Modula-2 Hypertext Browser

Anpassung eines bestehenden Hypertext Browsers für C++ an die Programmiersprache Modula-2

Diplomarbeit

Zur Erlangung des akademischen Grades eines

Magisters der Sozial- und Wirtschaftswissenschaften

in der Studienrichtung Wirtschaftsinformatik

an der
Johannes Kepler Universität Linz,
eingereicht bei
o. Univ. Prof. Dr. Gustav Pomberger
Linz, August 1992

eingereicht von

Thomas Prückler

8656565
Landwiedstraße 11
A-4020 Linz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den zitierten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche gekennzeichnet habe.

Linz, im August 1992 ....

Danksagung

Ich danke meinen Eltern, die mir meine Ausbildung ermöglicht haben und mir auch in schwierigen Zeiten immer beigestanden haben.

Ich danke Herrn Professor Pomberger und allen Angehörigen des Institutes für Wirtschaftsinformatik, die mir die Erstellung der Arbeit in einer inspirierenden Umgebung ermöglichten und mich durch Anregungen unterstützten. Besonders danke ich den Herren Dipl.-Ing. Stefan Schiffer und Dipl.-Ing. Rainer Weinreich für ihre Unterstützung während der Implementierungsphase.

Mein besonderer Dank gilt Herrn Dr. Johannes Sametinger. Er hat nicht nur durch die Erstellung von DOgMA diese Arbeit möglich gemacht, sondern hatte auch jederzeit eine Antwort auf meine vielen Fragen und gab mir viele wertvolle Anregungen, sowohl bei der Implementierung als auch bei der schriftlichen Ausarbeitung.

Schließlich möchte ich noch Herrn Diplominformatiker Hans Fröhlich danken, der während der Implementierungsphase eine geduldige Zielscheibe für meine "kleinen" Boshaftigkeiten war (Danke, Eskimo!).

Inhaltsverzeichnis

Kapitel 1

Einleitung

DOgMA ist ein Werkzeug zur Unterstützung der Programmierung, Dokumentation und Wartung von C++-Programmsystemen. Das Werkzeug kann sowohl zum Erstellen und Dokumentieren von Programmen eingesetzt werden, wobei beide Aktivitäten parallel ablaufen können, als auch zur Erforschung bereits bestehender Programmsysteme. Diese Aufgaben werden durch die in DOgMA verwendeten Konzepte des Literate Programming und Hypertext unterstützt.

Das Ziel dieser Diplomarbeit besteht darin, dieses Werkzeug so umzuprogrammieren, daß die Programmiersprache Modula-2 damit verarbeitet werden kann.

Dieses Kapitel gibt einen Überblick über die gesamte Arbeit. Zuerst wird die hinter dieser Arbeit stehende Motivation dargestellt. Der zweite Abschnitt beschreibt die Schwerpunkte, auf die bei der Ausarbeitung besonders Rücksicht genommen wurde. Der letzte Abschnitt gibt einen Überblick über den inhaltlichen Aufbau der restlichen Arbeit.

1.1 Motivation

Während der letzten Jahre gewann ein Teilgebiet der Informatik, welches zuvor meist etwas vernachlässigt wurde, immer mehr an Bedeutung: die Dokumentation. Der Grund dafür ist, daß der Anteil der Wartungstätigkeiten an Programmsystemen ständig steigt. So geben Gibson und Senn an, daß die Hälfte der Zeit von Programmierern von Wartungsaktivitäten verbraucht wird (vergleiche [GS89]). Die Hälfte des Zeitaufwandes für die Wartung wiederum entfällt auf das Einarbeiten in das zu ändernde Programmsystem. Daher gewinnen alle Möglichkeiten, diesen Einarbeitungsaufwand zu minimieren, mehr an Bedeutung.

Das Erstellen einer Programmdokumentation wurde als unbeliebte Tätigkeit angesehen, die meist "am Ende" der Programmentwicklung durchgeführt wurde – also aus Zeitgründen oft nur rudimentär oder gar nicht. Dieses Versäumnis wird dann zum "Bumerang", wenn das Programm geändert werden muß, da eine gute, aktuelle Dokumentation das Einarbeiten in ein (fremdes) Programmsystem *bedeutend* erleichtert. Ein weiterer Weg, um die Einarbeitung in ein Softwaresystem zu unterstützen, ist die Bereitstellung eines Werkzeuges, welches das Programmsystem und die dazugehörige Dokumentation in geeigneter Weise aufbereitet.

Eines dieser Werkzeuge ist DOgMA[1]. Dieses Werkzeug bereitet einerseits das Programmsystem in eine dem Menschen besser zugängige Form auf, der Darstellung als Hypertext, und erlaubt andererseits die konsistente Einbindung von Dokumentation (vergleiche [PPS92]). DOgMA wurde ursprünglich für die Bearbeitung von C++-Programmen konzipiert. Da aber die Art der Unterstützung, die DOgMA für C++-Programme realisiert, auch für andere Programmiersprachen wünschenswert ist, sollte mit dieser Arbeit untersucht werden, wie gut sich der für DOgMA gewählte Ansatz auf andere Programmiersprachen übertragen läßt.

[1]Der Name DOgMA leitet sich von "DOcumentation & MAintenancs" ab, wobei das "g" für das "&" steht.

1.2 Schwerpunkte

Dieser Abschnitt beschreibt die Schwerpunkte, auf die während der Durchführung der Arbeit besonderes Augenmerk gelegt wurde.

Den Hauptschwerpunkt stellt natürlich die Implementierung selbst dar, d.h. die eigentliche Anpassung des Werkzeuges, damit dieses Modula-2-Programme bearbeiten kann.

Daneben gibt es aber auch noch einige andere interessante Aspekte dieser Arbeit:

C++-DOgMA wurde so implementiert, daß eine strikte Trennung zwischen einem sprachabhängigem und einem sprachunabhängigem Teil besteht. Eine wichtige Frage ist also die, wie gut die sprachabhängige Funktionalität in dem als sprachabhängig gekennzeichneten Teil gekapselt ist, ob Änderungen im sprachunabhängigen Teil notwendig sind und, wenn ja, wie schwierig und umfangreich diese sind. Die Antworten auf diese Fragen sollen als Grundlage für Änderungen bei einem eventuellen Neuentwurf des ursprünglichen Werkzeuges dienen.

Weiters soll das Werkzeug während der praktischen Arbeit benutzt werden, sodaß die Praxistauglichkeit und Stabilität von DOgMA, sowohl von der C++- als auch von der Modula-2-Version, getestet werden kann. Dieser Praxistest soll auch als Basis zur Anregung weiterer nützlicher Funktionen und Erweiterungen dienen.

Eine andere interessante Frage ist die nach dem Beitrag, den DOgMA zur Wartung leisten kann, da sich DOgMA ausdrücklich als Werkzeug für Dokumentation und Wartung versteht. Da die während der Erstellung der Arbeit zu erfüllende Aufgabe durchaus auch in die Kategorie Wartung eingeordnet werden kann, kann daher eine aussagekräftige Antwort auf diese Frage erwartet werden.

In diesem Zusammenhang stellt sich dann noch die Frage, inwieweit sich der objektorientierte Ansatz auf die Wartung eines Programmes auswirkt, da DOgMA objektorientiert implementiert wurde.

1.3 Übersicht

Nach der Einführung in die Arbeit im ersten Kapitel beschreibt das zweite Kapitel die Konzepte, auf denen DOgMA basiert.

Das dritte Kapitel ist der Beschreibung der beiden beteiligten Programmiersprachen gewidmet. Nach je einem kurzen Überblick über Modula-2 und C++ werden die für das Werkzeug relevanten Gemeinsamkeiten und Unterschiede der beiden doch sehr verschiedenen Sprachen behandelt.

Kapitel vier beschreibt detailliert die Funktionalität und Benutzeroberfläche von Modula-2-DOgMA.

Kapitel fünf gibt einen Überblick über die für die Anpassung von DOgMA an die Programmiersprache Modula-2 relevanten Klassen des Programmsystemes und beschäftigt sich mit einigen Aspekten der Implementierung.

In Kapitel sechs werden die Erfahrungen bei der praktischen Anwendung sowohl von C++-DOgMA als auch mit der Modula-2-Version beschrieben. Dieses Kapitel enthält auch Vorschläge für Änderungen beziehungsweise Erweiterungen des Werkzeuges.

Kapitel sieben beinhaltet einige abschließende Betrachtungen, und es wird aus der Sichtweise des Autors nach der Beendigung der Implementierung noch einmal auf die Schwerpunkte eingegangen.

Kapitel 2

Konzepte

In diesem Kapitel werden die Konzepte, die in DOgMA angewendet werden, vorgestellt. Es handelt sich dabei einerseits um das Konzept des Literate Programming, welches durch eine Synthese von Quellcode und Dokumentation eine bessere Lesbarkeit und geringeren Zeitaufwand für das Verstehen eines Softwaresystems erreichen will und andererseits das Konzept der Darstellung von Texten in Form von Hypertexten. Im letzten Abschnitt dieses Kapitels werden die Synergiepotentiale dieser beiden Konzepte dargestellt.

2.1 Literate Programming

Es soll hier weder detailliert auf die hinter Literate Programming stehende Philosophie eingegangen werden, noch eine "Gebrauchsanweisung" gegeben werden (siehe zum Beispiel [Lin89a], [Lin89b], [Tun89], [BC90] und [BG92]). Dieser Abschnitt gibt einen Überblick über die technischen Aspekte der Anwendung von Literate Programming, soweit diese für DOgMA relevant sind.

Die Grundidee von Literate Programming besteht darin, ein Programm mit der dazugehörigen Dokumentation weniger als einen an einen Compiler gerichteten Text mit einem für den Menschen bestimmten Anhang – die Dokumentation – zu sehen, als vielmehr als ein an den Menschen gerichtetes "Literarisches Werk" (vergleiche [Knu84]).

Die Dokumentation wird dabei in Kapitel unterteilt, die jeweils auch Teile des Quellcodes enthalten können. Dieser Quellcode wird dabei nicht als Einheit präsentiert, sondern wird an den Stellen, an denen ein bestimmter Codeteil beschrieben wird, in die Dokumentation eingefügt. Der Quellcode wird also in Fragmente aufgeteilt. Vielfältige, vom System automatisch generierte Querverweise helfen dabei, den Überblick zu behalten.

Zur Demonstration des Literate Programming erstellte Knuth das WEB-System (vergleiche [Knu86]), mit dessen Hilfe Programmsysteme erstellt werden können. In diesem WEB-System sind sowohl Dokumentation als auch Quellcode in derselben Datei enthalten. Es existieren zwei Programme, die die Aufgabe haben, diese Datei den Bedürfnissen des jeweiligen Empfängers entsprechend aufzubereiten. "Tangle" filtert aus der WEB-Datei die Codefragmente heraus und ordnet diese zu einem Programm, welches dann von einem Standardcompiler übersetzt werden kann. "Weave" bereitet die Dokumentation für den Menschen auf, wobei auch die oben erwähnten Indizes erzeugt werden. Die Ergebnisdatei von Weave ist wiederum eine Programmdatei, die von dem Formatierprogramm TEX zur ausdruckbaren Dokumentation aufbereitet wird (vergleiche [BG92]).

Die Grundidee des Literate Programming wird kaum in Frage gestellt, wohl aber die Methode, die Knuth zu ihrer Verwirklichung anwendet. Die Kritikpunkte werden von Bishop und Gregson (vergleiche [BG92]) zusammengefaßt:

- Ein WEB-Programm beinhaltet Kommandos dreier Programmiersprachen: die Kommandos der eigentlichen Programmiersprache, TEX-Kommandos zur Formatierung der Dokumentation und WEB-Kommandos, um die anderen Kommandos zu verbinden. Dieses Vermengen von Kommandos wirkt sich negativ auf die Lesbarkeit des WEB-Codes aus, insbesonders da die Sprachen nicht nach einem einheitlichem Schema entworfen wurden.

- Der hohe Dokumentationswert des WEB-Codes entsteht erst nach seinem Ausdruck, da die Absatznumerierung und die Querverweise im ursprünglichen WEB-Code nur in einer schwierig lesbaren Form vorhanden sind. Der Programmierer hat die Wahl, sich entweder mit dem schwer lesbaren WEB-Code zu begnügen, oder den Code immer wieder auszudrucken, um den aktuellen Stand der Dokumentation in Händen zu haben.

- WEB wurde explizit für die Programmiersprache Pascal entworfen (vergleiche [Knu86]). Moderne Programmierparadigmen wie zum Beispiel modulare oder objektorientierte Programmierung wurden beim Entwurf von WEB nicht berücksichtigt, da diese Paradigmen auch in Pascal nicht verwirklicht wurden.

- WEB erzeugt keinen reinen, für den Menschen bestimmten Quellcode. Dieser ist aber dann wichtig, wenn vom Compiler gemeldete Fehler behoben werden sollen.

Es wurden Programmierumgebungen vorgeschlagen, die versuchen, diese Nachteile zu beseitigen ([BG92] und [BC90]). Auch DOgMA versucht dies, indem der Benutzer die Dokumentation und die Verbindung von Dokumentation und Quellcode interaktiv mittels einer von Textverarbeitungen bekannten Arbeitsweise erstellen kann und so weder WEB- noch TEX-Code eingeben muß. Ebenfalls wird das aktuelle Ergebnis sofort am Bildschirm dargestellt, sodaß ein oftmaliges Ausdrucken entfällt.

