davon ab, wie lange das Visualisierungssystem benötigt, den Zustand des Programmes zu ermitteln und zu zeichnen. Ein Watchpoint dient also dazu, einen Punkt im Programm anzugeben, an die Visualisierung auf den neuesten Stand gebracht werden soll.

Die folgende Tabelle zeigt eine Übersicht der verschiedenen Break- und Watchpoints, die in der 32-Bit Version verwendet werden.

Break- / Watchpoint	**wird ausgelöst durch**	**Visualisierung wird erneuert**	**Programm wird angehalten**
Code Breakpoint	Erreichen einer bestimmten Stelle im Programmcode	Ja	Ja
Data Breakpoint	Ändern bestimmter Daten	Ja	Ja
Code-Data Breakpoint	Erreichen einer bestimmten Stelle im Programmcode einer Methode, die für eine bestimmte	Ja	Ja

Break- / Watchpoint	wird ausgelöst durch	Visualisierung wird erneuert	Programm wird angehalten
	Instanz aufgerufen wurde		
User Breakpoint	Benutzer	Ja	Ja
Code Watchpoint	Erreichen einer bestimmten Stelle im Programmcode	Ja	Nein
Data Watchpoint	Ändern bestimmter Daten	Ja	Nein
Code-Data Watchpoint	Erreichen einer bestimmten Stelle im Programmcode einer Methode, die für eine bestimmte Instanz aufgerufen wurde	Ja	Nein

Zur Implementierung der unterschiedlichen Break- bzw. Watchpoints bedient sich das Visualisierungssystem der Debugging-Funktionen des Betriebssystems. Am Beginn aller Aktivitäten steht das Starten einer beobachteten Applikation. Dies geschieht mit Hilfe der Funktion *CreateProcess*, deren Schnittstelle wie folgt definiert ist.

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
                   LPTSTR lpCommandLine,
                   LPSECURITY_ATTRIBUTES lpProcessAttributes,
                   LPSECURITY_ATTRIBUTES lpThreadAttributes,
                   BOOL bInheritHandle,
                   DWORD dwCreationFlags,
                   LPCTSTR lpCurrentDirectory,
                   LPSTARTUPINFO lpStartupInfo,
                   LPPROCESS_INFORMATION lpProcessInformation);
```

Diese Funktion wird generell dazu verwendet, um ein Programm zu starten. Um ein Programm nach dem Start debuggen zu können, muß man im Parameter *dwCreationFlags* den Wert *DEBUG_PROCESS* übergeben.

Auf ein Programm, das wie beschrieben gestartet wurde, kann man danach die Debugging-Funktionen anwenden. Die wichtigsten Debugging-Funktionen sind jene zum Lesen bzw. Schreiben eines Speicherbereichs im untersuchten Programm. Es handelt sich dabei um die beiden Funktion *ReadProcessMemory* und *WriteProcessMemory*.

```
BOOL ReadProcessMemory (HANDLE  hProcess,
                        LPCVOID lpBaseAddress,
                        LPVOID  lpBuffer,
                        DWORD   nSize,
                        LPDWORD lpNumberOfBytesRead);
```