2.2 Hypertext

Unter "Hypertext" versteht man eine bestimmte Darstellungsform von Information. Traditionellerweise wird Information linear dargestellt, wie zum Beispiel in einem Buch oder in einer Programmdatei. Der menschliche Geist arbeitet jedoch nicht linear, sondern bereitet sein Wissen in kleine Abschnitte auf, die durch Assoziationen verbunden sind. Hypertext ist die Übertragung dieser Darstellungsweise auf die Darstellung von Information (vergleiche [SW88]).

In einem Hypertextsystem wird die Information mittels zweier Bausteine dargestellt (vergleiche [SK89]):

Knoten: die Knoten eines Hypertextsystems enthalten die Daten, die dargestellt werden sollen. Es kann sich dabei um Text, Graphik, Animationen oder Geräusche handeln. Jeder Knoten enthält eine Information, die in sich geschlossen ist.

Verbindungen (links): diese verbinden die Knoten des Hypertextsystems. Wenn in einem Knoten eine Verbindung aktiviert wird, verzweigt die Anzeige zu dem Knoten, auf den die Verbindung zeigt.

Man kann sich ein Hypertextsystem als einen gerichteten Graphen vorstellen, wobei die Verbindungen die Kanten des Graphens sind. Einen Vorläufer dieser Systeme stellt das Lexikon mit seinen Querverweisen dar. Auch dieses wurde nicht dafür konzipiert, linear gelesen zu werden, sondern ermöglicht es seinem Benutzer, die für ihn relevante Information durch Benutzung der Querverweise zu erhalten.

Der große Vorteil von Hypertextsystemen liegt darin, daß dem Benutzer die Sucharbeit abgenommen wird, sodaß er sich schnell und ohne Mühe eine bestimmte Information aus einer großen Anzahl von Informationen verschaffen kann.

2.3 Literate Programming und Hypertext

Ein Kerngedanke des Literate Programming besteht darin, ein Programm für den Menschen besser verständlich zu machen. Dies soll dadurch erreicht werden, daß der Quellcode zusammen mit der dazugehörigen Dokumentation in kleine, in sich geschlossene Abschnitte gegliedert wird. Die Beziehung zwischen diesen Abschnitten wird durch automatisch generierte Indizes hergestellt (vergleiche [Knu86]). Das Medium, auf dem die Dokumentation ausgegeben wird, unterliegt allerdings den gleichen Beschränkungen, denen schon die verschiedenen Ausgabearten des Quellcodes an sich unterliegen: es ist streng linear, dem Benutzer fällt die Aufgabe zu, sich mittels der Indizes durch das Dokument zu bewegen.

Konsequenterweise sollte versucht werden, diese Starrheit aufzubrechen und eine einfachere Form anzubieten, um sich Dokumentation und Code erschließen zu können. Hier bietet sich als optimale Ergänzung das Hypertext-Konzept an. Mit seiner Hilfe können die Indizes und vor allem deren Benutzung unter einer modernen Benutzeroberfläche verborgen werden, sodaß der Benutzer sich ohne Mühe durch die Dokumentation und das Programm bewegen kann.

Beide Konzepte zu vereinen ist insoferne einfach, als sie über eine wichtige Gemeinsamkeit verfügen: beide "zerlegen" den darzustellenden Sachverhalt in kleine, in sich abgeschlossene und in sich verständliche Einheiten. Hier bietet sich ein Ansatzpunkt, um beide Konzepte zu integrieren, was in DOgMA konsequent verwirklicht wurde.

Kapitel 3

C++ und Modula-2

In diesem Kapitel werden die Programmiersprachen C++ und Modula-2, für die je eine DOgMA-Version existiert, beschrieben. Es wird dabei weniger ein formaler Vergleich der beiden Sprachen bezüglich der Funktionalität und der unterschiedlichen Programmier-Paradigmen durchgeführt. Vielmehr werden die Sprachen so dargestellt, daß die für die Anpassung von DOgMA an Modula-2 wichtigen Unterschiede und Gemeinsamkeiten sichtbar werden.

Zuerst werden beide Sprachen kurz vorgestellt, wobei besonders die Strukturierung des Quellcodes in Klassen und Methoden einerseits sowie Module und Prozeduren andererseits behandelt wird. Der letzte Abschnitt vergleicht die oben erwähnten Strukturbausteine und beschreibt, wie die in C++-DOgMA verarbeiteten C++-Strukturen auf die Modula-2-Strukturen abgebildet wurden.

3.1 C++

C++ ist eine Erweiterung der prozeduralen Programmiersprache C, die von Dennis Ritchie entwickelt wurde, um das Betriebssystem Unix zu implementieren. C wurde nicht explizit für ein bestimmtes Anwendungsgebiet entworfen, eignet sich aber besonders für die maschinennahe Programmierung, wobei allerdings auf die Portabilität von Programmen großer Wert gelegt wird (vergleiche [KR83]).

C++ ist eine Obermenge von C, wobei das wichtigste neue Konstrukt das der Klasse ist. C++ ist eine sogenannte Hybrid-Sprache, d.h. eine objektorientierte Sprache, bei der nach Wunsch auch "normal prozedural" gearbeitet werden kann. Eine Klasse wird in C++ dazu benutzt, Daten und die auf diese Daten einwirkenden Operationen zusammenzufassen (vergleiche [Str91] und [Mit89]). Klassen können benutzt werden um neue Klassen zu erzeugen, wobei die neuen Klassen die Daten und Operationen der Oberklasse erben und ihrem Zweck gemäß abändern können.

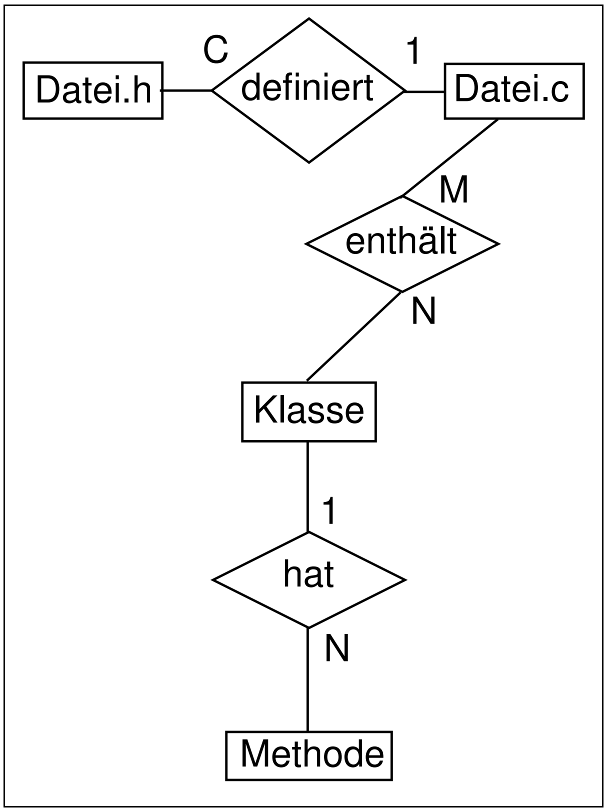

Abbildung 3.1: Strukturhierarchie von C++

Abbildung 3.1 zeigt die Struktur der Informationsbausteine von C++, die für DOgMA relevant sind. Unter einem Informationsbaustein soll in diesem Zusammenhang eine Form der Aggregierung von Anweisungen der Programmiersprache verstanden werden. Die Art der Darstellung lehnt sich an Entity-Relationship-Diagramme an (vergleiche z.B. [EN89]). In dieser Abbildung wurden nur die Konstrukte dargestellt, die für das Verständnis der Änderungen in DOgMA notwendig sind. Es wurde auf die Darstellung der Vererbung verzichtet, da diese für die Zuordnung der Informationsbausteine nicht relevant ist, ebenso wie auf die Darstellung einiger in C++ möglicher Optimierungen[1].

Zentraler Ausgangspunkt ist die Klasse. Jede Klasse hat mehrere Methoden, wobei eine Methode zu einer Klasse gehört[2]. Eine Methode ist die niedrigste Stufe der Aggregierung von Programmiersprachen-Anweisungen, von denen mehrere wiederum zu Klassen zusammengefaßt werden können. Eine "Datei.c" – also eine Implementierungsdatei – enthält ein oder mehrere Klassen[3], eine Klasse ist in einer oder mehreren Implementierungsdateien enthalten. Einer Implementierungsdatei kann eine "Datei.h" – eine Definitionsdatei – zugeordnet sein, die die Schnittstelle einiger oder aller Klassen in der Implementierungsdatei enthält.

3.2 Modula-2

Modula-2 wurde 1975 von Niklaus Wirth an der ETH Zürich entwickelt. Die Vorgänger von Modula-2 sind Pascal und Modula, die ebenfalls von Wirth entworfen wurden. Modula-2 ist eine prozedurale Sprache, die sowohl für Anwendungen des Systementwurfes auf hoher Ebene als auch für Anwendungen auf einer sehr niedrigen, maschinennahen Ebene konzipiert wurde. Ebenso wie C wurde auch Modula-2 ursprünglich benutzt, um ein Betriebssystem – in diesem Fall für den Arbeitsplatzrechner Lilith – zu implementieren (ver-

[1] In C++ ist es möglich, in die Definitionsdatei – in der Abbildung "Datei.h" – Methodenimplementierungen aufzunehmen, die vom Compiler dann speziell behandelt werden.

[2] Hier könnte man einwenden, daß eine vererbte Methode durchaus zu mehreren Klassen gehören kann. Für diese Betrachtung soll aber eine vererbte Methode als eine eigenständige Methode gelten.

[3] Würde die "Datei.c" keine Klassen enthalten wäre es nur mehr ein C-Programm und für diese Ausführungen nicht von Belang.

gleiche [Wir91]).

Das hervorstechenste Merkmal von Modula-2 ist das Konstrukt des Moduls (vergleiche [Wir91] und [Pom87]). Dieser wird dazu benutzt, um Daten und darauf agierende Prozeduren zu kapseln, sodaß man das Konstrukt des Moduls als einen Vorläufer der Klasse ansehen kann [4] (vergleiche [Rec90]).

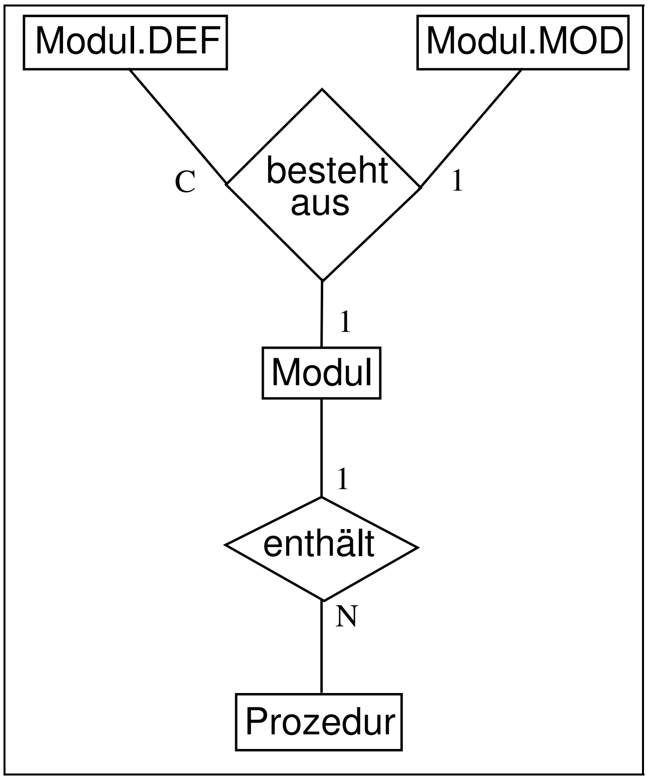

Abbildung 3.2: Strukturhierarchie von Modula-2

Abbildung 3.2 zeigt die Struktur von Modula-2 analog zu Abbildung 3.1. Ein Modula-2-Programm besteht aus einem oder mehreren Modulen. Ein Modul besteht aus einer Definition und der dazugehörigen Implementierung, die in zwei verschiedenen Dateien abgelegt werden, wobei alle Module bis auf den Hauptmodul eines Programmsystems eine Definierungsdatei besitzen. In jedem Modul können mehrere Prozeduren implementiert sein.

[4] Es wurden sogar Versuche unternommen, basierend auf Modula-2 einen objektorientierten Programmierstil zu verwirklichen (vergleiche [Bla89]).

3.3 Gegenüberstellung C++ und Modula-2

Von der Semantik und den dahinterstehenden Paradigmen aus gesehen sind Modula-2 und C++ als sehr verschieden anzusehen. Die Art und Weise, wie die Anweisungen der Programmiersprache aggregiert werden, ist allerdings sehr ähnlich – zumindest soweit, als diese Aggregierungen für DOgMA relevant sind. Abbildung 3.3 zeigt nebeneinander nochmals die beiden Strukturabbildungen. Dabei sind die Konstrukte, die aufeinander abgebildet wurden, in Ovalen zusammengefaßt.

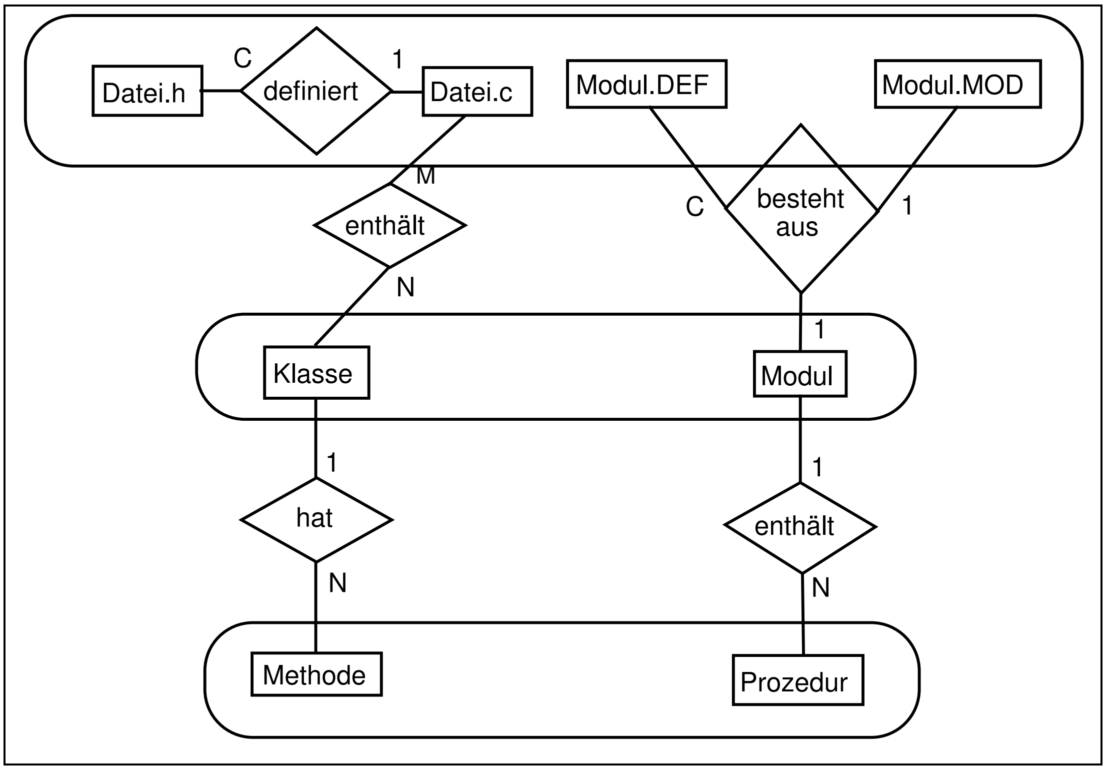

Abbildung 3.3: Vergleich der Strukturhierarchien

Folgende Zuordnungen von Informationsbausteinen wurden bei der Implementierung von Modula-2-DOgMA vorgenommen:

- Klasse → Modul
- Methode → Prozedur

- Datei.h → Modul.DEF
- Datei.c → Modul.MOD

Die Anpassung von DOgMA an Modula-2 war deshalb einfach, weil die Sprachstruktur von Modula-2 bedeutend einfacher ist als die Sprachstruktur von C++. Besonders die Beziehung der oben erwähnten Informationsbausteine untereinander ist in Modula-2 viel einfacher gestaltet:

- In C++ kann eine Klasse auf mehrere Dateien verteilt werden oder eine Datei kann mehrere Klassen beinhalten. In Modula-2 hingegen entspricht ein Modul, welches in Bezug auf DOgMA gleich wie eine C++-Klasse behandelt wird, genau einer Datei[5].
- In Modula-2 ist der Definitions- und der Implementierungsteil des Moduls klar getrennt. In C++ hingegen können sich zum Beispiel die Implementierungen von Methoden im Definitionsteil befinden.

Wäre DOgMA ursprünglich für Modula-2 implementiert worden, dann wäre der Aufwand für die Anpassungen ungleich größer gewesen.

Ein wichtiger Unterschied zwischen C++ und Modula-2 ist der, daß in C++ das Konzept der Vererbung verwirklicht wurde. Dies bedeutet, daß eine Klasse die von ihrer Oberklasse "ererbten" Daten und Methoden ohne weitere Vorkehrungen benutzen kann. Es besteht zwischen den Klassen also eine Beziehung der Art "erbt von", wobei eine Klasse jeweils genau eine Oberklasse hat. In Modula-2 fehlt dieses Konstrukt vollständig. Dieser Unterschied hat auf DOgMA keine Auswirkungen – die Funktionen, die sich mit der Vererbung beschäftigen, wurden einfach nicht benutzt.

Ein weiterer interessanter Aspekt ist der der Definition und Verwendung der Bezeichner in den beiden Programmiersprachen. Beiden Sprachen ist gemeinsam, daß für jeden Bezeichner genau eine Definition existiert und dieser Bezeichner beliebig oft verwendet werden kann. Die genaue Art und

[5]Oder zwei Dateien, wenn eine Definitionsdatei vorhanden ist.

Weise der Bezeichnerhandhabung kann in DOgMA einfach an die gewünschte Programmiersprache angepaßt werden.

Kapitel 4

Das Werkzeug Modula-2-DOgMA

Dieses Kapitel beschreibt das Werkzeug DOgMA und dessen Benutzung. Der Schwerpunkt liegt dabei in der Beschreibung der Benutzeroberfläche und deren Funktionalität. Zuerst wird jedoch Aufbau und Inhalt des Projektmoduls, mit dem gesteuert werden kann, welche Dateien in das Werkzeug geladen werden sollen, erläutert, da diese Beschreibung für das Verständnis der folgenden Abschnitte notwendig ist. Zuletzt werden die Möglichkeiten beschrieben, die Funktionalität von DOgMA von außen, also von der Betriebssystemebene aus, zu beeinflussen.

4.1 Projektmodul

Das Projektmodul ist jene Datei, deren Inhalt bestimmt, welche Quellcode- und Dokumentationsdateien von DOgMA geladen werden sollen. Der Name dieser Datei wird beim Öffnen eines Projekts angegeben (Kommando "Open Project"). Dann wird das Modul eingelesen und die darin enthaltenen Importanweisungen aufgelöst.

Die äußere Form des Projektmoduls wurde einem Modula-2-Modul nachempfunden. Folgendes Beispiel zeigt einen solchen Modul:

```
PROJECT MODULE ModulName;

  IMPORT Codedatei1;
  IMPORT MODULE Codedatei2;
  IMPORT DOCUMENTATION Dokumentationsdatei;

END ModulName.
```

Die Befehle des Projektmoduls haben folgende Bedeutung:

PROJECT MODULE ModulName kennzeichnet die Datei als Projektmodul. Der Name des Moduls – ModulName – wird zwar gegen das "END ModulName" geprüft, ansonsten aber nicht verwendet.

IMPORT Codedatei importiert die Datei "Codedatei.def". Die importierten Dateien können ebenfalls "IMPORT"-Anweisungen enthalten. In importierten Dateien wird zusätzlich noch die "FROM Codedatei IMPORT"-Anweisung verarbeitet. Wie auch bei der Übersetzung eines Programms durch den Compiler wird hier nur die Schnittstellenbeschreibung importiert.

Diese Anweisung sollte im Projektmodul dann verwendet werden, wenn nur die Schnittstellenbeschreibung eines Moduls vorhanden ist oder die Implementierung für den Bearbeiter nicht relevant ist. Grundsätzlich könnte auch die Anweisung "FROM Codedatei IMPORT" im Projektmodul mit denselben Auswirkungen wie "IMPORT Codedatei" verwendet werden, in Bezug auf die Lesbarkeit und Konsistenz ist diese Variante allerdings nicht empfehlenswert.

IMPORT MODULE Codedatei importiert die Datei "Codedatei.mod" und, so vorhanden, auch "Codedatei.def". Damit ist es möglich, auch Haupt- und Implementierungsmodule einfach und konsistent einlesen zu können. Die Anweisung "IMPORT MODULE" gehört nicht zur

Sprachdefinition von Modula-2 und sollte daher nicht in Programmdateien verwendet werden, obwohl DOgMA die Anweisung auch dort korrekt verarbeiten würde.

IMPORT DOCUMENTATION Dokumentationsdatei liest die Datei "Dokumentationsdatei.doc" ein. Auch hier werden eventuelle Unterkapitel automatisch eingelesen. "IMPORT DOCUMENTATION" gehört ebenfalls nicht zur Sprachdefinition von Modula-2.

Im Projektmodul können zwar alle Anweisungen von Modula-2 verwendt werden, sinnvoll erscheint dies allerdings nur für Kommentare.

Wenn ein Projekt eingelesen wird, liest DOgMA zuerst das Projektmodul ein. Ausgehend von den dort vorgefundenen Anweisungen werden die angeführten Dateien eingelesen. Wenn diese Dateien selbst Importanweisungen enthalten, werden diese wie oben beschrieben, im Bedarfsfall auch rekursiv, ausgeführt.

Es existiert ein Schutzmechanismus, der bewirkt, daß der Benutzer nur Dateien verändern kann, die Teil seines Arbeitsgebietes sind: Wenn eine Datei lediglich aufgrund einer Importanweisung in einer Programmdatei eingelesen wird, kann diese Datei nicht verändert werden. Dies bedeutet, daß alle Dateien, die der Benutzer verändern möchte, im Projektmodul aufgeführt werden müssen.

4.2 Benutzeroberfläche

In diesem Abschnitt wird die Benutzeroberfläche von Modula-2-DOgMA beschrieben. Zuerst werden die verschiedenen Fenster des Werkzeuges vorgestellt. Dann wird ausgehend von der Menüleiste und den anderen PopUp-Menüs auch die durch die Kommandos aktivierte Funktionalität behandelt, wobei sich die Gliederung dieser Beschreibung an der Gliederung der Menüeinträge orientiert. Diese Darstellungsart wurde deshalb gewählt, weil die Anordnung der Menüpunkte nicht wahllos, sondern nach Aufgaben geordnet vorgenommen wurde.

4.2.1 Applikationsfenster

Das Applikationsfenster (siehe Abbildung 4.1) stellt den Ausgangspunkt für die Benutzung von DOgMA dar. Ausgehend von diesem Menü können die einzelnen Programmteile durch Betätigen der entsprechenden Knöpfe aufgerufen werden.

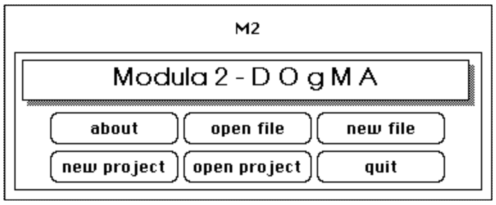

Abbildung 4.1: Applikationsfenster

about Durch Auswählen dieses Knopfes wird eine Dialogbox (siehe Abbildung 4.2) angezeigt, die Informationen über DOgMA und seine Programmierer enthält.

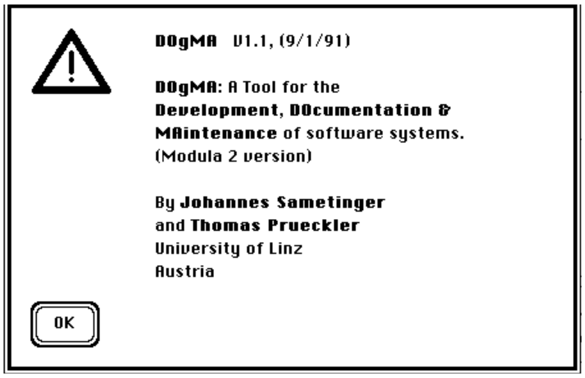

Abbildung 4.2: Dialogbox "About"

open file Dieses Kommando fragt zuerst nach einer bestehenden Datei, die dann im Dateieditor (siehe Abschnitt 4.2.6) bearbeitet werden kann.

Das Kommando öffnet den sogenannten Datei-Dialog (siehe Abbildung 4.3), mit dessen Hilfe in DOgMA alle Dateinamen eingegeben werden. In diesem Dialog kann aber nicht nur der Dateiname eingegeben werden, sondern es besteht auch die Möglichkeit, die Verzeichnisse nach einer bestimmten Datei zu durchsuchen und diese Datei auszuwählen, ohne den Namen textuell eingeben zu müssen.

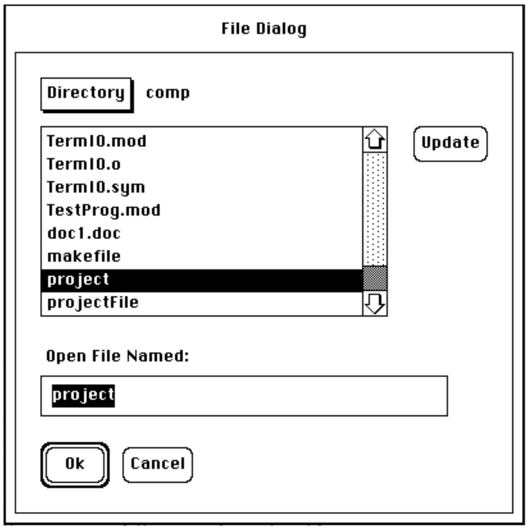

Abbildung 4.3: Dialog "File"

new file Dieses Kommando fragt nach dem Namen einer zu erstellenden Datei, die dann im Dateieditor bearbeitet werden kann.

new project Dieses Kommando erlaubt es, ein neues Projekt anzulegen. Nachdem im Datei-Dialog der Name des Projektmoduls angegeben wurde, wird dieses Modul erstellt. Dann wird ein Hypertextfenster für das neue Projekt geöffnet, wo dann mit den verschiedenen Kommandos des Menüs "Edit" das Programmsystem aufgebaut werden kann.