```
BOOL WriteProcessMemory(HANDLE  hProcess,
                        LPVOID  lpBaseAddress,
                        LPVOID  lpBuffer,
                        DWORD   nSize,
                        LPDWORD lpNumberOfBytesWritten);
```

Als Abschluß bleibt noch zu erwähnen, daß das Visualisierungssystem auch mit Symbolen arbeiten kann. Um zum Beispiel einen Breakpoint setzen zu können, ist es für den Benutzer angenehmer, wenn er den Quellcode vor sich hat und mit dem Quellcode statt mit dem disassemblierten Code arbeiten kann. Dazu ist es aber notwendig, daß das Visualisierungssystem die Debug-Informationen eines Programmes entschlüsseln kann.

Dabei handelt es sich um Informationen, die der Compiler auf Wunsch zusätzlich zum auszuführenden Code generiert. Darin werden alle Strukturen und Variablen beschrieben, sowie ein Zusammenhang zwischen dem Quellcode und dem übersetzten Code hergestellt. Mit letzterem kann man jeder Zeile des Quellcodes genau jene Befehle zuordnen, die der Quellcodezeile entsprechen. Damit weiß das Visualisierungssystem zum Beispiel, an welche Stelle im übersetzten Code ein Breakpoint gesetzt werden muß, wenn der Benutzer einen Breakpoint auf eine bestimmte Quellcodezeile setzt. Die Informationen über die Variablen und Strukturen benötigt das Visualisierungssystem nicht, weil es sich hier auf die RTTI stützt.

Wie diese Debug-Informationen aufgebaut und strukturiert sind, ist sehr genau in [BON97] beschrieben und wird daher hier nicht mehr diskutiert.

8.4 Heap Browser

Eine ausführliche Dokumentation der Implementierung des Heap Browsers findet man in [PFE97b]. Daher wird im folgenden auf eine Beschreibung der einzelnen Funktionen verzichtet und statt dessen die Funktionsweise des Heap Browsers im Detail erläutert.

Der Heap Browser ist eine wichtige Komponente des Visualisierungssystem. Für den Benutzer ist er wichtig, weil er dem Benutzer die Möglichkeit gibt, jene Objekte auszuwählen, die er gerne beobachten möchte. Aus technischer Sicht ist er von Bedeutung, weil er die RTTI des untersuchten Programmes auswertet. Bevor die technischen Details geklärt werden, sollte man einen Blick darauf werfen, wie der Heap Browser zu bedienen ist.

Dafür steht eine eigene Version des Heap Browsers zur Verfügung. Die Unterschiede zu jener Version, die im Visualisierungssystem eingebaut ist, liegen darin, daß dieser Heap Browser eine

eigenständige Applikation ist und daß er nur die Objekte anzeigt und nicht auswählen läßt. An ihm kann man aber sehr schön zeigen, wie einfach der Heap Browser zu bedienen ist.

Einen ersten Eindruck des Heap Browsers liefert die Abbildung 67. Darin sieht man das Hauptfenster des Heap Browsers. Man kann erkennen, daß sich in der obersten Reihe des Heap Browsers drei Knöpfe befinden. Der Knopf *Refresh* dient zum Erneuern der Anzeige, *Back* dient zum Zurückgehen zur vorigen Seite und *Top-Level* zum Sprung auf die Hauptseite.

Der Begriff *Seite* wird hier für jene zusammengehörigen Informationen verwendet, die der Heap Browser im unteren Bereich anzeigt. Im konkreten Beispiel werden die Felder eines Records *ScreenPane* angezeigt, der im Modul Display des Programmes Life.exe verwendet wird. Wie bei einem Webbrowser kann der Benutzer durch die zur Verfügung stehenden Seiten navigieren. Die Seiten werden dabei immer dynamisch aufgebaut.

In der Abbildung sieht man einen Link auf dem Feld *keys*. Klickt der Benutzer darauf, bekommt er eine Seite angezeigt, in der alle Elemente des Array *keys* enthalten sind. Ebenso zeigt der Heap Browser für jeden Zeiger einen Link an, mit dem der Benutzer verfolgen kann, auf welchen Inhalt der Zeiger verweist.

Es wurde bereits erwähnt, daß der Knopf *Top-Level* zur Hauptseite springt. Auf einer Hauptseite werden alle globalen Variablen eines Moduls angezeigt. Daher gibt es mehrere davon. Jedes Modul eines Programmes hat seine eigene Hauptseite. Über die Comboboxen *Program* und *Module* kann der Benutzer auswählen zu welcher Hauptseite der Knopf *Top-Level* springen soll.

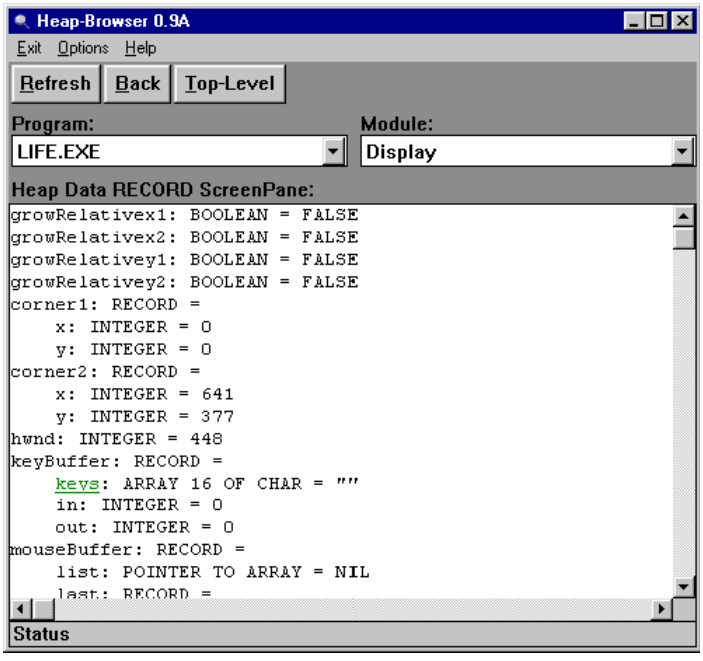


Abb. 68: Heap BrowserDer im Visualisierungssystem eingebaute Heap Browser unterscheidet sich nur unwesentlich von bisher gezeigten Version. Wie schon kurz erwähnt liegt der wesentliche Unterschied darin, daß der Benutzer in ihm Objekte zur Visualisierung auswählen kann. Einen Eindruck davon liefert die Abbildung 69. Sie zeigt ein zusätzliches Element unter der Anzeige der Seiten, nämlich eine Liste der auswählten Objekte. Außerdem gibt es zwei zusätzliche Knöpfe *Add* und *Remove*, mit denen Objekte für die Visualisierung selektiert bzw. deselektiert werden können.

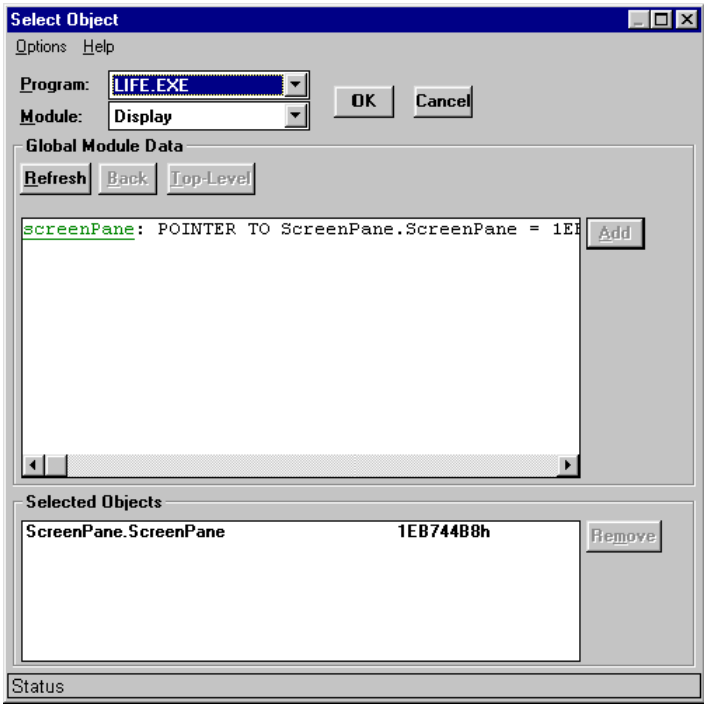


Abb. 70: Heap Browser im VisualisierungssystemNach einem kurzen Ausflug in die Bedienung des Heap Browsers ist es jetzt an der Zeit, die technischen Details zu erläutern. Für die Erledigung seiner Aufgaben arbeitet der Heap Browser sehr eng mit dem Laufzeitsystem zusammen.

Wenn nämlich der Heap Browser gestartet wird, muß er die Combobox *Program* füllen. Zu diesem Zweck muß er Kenntnis von allen gerade laufenden Programmen haben. Diese Information liefert ihm eine Tabelle, die das Laufzeitsystem für seine eigenen Zwecke aufbaut. In dieser Tabelle sind die Namen aller laufenden Programme sowie eine Liste der Module, aus denen die Programme bestehen, eingetragen. Die Module sind nicht in Form eines Namens gespeichert, sondern als Referenz auf den Moduldeskriptor. Der Moduldeskriptor ist eine Information, die der Compiler zusätzlich zum Code generiert und die ein Modul beschreibt. Eine ausführliche Beschreibung des Moduldeskriptors findet man im Unterkapitel Laufzeitsystem.

Das Visualisierungssystem entnimmt dem Moduldeskriptor den Namen des Moduls und füllt damit die zweite Combobox *Module*. Abbildung 71 zeigt den Aufbau dieser Tabelle graphisch. Links befindet sich die Tabelle mit den Namen aller laufenden Programme. In diesem Beispiel läuft gerade ein Programm namens *hello.exe*. Unter der Überschrift *MDPs* sind die Referenzen auf die Moduldeskriptoren eingetragen, aus denen das Programm hello.exe besteht. In diesem Beispiel sind es die Module x und y.

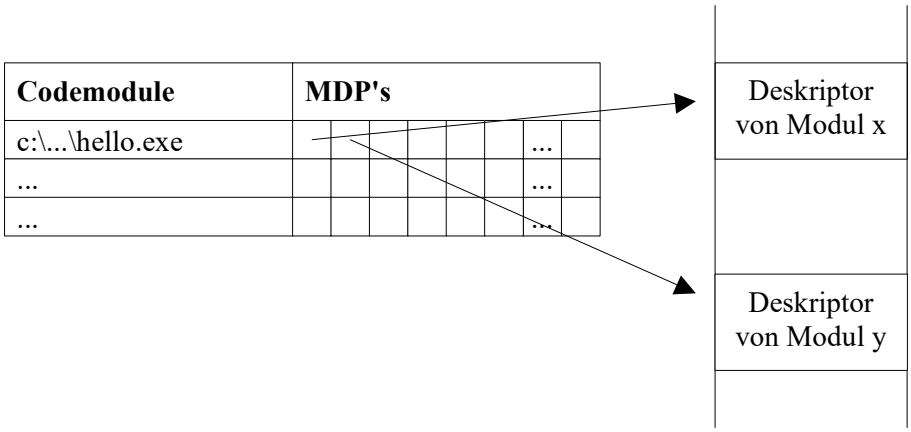


Abb. 72: Tabellen des Laufzeitsystems

Wenn der Benutzer das Programm *hello.exe* in der Combobox *Program* auswählt, dann sucht der Heap Browser alle Moduldeskriptoren zu diesem Programm und zeigt den Namen der Module in der Combobox *Module* an. Wählt der Benutzer danach ein Modul aus, dann zeigt der Heap Browser alle globalen Variablen dieses Moduls an. Damit der Heap Browser weiß, welche globalen Variablen es in einem bestimmten Modul gibt, analysiert er die Run-Time Type Information (RTTI) dieses Moduls. Die RTTI ist eine vom Compiler generierte Zusatzinformation, die unter anderem den Aufbau der globalen Daten beschreibt. Für den Heap Browser sind diese Informationen leicht zu finden, weil sie direkt vor dem Moduldeskriptor im Speicher liegen.

Die RTTI beschreibt aber nur den Aufbau der globalen Daten. Die konkreten Werte sind im globalen Datensegment enthalten. Wie aus den Abbildungen 73 und 74 zu ersehen ist, zeigt der Heap Browser auch die konkreten Werte der jeweiligen Objekte an. Um diese Werte zu ermitteln, bedient sich der Heap Browser einer weiteren Tabelle des Laufzeitsystems, in der alle globalen Datensegmente verzeichnet sind. Jeder Eintrag in dieser Tabelle enthält nicht nur eine Adresse des globalen Datensegmentes sondern auch eine Referenz auf den zugehörigen Moduldeskriptor. Durch Vergleich der Adressen des Moduldeskriptor der beiden Tabellen findet der Heap Browser das passende globale Datensegment zu einem Modul und somit auch die globalen Daten des Moduls.

Der Heap Browser zeigt dann die globalen Daten des Moduls an, wobei er für Zeiger und Arrays einen Link einzeichnet. Wenn der Benutzer auf einen solchen Link klickt, dann zeigt der Heap Browser die dahinter liegende Information an. Bei einem Zeiger ist das die Struktur, auf den der Zeiger verweist. Bei einem Array ist das der Inhalt der einzelnen Elemente des Arrays.

Im beiden Fällen analysiert der Heap Browser die zugehörige RTTI. Bei einem Zeiger untersucht er die RTTI jenes Speicherblockes auf den der Zeiger verweist. Die RTTI eines Speicherblockes ist ähnlich wie beim Moduldeskriptor leicht zu finden. Direkt vor dem Speicherblock steht ein Zeiger auf den Typdeskriptor des Speicherblocks und direkt vor dem Typdeskriptor steht die RTTI, die den Speicherblock beschreibt. Zur Verdeutlichung ist dieser Umstand in Abbildung 75 wiedergegeben.

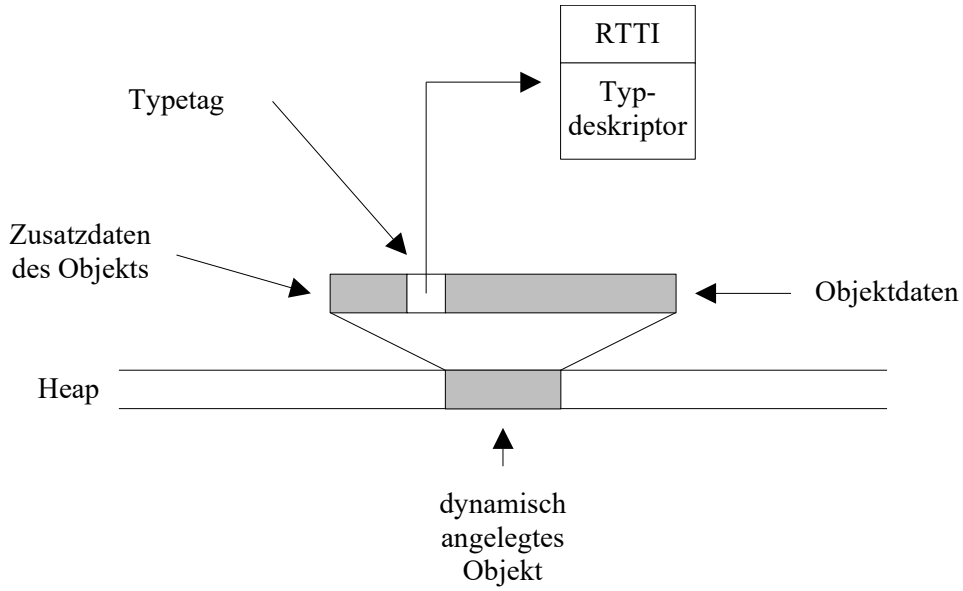


Abb. 76: RTTI eines dynamisch angelegten Objektes

8.5 Klassenbibliothek für Visualisierungsobjekte

Eine der wichtigsten Ideen des Visualisierungssystem sind die Visualisierungsobjekte. Das Visualisierungssystem kann beliebig viele dieser Visualisierungsobjekte verwalten. Sie sind dafür verantwortlich, daß die ausgewählten Objekte überhaupt dargestellt werden, wobei jedes Visualisierungsobjekt genau für ein Objekt zuständig ist.

Im folgenden werden die technischen Hintergründe und Entwurfsentscheidungen der Visualisierungsbibliothek diskutiert. Die Schnittstellen und Funktionen der einzelnen Klassen werden allerdings nicht erläutert, da es hierzu eine vollständige Arbeit gibt. Nähere Details kann man in [JOE99] nachlesen.

Die Abbildung 77 soll helfen, den Zusammenhang zwischen Visualisierungsobjekten und visualisierten Objekten besser erläutern zu können. Sie zeigt eine typische Situation. Rechts sind zwei Programme namens P1 und P2 gezeichnet, von denen 4 bzw. 2 Objekte gerade visualisiert werden. Die Objekte sind der Einfachheit halber von 1 bis 6 durchnumerieret. Im Visualisierungssystem existieren dann ebenfalls sechs Visualisierungsobjekte, wobei jedes davon genau ein Objekt visualisiert. Die Zuständigkeit wird in der Zeichnung durch Pfeile vom Visualisierungsobjekt zum visualisierten Objekt dargestellt. In Wirklichkeit existiert diese Verbindung auch. Die Visualisierungsobjekte speichern natürlich die Adresse des Objektes, für das sie verantwortlich sind, damit sie regelmäßig den Zustand des Objektes ermitteln können.

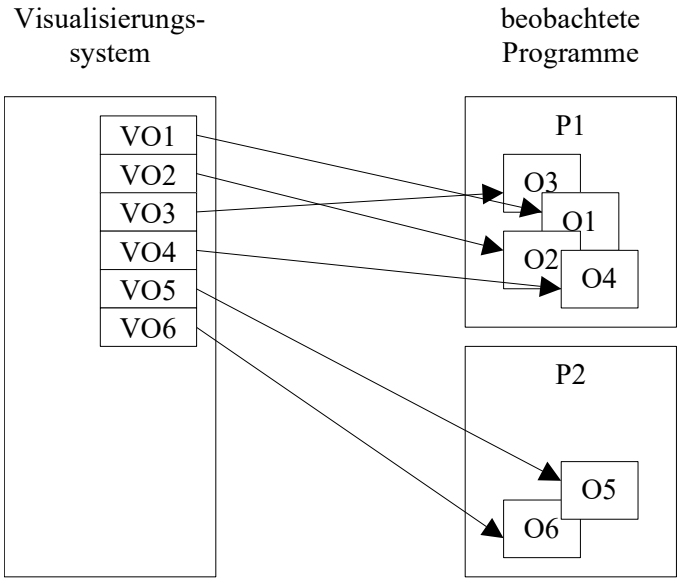


Abb. 78: Zusammenhang zwischen Visualisierungsobjekten und visualisierten Objekten

Die wesentliche Aufgabe eines Visualisierungsobjektes liegt nun darin, den Zustand eines Objektes zu ermitteln und das Objekt entsprechend seines Zustandes darzustellen. Wenn also zum Beispiel eine lineare Liste sechs Elemente enthält, dann ist es die Aufgabe des dafür zuständigen Visualisierungsobjektes die Anzahl der Elemente zu ermitteln und die lineare Liste mit ihren sechs Elementen zu zeichnen. Das Visualisierungssystem weiß also selbst nichts über die darzustellenden Objekte, sondern überläßt die Visualisierung vollständig den Visualisierungsobjekten. Es ist lediglich dafür zuständig, die Visualisierungsobjekte in geeigneter Weise zu verwalten.

Die Visualisierungsobjekte stammen aus einer Klassenbibliothek. Die Klassenbibliothek ist so aufgebaut, daß alle Visualisierungsklassen direkt oder indirekt von einer Basisklasse abgeleitet werden. Diese Basisklasse heißt VisBase und wird selbst nicht zur Visualisierung herangezogen. Es ist eine abstrakte Klasse, die die Schnittstelle zum Visualisierungssystem vorgibt. Jede von ihr abgeleitete Klasse ist für die Visualisierung einer bestimmten Klasse von Objekten zuständig und

kennt den Aufbau der Objekte und kann daher eine passende Darstellung wählen. In Abbildung 79 ist eine vereinfachte Klassenhierarchie der Visualisierungsklassen dargestellt.

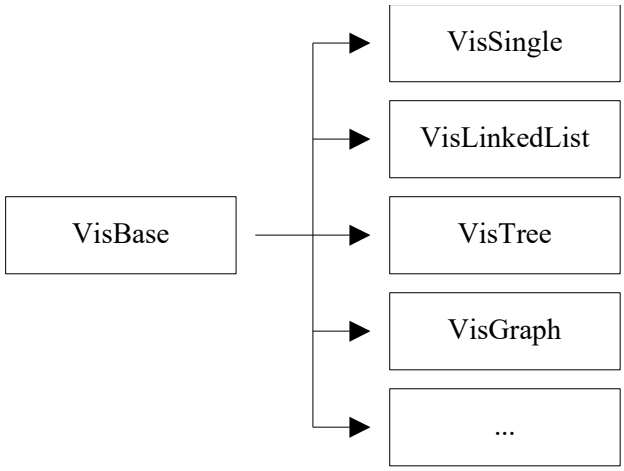


Abb. 80: Klassenhierarchie der Visualisierungsklassen

Das Visualisierungssystem stellt also bereits eine Bibliothek von Visualisierungsklassen zur Verfügung, mit denen man sofort loslegen kann. Es ist aber auch leicht möglich, neue Visualisierungsklassen zu implementieren. Dazu muß man lediglich eine neue Klasse von eine der bestehenden ableiten und die Routinen zum Ermitteln des Zustandes und zum Zeichnen des Objektes überschreiben.

Der Grund für eine neue Visualisierungsklasse liegt nicht immer nur darin, daß man ein neue Art von Objekten visualisieren will. Meistens liegt der Grund darin, daß man eine andere Art der Visualisierung für eine bereits bestehende Klasse implementieren will. In diesem Fall ist die Erstellung einer neuen Visualisierungsklasse sogar noch einfacher, weil nur die Funktionen zum Zeichnen des Objektes überschrieben werden müssen. Dadurch ist es also leicht möglich, eine eigene Visualisierung zu implementieren. Es muß lediglich eine neue Klasse abgeleitet werden und die Visualisierungsfunktion überschrieben werden.

Dadurch kann es aber passieren, daß es für ein bestimmtes Objekt mehr als eine Visualisierungsklasse geben kann. Das bedeutet aber nichts anderes als, daß es verschiedene Visualisierungen bzw. Darstellungen für ein und dasselbe Objekt geben kann. Dies war natürlich auch ein Ziel der Entwicklung. Der Benutzer sollte die Möglichkeit haben, aus verschiedenen Darstellungsarten auswählen zu können. Durch Ableitung einer neuen Visualisierungsklasse kann er eigene Darstellungsarten hinzufügen und dadurch das System seinen Wünschen anpassen und erweitern.

Wirft man einen genaueren Blick auf die Beziehung zwischen Visualisierungsklasse und visualisierten Objekten, dann erkennt man eine interessante Eigenschaft, die sich aus der Vererbung von Klassen ergibt. Eine Klasse kann nämlich nicht nur jene Objekte visualisieren, für die sie zuständig ist, sondern auch alle Objekte von abgeleiteten Klassen. Dies kann man am besten anhand eines Beispiels demonstrieren. Für dieses Beispiel soll angenommen werden, daß es eine Klasse A gibt, von der zwei Klassen abgeleitet wurden, nämlich B und C. Für die Klasse A gibt es eine Visualisierungsklasse namens VA, die Objekte dieser Klasse visualisieren kann. Da Objekte der Klasse B und C Erweiterungen von A sind, ist es nur verständlich, daß die Visualisierungsklasse VA auch diese Objekte visualisieren kann. Das bedeutet im allgemeinen, daß die Visualisierungsklasse VA nicht nur Objekte der Klasse A darstellen kann, sondern auch Objekte von allen abgeleiteten Klassen der Klasse A.

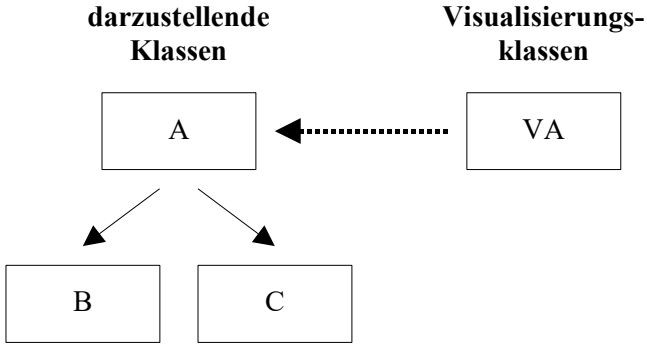


Abb. 81: Zusammenhang zwischen einer Visualisierungsklasse und der Klasse der dargestellten Objekte

Natürlich kann die Visualisierungsklasse VA nicht auf Erweiterungen in den abgeleiteten Klassen B oder C eingehen, sondern verwendet dieselbe Darstellung wie für Objekte der Klasse A. Dies hat aber den großen Vorteil, daß die vorhandenen Visualisierungsklassen jederzeit für abgeleitete Klassen verwendet werden können, wenn man für die Erweiterungen keine spezielle Darstellung benötigt.

Durch den eben besprochenen Entwurf der Klassenbibliothek ergeben sich also folgende zwei wichtige Eigenschaften:

- Die vorhandenen Standardklassen können nicht nur für jene Objekte verwendet werden, für die sie vorgesehen sind, sondern auch für Objekte abgeleiteter Klassen. Das bedeutet, daß es nicht immer notwendig ist, eigene Visualisierungsklassen zu implementieren, wenn man eigene Klassen visualisieren will. Solange man mit der Visualisierung der Standardklassen zufrieden ist, ist es nicht notwendig, neue Visualisierungen zu implementieren.
- Sollte man aber spezielle Darstellungsformen wünschen, dann kann man jederzeit eine neue Visualisierungsklasse von einer bestehenden ableiten. Man braucht dann nur die

Visualisierungsroutinen überschreiben, um eine andere Darstellung zu bekommen. Alle anderen Eigenschaften kann man von der bestehenden Visualisierungsklasse erben.

Für den Fall, daß die Standardklassen um neue Klassen erweitert werden sollen, sollte die Einbindung neuer Klassen leicht vonstatten gehen. Dazu sind die Visualisierungsklassen in Dynamic Link Libraries (DLL) implementiert. Da DLLs nicht zum Programm gebunden werden, sondern erst beim Starten eines Programmes geladen werden, ist es sehr leicht möglich, durch Hinzufügen neuer DLLs die Visualisierung zu erweitern.

Damit das Visualisierungssystem weiß, welche Visualisierungsklasse in welcher DLL implementiert ist, gibt es eine sogenannte Visualisierungsdatenbank. Darin sind folgende Informationen gespeichert:

- Name der Visualisierungsklasse: Neben dem eigentlichen Klassennamen ist auch der Modulname gespeichert. Der Name hat also die Form <Modulname>.<Klassenname>.
- Versionsnummer
- Name der Klasse, dessen Objekte die Visualisierungsklasse darstellen kann. Für diesen Klassennamen gelten dieselben Anmerkungen wie für den vorigen Klassennamen.
- Dateiname der DLL, in der die Visualisierungsklasse implementiert ist.

Die folgenden beiden Zeilen zeigen typische Einträge in der Visualisierungsdatenbank, die wegen der einfachen Wartbarkeit als Textdatei implementiert ist. Der erste Eintrag besagt, daß Objekte der Klasse *OListT* im Modul *OList* durch Visualisierungsobjekte der Klasse *StructureT* im Modul *VList* dargestellt werden können. Die Visualisierungsklasse ist in der DLL D:\Visual~1\Proj4.03\-VisDLL.DLL implementiert.