Alternativ dazu kann ein neues Projekt auch händisch erstellt werden, indem ein Projektmodul (siehe Abschnitt 4.1) mit dem Dateieditor oder einem beliebigem anderen Editor vorbereitet wird.

open project In der Dialogbox wird der Name des Projektmoduls ausgewählt, welches von DOgMA bearbeitet werden soll. Dann wird dieses Projektmodul eingelesen und die darin angeführten Dateien werden geladen (siehe Abschnitt 4.1 und Abschnitt 4.2.4.3 - Kommando "Read

Project"). Wenn alle Dateien eingelesen sind, wird das Hypertextfenster geöffnet und der Benutzer kann mit seiner Arbeit beginnen.

quit Dieser Knopf schließt das Applikationsfenster und damit gleichzeitig auch die Anwendung.

Da das Applikationsfenster wichtige Steuerfunktionen ausführt, ist es immer am Bildschirm präsent. Bei der Bearbeitung kann es vorkommen, daß das Applikationsfenster durch andere Fenster überlagert wird und damit nicht mehr benutzt werden kann. Für diesen Fall steht sowohl im Hypertextfenster als auch im Dateieditor das Kommando "Show Application Window" zur Verfügung, welches das Applikationsfenster über allen anderen Fenstern positioniert, sodaß es sofort sichtbar wird.

4.2.2 Hypertextfenster

Das Hypertextfenster (siehe Abbildung 4.4) ist jenes Fenster, in dem der Benutzer ein Programmsystem untersuchen und bearbeiten kann. Dieses Fenster ist in folgende Komponenten gegliedert:

Menüleiste Die Menüleiste wird dazu benutzt, die Kommandos von DOgMA zu aktivieren. Die Menüleiste wird in Abschnitt 4.2.4 genau beschrieben.

Informationsbox Die Informationsbox zeigt wichtige Informationen über den Text, der gerade im Editor angezeigt wird, an. Dies sind:

- **Displayed Text** Der Titel des Textes, d.h. der Name eines Moduls, eines Kapitels oder einer Prozedur.
- **File** Die Datei, in der der Text zu finden ist. Durch Anklicken des Dateinamens wird der Text der Datei im Editor angezeigt.
- **Module** Das Modul, zu dem die Prozedur gehört.

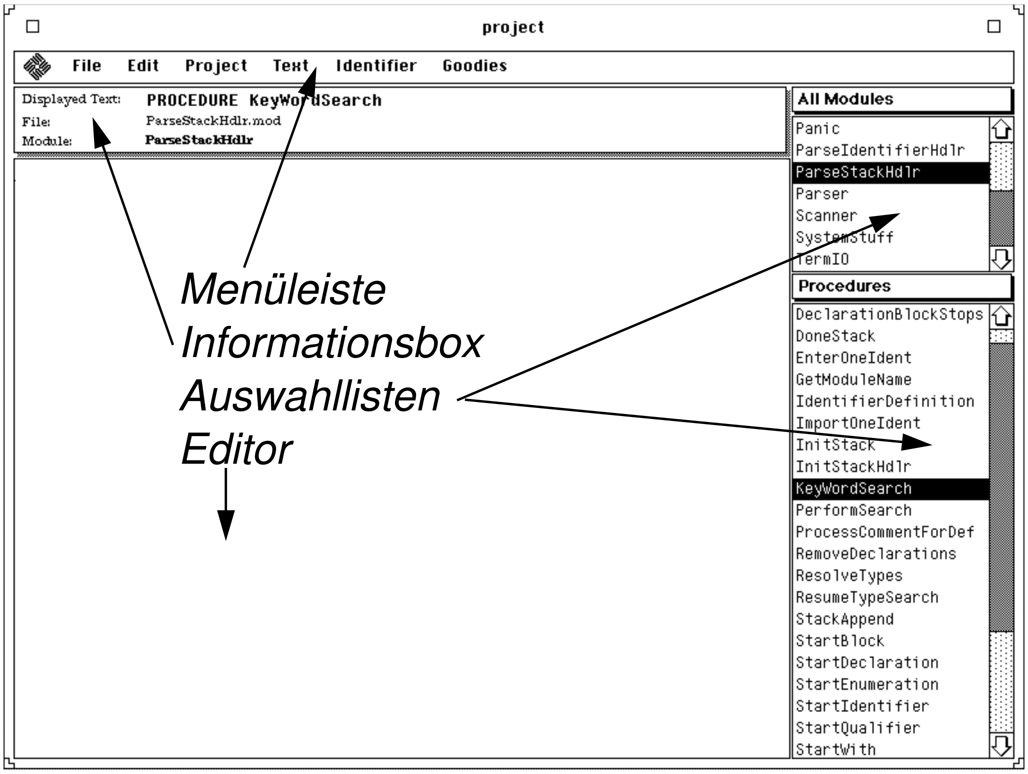

Abbildung 4.4: Hypertextfenster

Alle diese Angaben dienen dazu, dem Benutzer anzuzeigen, wo er sich im Programmsystem gerade befindet. Zusammen mit den Kommandos "History" und "Branch Back" sollen sie erreichen, daß sich besonders der unerfahrene Benutzer nicht im Netz der Hypertext-Knoten verirrt, oder zumindest sicherstellen, daß er sich im Bedarfsfall schnell einen Überblick verschaffen kann.

Auswahllisten Die Auswahllisten zeigen die zur Verfügung stehenden Texte an, die durch einfaches Anklicken ihres Namens in den Editor geladen werden können. Die Texte können dabei nach verschiedenen Kriterien aufbereitet werden. Für eine genaue Beschreibung dieser Listen siehe Abschnitt 4.2.3.

Editor Im Editor wird der Text, der durch Kommandos oder durch die Auswahllisten ausgewählt wurde, angezeigt und kann bearbeitet werden. Es stehen dafür die üblichen Funktionen (ausschneiden, kopieren, einfügen, suchen passender Klammern) zur Verfügung.

Zusätzlich wird in der Titelzeile des Hypertextfensters der Name des Projektmoduls angezeigt – in Abbildung 4.4 ist der Name des Projektmoduls "project". Dadurch können verschiedene Programmsysteme gleichzeitig auf dem Bildschirm dargestellt werden, ohne daß der Benutzer die Übersicht verliert.

4.2.3 Navigation

Die Navigation im Hypertextsystem geschieht hauptsächlich mit Hilfe der beiden Auswahllisten. Daneben dienen noch einige Kommandos in der Menüleiste (z.B. "History", "Branch", "Show Definition" - siehe weiter unten) der Navigation.

Die Auswahllisten stellen die verfügbaren Texte dar und ordnen diese nach verschiedenen Gesichtspunkten. In der oberen Liste werden alle verfügbaren Knoten der entsprechenden Kategorie dargestellt, während die untere Auswahlliste die Knoten anzeigt, die in einer gewissen Beziehung zu dem in der oberen Liste selektierten Knoten stehen.

Wenn der Benutzer einen Eintrag aus einer der beiden Listen auswählt, wird zu dem entsprechenden Knoten verzweigt und der dazugehörige Text angezeigt. Eine Besonderheit stellt dabei das Auswählen einer Prozedur dar: ist eine Prozedur nur definiert, d.h. die Datei, die die Implementierung der Prozedur enthält, wurde nicht geladen, dann wird diese Prozedur in der Auswahlliste kursiv angezeigt. Durch Auswählen eines solchen kursiven Eintrags wird zur Definition der Prozedur verzweigt während ansonsten der Text der Prozedur in den Editor geladen wird.

Die Kategorien werden durch ein PopUp-Menü eingestellt, welches erscheint, wenn in der Titelzeile der Auswahllisten die linke Maustaste gedrückt wird. Die Kategorien der unteren Auswahlliste werden dabei von derjenigen Kategorie bestimmt, die für die obere Auswahlliste gewählt wurde.

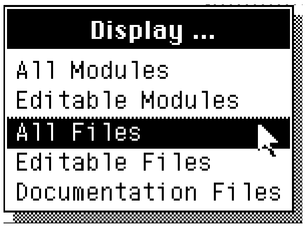

Abbildung 4.5: PopUp-Menü der oberen Auswahlliste

Für die obere Auswahlliste (siehe Abbildung 4.5) stehen folgende Kategorien zur Verfügung:

All Modules zeigt alle Module an, die geladen wurden. Wenn ein Eintrag ausgewählt wird, wird das Definitionsmodul oder, bei einem (Haupt-) Modul, der Modultext angezeigt. Die untere Liste zeigt die Prozeduren an, die in diesem Modul enthalten sind.

Editable Modules Wie "All Modules", allerdings mit dem Unterschied, daß nur Module angezeigt werden, für die auch das entsprechende Implementierungsmodul geladen worden ist. Für den Steuerungsmechanismus, der bestimmt welche Module geladen werden sollen siehe Abschnitt 4.1.

All Files zeigt alle geladenen Dateien an. Die untere Auswahlliste (siehe Abbildung 4.6) bietet die Kategorien "Imported Modules" und "Substitutes" an. "Imported Modules" sind alle Dateien, die von der gerade betrachteten Datei importiert werden. Als "Substitutes" werden solche Texte bezeichnet, die während des Einlesevorganges aus dem gerade angezeigten Text herausgeschnitten wurden. Diese herausgeschnittenen Texte werden als eigenständige Texte behandelt und können durch Auswahl ihres Namens in der Auswahlliste angezeigt werden. Der Quellcode von Prozedurimplementierungen wird beispielsweise als ein "Substitute" behandelt, ebenso wie Dokumentationskapitel. Der Benutzer kann diesen Mechanismus auch auf beliebige Textstücke anwenden: der ausgeschnittene Text wird mit einem eigenen Namen versehen, der an der Stelle angezeigt wird, an der der Text ursprünglich stand. In DOgMA werden solche Texte als "kollabierte Texte" bezeichnet.

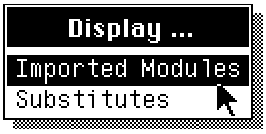

Abbildung 4.6: PopUp-Menü der unteren Auswahlliste

Editable Files zeigt alle Dateien an, die verändert werden können. In DOgMA können nur die Dateien editiert werden, die durch eine Anweisung des Projektmoduls (vergleiche Abschnitt 4.1) eingelesen wurden. Dateien, die nur deshalb geladen wurden, weil eine andere Datei diese benutzt, können nur angezeigt werden. Durch diesen Mechanismus wird sichergestellt, daß der Benutzer nur die Dateien seines Arbeitsbereiches ändern kann.

Documentation Files zeigt die Dokumentationsdateien an.

4.2.4 Menüleiste

Die Menüleiste mit ihren PopUp-Menüs wird dazu benutzt, die von DOgMA angebotenen Kommandos zu aktivieren. Die Kommandos sind in folgende sechs Gruppen eingeteilt:

File Dieser Menüpunkt bietet Kommandos zum Laden und Sichern einzelner Dateien sowie einige Kommandos zur Manipulation des Applikationsfensters.

Edit Das Editmenü bietet die Kommandos für das Editieren von Texten (ausschneiden, einfügen, kopieren), Kommandos zum Einfügen und Löschen von Informationsknoten, sowie das möglicherweise wichtigste Kommando: das Kommando zum Rückgängigmachen des vorhergehenden Kommandos.

Project Die Kommandos dieses Menüpunkts beziehen sich jeweils auf das gesamte Projekt. Es werden Kommandos zum Sichern und Analysieren des Projektes sowie zum Anpassen der Darstellungsweise im Editor angeboten.

Text Die Kommandos dieses Menüpunktes dienen zum Navigieren im Netz der Hypertextknoten und zum Manipulieren von Textstücken.

Identifier Hier sind die Kommandos zusammengefaßt, die sich mit der Analyse und Darstellung einzelner Bezeichner eines Programmsystems beschäftigen.

Goodies Dieser Menüpunkt bietet spezielle Kommandos zur Steuerung der Darstellung des Textes im Editor.

Unter dem sich in der Menüleiste ganz links befindlichem Sun-Symbol verbirgt sich ein Menüpunkt, mit dessen Hilfe die About-Meldung (siehe Abschnitt 4.2.1 und Abbildung 4.2) angezeigt werden kann.

In manchen Abbildungen von Menüs sind gewisse Kommandos grau dargestellt. Dies bedeutet, daß das Bild zu einem Zeitpunkt gemacht wurde, an dem das entsprechende Kommando nicht ausgeführt werden konnte. Immer wenn ein Kommando nicht ausgeführt werden kann wird dieser Punkt im Menü automatisch grau dargestellt.