```
OList.OListT;1;VList.StructureT;D:\VISUAL~1\PROJ4.03\VISDLL.DLL
OTree.OTreeT;1;VTree.StructureT;D:\VISUAL~1\PROJ4.03\VISDLL.DLL
```

8.6 Kern

Der letzte Teil des Visualisierungssystem ist der Kern des Systems, der alle anderen Teile miteinander verbindet.

Neben der Koordination der anderen Teile des Visualisierungssystems kümmert sich der Kern vor allem um die Verwaltung der Visualisierungsobjekte. Dies ist gar nicht so einfach, weil der Kern dabei auch berücksichtigt, wie die Objekte zueinander in Beziehung stehen.

Es ist nicht abwegig, daß ein beobachtetes Objekt eine Referenz auf ein anderes Objekt hat, das ebenfalls gerade vom Visualisierungssystem beobachtet wird. Ein einfaches Beispiel dafür wäre eine lineare Liste, von der alle Elemente gerade visualisiert werden. In einer typischen Implementierung einer linearen Liste hat das erste Element der Liste einen Zeiger auf das zweite Element, das zweite auf das dritte, usw.

Solche Referenzen sind wichtige Informationen, die auch visualisiert werden sollten. Daher sollte das Visualisierungssystem solche Referenzen in geeigneter Form darstellen können. Als Darstellungsform wurde ein Pfeil gewählt, der von jenem Objekt ausgeht, in dem die Referenz gespeichert ist, und der zu jenem Objekt zeigt, auf das die Referenz verweist. Zusätzlich kann man den Namen der Referenz anzeigen lassen. Dabei wird einfach der Name jener Variablen verwendet, in der die Referenz gespeichert ist.

Das Visualisierungssystem kann also Referenzen zwischen Objekten als Pfeile darstellen. Es bleibt aber die Frage offen, wie es zu diesen Informationen kommt. Lieferant für diese Information ist das Visualisierungsobjekt selbst. Es meldet dem Kern alle Referenzen seines Objektes. Der Kern baut aus diesen Informationen spezielle Referenzobjekte auf, wobei für jeweils zwei Objekte, die sich referenzieren, ein Referenzobjekt erstellt wird. Das Referenzobjekt enthält die Information über die zwei an der Referenz beteiligten Objekte sowie den Namen der Referenz.

Da es nicht vorhersehbar ist, wie die beobachteten Objekte miteinander in Beziehung stehen, muß das Visualisierungssystem die Referenzen möglichst flexibel speichern. Die flexibelste Datenstruktur ist ein allgemeiner Graph, in dem jedes Objekt jedes andere Objekt referenzieren kann. Daher wurde für die Speicherung der Visualisierungs- und Referenzobjekte eine Datenstruktur gewählt, wie sie für Graphen verwendet wird, nämlich die Adjazenzmatrix.

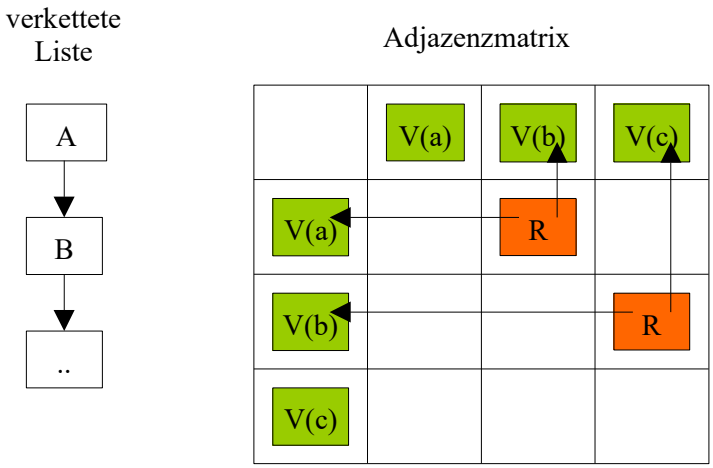


Abb. 82: Adjazenzmatrix des Visualisierungssystems

Für die Objekte A, B und C wurden Visualisierungsobjekte namens V(a), V(b) und V(c) angelegt. Um die Information über den Zeiger vom Objekt A zum Objekt B in der Adjazenzmatrix zu speichern, wird im Schnittpunkt des Objektes V(a) mit dem Objekt V(b) das Referenzobjekt gespeichert, das seinerseits wieder auf die beiden Visualisierungsobjekte verweist. Ebenso wird für den Zeiger vom Objekt B auf das Objekt C verfahren.

Dadurch weiß der Kern welche Objekte sich referenzieren und kann die entsprechenden Pfeile zeichnen. Das Zeichnen der Objekte erfolgt in einer ganz bestimmten Reihenfolge. Zuerst werden die Pfeile gezeichnet und danach die Objekte, wobei das Zeichnen der Objekte den Visualisierungsobjekten überlassen wird. Der Grund für diese Reihenfolge liegt darin, daß Pfeile nicht über Objekte gehen sollten. Dadurch daß Pfeile vorher gezeichnet werden, ist sichergestellt, daß Objekte an derselben Stelle Pfeile überdecken.

9 Anwendungen

Das folgende abschließende Kapitel widmet sich den möglichen Anwendungen des Visualisierungssystem, von denen drei herausgegriffen und näher beschrieben werden.

Neben dem naheliegenden Einsatzgebiet als graphischer Debugger mit besonderen Fähigkeiten, bietet sich auch der Einsatz im Unterricht an. Dort kann das Werkzeug helfen, neue Datenstrukturen und ihre Operationen zu verstehen. Schließlich soll aber auch der Einsatz als Reengineering Werkzeug nicht unerwähnt bleiben. Dadurch, daß der Quellcode des untersuchten Programmes nicht vorliegen muß, kann es sich auch für die Erforschung unbekannter Programme bewähren.

9.1 Didaktik

Im Unterricht kann das Visualisierungswerkzeug große Unterstützung bieten. Studenten der Studienrichtung Informatik müssen im Laufe ihres Studiums unter anderem viele verschiedene Datenstrukturen kennenlernen. Um neue Datenstrukturen leichter zu erlernen, greifen viele zu bildlichen Darstellungen. Auch Lehrer helfen sich beim Erklären von Datenstrukturen mit graphischen Darstellungen. Frei nach dem Sprichwort "Ein Bild sagt mehr als tausend Worte" läßt sich eine gezeichnete Datenstruktur einfacher erfassen.

An diesen Punkt kann das Visualisierungssystem helfen. Es kann auf jeden Fall den Lehrer unterstützen. Der Lehrer muß nicht mehr die Datenstrukturen selbst zeichnen, sondern kann diese Aufgabe dem Visualisierungssystem überlassen. Dazu muß der Lehrer lediglich die Datenstruktur implementieren, die er unterrichten will. Falls die vorhandenen Visualisierungen nicht passen, muß er zusätzlich noch eine eigene Visualisierung implementieren. Letzteres sollte aber eher selten der Fall sein, weil das Visualisierungssystem schon mit einer Reihe vorgefertigter Visualisierungen ausgestattet ist.

Jetzt kann der Lehrer seine Implementierung der Datenstruktur im Unterricht demonstrieren und mit dem Visualisierungssystem erklären, wie die Datenstruktur aussieht. Bei jeder Veränderung der Datenstruktur kann er den Algorithmus Schritt für Schritt durchgehen und den Studenten zeigen, wie sich die Datenstruktur verändert. Er kann also jede Operation an einer Datenstruktur live vorführen und gleichzeitig darauf hinweisen, welche Auswirkungen eine Operation hat.

Der Lehrer kann aber auch noch einen Schritt weitergehen. Er kann die Implementierung sogar verändern und mit dem Visualisierungssystem ohne zusätzlichen Aufwand darlegen, welche Auswirkungen die Veränderung des Algorithmus auf die Datenstruktur hat. Insbesondere kann er typische Fehler einbauen, wie sie von Studienanfängern gerne gemacht werden, und anhand des fehlerhaften Algorithmus nachweisen, daß die Datenstruktur falsch aufgebaut wird.

Außerdem kann man das Visualisierungssystem den Studenten in die Hand geben. Es ist durchaus üblich, daß Studenten zur Übung vorgegebene Datenstrukturen selbst implementieren müssen. Dabei passieren immer wieder Fehler. Der Grund liegt meist darin, daß sie die Datenstruktur nicht vollständig verstanden haben. Mit dem Visualisierungssystem können sie den Fehler leichter einkreisen, weil sie ja sehen, wie die Datenstruktur aussieht. Außerdem können sie ihren Quellcode auch Schritt für Schritt durchgehen und dadurch erkennen, welche Operation fehlerhaft ist.

Die Studenten können ihre Fehler mit dem Visualisierungssystem nicht nur leichter finden, sondern sie können auch selbst erkennen, welchen Fehler sie genau gemacht haben. Da man bekanntlich aus Fehlern lernt, sollte das Visualisierungssystem helfen, die Implementierung einer Datenstruktur besser zu verstehen.

Diese Idee soll nun an einem einfachen Beispiel erläutert werden. Es wurde ein sehr einfaches Beispiel gewählt, um die Wirkung des Visualisierungssystem möglichst klar herausarbeiten zu können. In diesem Beispiel geht es um die Implementierung einer einfach verketteten Liste. Das ist eine Datenstruktur, die meist schon sehr früh im Studium unterrichtet wird. Sie ist Voraussetzung für weiterführende Datenstrukturen wie zum Beispiel Bäume und für ihr Verständnis benötigt man im wesentlichen nur Kenntnisse über Zeiger.

Eine typische Deklaration der für eine einfach verkettete Liste benötigten Datentypen wird durch folgendes Codefragment wiedergegeben, das in Oberon-2 verfaßt ist. Als Inhalt eines Listenelementes wird der Einfachheit halber ein Integer-Wert verwendet.

```
TYPE
  ListP = POINTER TO ListT;
  ListT = RECORD
    data: INTEGER;
    next: ListP;
  END;
VAR
  list: ListP;
```

Auch die visualisierte Darstellung einer solchen Liste ist nicht kompliziert. Jedes Element der Liste wird durch ein Rechteck symbolisiert. Der Zeiger von einem Element zum nächsten wird durch

einen Pfeil dargestellt. Das Ende einer verketteten Liste wird meist durch den griechischen Buchstaben Ω, dem letzten Buchstaben des griechischen Alphabets, angedeutet. Abbildung 83 zeigt eine solche Darstellung von einer Liste mit drei Elementen, wobei das erste Element den Wert 1, das zweite den Wert 2 und das dritte den Wert 3 trägt. Die verkettete Liste ist also sortiert.

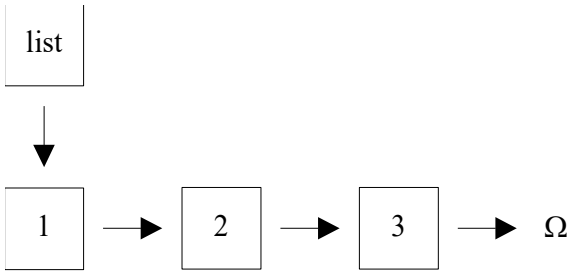


Abb. 84: Visualisierung einer verketteten Liste

Nun soll in die sortierte Liste ein neuer Wert eingefügt werden. Dazu muß man die Operation Hinzufügen implementieren. In diesem Beispiel soll als kleine Erschwernis die Sortierung der Liste beibehalten bleiben. Diese Funktion kann man folgendermaßen implementieren:

```
PROCEDURE Add(new: INTEGER);
VAR h, h1: ListP;
BEGIN
  (* -- leere Liste oder an erster Position einfügen -- *)
  IF (list = NIL) OR (new < list.data) THEN
    h := list;
    NEW(list);
    list.data := new;
    list.next := h;
  ELSE
    (* -- Suche passende Einfügestelle -- *)
    h := list;
    WHILE (h.next # NIL) AND (h.next.data <= new) DO
      h := h.next
    END;
    (* -- neues Element zwischen h und h.next einfügen -- *)
    h1 := h.next;
    NEW(h.next);
    h.next.data := new;
    h.next.next := h1;
  END;
END Add;
```

Läßt man diese Funktion mit dem Visualisierungssystem für ein neues Element mit dem Wert 2 durchlaufen, dann könnte die Anzeige im Visualisierungssystem wie in Abbildung 85 aussehen. Die ersten beiden Schritte zeigen, wie eine passende Stelle gesucht wird. Das Element, auf das die Variable h zeigt, wird dabei eingefärbt. Die nächsten drei Schritte zeigen, wie das neue Element hinter dem Element, auf das die Variable h zeigt, eingefügt wird. Recht schön sind dabei die Zeigeroperationen mit Hilfe der Variablen h1 zu sehen. Der letzte Schritt zeigt schließlich das Endergebnis.

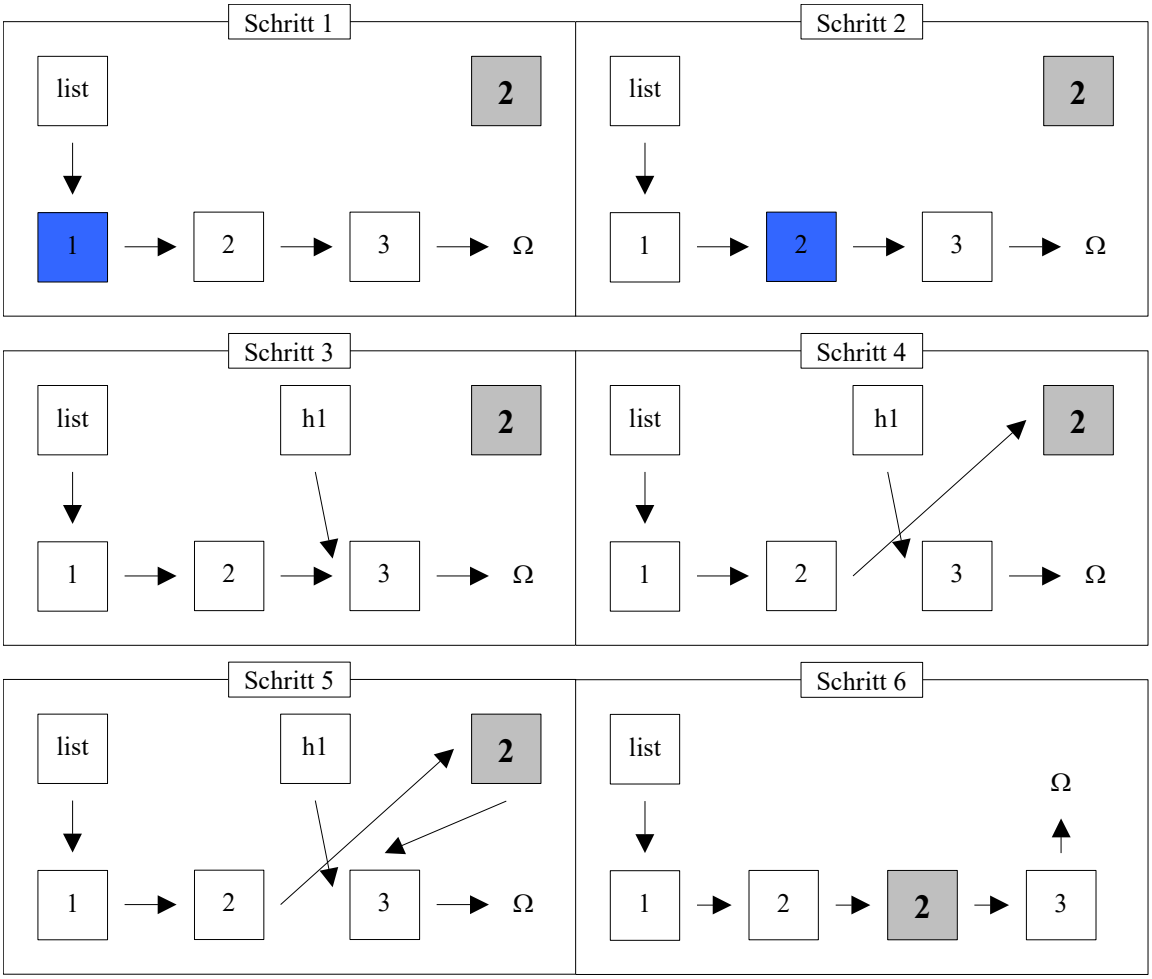


Abb. 86: Visualisierung einer Listenoperation

Studienanfänger machen bei diesem Beispiel gerne Fehler bei den Zeigeroperationen. Häufig passiert es, daß die Reihenfolge der Zeigeroperationen nicht korrekt ist und dadurch Elemente verlorengehen oder daß zum Beispiel eine Hilfsvariable vergessen wird. Letzteres soll im folgenden genauer betrachtet werden.

Wenn die Hilfsvariable h1 vergessen wird, dann schaut das Einfügen eines Elementes folgendermaßen aus:

```
(* -- neues Element zwischen h und h.next einfügen -- *)
NEW(h.next);
h.next.data := new;
h.next.next := NIL;  (* hier steht manchmal auch h.next.next := h *)
```

Die Visualisierung dieses Codestück würde daher wie in Abbildung 87 aussehen. Darin sind nur jene Schritte wiedergegeben, die sich durch die kleine Codeänderung verändert haben. Wie zu erwarten war, zeigt auch die Darstellung im Visualisierungssystem, daß der Rest der Liste abgehängt wird und dadurch verlorengeht.

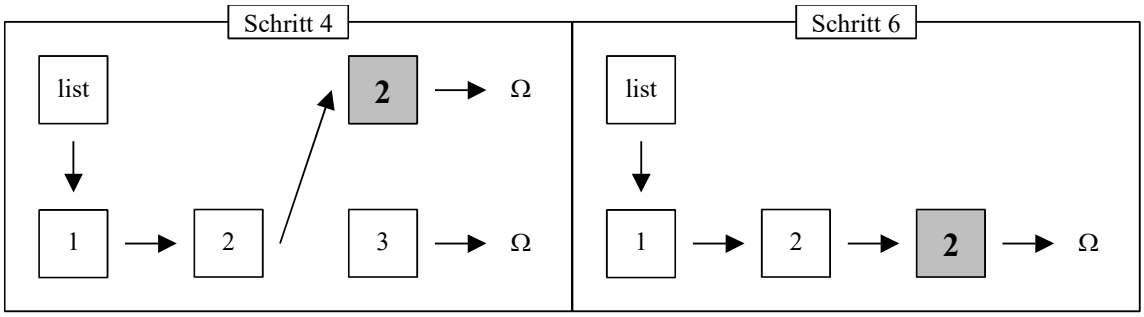


Abb. 88: Visualisierung einer fehlerhaften Listenoperation

Durch diese visuelle Unterstützung sollte es dem Studenten leicht fallen, seinen Fehler zu finden. Insbesondere kann er erkennen, daß es wichtig ist, sich den Zeiger auf das nächste Element vor einer Veränderung zu merken.

Zusammenfassend kann man also feststellen, daß das Visualisierungssystem nicht nur den Lehrer beim Erklären neuer Datenstrukturen unterstützen kann, sondern auch Studenten helfen kann, ihre eigenen Fehler besser zu verstehen.

9.2 Software Reengineering

Ein weiteres interessantes Einsatzgebiet des Visualisierungssystems ist das Gebiet des Software Reengineering (siehe [ARN94], [MÜL97] und [KOS97]). Dieses Gebiet befaßt sich mit der Analyse bestehender Programme und der Anpassung dieser Programme an neue Gegebenheiten.

Solche Aufgabenstellungen hat sicherlich jeder Programmierer schon einmal zu bewältigen gehabt. Typisch für diese Aufgabenstellung ist, daß der ursprüngliche Autor des Programmes meist nicht mehr greifbar ist, weil er zum Beispiel die Firma verlassen hat. Meist kommt dann auch noch hinzu, daß die Dokumentation nur sehr mangelhaft ist. Viele Projekte werden unter großem Zeitdruck entwickelt und legen keinen großen Stellenwert auf eine korrekte und vollständige

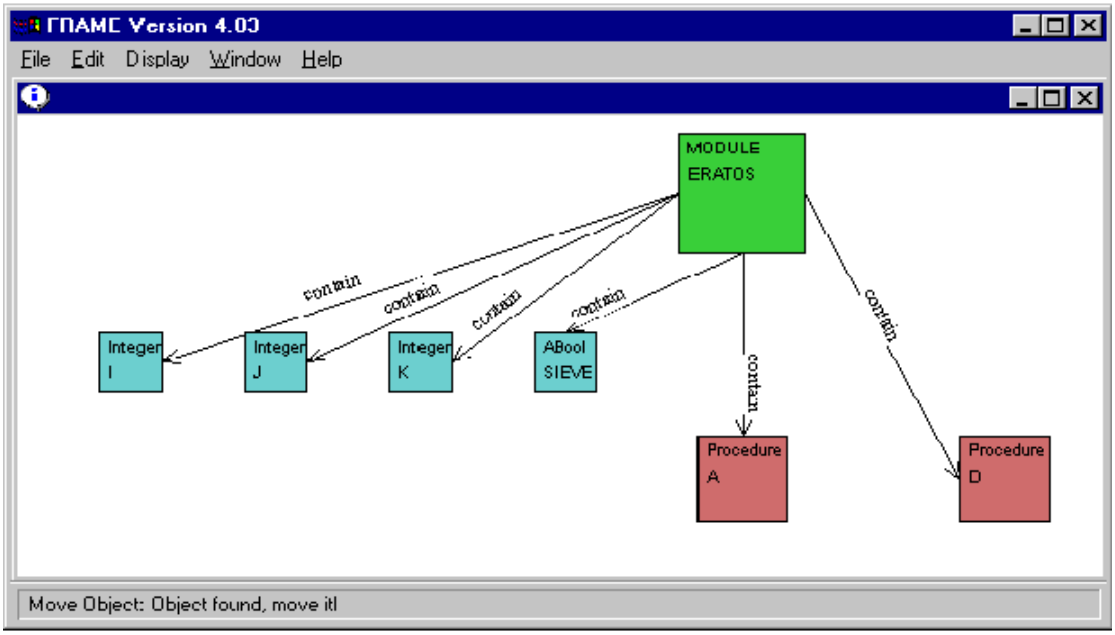
Dokumentation. Um die Funktionalität eines Programmes in einer solchen Situation zu ermitteln, bleibt dann meist nur die Analyse des Quellcodes übrig. Wenn der Autor sich aber keine Mühe beim Verfassen von Kommentaren gegeben hat, kann dies ein schwieriges Unterfangen werden.

In einer solchen Situation soll das Visualisierungssystem helfen. Man kann das unbekannte Programm laufen lassen und sich gleichzeitig anschauen, welche Objekte das Programm anlegt und wie sie miteinander verknüpft sind. Gerade in objektorientierten Programmen spielt die Interaktion der einzelnen Objekte untereinander eine große Rolle und ist für das Verständnis eines Programmes von entscheidender Bedeutung.

Das Visualisierungssystem bietet aber noch einen darüber hinausgehenden Vorteil. Der Quellcode des untersuchten Programmes muß gar nicht vorhanden sein. Da für die Visualisierung keine Veränderungen am Quellcode notwendig sind, können auch Programme untersucht werden, deren Quellcode nicht zur Verfügung steht. Dabei geht es gar nicht darum, verbotenerweise Programme der Konkurrenz zu analysieren. Sogar in selbst geschriebenen Programmen kann es Teile geben, für die der Quellcode nicht zugänglich ist. Dies ist zum Beispiel der Fall, wenn man eine Bibliothek von einem anderen Hersteller kauft und in sein Programm einbindet. Dann bekommt in der Regel für die Bibliothek keinen Quellcode. Trotzdem ist aber die Analyse des Programmes mit dem Visualisierungssystem möglich.

Für die Visualisierung eines unbekannten Programmes kann man in der Regel die zur Verfügung gestellten Visualisierungsklassen verwenden. Bei Bedarf kann man aber auch eigene Visualisierungen hinzufügen, da eine neue Visualisierungsklasse sehr leicht hinzugefügt werden kann.

Ein Beispiel dafür findet man in [MDJ99]. Dort wird beschrieben wie das Visualisierungssystem für die Analyse eines Compilers verwendet wurde. Der Compiler lag zwar im Quellcode vor, was die Angelegenheit ein wenig vereinfachte, aber für die Visualisierung selbst wäre das nicht notwendig gewesen. Ein Beispiel dieser Arbeit ist in Abbildung 89 dargestellt. Sie zeigt die Symbolliste für ein einfaches Programm.



9.3 Abb. 90: Visualisierung einer SymbollisteGraphischer Debugger

Als letztes der drei besprochenen Anwendungsgebiete wird der Einsatz als graphischer Debugger besprochen. Im Gegensatz zum vorher beschriebenen Einsatzgebiet ist der Quellcode des zu analysierenden Programmes beim Debuggen immer vorhanden. Das Visualisierungssystem kann wie ein Debugger eingesetzt werden, nur daß es zusätzlich die Datenstrukturen visualisieren kann. Das bedeutet, daß ein Programmierer nicht die Zeiger eines binären Baumes mühsam selbst durchgehen muß. Statt dessen kann er dem Visualisierungssystem bekanntgeben, daß es den gesamten Baum auf einmal anzeigen soll. Das erhöht nicht nur die Übersichtlichkeit, weil die Datenstruktur als Ganzes wahrgenommen werden kann, sondern erleichtert auch die Fehlersuche.

Ob die mitgelieferten Visualisierungsklassen ausreichen, hängt vom jeweiligen Anwendungsfall ab. In der Regel werden sie genügen. Es ist aber auch nicht abwegig, daß man sich eine eigene Visualisierung zusammenstellt. Dies wird vor allem dann hilfreich sein, wenn das untersuchte Programm eine gewisse Größe annimmt und so viele Objekte anlegt, daß der Überblick nicht mehr so leicht möglich ist. In diesem Fall ist es sicherlich ratsam eine eigene Visualisierung zu implementieren, die die Objekte etwas abstrakter darstellt. Bei einem Baum kann es zum Beispiel genügen, daß nur die Anzahl der im Baum enthaltenen Elemente und nicht jedes einzelne Element

angezeigt wird. Diese Darstellung ist sehr einfach hinzuzufügen. Erst wenn man einen Fehler im Baum vermutet, kann man auf eine detaillierte Darstellung umschalten, wie sie zum Beispiel das Visualisierungssystem mit sich bringt.

10 Anhang A: Spezifikationen

In diesem Kapitel wird die Compiler-Schnittstelle sowie die Editor-Schnittstelle von Pow! im Detail erläutert.

10.1 Compiler-Schnittstelle

Für jeden Compiler, der mit Pow! zusammenarbeiten soll, muß eine entsprechende DLL entwickelt werden. Dies kann von jedem Windows-kundigen Programmierer erledigt werden. Man braucht dazu keine Interna über Pow! wissen. Es genügt, eine DLL zu entwickeln, die der nachfolgenden Spezifikation entspricht.

Damit Pow! eine DLL überhaupt zur Kenntnis nimmt, muß sie im selben Verzeichnis liegen wie die Datei pow.exe und die Dateiendung *cll* besitzen. Ruft man den Menüpunkt "Options, Preferences" auf, dann öffnet sich eine Dialogbox, in der unter anderem alle von Pow! gefundenen Compiler-Schnittstellen-Module angezeigt werden. Dabei wird nur der Dateiname ohne Pfad und Dateiendung angezeigt.

Damit Pow! mit der Compiler-Schnittstelle auch arbeiten kann, müssen folgende Funktionen unter den folgenden Exportzahlen (Ordinals) von der DLL exportiert werden:

Funktionsname	Exportnummer
InitInterface	1
AboutCompiler	2
CompileOptions	3
CompileFile	4
CheckDepend	5
FileWasCompiled	6
LinkerOptions	7
Link	8
DirectoryOptions	9
NewProject	10
WriteOptions	11
ReadOptions	12
GetExtensions	13
HelpCompiler	14
EditorSyntax	15

Funktionsname	Exportnummer
EditorComment	16
ExitInterface	17
GetExecutable	18
NewProjectName	19
ChangeModuleName	20
SourceAvailable	28
MustBeBuilt	29
CheckIfYounger	30
GetHelpFile	31
GetTarget	32

10.1.1 Funktionen

Die Funktionen der Compiler-Schnittstelle kann man in verschiedene Bereiche einteilen. Neben dem klassischen Bereich des Übersetzens gibt es noch andere Aufgaben, wie zum Beispiel das Linken eines Programmes, oder das Verwalten von Projekten.

- Initialisierung und Terminierung
- Übersetzen
- Linken
- Verzeichnisse
- Projekte
- Hilfe
- Editor

10.1.1.1 Initialisierung und Terminierung

Hierbei handelt es sich um zwei Funktionen, die nach dem Laden und vor dem Entladen einer Compiler-Schnittstelle aufgerufen werden.