4.2.4.1 File

Die Kommandos dieses Menüs (siehe Abbildung 4.7) dienen zum Laden und Sichern von Dateien, zum Schließen und Duplizieren des Hypertextfensters sowie zum Zeigen des Applikationsfensters.

Load File ... [1] Dieser Menüpunkt dient dazu, einzelne Dateien zusätzlich zu den bereits geladenen Dateien einzulesen. Diese Datei wird grundsätzlich gleich behandelt wie eine Datei, die mittels des Projektmoduls

[1] Drei Punkte "..." nach einem Menüpunkt bedeuten immer, daß der Befehl nicht unmittelbar ausgeführt wird, sondern daß der Benutzer zuerst noch gewisse Daten eingeben muß – in diesem Fall den Namen der Datei.

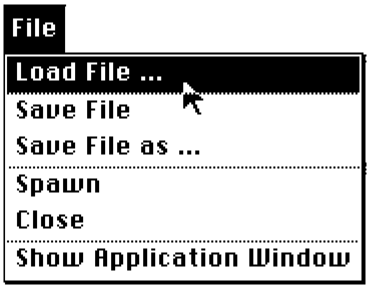

Abbildung 4.7: Menü "File"

eingelesen wird, es werden also eventuell von dieser Datei importierte Dateien ebenfalls importiert und die Datei mit den Prozeduren, so vorhanden, in das Hypertextsystem eingebunden.

Das Kommando "Load File" ist dann sinnvoll, wenn Dateien bearbeitet werden sollen, die nur bei dieser speziellen Arbeitssitzung von Interesse sind.

Save File Durch dieses Kommando wird die Datei auf die Festplatte gesichert, zu der der angezeigte Hypertext gehört – egal ob es sich bei dem Text um eine Prozedur oder um den Dateitext selbst handelt.

Immer wenn während der Arbeit eine gewisse Anzahl von Änderungen durchgeführt worden sind ist es ratsam, diese durch Abspeichern der Datei zu sichern. Dazu kann dieses Kommando verwendet werden.

Save File as ... Funktioniert analog zu "Save File", allerdings mit der Ausnahme, daß hier die Datei auf der Festplatte nicht überschrieben, sondern eine neue Datei angelegt wird, deren Name der Benutzer in dem Datei-Dialog (siehe Abbildung 4.3) eingibt.

Spawn Das Kommando "Spawn" öffnet ein weiteres Hypertextfenster, welches auf dieselbe Datenbasis zugreift. Dadurch wird es möglich, verschiedene Stellen des Programmsystems nebeneinander zu untersuchen und zu bearbeiten. Dies erlaubt es auch, Quellcode und Dokumentation parallel zu betrachten. Es können beliebig viele Hypertextfenster für eine Datenbasis gleichzeitig geöffnet werden.

Close Durch Auswahl dieses Menüpunktes kann das Hypertextfenster geschlossen werden. Sollte es sich um das letzte Hypertextfenster für eine bestimmte Datenbasis handeln und sollten seit der letzten Sicherung Änderungen vorgenommen worden sein, wird der Benutzer gefragt, ob diese Änderungen gesichert werden sollen.

Show Application Window Dieser Menüpunkt zeigt das zu diesem Hypertextfenster gehörige Applikationsfenster an. Das Fenster, welches unter mehreren Schichten sich überlappender Fenster verborgen sein kann, wird benötigt, um Dateien zu editieren, weitere Projekte einzulesen oder die Applikation zu verlassen.

4.2.4.2 Edit

Das Editmenü (siehe Abbildung 4.8) bietet Kommandos zum Editieren, sowie zum Einfügen und Löschen von Informationsknoten.

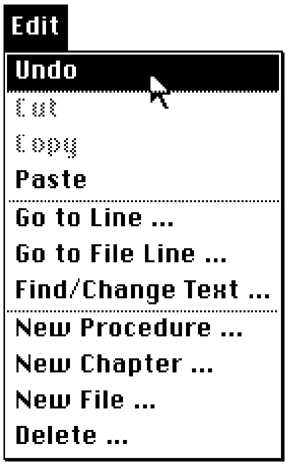

Abbildung 4.8: Menü "Edit"

Undo Dieses Kommando macht den Effekt des unmittelbar zuvor ausgeführten Kommandos rückgängig. Wenn es ein zweites Mal aufgerufen wird, dann macht es sich selbst rückgängig, d.h. das zuvor rückgängig gemachte Kommando wird wieder ausgeführt.

Dieses Kommando ist besonders in der Eingewöhnungsphase von großer Wichtigkeit, da es dem Benutzer erlaubt, das Werkzeug durch Probieren zu erkunden, ohne negative Effekte für seine Arbeit fürchten zu müssen. Auch später können Fehler einfach ausgemerzt werden, was dem Benutzer ein höheres Gefühl an Sicherheit vermittelt.

Cut Löscht den markierten Text aus dem Editor und legt diesen in der Zwischenablage[2] ab, sodaß er später mit "Paste" an einer anderen Stelle wieder eingefügt werden kann.

Copy Kopiert den im Editor markierten Text in die Zwischenablage, wo er für ein "Paste" zur Verfügung steht.

Paste Fügt den Text, der sich in der Zwischenablage befindet, an der durch die Einfügemarke bestimmten Stelle im Editor ein.

Go to Line ... Positioniert die Einfügemarke in der Zeile, die im Dialog (siehe Abbildung 4.9) eingegeben wurde. Die Nummer der Zeile bezieht sich dabei auf den Text, der im Editor angezeigt wird.

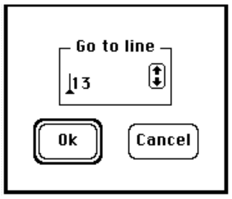

Abbildung 4.9: Dialog "Go to Line"

Go to File Line ... Dieses Kommando gleicht "Go to Line" mit dem einzigen Unterschied, daß sich die Zeilenangabe nicht auf den Text bezieht, der gerade angezeigt wird, sondern auf die Datei, zu der der Text gehört.

Wenn der Quellcode durch den Compiler übersetzt wird, erzeugt dieser im Fall, daß ein Übersetzungsfehler auftritt, eine Fehlermeldung. Diese Fehlermeldung besteht aus der Zeilennummer, in der der Fehler auftrat

[2] Die Zwischenablage ist ein unsichtbarer Puffer, der für die Standard-Editierfunktionen – cut, copy und paste – benutzt wird.

und aus einer Nachricht, die auf die Art des Fehlers und seine möglichen Ursachen hindeutet. Durch "Go to File Line" kann nun unmittelbar auf die in der Fehlermeldung angegebene Zeilennummer verzweigt werden.

Find/Change Text ... Dieses Kommando bietet die übliche Funktionalität, um nach Zeichenketten im Text zu suchen und diese eventuell gegen andere Zeichenketten austauschen zu können.

New Procedure ... Dieser Menüpunkt erlaubt es, eine neue Prozedur in das Hypertextsystem aufzunehmen. In einem Dialog gibt der Benutzer den Namen der zu erstellenden Prozedur an sowie das Modul, welches die Prozedur enthalten soll. Danach werden die Daten in das allgemeine Datensystem eingefügt und der Prozedurtext angezeigt. Wenn eine Prozedur neu erstellt wird, wird sie mit einem Standardtext versehen, der über Parameter einstellbar ist (siehe Abschnitt 4.3).

New Chapter ... Der Benutzer gibt an, wie das einzufügende Kapitel heißen soll, welches Kapitel ihm übergeordnet ist und in welche Datei das Kapitel geschrieben werden soll.

Abbildung 4.10 zeigt stellvertretend für alle "New"-Kommandos die Dialogbox, die für die Eingabe des Names verwendet wird.

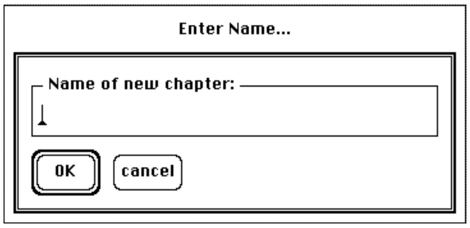

Abbildung 4.10: Dialog "New Chapter"

New File ... Dieser Menüpunkt dient sowohl zum Anlegen von Dokumentations- als auch von Programmdateien. Abhängig von der Erweiterung[3] wird der zu erzeugende Dateityp – Modul oder Dokumentationsdatei –

[3]Als Erweiterung bezeichnet man den Teil des Dateinamens, der nach dem Punkt folgt. Unter Unix ist die Erweiterung kein notwendiger Bestandteil des Dateinamens – wie zum Beispiel unter MS-DOS – wird aber zur Kennzeichnung von Dateien verwendet.

festgelegt und die so erzeugte Datei – mit einem Standardtext versehen – in das Datensystem eingefügt.

Delete ... Löscht beliebige Knoten (Prozeduren, Module, Dateien, Dokumentationskapitel) aus dem Datensystem.

4.2.4.3 Project

Die Kommandos dieses Menüs (siehe auch Abbildung 4.11) beziehen sich auf das gesamte Projekt. Es werden Kommandos zum Sichern und Analysieren des Projektes sowie zum Anpassen der Darstellungsweise im Editor angeboten.

Abbildung 4.11: Menü "Project"

Show Project File Zeigt das Projektmodul im Editor an.

Read Project Der "Project File", in diesem Fall das Projektmodul (siehe auch Abschnitt 4.1) enthält eine Liste alleri Dateien, die vom Werkzeug eingelesen werden sollen. Die angeführten Dateien werden automatisch geladen und verarbeitet, wenn ein Projekt geöffnet wird (vergleiche Abschnitt 4.2.1). Wenn aber nach dem Einlesen weitere Dateien in den Projektmodul eingefügt werden, dann kann das gesamte System mit Hilfe dieses Kommandos neu eingelesen werden.

Die Informationen über das Programmsystem nach einem "Read Project" reichen aus, damit dieses als Hypertext-Netz, bestehend aus Prozeduren, Modulen und Dokumentationskapiteln für die Naviation erzeugt werden kann. Danach kann sich der Benutzer mit Hilfe der Auswahllisten durch das Programmsystem bewegen und die Texte editieren.

Wenn noch zusätzliche Informationen über das Programmsystem benötigt werden, muß das Kommando "Parse Project" ausgeführt werden. Nach Ausführung von "Parse Project" stehen Informationen über jeden einzelnen Bezeichner des Programmsystemes zur Verfügung (siehe auch Abschnitt 4.2.4.5).

Parse Project Nachdem ein Projekt eingelesen wurde, können durch Aufruf dieses Kommandos weitere Informationen über die Bezeichner des Quellcodes generiert werden. Diese Informationen werden für die Ausführung der Kommandos des Abschnittes 4.2.4.5 benötigt.

Der Parse-Prozeß benötigt einige Zeit und die von ihm erzeugten Informationen werden gewöhnlich nur in der Einarbeitungsphase benutzt. Deshalb wurde dem Benutzer die Möglichkeit gegeben, selbst zu entscheiden, ob er diese zusätzlichen Informationen benötigt oder nicht.

Save Project Wenn einige Änderungen durchgeführt worden sind, ist es manchmal schwierig festzustellen, in welchen Dateien diese Änderungen vorgenommen wurden, besonders da der Benutzer von DOgMA weniger Dateien, als vielmehr Module und Prozeduren bearbeitet. Wenn "Save Project" ausgewählt wird, wird der Benutzer bei jeder Datei, die seit der letzten Sicherung verändert wurde, gefragt, ob diese Datei gesichert werden soll. Dadurch muß sich der Benutzer nicht darum kümmern, in welcher Datei er Änderungen vorgenommen hat und kann trotzdem sicher sein, daß alle Änderungen korrekt abgespeichert werden.

Close Project Dieser Menüpunkt führt zuerst ein "Save Project" aus, es werden also alle geänderten Dateien gespeichert, und schließt dann alle Hypertextfenster, die zu dem betreffenden Projekt gehören.

Define Styles ... Dieses Kommando zeigt einen Dialog (siehe Abbildung 4.12), der es erlaubt, den Stil der Textanzeige im Editor nach eigenen Vorstellungen zu beeinflussen. Durch die unterschiedliche Darstellung von z.B. Schlüsselwörtern und Bezeichnern kann sich der Benutzer noch schneller einen Überblick über den Programmtext verschaffen.

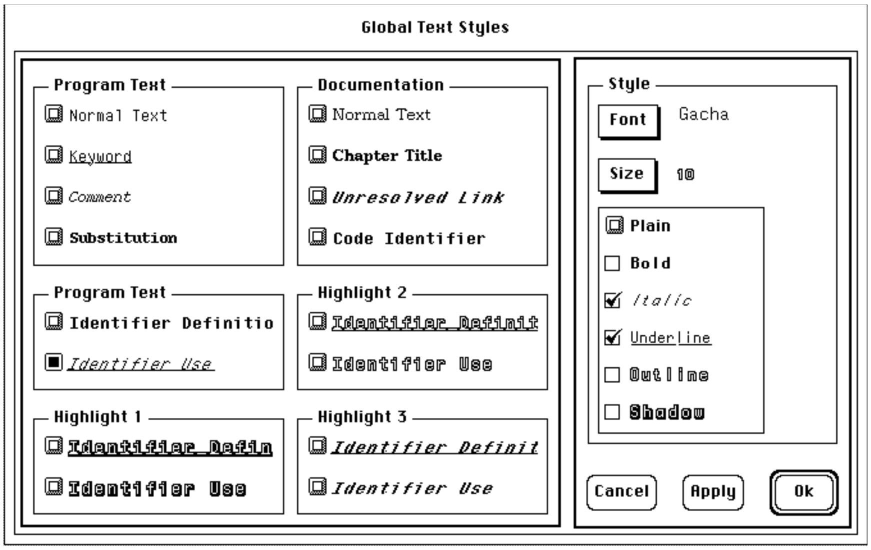

Abbildung 4.12: Dialog "Stile"

Für folgende Textarten kann jeweils einzeln die Darstellungsform gewählt werden:

- Schlüsselwörter, Kommentare, Ersetzungen, Bezeichnerdefinitionen und Bezeichnerverwendungen.
- Kapiteltitel, Bezeichner im Dokumentationstext und Verweise auf Quellcode in der Dokumentation, für die der Quellcode nicht geladen wurde.
- Drei Stile, mit denen für jeweils einen Bezeichner dessen Deklaration und Verwendung speziell hervorgehoben werden kann (siehe auch die Kommandos "Use Highlight 1", "Use Highlight 2" und "Use Highlight 3" im Menü Goodies).

Abbildung 4.13 zeigt an einem Stück Quellcode die Auswirkungen der in Abbildung 4.12 gewählten Stilparameter. Hier wurden die Schlüsselwörter unterstrichen, Bezeichnerdefinitionen fett und Bezeichnerverwendungen kursiv und unterstrichen dargestellt. Alle übrigen Abbildungen dieser Arbeit zeigen die Standard-Stileinstellungen. Leider be-

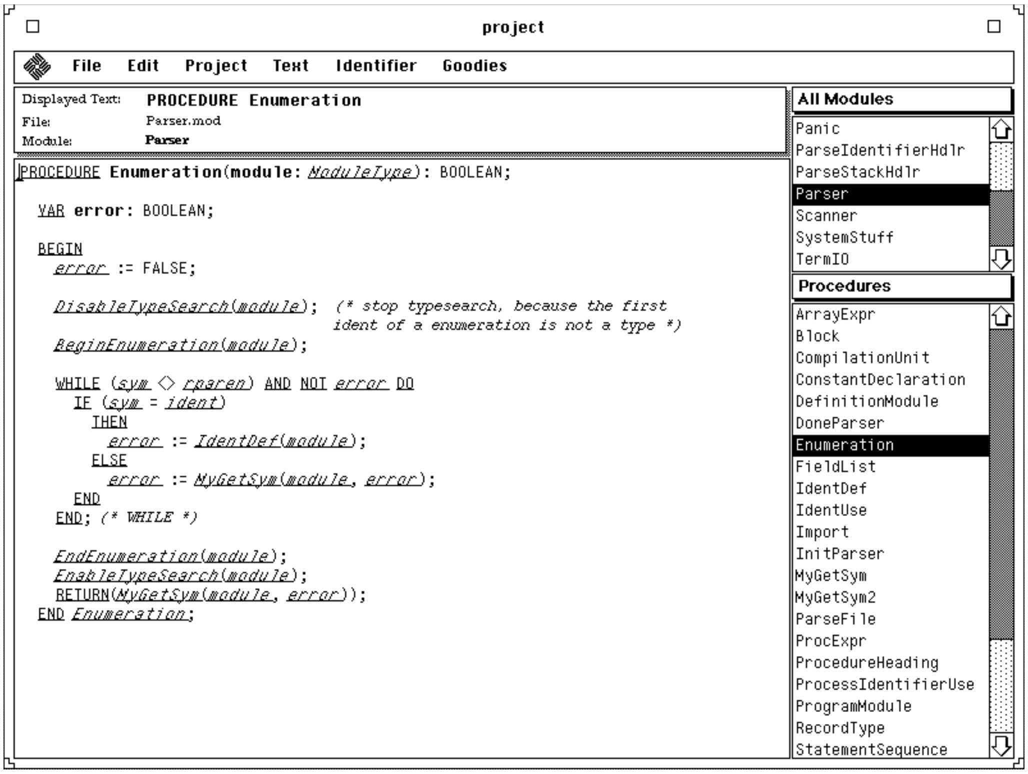

Abbildung 4.13: Angewendeter Stil

steht nicht die Möglichkeit, eine solche Einstellung in einer Datei abzuspeichern, sodaß bei jedem Laden des Programmsystems die Stile neu definiert werden müssen.

History ... Besonders in der Einarbeitungsphase geschieht es manchmal, daß der Benutzer die Übersicht über den Weg, den er durch das System genommen hat, verliert. Deshalb sind Funktionen, die es dem Benutzer erlauben, den bisher zurückgelegten Weg nachzuvollziehen von großer Wichtigkeit (vergleiche [Nie90]). Diese Funktionalität wird in DOgMA durch die Kommandos "Branch Back" und "History" implementiert.

Das Kommando "History" zeigt in einer Dialogbox (siehe Abbildung 4.14) die Knoten, d.h. die Module, Prozeduren oder Kapitel an, die der Benutzer bisher bearbeitet hat.

Durch Auswählen eines Eintrags der Liste mit der Maus oder der Tastatur wird der entsprechende Text im Editor angezeigt.

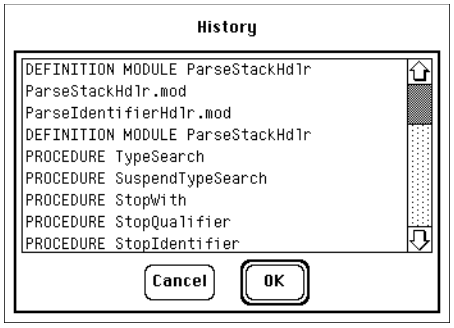

Abbildung 4.14: Dialog "History"

Diese Funktion ist auch dann nützlich, wenn verschiedene Stellen im Quellcode immer wieder bearbeitet werden sollen, da sie ein direktes Verzweigen zu einem anderen Informationsknoten erlaubt.

4.2.4.4 Text

Die Kommandos dieses Menüpunktes (siehe Abbildung 4.15) dienen zum Navigieren im Netz der Hypertextknoten und zum Manipulieren von Textstücken.

Abbildung 4.15: Menü "Text"

Branch Wenn die Programmdateien eingelesen werden, wird der Text einer Prozedur in einen eigenen Hypertext geladen und die Stelle, an der sich die Prozedur in der Datei befand, wird durch ein "Ersetzungszeichen" – den fett dargestellten Prozedurkopf – gekennzeichnet. Wenn ein Ersetzungszeichen mit der Maus ausgewählt wird, kann durch dieses Kommando zu dem Prozedurtext verzweigt werden. Die gleiche Möglichkeit besteht auch bei Import-Anweisungen, die ebenfalls als Ersetzungszeichen im Text dargestellt werden.

Branch Back Dieses Kommando erlaubt es, den unmittelbar vorher bearbeiteten Text wieder in den Editor zu laden. "Branch Back" kann mehrmals aufgerufen werden, sodaß die bereits untersuchten Knoten nochmals betrachtet werden können.

Copy Text Zusammen mit "Paste Text" ermöglicht es dieses Kommando, Teile des Quellcodes in die Dokumentation einzufügen. Auf diese Weise können allerdings nur ganze Texte, also Module, Prozeduren oder kollabierte Texte (siehe Kommando "Collapse Text") eingebunden werden.

Paste Text Fügt den mit "Copy Text" ausgewählten Text an der Stelle des Einfügungszeichens in die Dokumentation ein. Die notwendigen Hypertext-Querverweise werden automatisch erzeugt.

Ein Beispiel für eine mit diesen Kommandos erstellte Dokumentation zeigt Abbildung 4.16, die den Text der Prozedur "BeginModule" und mehrere Bezeichner, die fett dargestellt sind, im Dokumentationstext enthält.

Collapse Text Durch dieses Kommando können längere Texte oder Codestücke strukturiert werden. Zuerst wird ein Stück Text ausgewählt und dann dieses Kommando aktiviert. Für den ausgewählten Text wird ein eigener Hypertextknoten mit den entsprechenden Querverweisen angelegt. An der Stelle, an der der kollabierte Text ursprünglich stand, befindet sich nun der fett dargestellte Texttitel.

Highlight Locals Dieser Menüpunkt erlaubt es, die lokalen Bezeichner des angezeigten Programmtextes in besonderer Weise hervorzuheben. Ein Bezeichner ist relativ zum angezeigten Text dann lokal, wenn er in diesem Text deklariert worden ist. So ist zum Beispiel der Name einer Prozedur ein lokaler Bezeichner in ihrem Modul und ebenfalls eine in einer Prozedur deklarierte Variable für diese Prozedur. Durch das

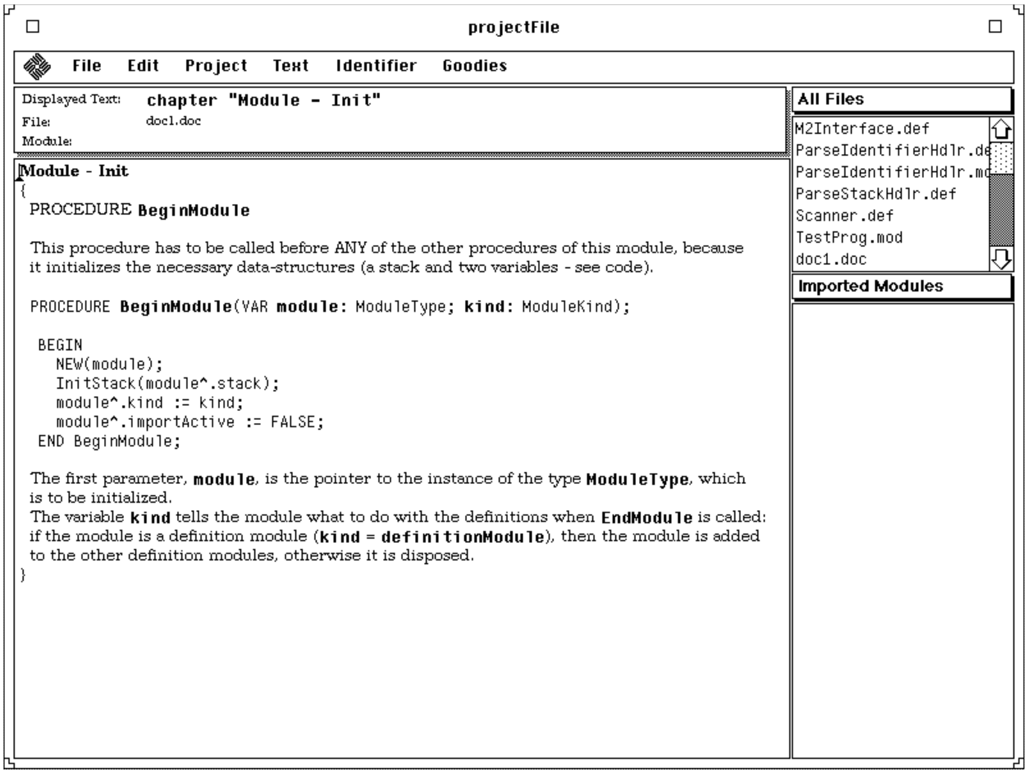

Abbildung 4.16: Dokumentationskapitel

Hervorheben dieser Bezeichner können diese von anderen, globalen Bezeichnern auf einen Blick unterschieden werden. Ein Beispiel der Anwendung dieses Kommandos zeigt Abbildung 4.17, wo drei lokale Bezeichner – TypeDeclaration, module und error – hervorgehoben sind. Dieses Kommando kann nur dann ausgeführt werden, wenn das Projekt geparst wurde.

Eine weitere interessante Information ist die, an welchen anderen Stellen im Programmtext bestimmte Bezeichner verwendet werden. Deshalb wird in den Auswahllisten die Anzahl der hervorgehobenen lokalen Bezeichner, die im jeweiligen Text verwendet werden, neben dem Namen des Textes angezeigt. Für Prozeduren wird einfach die Anzahl der Verwendungen angezeigt, für Module wird die Anzahl der Verwendungen im Modulkörper und die Summe der Verwendungen in den Prozeduren des Modules angezeigt. Dieses Kommando kann nur dann ausgeführt werden, wenn das Projekt geparst wurde.