```
HANDLE InitInterface(LPSTR compilerName,
                     LPSTR powDir,
                     DWORD ddeInstId);
```

Diese Funktion wird von Pow! immer beim Öffnen eines neuen Interfaces (also beim Programmstart oder beim Wechseln des Compilers im Preferences-Dialog) aufgerufen. Die DLL sollte hier ihre globalen Initialisierungen durchführen (Default-Options,...).

Der Parameter *compilerName* enthält den Namen der von Pow! ausgewählten Compilerinterface DLL, *powDir* den Pfad zur Pow!-Applikation. *ddeInstId* enthält den DDE Instance Handle von Pow!, der für Client Transactions verwendet werden kann, ohne die DDEML selber initialisieren zu müssen (notwendig, da unter Win32 eine Applikation keine Objekte zwischen zwei DDEML Instanzen austauschen darf).

DLLs unter Windows haben für alle Instanzen nur ein einziges Datensegment. Damit nun mehrere Instanzen von Pow! auf dieselbe DLL zugreifen können, muß diese ihre gesamten globalen Daten in einen Speicherblock verpacken, der in der Funktion *InitInterface* angelegt wird. Der Handle dieses Speicherblocks wird als Rückgabewert an Pow! übergeben und von dort aus jeder Funktion der Compilerinterface DLL als Kontext mitgegeben. Die DLL selber darf also keine globalen Daten mehr haben, sondern darf nur mehr auf dem für die jeweilige Pow!-Instanz spezifischen Speicherblock operieren.

```
void ExitInterface(HANDLE hDat);
```

Pow! wird beendet und fordert die Compiler-DLL auf, sämtliche benutzten Ressourcen und den gesamten angelegten Speicher freizugeben (unter anderem auch den globalen Datenblock *hDat*).

10.1.1.2 Übersetzen

```
BOOL AboutCompiler(HANDLE hDat, HWND hwnd);
```

Diese Funktion soll einen Dialog anzeigen, der den Urheber der Compiler-Schnittstelle und des Compilers wiedergibt. Sie wird aufgerufen, wenn die Compiler-Schnittstelle gewechselt wird und wenn Pow! gestartet wird. Als Vaterfenster soll das Fenster in *hwnd* verwendet werden.

```
BOOL CompileOptions(HANDLE hDat, HWND hwnd);
```

Die DLL sollte hier einen Compiler-Options Dialog mit dem Parent *hwnd* aufbauen. Die eingestellten Optionen müssen global gehalten werden. Beim Aufruf des Compilers sind sie entsprechend in Parameter umzuwandeln. Ein Rückgabewert von TRUE bedeutet, daß die Optionen geändert wurden (also die Projektdatei neu zu schreiben ist), FALSE bedeutet keine Veränderung.

```
BOOL CompileFile(HANDLE  hDat,
                 LPSTR   file,
                 FARPROC msg,
                 FARPROC err,
                 HWND    fromWnd,
                 FARPROC first,
                 FARPROC next,
                 FARPROC fileOpen,
                 FARPROC fileRead,
                 FARPROC fileClose);
```

Diese Funktion soll die Datei in *file* übersetzen. Treten dabei Fehler oder Warnungen auf, so sollen diese über die Callback-Funktionen *err* und *msg* an Pow! zurückgemeldet werden. Diese beiden Funktionen sind folgendermaßen definiert:

```
void FAR PASCAL msg(LPSTR txt);
```

Mit der Callback-Funktion *msg* werden allgemeine Nachrichten an Pow! zurückgemeldet, die in einem Textfenster ausgegeben werden. Sie haben keine Auswirkung, sondern dienen nur der Information des Benutzers. Hiermit kann zum Beispiel 'Compiling test.o ...', 'Linking ... ', usw. ausgegeben werden.

```
void FAR PASCAL err(int num,
                    int line, int col,
                    BOOL warn, LPSTR txt);
```

Mit dieser Callback-Funktion wird dem Pow! ein Fehler beim Übersetzen der Datei mitgeteilt. Der Fehler Nummer *num* (Klartext in *txt*) ist in Zeile *line* und Spalte *col* gefunden worden. (*warn* = TRUE signalisiert ein Warning, *warn* = FALSE weist auf einen Fehler hin)

Falls die Datei momentan in einem offenen Edit-Fenster gehalten wird (Window-Handle *fromWnd* ungleich Null), dann kann der Puffer direkt compiliert werden. *first* liest dann den ersten Teil des Puffers und *next* weitere Teile. Wenn von einer Datei gelesen werden soll (*fromWnd* = 0), dann muß die Datei zuerst mit *fileOpen* geöffnet werden, der Puffer mit der Funktion *fileRead* ausgelesen und die Datei dann wieder mit *fileClose* geschlossen werden (damit wird dem Editor das Auslesen der Datei überlassen, wodurch aus beliebigen Dateiformaten compiliert werden kann).

Rückgabewert: TRUE, wenn sich das Interface des Moduls geändert hat, also alle abgeleiteten Module ebenfalls neu zu übersetzen sind; sonst FALSE.

Dabei gilt:

```
long FAR PASCAL first(HWND hwnd, LPSTR buf, long size);
```

Die ersten *size* Bytes des Edit-Fensters *hwnd* werden in den Puffer kopiert und die tatsächliche Anzahl Bytes zurückgegeben.

```
long FAR PASCAL next(HWND hwnd, LPSTR buf, long size);
```

Die nächsten *size* Bytes des Edit-Fensters *hwnd* werden in den Buffer kopiert und die tatsächliche Anzahl Bytes zurückgegeben.

```
int FAR PASCAL fileOpen(LPSTR fileName);
```

Die Datei *fileName* wird vom Editor in dessen Format geöffnet ein Handle retourniert, mit dem man in den folgenden Funktionen darauf zugreifen kann.

```
long FAR PASCAL fileRead(int handle, LPSTR buf, long size);
```

Die ersten *size* Bytes der Datei *handle* werden in den Puffer kopiert und die tatsächliche Anzahl Bytes zurückgegeben.

```
void FAR PASCAL fileClose(int handle);
```

Die Datei *handle* wird wieder geschlossen und der Editor kann temporär angelegte Daten freigeben.

```
void CheckDepend(HANDLE  hDat,
                 LPSTR   file,
                 FARPROC depends,
                 HWND    fromWnd,
                 FARPROC first,
                 FARPROC next,
                 FARPROC fileOpen,
                 FARPROC fileRead,
                 FARPROC fileClose);
```

Diese Funktion soll alle Module ermitteln, die von der Datei, deren Name in *file* übergeben wird, benutzt werden. Die Parameter und Funktionen zum Einlesen des Quelltextes entsprechen denen der Funktion *CompileFile*. Für jedes Modul, das von der Datei *file* benutzt wird, soll die Callback-Funktion *depends* aufgerufen werden, die folgendermaßen definiert ist:

```
void FAR PASCAL depends (LPSTR module);
```

Übergibt eine Abhängigkeit, d.h. *module* muß vor *file* übersetzt werden.

```
BOOL FileWasCompiled(HANDLE hDat, LPSTR file);
```

Diese Funktion wird von Make aufgerufen um zu überprüfen ob eine Datei bereits in übersetzter Form vorliegt (egal ob in der aktuellen Version!).

Rückgabewert: TRUE, wenn die Objektdatei existiert; FALSE, sonst

```
BOOL CALLBACK SourceAvailable (HANDLE hData,
                               LPSTR module, LPSTR file)
```

Diese Funktion soll TRUE zurückgeben, wenn die in *module* übergebene Quelldatei existiert. In diesem Fall soll in *file* der Pfad und der Name der Quelldatei zurückgegeben werden.

```
BOOL CALLBACK MustBeBuilt (HANDLE hData, LPSTR file)
```

Diese Funktion soll TRUE zurückgeben, wenn die in *file* übergebene Quelldatei übersetzt werden muß. Das kann zum Beispiel passieren, wenn es zu *file* keine übersetzte Datei gibt oder wenn *file* jünger ist als die übersetzte Datei.

```
BOOL CALLBACK CheckIfYounger (HANDLE hData,
                              LPSTR module,
                              LPSTR client)
```

Diese Funktion soll TRUE zurückliefern, wenn die Datei *client* nicht existiert oder *client* älter ist als die Datei *module*.

10.1.1.3 Linker

```
BOOL LinkerOptions (HANDLE hDat, HWND hwnd);
```

Ähnlich den Compiler-Options muß die DLL hier eine Dialogbox zum Eingeben der Linker-Parameter öffnen. Die eingegebenen Parameter sind ebenfalls global in der DLL zu halten, sie werden beim Aufruf des Linkers gebraucht. Wichtige Optionen sind unter Windows die Import- und Exportliste, Stack- und Heapsize, DLL- oder EXE-target und einige Switches (Debug-Information, CASE-Sensitivität,...).

Der Parameter *hwnd* ist der Handle des Parent-Fensters; der Rückgabewert zeigt wieder an, ob Änderungen durchgeführt wurden (TRUE) oder nicht.

```
void Link(HANDLE hDat, LPSTR file, LPHANDLE flist, FARPROC msg);
```

Die Datei *file* soll aus den Dateien *flist* gebunden werden. *flist* ist die im Projekt angegebene Liste von zugehörigen Dateien und über die Zugriffsprozeduren in der Powsupp.DLL anzusprechen. Alle Messages sind an die schon aus *Compile* bekannte Funktion *msg* zu übergeben.

Die Funktionen, mit denen die lineare Liste *flist* verarbeitet werden kann, sind im folgenden Kapitel erläutert. Sie entstammen der DLL PowSupp.dll.

10.1.1.4 Verzeichnisse

```
BOOL DirectoryOptions (HANDLE hDat, HWND hwnd);
```

Diese Funktion öffnet eine Dialogbox zum Eingeben der Verzeichnisse für den Compiler und den Linker.

Rückgabewert: TRUE bei Änderungen, FALSE sonst.

10.1.1.5 Projekte

```
void NewProject(HANDLE hDat);
```

Diese Funktion wird aufgerufen, wenn Pow! ein neues Projekt erstellt. Die Compiler-Schnittstelle sollte ihre globalen Variablen auf Default-Werte setzen.

```
BOOL WriteOptions(HANDLE hDat, LPSTR prjName, HFILE file);
```

Immer wenn Pow! Projektdaten speichert, wird die Compiler-DLL aufgefordert, die eigenen Daten dazuzuschreiben. Die Datei kann über den Handle *file* angesprochen werden. Die Datei soll nicht geschlossen werden. Der Parameter *prjName* gibt den Pfad und den Namen des Projekts bekannt

Rückgabewert: TRUE, wenn die Daten geschrieben wurden, FALSE bei einem Fehler.

```
BOOL ReadOptions(HANDLE hDat, LPSTR prjName, HFILE file);
```

Diese Funktion liest die mit *WriteOptions* gesicherten Daten wieder zurück. Handhabung und Parameter wie oben.

Rückgabewert: TRUE, wenn die Daten gelesen werden konnten, FALSE bei einem Lesefehler.

```
void CALLBACK NewProjectName (HANDLE hData,LPSTR prjName)
```

Nach Aufruf dieser Funktion soll der in *prjName* übergebene Name (inkl. Pfad) als neuer Projektname verwendet werden. (für Default-Projects in Oberon-2!)

```
void CALLBACK ChangeModuleName(HANDLE  hData,
                               HWND    hwnd,
                               FARPROC replace,
                               LPSTR   modname,
                               LPSTR   dstname)
```

Diese Funktion soll den Modulnamen *modname* in *dstname* in einer Quelldatei ändern. Die Quelldatei ist im Editorfenster *hwnd* geladen. Mit der Funktion *replace* kann man die Änderungen direkt im Editor durchführen. Die Funktion *replace* ist folgendermaßen definiert:

```
int Replace(HWND  edit,
            LPSTR text,
            LPSTR new,
            int   matchcase,
            int   down,
            int   words,
            int   all,
            int   ask);
```

Diese Funktion ersetzt den Text, der im Parameter *text* übergeben wird, durch den Text des Parameters *new* im Fenster *edit*. Die Ersetzung berücksichtigt die Groß-/Kleinschreibung, wenn *matchcase* 1 ist. *down* gibt die Richtung der Suche an. Wenn 1 übergeben wird, wird von der aktuellen Cursorposition bis zum Dateiende gesucht. Sonst wird in der umgekehrten Richtung gesucht. Wenn *word* 1 ist, werden nur ganze Worte ersetzt. Der Wert 1 in *all* bewirkt, daß alle Vorkommen ersetzt werden. *ask* gleich 1 bewirkt schließlich, daß der Benutzer gefragt wird, ob die Ersetzung vorgenommen werden soll.

Beispiel für Oberon-2: Für Oberon-2 Module soll diese Funktion den Text *MODULE modname* in *MODULE dstname* und *END modname.* in *END dstname.* ändern.