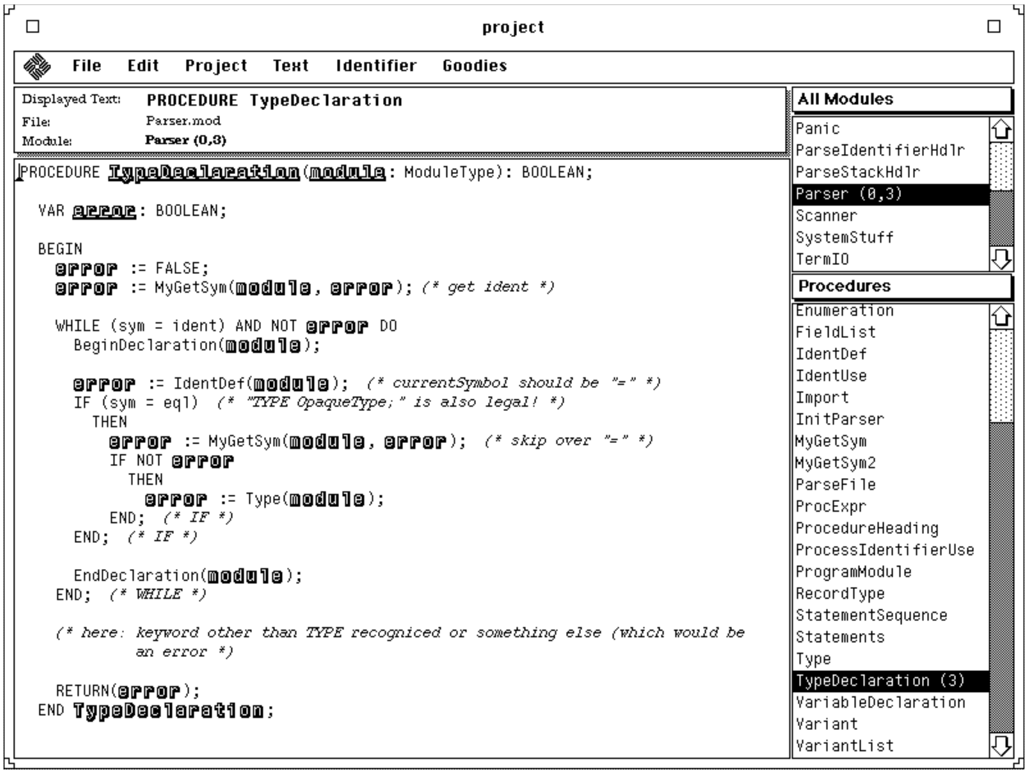

Abbildung 4.17: Hervorgehobene lokale Bezeichner

Lowlight Locals Macht den Effekt von "Highlight Locals" rückgängig.

Show Documentation Wenn dieses Kommando im Menü nicht grau (wie in Abbildung 4.15), sondern fett dargestellt ist, bedeutet dies, daß für den im Editor angezeigten Text eine entsprechende Dokumentation existiert. Durch Auswählen des Kommandos kann zu dieser Stelle in der Dokumentation verzweigt werden.

Show Code Wenn im Editor der Text eines Dokumentationskapitels angezeigt wird, dann wird im Menü anstelle des Kommandos "Show Documentation" das Kommando "Show Code" angeboten, welches die analoge Funktion zu "Show Documentation", allerdings in die Gegenrichtung, also von Dokumentation zu Code, anbietet.

4.2.4.5 Identifier

Hier sind die Kommandos zusammengefaßt, die sich mit der Analyse und der Darstellung einzelner Bezeichner eines Programmsystems beschäftigen. Alle Kommandos dieses Menüs (siehe Abbildung 4.18) sind nur nach dem Parsen des Programmsystems mit "Parse Project" verfügbar.

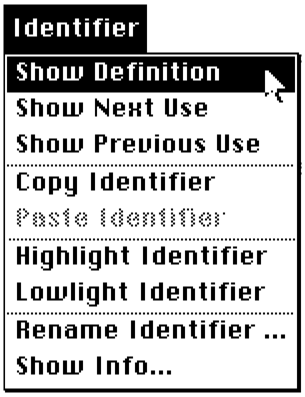

Abbildung 4.18: Menü "Identifier"

Alle Kommandos funktionieren sowohl im Quellcode als auch in der Dokumentation, sodaß ein konsistentes Arbeiten ermöglicht wird. Außerdem funktionieren sie sowohl für Typen- und Variablennamen als auch für Prozedur- und Modulnamen.

Show Definition Die Definition eines Bezeichners und oft auch deren näheres Umfeld sind für das Verständnis des Quellcodes von großer Wichtigkeit. Dieses Kommando ermöglicht es, zu der Stelle der Definition des Bezeichners, der gerade selektiert ist, zu verzweigen.

Show Next Use Verzweigt zu der Stelle im Quellcode, an der der selektierte Bezeichner das nächste Mal verwendet wird.

Show Previous Use Verzweigt zu der Stelle im Quellcode, an der der selektierte Bezeichner zuletzt verwendet wurde.

Copy Identifier Dieses Kommando wird zusammen mit "Paste Identifier" dazu verwendet, einen einzelnen Bezeichner in die Dokumentation einzufügen. Die nötigen Hypertext-Querverweise werden automatisch erzeugt.

Paste Identifier Fügt den mit "Copy Identifier" selektierten Bezeichner an der Stelle, an der sich die Einfügemarke befindet, in die Dokumentation ein.

Highlight Identifier Die Funktionsweise dieses Kommandos ist vergleichbar mit der des Kommandos "Highlight Locals", mit dem Unterschied, daß hier nur der selektierte Bezeichner besonders hervorgehoben wird. Ebenso wie bei "Highlight Locals" wird auch hier die Anzahl der Verwendungen des Bezeichners in den Auswahllisten dargestellt.

Lowlight Identifier macht den Effekt von "Highlight Identifier" wieder rückgängig.

Rename Identifier ... Bei der herkömmlichen Arbeitsweise mit dateiorientierten Editoren ist das Ändern des Namens eines Bezeichners schwierig, besonders wenn unterschiedliche Bezeichner den gleichen Namen haben. Dann muß nach jedem Auftreten des Namens gesucht werden und dann untersucht werden, ob es sich bei diesem Auftreten des Namens um den gesuchten Bezeichner handelt. Es muß nicht betont werden, daß dies besonders bei großen Systemen eine gefährliche Fehlerquelle darstellt.

DOgMA bietet die Möglichkeit, gezielt den Namen eines Bezeichners zu ändern, wobei garantiert wird, daß nur dieser Bezeichner und nicht auch eventuelle "Namensvettern" abgeändert werden. Wenn der Bezeichner in der Dokumentation verwendet wird, wird er dort ebenfalls geändert. Dieses Kommando kann auch auf Bezeichner in der Dokumentation angewendet werden.

Abbildung 4.19 zeigt die Dialogbox, in der der neue Name des Bezeichners angegeben wird.

Show Info ... Dieser Menüpunkt zeigt ein Dialogfenster mit einigen Angaben über einen ausgewählten Bezeichner. Ein Beispiel dafür zeigt Abbildung 4.20, wobei unterhalb des Dialogfensters der ausgewählte Bezeichner – in diesem Fall "definitionMark" – zu sehen ist.

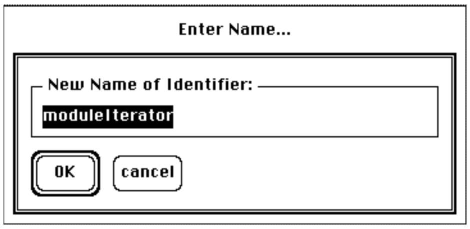

Abbildung 4.19: Dialog "Rename Identifier"

Neben dem Namen des Bezeichners ist auch die Zeile, in der der Bezeichner deklariert worden ist, zu sehen, womit in den meisten Fällen auch die Information über den Typ des Bezeichners vorhanden ist. Weiters kann man ablesen, in welcher Datei die Deklaration zu finden ist. Die letzte Information ist ein Kommentar, falls sich unmittelbar nach dem Bezeichner einer befindet. Dabei wird unterstellt, daß ein Kommentar, der sich hinter einem Bezeichner befindet, eine kurze Beschreibung dieses Bezeichners enthält.

Alle diese Informationen tragen dazu bei, sich schnell einen Überblick über die wichtigsten Daten, die einen bestimmten Bezeichner betreffen, verschaffen zu können.

4.2.4.6 Goodies

Dieser Menüpunkt (siehe Abbildung 4.21) bietet spezielle Kommandos zur Steuerung der Darstellung des Textes im Editor.

Use Highlight 1

Use Highlight 2

Use Highlight 3 Diese 3 Kommandos, von denen immer nur eines ausgewählt sein kann, bestimmen den Anzeigestil, der beim nächsten Aufruf von "Highlight Identifier" verwendet wird. Es besteht die Möglichkeit, diesen Stil mit dem "Define Styles"-Dialog den eigenen Vorstellungen anzupassen.

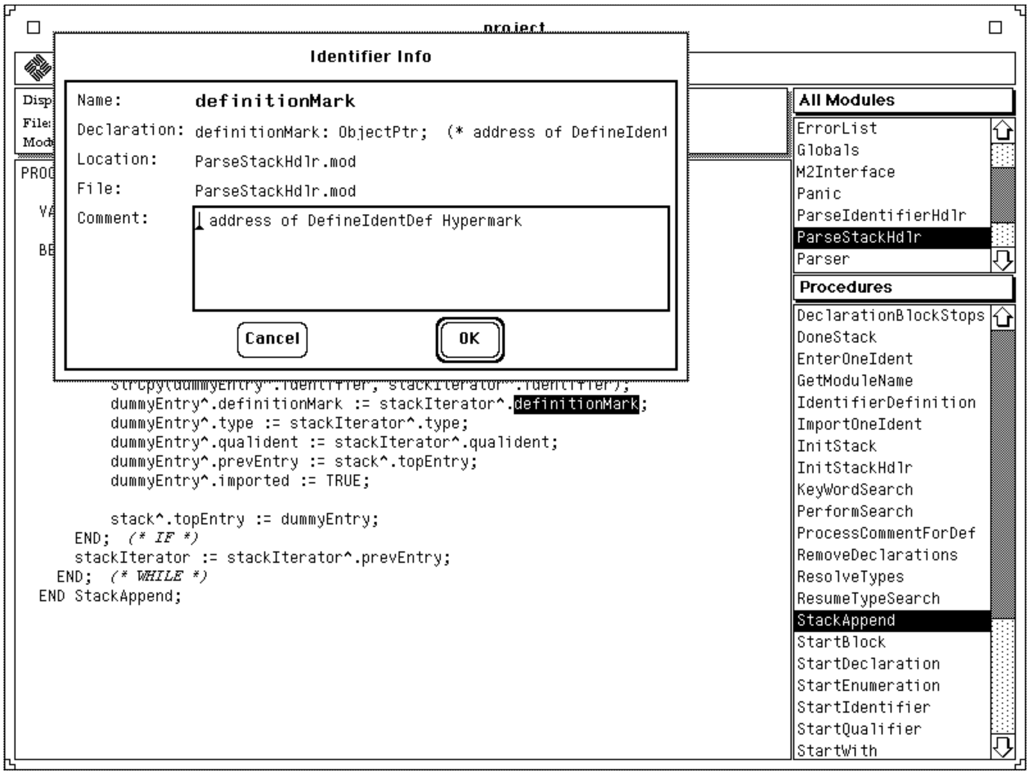

Abbildung 4.20: Bezeichnerdefinition

Show Comments Macht den Effekt von "Hide Long Comments" und "Hide All Comments" rückgängig.

Hide Long Comments Manchmal ist es für das Verständnis eines des Quellcodes nützlich, die Kommentare ausblenden zu können, um sich auf den Programmtext zu konzentrieren. Mit diesem Kommando können alle *langen* Kommentare ausgeblendet werden. Ein Kommentar ist in diesem Zusammenhang dann lang, wenn er sich über mehr als eine Zeile erstreckt. An der Stelle der Kommentare wird nur mehr "(*Comment*)" angezeigt.

Hide All Comments Dieses Kommando verbirgt alle Kommentare, also auch die, die von "Hide Long Comments" nicht verborgen werden. Den Effekt dieses Kommandos zeigt Abbildung 4.22.

Abbildung 4.21: Menü "Goodies"

4.2.5 PopUp-Menü

Wenn sich der Mauszeiger über dem Editor befindet und die rechte Maustaste gedrückt wird erscheint ein PopUp-Menü (siehe Abbildung 4.23).

PopUp-Menüs sind z.B. in den Benutzeroberflächen SunView, wo es keine anderen Menüs gibt, und auch in etwas geringerem Ausmaß in OpenWindows zu finden. In DOgMA werden in diesem Menü häufig benutzte Kommandos angeboten, die auf diese Weise sehr schnell ausgewählt werden können.

Alle Kommandos dieses Menüs sind auch in der Menüleiste zu finden. Eine detailliertere Beschreibung der einzelnen Kommandos ist in der Beschreibung der Menüleiste (siehe Abschnitt 4.2.4) zu finden.

undo macht den Effekt des letzten Kommandos rückgängig. Dieses Kommando ist in der Menüleiste in der Gruppe "Edit" zu finden.

cut löscht den ausgewählten Text in die Zwischenablage. Gruppe "Edit".

copy kopiert den ausgewählten Text in die Zwischenablage. Gruppe "Edit".

paste kopiert den Text von der Zwischenablage an die Stelle, an der sich die Einfügemarke befindet. Gruppe "Edit".

select all selektiert den gesamten Text im Editor.