```
BOOL GetExecutable(HANDLE hDat, LPSTR exe);
```

Diese Funktion soll im Parameter *exe* den Befehl zurückliefern, mit dem das erzeugte Programm gestartet werden kann. Für Compiler-Schnittstellen, die direkt ausführbare Dateien (EXE oder DLL) erzeugen, wird der Pfad und der Dateiname des erzeugten Programmes zurückgeliefert. Compiler-Schnittstellen, die zum Beispiel Code für einen Interpreter erzeugen, liefern in *exe* den gesamten Befehl zurück, mit dem der Interpreter mit dem Projekt gestartet werden. Der Rückgabewert soll TRUE sein, wenn der Befehl in *exe* ausführbar ist, sonst FALSE.

```
BOOL CALLBACK GetTarget(HANDLE hData, LPSTR exe)
```

GetTarget soll im Parameter *exe* den Pfad und Dateinamen der Datei zurückliefern, die durch Build oder Make von diesem Projekt erzeugt wird. In der Regel handelt es sich dabei um ausführbare Dateien, wie zum Beispiel Programme oder Dynamic Link Libraries. Außerdem soll diese Funktion TRUE zurückliefern, wenn es sich bei der in *exe* angegebenen Datei um ein Programm (EXE-Datei) handelt. Sonst soll sie FALSE zurückliefern.

10.1.1.6 Hilfe

```
BOOL HelpCompiler(HANDLE hDat,
                  HWND   hwnd,
                  LPSTR  powDir,
                  WORD   wCmd,
                  DWORD  dwData);
```

Diese Funktion soll die zum Compiler mitgelieferte Hilfedatei aufrufen. Die Parameter *hwnd*, *wCmd* und *dwData* haben dieselbe Bedeutung wie in der Windows-Funktion *WinHelp*, die Interface-DLL muß den Aufruf an die richtige Hilfedatei weiterleiten. In *powDir* wird das Pow!-Arbeitsverzeichnis übergeben, in dem auch die Hilfedatei installiert werden sollte.

Rückgabewert: TRUE, wenn die DLL eine eigene Hilfedatei unterstützt, FALSE sonst.

```
void CALLBACK GetHelpFile(HANDLE hData, LPSTR name)
```

Diese Funktion soll im Parameter den Pfad und Namen einer zum Compiler gehörigen Hilfedatei zurückliefern. Diese Funktion scheint zurzeit nicht in Verwendung zu sein!

10.1.1.7 Editor

Die Editor-Schnittstelle von Pow! sieht vor, daß Editoren den Quellcode zur besseren Lesbarkeit in verschiedenen Farben darstellen können (Syntax Coloring). Dazu soll die Compiler-Schnittstelle bestimmte Eigenschaften der Sprache, wie zum Beispiel Schlüsselworte und den Aufbau von Kommentaren, bekanntgeben.

```
void EditorSyntax(HANDLE  hDat,
                  LPLONG  caseSensitive,
                  FARPROC enumKeys);
```

Alle Schlüsselwörter der Sprache sind an Pow! zu übergeben, und zwar in einer Enumeration über die Funktion *enumKeys*. Wenn *caseSensitive* auf einen Wert ungleich 0 gesetzt wird, dann bedeutet

dies, daß Schlüsselwörter genau mit der übergebenen Groß-/Kleinschreibung übereinstimmen müssen.

Dabei gilt:

```
void FAR PASCAL enumKeys (LPSTR key);
```

Pow! wird ein einzelnes Schlüsselwort *key* mitgeteilt.

```
void EditorComment(HANDLE hDat,
                   LPLONG nested,
                   LPSTR  commentOn,
                   LPSTR  commentOff);
```

In *commentOn* sind jene Zeichen, mit denen ein Kommentar beginnt (z.B. '/*'), und in *commentOff* das Gegenstück (z.B. '*/') zurückzugeben. Zeigt *commentOff* auf einen Nullstring, werden die Kommentare am Ende der Zeile automatisch beendet. Mehrfachnennungen sind durch Leerzeichen zu trennen. Pow! nimmt dann eine Gruppierung vor: Erster Teil von *commentOn* zu erstem Teil von *commentOff*, usw. Wenn *nested* auf 0 gesetzt wird, dann können Kommentare nicht geschachtelt werden. Jeder andere Wert bedeutet das Gegenteil.

```
void GetExtensions(HANDLE hDat,
                   LPEXT  far *srcExt,
                   LPINT  srcN,
                   LPEXT  far *addExt,
                   LPINT  addN);
```

Der File-Open Dialog bietet in einer Combobox verschiedene Dateinamenserweiterungen an (z.B.: *.mod, *.c, ...). Um Pow! optimal an eine Sprache anzupassen, soll diese Funktion die passenden Dateinamenserweiterungen an Pow! zurückmelden. *srcExt* soll auf die Dateinamenserweiterungen für Quelldateien und *addExt* auf jene für den Dialog Project/Edit zeigen. *srcN* soll die Anzahl der Dateinamenserweiterungen in *srcExt* und *addN* den Umfang von *addExt* enthalten.

Die Anzahl der Einträge ist mit jeweils 10 limitiert! Der Typ LPEXT ist folgendermaßen definiert:

```
#define MAXPATHLENGTH 256
typedef struct {  /* list of file extensions */
    char ext[MAXPATHLENGTH];
    char doc[256];
} EXT;
typedef EXT far *LPEXT;
```

10.1.2 Hilfsfunktionen aus PowSupp.dll

Die folgenden Funktionen sind in der DLL PowSupp implementiert und bieten Zugriff auf die Datenstruktur, die für den Parameter *flist* in der Funktion Link der Compiler-Schnittstelle verwendet wird. Es handelt sich dabei um eine einfach verkettete Liste.

Im folgenden werden jene Funktionen beschrieben, die in der Compiler-Schnittstelle benötigt werden, um mit dem Parameter *flist* der Funktion Link arbeiten zu können.

```
int FAR PASCAL CountList(HANDLE list);
```

Die Funktion *CountList* gibt die Anzahl der Einträge, die in der linearen Liste gespeichert sind, zurück.

Verwendet man diese Funktion in der Funktion Link der Compiler-Schnittstelle und übergibt ihr den Parameter *flist*, liefert diese Funktion die Anzahl der Dateien zurück, die für das Linken benötigt werden.

```
int FAR PASCAL GetElem(HANDLE list, int i, long adr)
```

Die Funktion *GetElem* ermittelt das i. Element der linearen Liste und kopiert die Daten an jene Adresse, die im Parameter adr übergeben wird. Der Rückgabewert dieser Funktion verrät, wie viele Bytes die Daten des i. Eintrages umfassen. Wird 0 zurückgegeben, dann konnte der i. Eintrag nicht gefunden werden.

Für die Verwendung in der Funktion Link der Compiler-Schnittstelle bedeutet dies, daß der Name der i. Datei zurückgegeben wird.

10.2 Editor-Schnittstelle

Der Editor ist nicht Bestandteil des Pow!-Kerns, sondern ist in einem eigenständigen Modul implementiert. Pow kann! somit verschiedene Editoren verwenden. Zurzeit werden zum Beispiel zwei Editoren mit Pow! ausgeliefert, nämlich PowEdit und Boosted. PowEdit ist sehr einfach und verwendet nur die Edit-Control von Windows und stellt damit wenig Unterstützung für den Programmierer zur Verfügung. Der Editor Boosted kann dagegen zum Beispiel Kommentare farbig darstellen und die Einrückung in blockstrukturierten Programmiersprachen selbst vornehmen. Pow! kann aber nicht gleichzeitig verschiedene Editoren verwenden. Es ist also nicht möglich, eine Datei mit Editor x und gleichzeitig eine zweite Datei mit Editor y zu editieren. Die Entscheidung,

welcher Editor verwendet werden soll, kann der Benutzer über den Dialog im Menü Options/Preferences bekanntgeben, muß sie aber vorab fällen.

Editoren sind in Dynamic Link Libraries (DLLs) implementiert. Der Kern lädt die eingestellte DLL und kommuniziert mit dem Editor über eine im folgenden beschriebene Schnittstelle. Im Gegensatz zur üblichen Endung *dll* müssen DLLs, die als Editor in Pow! arbeiten sollen, die Endung *ell* haben. In die Combobox zum Auswählen eines Editor im Menü Options/Preferences werden nämlich nur jene Dateien angezeigt, die über die Endung *ell* verfügen.

Die Schnittstelle zwischen Kern und Editor besteht im wesentlichen aus einer Reihe von Funktionen, die von der DLL exportiert werden müssen. Diese Funktionen werden im folgenden beschrieben. Dazu werden sie in einer C-ähnlichen Syntax angeschrieben.

Die Aufgabenteilung zwischen Kern und Editor sieht folgendermaßen aus. Der Kern öffnet und verwaltet ein leeres MDI Child Window. Innerhalb dieses Fensters kann der Editor beliebige Fenster anlegen und verwalten. Im einfachsten Fall legt der Editor zum Beispiel eine Edit-Control über den gesamten Client-Bereich des MDI Child Window an. Der Editor legt also seine Fenster als Kinder des MDI Child Window an.

10.2.1 Funktionen

Im folgenden werden alle Funktionen der Editor-Schnittstelle samt ihrer Schnittstelle beschrieben. Bei den meisten Funktionen wird ein Fenster-Handle übergeben. Dabei handelt es sich um den Handle des MDI Child Window.

```
int InterfaceVersion (void);
```

Der Editor muß die Versionsnummer der Editor-Schnittstelle zurückgeben, die er implementiert. Ein Wert von 180, zum Beispiel, bedeutet dabei die Versionsnummer 1.80.

```
void NewEditWindow (HWND parent, int readOnly);
```

Wenn der Kern ein neues Editierfenster öffnen will, ruft er diese Funktion auf. Der Editor muß daraufhin ein leeres Fenster als Kind des Fensters *parent* anlegen. Wenn *readOnly* ungleich Null ist, dann soll der Benutzer in diesem Fenster keine Eingaben tätigen können. Dieses Fenster dient dann nur zur Ausgabe von Textinformationen, wie zum Beispiel Fehlermeldungen vom Compiler.

```
void CloseEditWindow (HWND edit);
```

Mit dieser Funktion wird das Fenster mit dem Handle *edit* geschlossen. Es muß dabei nicht berücksichtigt werden, ob seit dem letzten Speichern Änderungen an der Datei eingegeben wurden. Das Fenster kann ohne Rücksicht darauf geschlossen werden.

```
int HasChanged (HWND edit);
```

Diese Funktion gibt einen Wert ungleich Null (TRUE) zurück, wenn die im Fenster *edit* angezeigte Datei seit dem letzten Speichern verändert worden ist.

```
int LoadFile (HWND edit, LPSTR name);
```

Wenn der Pow!-Benutzer eine Quelldatei öffnet, wird die Funktion *LoadFile* zum Laden einer Datei aufgerufen. Der Editor muß die in *name* übergebene Datei laden und im Fenster *edit* anzeigen.

```
int SaveFile (HWND edit, LPSTR name);
```

Die Funktion *SaveFile* ist das Pendant zu *LoadFile*. Sie speichert den im Fenster *edit* befindlichen Text in einer Datei mit dem Namen, der in *name* übergeben wird, ab. Der Aufruf der Funktion *HasChanged* muß nach dem erfolgreichen Speichern solange TRUE zurückgeben, bis der Benutzer den Text im Fenster *edit* verändert.

```
void GetCursorpos (HWND edit, LPLONG row, LPLONG col);
```

Diese Funktion liefert die aktuelle Position der Textmarke des Fensters *edit* in den Parametern *row* und *col* zurück.

```
int Cut (HWND edit);
```

Wenn ein Text im Fenster *edit* selektiert ist, gibt diese Funktion den Wert 1 zurück, kopiert den im Fenster *edit* selektierten Text in die Zwischenablage und löscht den selektierten Text. Ist kein Text selektiert, gibt diese Funktion den Wert 0 zurück.

```
int Copy (HWND edit);
```

Wenn ein Text im Fenster *edit* selektiert ist, gibt diese Funktion den Wert 1 zurück und kopiert den im Fenster *edit* selektierten Text in die Zwischenablage. Ist kein Text selektiert, gibt diese Funktion den Wert 0 zurück.

```
int Paste (HWND edit);
```

Wenn Text in der Zwischenablage enthalten ist, wird dieser Text an der aktuellen Position der Textmarke im Fenster *edit* eingefügt und der Wert 1 zurückgegeben. Ist kein Text in der Zwischenablage vorhanden, wird der Wert 0 zurückgegeben.

```
int Clear (HWND edit);
```

Clear löscht den im Fenster *edit* selektierten Text und gibt den Wert 1 zurück. Wenn kein Text selektiert ist, gibt die Funktion den Wert 0 zurück.

```
int CanUndo (void);
```

Die Funktion *CanUndo* wird vom Kern aufgerufen, um herauszufinden, ob der Editor den letzten Befehl rückgängig machen kann (Undo) und ob ein rückgängig gemachter Befehl wiederhergestellt werden kann (Redo). Die Funktion gibt die Fähigkeiten im Rückgabewert zurück. Ein Wert 0 bedeutet, daß der Editor keine der beiden Fähigkeiten unterstützt, 1 bedeutet, daß der Editor Undo unterstützt und 2 bedeutet, daß der Editor beides unterstützt.

```
void Undo (HWND edit);
```

Diese Funktion macht den letzten Befehl im Fenster *edit* rückgängig.

```
void Redo (HWND edit);
```

Diese Funktion stellt den letzten rückgängig gemachten Befehl im Fenster *edit* wieder her.

```
int GotoPos (HWND edit, long row, long col);
```

Diese Funktion setzt die Textmarke im Fenster *edit* an eine bestimmte Position. Die Position wird durch die Zeile *row* und Spalte *col* spezifiziert. Für die erste Zeile bzw. erste Spalte muß der Wert 1 übergeben werden. Wird der Wert -1 als Zeile und Spalte übergeben, wird die Textmarke an das Ende des Textes gesetzt.

```
int Search (HWND edit, LPSTR text,
            int matchcase, int down, int words);
```

Die Funktion *Search* sucht im Fenster *edit* nach dem Text, der im Parameter *text* übergeben wird. Wird der gesuchte Text gefunden, wird er selektiert und die Funktion gibt 1 zurück. Sonst gibt sie 0 zurück. Wenn im Parameter *matchcase* ein Wert ungleich Null übergeben wird, wird die Groß- und Kleinschreibung bei der Suche berücksichtigt. Im Parameter *down* kann die Suchrichtung angegeben werden. Der Wert 1 bedeutet, daß der ab der aktuellen Position der Textmarke bis zum

Ende der Datei gesucht werden soll. 0 bedeutet dagegen, daß bis zum Anfang der Datei gesucht wird. Im Parameter *words* wird angegeben, ob nur ganze Worte gesucht werden sollen. Ein Wert 1 in diesem Parameter bedeutet, daß nach dem Suchtext nur als ganzes Wort gesucht wird.

```
int Replace(HWND edit, LPSTR text, LPSTR new,
            int matchcase, int down, int words, int all, int ask);
```

Die Funktion *Replace* sucht wie die Funktion *Search* im Fenster *edit* nach einem Text, der im Parameter *text* übergeben wird. Allerdings ersetzt diese Funktion den gefunden Text durch den Text, der im Parameter *new* übergeben wird. Als Ergebnis gibt die Funktion die Anzahl der Ersetzungen zurück.

Die Parameter *matchcase*, *down* und *words* haben dieselbe Bedeutung wie bei der Funktion *Search*. Der Parameter *all* gibt an, ob das Ersetzen für die gesamte Datei im Fenster *edit* durchgeführt werden soll. Ein Wert 1 bewirkt, daß alle Fundstellen durch den neuen Text ersetzt werden. Der Wert 0 bewirkt dagegen, daß nur der nächste Fund ersetzt wird. Ein Wert 1 im Parameter *ask* teilt dem Editor mit, den Benutzer vor jeder Ersetzung zu fragen, ob der gefundene Text ersetzt werden soll.

```
void EditOptions (void);
```

Diese Funktion öffnet einen Dialog, in dem der Benutzer Optionen für den Editor einstellen kann. Nach dem Schließen des Dialogs muß die Funktion dafür sorgen, daß die Einstellungen gespeichert werden. Typischerweise kann der Benutzer in diesem Dialog die Schriftart und -größe im Editorfenster einstellen.

```
void Keywords (int caseSensitive, LPSTR words);
```

Wenn der Editor Schlüsselworte der verwendeten Programmiersprache hervorheben will, dann benötigt er die Informationen, die beim Aufruf dieser Funktion übergeben werden. Der Kern ruft diese Funktion auf, wenn der Editor geladen wird und jedesmal wenn das Compilerinterface geändert wird. Soll der Editor diese Funktionalität nicht unterstützen, dann kann diese Funktion leer bleiben.

Der Parameter *caseSensitive* gibt an, ob die Programmiersprache Groß- und Kleinschreibung unterscheidet. Dabei bedeutet ein Wert ungleich Null, daß die Groß- und Kleinschreibung beachtet wird. Im zweiten Parameter wird eine Liste von Schlüsselworten übergeben. Die Liste besteht aus aneinander gereihten Zeichenketten, die mit einem Nullbyte abgeschlossen sind. Nach der letzten Zeichenkette signalisiert ein weiteres Nullbyte das Ende der Liste. Am Ende der Liste befinden

sich somit zwei Nullbytes hintereinander, nämlich das Nullbyte der letzten Zeichenkette und das Nullbyte der Liste.

```
void Comments (int nested, LPSTR on, LPSTR off, LPSTR string);
```

Ähnlich wie die Funktion *Keywords* hilft diese Funktion dem Editor sich auf die verwendete Programmiersprache einzustellen. Dem Editor wird damit mitgeteilt, welche Kommentare die Programmiersprache versteht. Damit kann der Editor den Kommentar zum Beispiel in einer anderen Farbe darstellen. Soll der Editor diese Funktionalität nicht unterstützen, dann kann diese Funktion leer bleiben.

Die Parameter *on* und *off* geben an, wie Kommentare beginnen und enden. Beide Parameter können eine Liste von Zeichenketten enthalten. Die einzelnen Einträge werden durch Leerzeichen voneinander getrennt. Das Ende der Liste wird durch ein Nullbyte signalisiert.

Der Einträge der beiden Parameter korrespondieren miteinander. Das bedeutet, daß der erste Eintrag in *off* angibt, wie der Kommentar, der mit der ersten Zeichenfolge von *on* begonnen wurde, beendet wird. Der korrespondierende Eintrag in *off* kann auch leer sein. Das bedeutet, daß solche Kommentare bis zum Zeilenende gehen.

Mit dem Parameter *nested* wird der Editor darüber informiert, ob Kommentare auch geschachtelt werden können. Ein Wert ungleich Null bedeutet dabei, daß dies der Fall ist.

Im letzten Parameter werden ebenfalls die Start- und Endzeichenketten für Kommentare übergeben.

```
void SetCommandProcedure (FARPROC command);
```

Mit dieser Funktion kann man eine Funktion bestimmen, die der Editor jedesmal aufruft, wenn eine spezielle Taste, z.B. ALT-F1, gedrückt wird. An die aufgerufene Funktion wird der Text der aktuellen Selektion übergeben. Sie ist folgendermaßen zu definieren: void Command (LPSTR selection).

```
void SetHelpFile (LPSTR name);
```

Diese Funktion übergibt dem Editor den Dateinamen der Hilfedatei des Compilers. Damit kann der Editor beim Drücken der Taste F1 die Hilfe des Compilers aufrufen und beim Aufruf den selektierten Text übergeben, um zum Beispiel Informationen zu einer bestimmten Funktion anzuzeigen.

```
long GetFirstBuffer (HWND edit, LPSTR buf, long size);
```

Die beiden Funktionen *GetFirstBuffer* und *GetNextBuffer* werden gemeinsam verwendet. Beide dienen dazu, den Text, der sich gerade im Fenster *edit* befindet, zu ermitteln. Jeder Aufruf liefert dabei einen gewissen Teil des Textes zurück.

Wie der Name schon sagt, muß zuerst die Funktion *GetFirstBuffer* aufgerufen werden. Sie liefert den ersten Teil zurück Danach muß die Funktion *GetNextBuffer* aufgerufen werden, die alle weiteren Teile zurückliefert. Wenn die Funktion *GetNextBuffer* 0 zurückliefert, gibt es keine weiteren Teile mehr.

Der Parameter *size* gibt an, wie groß der Speicherbereich ist, auf den der Parameter *buf* zeigt. Beide Funktionen geben die Anzahl der Bytes zurück, die tatsächlich in den Speicherbereich des Parameters *buf* kopiert wurden.

```
long GetNextBuffer (HWND edit, LPSTR buf, long size);
```

Siehe Funktion *GetFirstBuffer*.

```
int GeneratesAscii (void);
```

Üblicherweise werden Quelldateien als ASCII-Text gespeichert. Dies erwarten auch die meisten Compiler. Die Editor-Schnittstelle von Pow! sieht aber auch vor, daß der Editor seine Quelldateien in einem anderen Format speichern kann. Dann hätten aber die meisten Compiler das Problem, daß sie mit den Quelldateien nichts anfangen können. Für diesen Fall müssen die Funktionen *GeneratesAscii*, *LoadOpen*, *LoadRead* und *LoadClose* implementiert werden. Die letzten drei Funktionen werden dem Compiler zur Verfügung gestellt. Sie wandeln eine Quelldatei vom Format des Editors in einen ASCII-Text um, sodaß der Compiler die Quelldateien auch übersetzen kann.

Mit der Funktion *GeneratesAscii* teilt der Editor mit, ob er Quelldateien als ASCII-Text speichert. In diesem Fall muß diese Funktion 1 zurückliefern. Sonst gibt sie 0 zurück.

```
int LoadOpen (LPSTR fileName);
```

Diese Funktion muß nur dann implementiert werden, wenn der Editor die Quelldateien nicht als ASCII-Text speichert. In diesem Fall öffnet diese Funktion die Datei mit dem Namen *fileName* und liefert einen Datei-Handle als Ergebnis zurück.

```
long LoadRead (int handle, LPSTR buf, long size);
```

Diese Funktion liest von einer zuvor mit *LoadOpen* geöffneten Quelldatei maximal *size* Bytes ein und kopiert sie in den Speicherbereich, auf den der Parameter *buf* zeigt. Dabei wird die Quelldatei

in einen ASCII-Text konvertiert. Als Ergebnis liefert die Funktion die Anzahl der Bytes, die tatsächlich in den Speicherbereich kopiert wurden. Sind weniger Bytes als im Parameter *size* angegeben kopiert worden, dann ist das Ende der Datei erreicht.

```
void LoadClose (int handle);
```

Diese Funktion schließt eine mit *LoadOpen* geöffnete Datei.

```
HGLOBAL GetText (HWND edit);
```

Diese Funktion legt einen großen Speicherbereich mit der Funktion *GlobalAlloc* an und kopiert in diesen Speicherbereich die ASCII-Repräsentation der gesamten Quelldatei, die im Fenster *edit* geöffnet ist. Als Ergebnis liefert diese Funktion einen Handle auf den angelegten Speicherbereich zurück. Diese Funktion muß ebenfalls nicht implementiert werden, wenn die Quelldateien vom Editor im ASCII-Format gespeichert werden.

```
int PrintWindow (HWND edit);
```

Die Funktion *PrintWindow* druckt die im Fenster *edit* geöffnete Quelldatei aus. Wenn die Datei erfolgreich ausgedruckt wurde, gibt die Funktion 1 zurück, sonst wird 0 zurückgegeben.

```
int InsertText (HWND edit, LPSTR text);
```

Diese Funktion fügt den Text, auf den der Parameter *text* zeigt, an der aktuellen Cursorposition im Fenster *edit* ein. 1 wird zurückgegeben, wenn der Text eingefügt wurde, sonst wird 0 zurückgegeben.

```
int AddText (HWND edit, LPSTR text);
```

Die Funktion *AddText* fügt den im Parameter *text* übergebenen Text an das Ende der Quelldatei an, die im Fenster *edit* geöffnet ist. Konnte die Operation erfolgreich abgeschlossen werden, wird 1 zurückgegeben, sonst 0.

```
void ResizeWindow (HWND edit, int dx, int dy);
```

Mit dieser Funktion wird der Editor angewiesen, das Fenster *edit* auf eine bestimmte Größe zu ändern. Der Parameter *dx* gibt dabei die Breite in Pixel an und *dy* die Höhe in Pixel.

```
int HasSelection (HWND edit);
```

Mit dieser Funktion prüft Pow!, ob im Fenster *edit* gerade ein Text selektiert ist. Wenn dies der Fall ist, muß der Editor 1 zurückgeben, sonst 0.

```
void ResetContent (HWND edit);
```

Diese Funktion löscht den gesamten Inhalt des Fensters *edit*. Somit ist das Fenster *edit* nach Aufruf dieser Funktion vollkommen leer.

```
int GetLine (HWND edit, int line, int max, LPSTR buf);
```

Mit dieser Funktion kann man den Text einer bestimmten Zeile im Fenster *edit* ermitteln. Der Parameter *line* gibt die Zeile an, wobei mit 1 zu zählen begonnen wird. *max* gibt die maximale Größe des Speicherbereichs an, auf den der Parameter *buf* zeigt. Es werden somit nie mehr als *max* Zeichen kopiert, auch wenn die Zeile länger ist. Als Ergebnis gibt die Funktion die Anzahl der Bytes zurück, die in den Speicherbereich kopiert wurden, wobei das abschließende Nullbyte nicht mitgezählt wird.

```
void ShowHelp (HWND wnd);
```

Wenn diese Funktion aufgerufen wird, soll der Editor eine Hilfe anzeigen. Im Parameter *wnd* wird der Handle des Pow!-Fensters übergeben.

10.2.2 Optionale Funktionen

Während die bis hierher beschriebenen Funktionen immer exportiert werden müssen, sind die im folgenden beschriebenen Funktionen optional. Das bedeutet, daß Pow! prüft, ob diese Funktionen vorhanden sind. Wenn sie nicht vorhanden sind, wird im Gegensatz zu den bisherigen Funktionen keine Fehlermeldung ausgegeben.

```
void UnloadEditor (void);
```

Mit dieser Funktion wird dem Editor mitgeteilt, daß der Editor nicht mehr benötigt wird. Dies passiert, wenn Pow! beendet wird oder wenn der Benutzer einen anderen Editor verwenden will. Der Editor soll beim Aufruf dieser Funktion seine Ressourcen freigeben.

10.2.3 Nachrichten

Pow! erwartet, daß der Editor beim Eintreten bestimmter Ereignisse Nachrichten versendet. Die Nachrichten werden an das Vaterfenster von *edit* gesendet. Im folgenden werden die Nachrichten beschrieben:

PEM_SHOWLINENR = (WM_USER+1000)

Diese Nachricht wird gesendet, wenn sich die aktuelle Position der Textmarke verändert hat. Dies passiert nicht nur beim Eingeben von Text oder beim Bewegen der Textmarke, sondern auch wenn zum Beispiel ein anderes Fenster aktiv wird. Diese Nachricht wird also immer dann gesendet, wenn Pow! die Position der Textmarke in der Statuszeile ändern soll. In *wParam* wird dabei die Spalte und in *lParam* die Zeile übergeben.

PEM_SHOWINSERTMODE = (WM_USER+1001)

Diese Nachricht wird gesendet, wenn Pow! den Einfügestatus in der Statuszeile ändern soll. In *wParam* wird dabei der aktuelle Status übergeben. Der Wert 0 bedeutet, daß sich der Editor im Modus Überschreiben befindet und 1 bedeutet, daß sich der Editor im Modus Einfügen befindet.

PEM_SHOWCHANGED = (WM_USER+1002)

Ähnlich wie die vorhergehende Nachricht, wird diese Nachricht immer dann gesendet, wenn sich der Änderungsstatus ändert. Der Änderungsstatus zeigt an, ob die Datei seit dem letzten Speichern geändert wurde oder nicht. Dieser Status wird in *wParam* übergeben. Der Wert 1 bedeutet, daß die Datei geändert wurde und 0 bedeutet, daß sie nicht geändert wurde.

PEM_DOUBLECLICK = (WM_USER+1003)

Diese Nachricht zeigt an, daß der Benutzer einen Doppelklick im Editorfenster ausgeführt hat.

Die Nachricht PEM_DOUBLECLICK sollte für Editorfenster, die als lesbar (*readonly*) geöffnet wurden, auch dann gesendet werden, wenn der Benutzer die Enter-Taste drückt. Pow! verwendet nämlich *readonly* Editorfenster für die Ausgabe von Fehlermeldungen. Drückt der Benutzer in einem solchen Fenster die Enter-Taste, dann springt Pow! automatisch zum nächsten selektierten Fehler.

10.2.4 Weitere Konventionen

Es wird angenommen, daß das Fenster zum Editieren des Textes die ID 0x0CAC hat. Pow! sendet die Benachrichtigungen EN_CHANGED und EN_ERRSPACE an dieses Fenster.

11 Literaturverzeichnis

[AG96] Arnold Ken, Gosling James. The Java Programming Language. Addison-Wesley, 1996. ISBN 0-201-63455-4.

[ARN94] Arnold Robert S. Software Reengineering: A Quick History. Communications of the ACM, Vol. 37, No. 5, 13-14. May 1994

[BCF95] Brandis M., Crelier R., Franz M., Templ J. The Oberon System Family. Software – Practice and Experience, Vol. 25, No. 12, 1331-1366. December 1995.

[BH90] Bocker H. D., Herczeg J.. Browsing through program execution, Interact' 90, pp. 991-996, 1990

[BH91] Brown Marc H., Hershberger John. Color and Sound in Algorithm Animation, IEEE Computer, 25(12):52-63, October 1991. Also appeared in 1991 IEEE Workshop on Visual Languages and as SRC Research Report 76a.

[BON97] Boninsegna Werner. Symbolischer Debugger für Windows NT. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität, Linz, 1997

[BRO92] Brown Marc H. (1992). Zeus: A System for Algorithm Animation and Multi-view Editing (Research Report No. 75). DEC Systems Research Center, Palo Alto, CA.

[BRO93] Brockhaus-Enzyklopädie in vierundzwanzig Bänden, Band 20. Neunzehnte Auflage. F.A. Brockhaus GmbH, Mannheim, 1993.

[BRO94] Brockhaus-Enzyklopädie in vierundzwanzig Bänden, Band 23. Neunzehnte Auflage. F.A. Brockhaus GmbH, Mannheim, 1994.

[CYG01] Cygwin. http://sourceware.cygnus.com/cygwin/index.html. 23.05.2000.

[CYG02] The Cygwin FAQ. http://sourceware.cygnus.com/cygwin/faq/. 23.05.2000

[CYG03] The Cygwin FAQ. http://sourceware.cygnus.com/cygwin/faq/faq.html#SEC1. 23.05.2000

[DIC99] Dicklberger C. POWERED – Ein Resource-Editor für Pow!. Diplomarbeit, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität, Linz, 1999

[DJM00] Dietmüller Peter R., Jöbstl Markus, Mühlbacher Jörg R., Zwicknagel Wolfgang. Real-time Visualisation of Object Structures for Semantic Validation. Accepted for: Proceedings of 26th Euromicro Conference, 2000.

[DM90] Ding Chen, Mateti Prabhaker. A framework for the automated drawing of data structure diagrams. IEEE Transactions on Software Engineering, Vol. 16, No. 5, 543-557, May 1990.

[ECM90] European Computer Manufacturing Association (ECMA), Standard ECMA-149: Portable Common Tool Environment (PCTE) Abstract Specification, Final Draft, 1990

[ESC92] Eschelbeck Gerhard. Implementierung einer objektorientierten C++ Klassenbibliothek für grafische Prozeßvisualisierungen unter MS-Windows und deren Anwendung als Ethernet-Monitor in einem Client-Server-Modell. Diplomarbeit, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität, 1992.

[FEL79] Feldman S. I.. Make – a program for maintaining computer programs. Software – Practice and Experience, Vol. 9, No. 4, 1979.

[FM97] Fortner Brand, Meyer Theodore E.. Number by Colors, A Guide To Using Color To Understand Technical Data. Springer Verlag, New York. 1997.

[FOR93] Ford Lindsey. Interactive Learning and Researching with Visualization. Department of Computer Science, University of Exeter. Research Report No. 274, 1993. ftp://ftp.dcs.ex.ac.uk/pub/media/sv/ifip.ps.Z (21.06.00)

[GHV95] Gamma E., Helm R., Vlissides J.. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA. Addison-Wesley, 1995

[GJS96] Gosling James, Joy Bill, Steele Guy. The Java Language Specification. Addison-Wesley, 1996. ISBN 0-201-63451-1.

[GNU01] The GNU Project. http://www.gnu.org (26.05.00).

[GSC96] Gschnell Christian. Portable Programmierung. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1996.

[GY96a] Gosling James, Yellin Frank, The Java Team. The Java Application Programming Interface, Volume 1: Core Packages. Addison-Wesley, 1996. ISBN 0-201-63453-8.

[GY96b] Gosling James, Yellin Frank, The Java Team. The Java Application Programming Interface, Volume 2: Window Toolkit and Applets. Addison-Wesley, 1996. ISBN 0-201-63459-7

[HAB95] Hable Richard. Ein Oberon-2 Compiler für 80960-Prozessoren. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1995

[HEL99] Helml Thomas. GNU-C++ Compiler Interface für Pow!, Version 1.0. Programmierprojekt am Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), 1999.

[HOL98] Holzleitner Gernot. OLE Anbindung an Pow!. Diplomarbeit, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1998.

[JDK01] Java Development Kit (JDK) Development Tools. http://java.sun.com/products/jdk/-1.2/docs/tooldocs/tools.html (29.05.00)

[JL96] Jones Richard, Lins Rafael. Garbage Collection – Algorithms for Automatic Dynamic Memory Managament. John Wiley & Sons. 1996

[JOE99] Jöbstl Markus. Transport semantischer Informationen in der Programmvisualisierung. Diplomarbeit, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1999.

[JS94] Jerding Dean F., Stasko John T. Using Visualization to Foster Object-Oriented Program Understanding. Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-94-33, July 1994.

[KIR95] Kirk Brian (Ed.). The Oakwood Guidelines for Oberon-2 Compiler Developers. Robinson Associates, Red Lion House, St. Mary's Street Painswick GLOS, GL6 6QR. ftp://ftp.fim.uni-linz.ac.at/pub/soft/pow-oberon2/oakwood/oakwood.ps.zip (29.05.00).

[KM92] Kirk B., Mühlbacher J. R.. Oberon for Everyone. Second European Modula-2 Conference, De Montfort University, Leicester, UK, 1992, 93-105.

[KM95] Koskimies K., Mössenböck H. Scenario-Based Browsing of Object Oriented Systems with Scene. Technical Report No. 4, August 1995, Institut für Informatik, Altenbergerstr. 69, A-4040 Linz

[KNU73] Knuth Donald E. The Art of Computer Programming, volume I: Fundamental Algorithms, chapter 2. Addison-Wesley. second edition, 1973.

[KOS97] Rainer Koschke. Reengineering Bibliography. Institute of Computer Science, University of Stuttgart. http://www.informatik.uni-stuttgart.de/ifi/ps/reengineering (29.05.00)

[KRE99] Kreuzeder, Ulrich. POW! - The Programmers Open Workbench. Journal of Systems Architecture, 45 (11) (1999 (submitted 1997)) pp. 909-918.

[LCC01] Jacob Navia (jacob@jacob.remcomp.fr). LCC-Win32: a free compiler system for Windows. http://www.cs.virginia.edu/~lcc-win32/. 23.05.2000.

[LEI00] Leisch B. Vorabversion der Dissertation, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität, Linz, 2000.

[LEI93] Leisch B. OPAL – Oberon Portable Applications Library. SYSPRO 53/93, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität, Linz, 1993.

[LY96] Lindholm Tim, Yellin Frank. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[MDJ99] Mühlbacher Jörg R., Dietmüller Peter R., Jöbstl Markus. Extendable Object Visualisation for Software Reengineering. In: Proceedings of the 25th Euromicro Conference, Volume II, 229 - 236. IEEE, 1999. ISBN 0-7695-0321-7.

[MIC92] Microsoft Corporation. Guide To Programming for the Microsoft Windows Operating System, Microsoft Windows Version 3.1. Microsoft Corporation, 1992

[MIN01] Jan-Jaap van der Heijden (J.J.vanderHeijden@student.utwente.nl). GNU MingW32 Tools. http://agnes.dida.physik.uni-essen.de/~janjaap/mingw32/index.html (23.05.00).

[MIN02] MinGW – Minimalist GNU for Windows. http://www.mingw.org (23.05.00).

[MLK95] Mühlbacher J. R., Leisch B., Kreuzeder U. Programmieren in Oberon-2 unter Windows. Carl Hanser Verlag, 1995.

[MLK97] Mühlbacher J.R., Leisch B., Kirk B., Kreuzeder U. Oberon-2 Programming with Windows. Springer Verlag, 1997.

[MOE93] Mössenböck Hanspeter. Objektorientierte Programmierung in Oberon-2. Springer Verlag, 1993.

[MOE94] Mössenböck Hanspeter. Extensibility in the Oberon System. Nordic Journal of Computing 1(1994), 77-93.

[MOH88] Moher Thomas G.. PROVIDE: A process visualization and debugging environment. IEEE Transactions on Software Engineering, 14(6):849-857, June 1988.

[MÜL97] Müller Bernd. Reengineering - eine Einführung. Teubner, Stuttgart, 1997.

[MW91a] Mössenböck H., Wirth N. Differences between Oberon and Oberon-2. Structured Programming, Vol. 12, No. 4, 175-177, 1991.

[MW91b] Mössenböck H., Wirth N. The Programming Language Oberon-2. Structured Programming, Vol. 12, No. 4, 179-195, 1991.

[PBS93] Price Blaine A., Baecker Ronald M., Small Ian S. A principled taxonomy of software visualization. Journal of Visual Languages and Computing, Vol. 4, No. 3, 211-266. September 1993.

[PFE97a] Pfeifer Bernhard Erich. Compiler-Einbindung in Pow!. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1997.

[PFE97b] Pfeiffer Michael. Heap-Browser, Version 0.6a, Programmdokumentation. Programmierprojekt Systemprogrammierung SS 1997, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1997

[PHK93] Pauw Wim De, Helm Richard, Kimelman Doug, Vlissides John. Visualizing the behavior of object-oriented systems. In Proceedings of the ACM OOPSLA '93 Conference, pages 326-337, Washington, D.C., October 1993.

[PHT91] Pfister Cuno (ed.), Heeb Beat, Templ Josef. Oberon Technical Notes. Report 156. ETH Zürich, March 1991.

[PIE96] Pietrek M., Ein Blick unter die neue Haube – Windows NT 4.0 für Programmierer, Microsoft Systems Journal 8/96

[PKV94] Pauw Wim De, Kimelman Doug, Vlissides John. Modeling Object-Oriented Program Execution. ECOOP ‘94 Proceedings, Springer Verlag, 1994.

[RAT99] Rathmayr Andreas. Konzeption und Implementierung einer Testumgebung für das Pow!-Werkzeug. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1999.

[REI85] Reiss Steve P.. Pecan: Program development systems that support multiple views. IEEE Transactions on Software Engineering, SE-11(3):276-285, March 1985.

[REI91] Reiser M. The Oberon System: Users Guide and Programmers Manual. Addison-Wesley, 1991

[RP97] Rechenberg Peter, Pomberger Gustav (Hrsg.). Informatik-Handbuch. Carl Hanser Verlag, 1997.

[RW91] Reiser Martin, Wirth Niklaus. Programming in Oberon: Steps Beyond Pascal and Modula-2. Addison-Wesley, 1991.

[RW94] Reiser Martin, Wirth Niklaus. Programmieren in Oberon: Das neue Pascal. Addison-Wesley, 1994.

[SAR96] Sarma Debabrata. DLLs for Beginners. November 1996. Microsoft Developer Network CD, April 2000.

[SB94] Sarkar Manojit, Brown Marc H. Graphical Fisheye views. Communications of the ACM, Vol. 37, No. 12, 73ff. December, 1994.

[SCH97] Schakerl Christian. Ein objektorientierter Linker für Windows NT. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1997.

[SDK01] Microsoft Platform Software Development Kit. Microsoft Developer Network CD, April 2000. Microsoft Corporation.

[SOL98] Solomon David A. Inside Windows NT, 2nd edition. Microsoft Press, 1998.

[SP92] Stasko John T., Patterson Charles. Understanding and characterizing software visualization systems. In Proceedings of the 1992 IEEE Workshop on Visual Languages, pages 3-10, Seattle, WA, September 1992.

[SS92] Shilling John J., Stasko John T. Using Animation to Design, Document and Trace Object-Oriented Systems, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-92-12, June 1992.

[SS94] Shilling John J., Stasko John T. Using animation to design, document and trace object-oriented systems. Object Oriented Systems, 1(1): 5-19, September 1994.

[STA92] Stasko John T. Animating Algorithms with XTANGO, SIGACT News, Vol. 23, No. 2, Spring 1992, pp. 67-71.

[SW67] Schorr H., Waite W. An efficient machine independent procedure for garbage collection in various list structures. Communication of the ACM, 10(8):501-506, August 1967.

[TGP89] Taenzer D., Ganti M., Podar S. Object-Oriented Software-Reuse: The Yo-yo Problem. Journal of Object Oriented Programming, 2(3), 1989, 30-35.

[WG89] Wirth N., Gutknecht J. The Oberon-System. Software – Practice and Experience, Vol. 19, No. 9., 857-893, 1989.

[WG92] Wirth N., Gutknecht J. Project Oberon: The Design of an Operating System and Compiler. Addison-Wesley, 1992

[WIE93] Wiesinger Bernhard,. Entwicklung objektorientierter Programme für Fenstersysteme unter spezieller Berücksichtigung von MS-Windows und C++. Diplomarbeit, Forschungsinstitut für Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, 1993

[WIL92] Wilson Paul R. Uniprocessor Garbage Collection Techniques. In: Yves Bekkers and Jacques Cohen (Ed.). Proceedings of International Workshop on Memory Management. Lecture Notes in Computer Science, Vol. 637. Springer-Verlag, 1992. ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps (26.05.00).

[WIL94] Wilson Paul R. Uniprocessor Garbage Collection Techniques, 1994. Submitted to ACM Computing Surveys. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps (26.05.00).

[WIR77a] Wirth Niklaus. Modula - A Language for Modular Multiprogramming. Software - Practice and Experience, Vol. 7, No. 1, 3-35, 1977.

[WIR77b] Wirth Niklaus. The Use of Modula. Software Practice and Experience, Vol. 7, No. 1, 37-65, 1977.

[WIR77c] Wirth Niklaus. Design and Implementation of Modula. Software - Practice and Experience, Vol. 7, No. 1, 67-84, 1977.