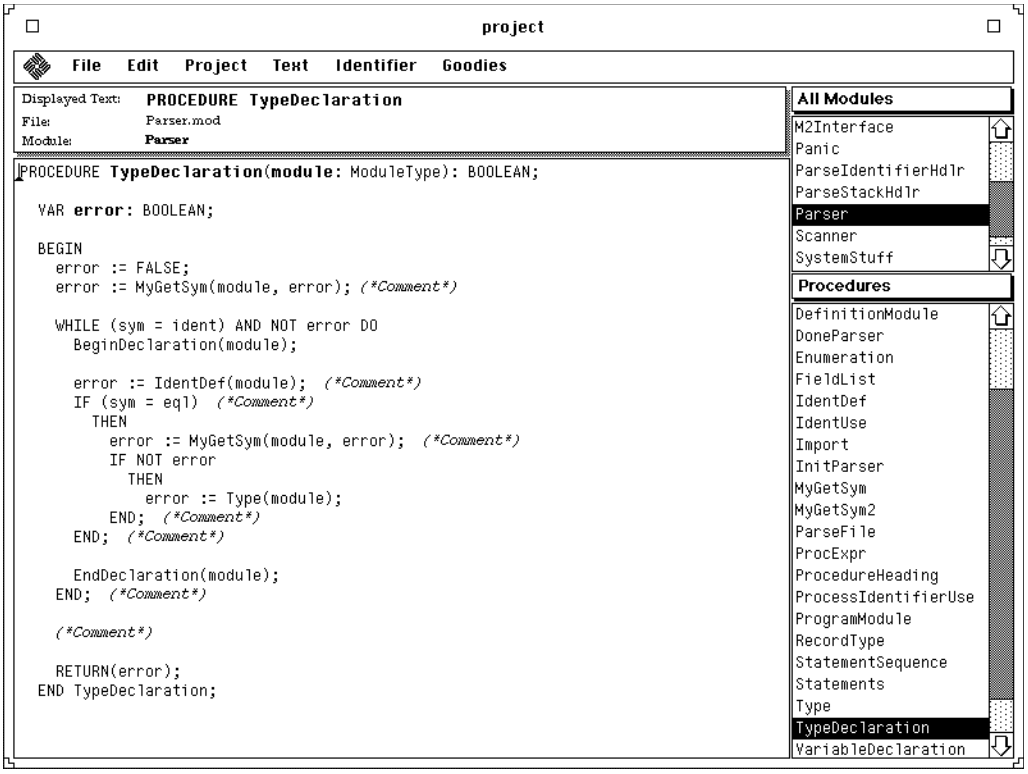

Abbildung 4.22: Ausgeblendete Kommentare

find/change ... zeigt einen Dialog zum Suchen und Ersetzen von Zeichenketten. Gruppe "Edit".

save file speichert die Datei, zu der der im Editor angezeigte Text gehört, unter ihrem Namen ab, sodaß die Datei auf der Festplatte überschrieben wird. Gruppe "File".

save file as ... speichert die Datei, zu der der im Editor angezeigte Text gehört, unter einem neuen Namen ab. Gruppe "File".

save project sichert alle Dateien des Projektes, die seit der letzten Sicherung verändert worden sind. Gruppe "Project".

Abbildung 4.23: PopUp-Menü

4.2.6 Dateieditor

Neben dem oben beschriebenen Hypertexteditor bietet DOgMA zusätzlich noch einen Dateieditor an, mit dessen Hilfe Dateien auf herkömmliche Weise bearbeitet werden können. Abbildung 4.24 zeigt neben dem Editor auch einen seiner Verwendungszwecke: die Erstellung der make-Datei. Diese dient zusammen mit anderen Dateien dazu, den Prozeß der Übersetzung eines Programmes zu automatisieren (für nähere Erläuterungen siehe Abschnitt 5.4.3).

Es ist insoferne wichtig, einen solchen Editor zur Verfügung zu stellen, als im Zuge der Arbeit mit einem Programmsystem auch Dateien bearbeitet werden müssen, die nicht Programmdateien im Sinne von DOgMA sind. Andererseits kann es auch notwendig sein, Programmdateien mit einem konventionellen Editor zu bearbeiten, zum Beispiel um diese für einen Ausdruck vorzubereiten. Natürlich kann für diese Aufgaben auf eine Vielzahl verschiedener Editoren zurückgegriffen werden. Unglücklicherweise existiert aber kein Editor, dessen Benutzeroberfläche zu der von DOgMA kompatibel ist, was bedeutet, daß mit mehreren verschiedenen Benutzeroberflächen gleichzeitig gearbeitet werden müßte.

Um diesem Mißstand abzuhelfen, wurde DOgMA mit diesem Dateieditor ausgestattet, dessen Benutzeroberfläche sich am Erscheinungsbild des Hypertextfensters orientiert. Der Editor wird vom Applikationsfenster (siehe

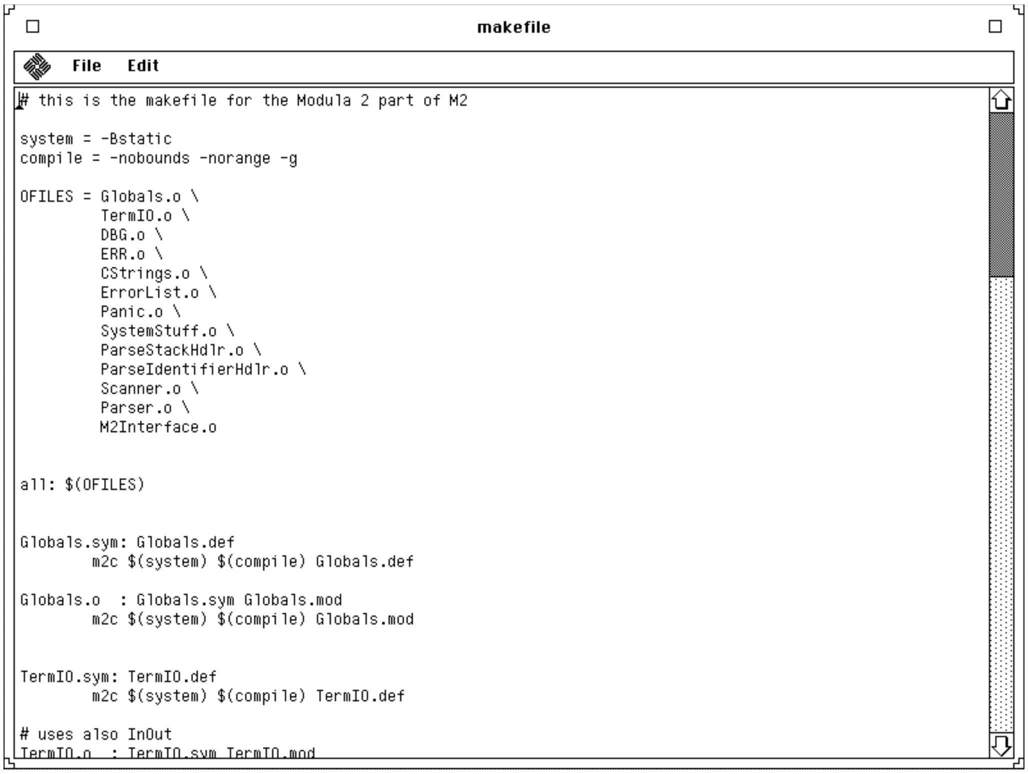

Abbildung 4.24: Dateieditor

Abschnitt 4.2.1) aus mit den Befehlen "Open File" oder "New File" aktiviert. Der Editor bietet die üblichen Funktionen an, die in zwei Befehlsgruppen, "File" und "Edit", gegliedert wurden.

4.2.6.1 File

In dieser Befehlsgruppe (siehe Abbildung 4.25) befinden sich diejenigen Befehle, die sich mit der Steuerung des Editors selbst beschäftigen.

Save File sichert die Arbeitsdatei auf die Festplatte und überschreibt dabei die sich dort befindliche Kopie.

Save File as ... speichert die Arbeitsdatei unter einem neuen Namen, der im üblichen Dateidialog (siehe Abbildung 4.3) eingegeben wird, ab.

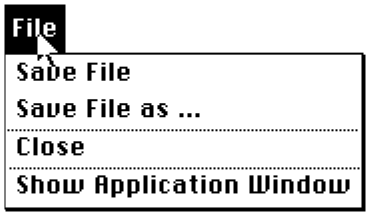

Abbildung 4.25: Editor-Menü "File"

Close speichert, wenn notwendig, die Arbeitsdatei und schließt das Editorfenster.

Show Application Window zeigt das zum Editor gehörende Applikationsfenster am Bildschirm an. Dieses Kommando ist dann nützlich, wenn das Applikationsfenster durch andere Fenster verdeckt worden ist.

4.2.6.2 Edit

In dieser Befehlsgruppe (siehe Abbildung 4.26) befinden sich diejenigen Kommandos, die die eigentlichen Editieroperationen unterstützen.

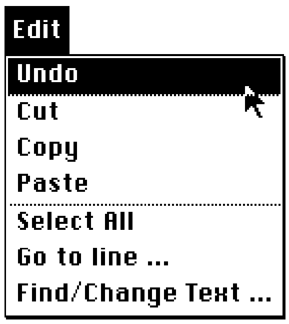

Abbildung 4.26: Editor-Menü "Edit"

Undo macht den Effekt des unmittelbar zuvor ausgeführten Kommandos rückgängig.

Cut löscht den ausgewählten Text aus dem Editor und fügt diesen Text in die Zwischenablage ein.

Copy kopiert den ausgewählten Text in die Zwischenablage.

Paste fügt den Text, der sich in der Zwischenablage befindet, an der Stelle in die Arbeitsdatei ein, die durch die Einfügemarke gekennzeichnet ist.

Select All markiert den gesamten Text der Arbeitsdatei.

Go to line ... positioniert die Einfügemarke in der Zeile der Arbeitsdatei, deren Nummer der im Dialog angegebenen entspricht.

Find/Change Text ... zeigt einen Dialog, der es erlaubt, nach Zeichenketten zu suchen und diese bei Bedarf gegen eine andere Zeichenkette auszutauschen.

4.3 Parametrisierung

Das Verhalten von DOgMA kann zu einem gewissen Grad von außen beeinflußt werden. Unter Unix stehen zu diesem Zweck sogenannte "Umgebungsvariablen" zur Verfügung. Diese Variablen werden vom Kommandoprozessor verwaltet und sind allen Applikationen, die von diesem Kommandoprozessor gestartet werden, zugängig.

Für DOgMA können diese Variablen beispielsweise in der Datei ".cshrc" definiert werden, die bei jedem Start eines Kommandoprozessors ausgeführt wird. Diese Werte können dann nachträglich noch händisch angepaßt werden.

Folgende Parameter können für DOgMA spezifiziert werden:

- Bei großen Programmsystemen können die Programmdateien über mehrere Verzeichnisse verteilt sein. Damit die Dateien, die sich in anderen Verzeichnissen als dem Verzeichnis des Projektmoduls befinden, geladen werden können, müssen diese Verzeichnisse in einen Suchpfad aufgenommen werden. Dies wird durch Definition der Umgebungsvariable "INCLUDE1" bis "INCLUDE9" erreicht, die auf das zu untersuchende Verzeichnis zeigt.

- Mit der Variable "HISTORY_SIZE" kann angegeben werden, wieviele Einträge in der Liste der Dialogbox "History" (siehe Abbildung 4.14) angezeigt werden sollen. Wenn man mit einem Programmsystem noch nicht vertraut ist, sollte man diesen Wert erhöhen, um mehr Navigationsschritte rückgängig machen zu können.

- Mit der Variable "LIST_WIDTH" kann die Breite der Auswahllisten (siehe Abschnitt 4.2.3) angegeben werden. Dies ist dann wichtig, wenn die verwendeten Bezeichner sehr lang sind, sodaß sie mit der Voreinstellung nicht mehr unterschieden werden können. Ansonsten kann auch die Breite herabgesetzt werden, um Platz am Bildschirm zu sparen.

- Durch die Variable "LIST_AT_RIGHT" kann eingestellt werden, ob die Auswahllisten rechts (Voreinstellung) oder links vom Editor angezeigt werden sollen.

- Die Variable "ASK_FOR_SAVE" bestimmt, ob beim Sichern einer Datei der Benutzer zuvor um Erlaubnis gefragt werden soll (Voreinstellung) oder nicht.

- Weiters existieren Umgebungsvariablen, mit denen der Text bestimmt werden kann, der bei der Neuerstellung von Modulen und Prozeduren in diese automatisch eingefügt wird. Es handelt sich dabei um die Variablen "PROCEDURE", "MODULE", "DEFMOD", "IMPMOD", "DOCU_FILE" und "CHAPTER".

  In diesen Zeichenketten kann als Parameter ein "%s" enthalten sein, welches im Text dann durch den Namen des Textes ersetzt wird.

  Durch diese Parametrisierbarkeit kann das Werkzeug an die verschiedenen Vorlieben des Benutzers oder auch an Projektstandards angepaßt werden.

Kapitel 5

Implementierung

Dieses Kapitel beschäftigt sich mit einigen Aspekten der Implementierung von Modula-2-DOgMA. Dabei wird hauptsächlich auf die Änderungen gegenüber C++-DOgMA eingegangen, sowie auf die für deren Verständnis notwendigen Teile des Programmsystemes. Für eine ausführliche Beschreibung von C++-DOgMA vergleiche [Sam91].

Zuerst wird ein Überblick über die Implementierung von DOgMA und dessen wichtigsten Datenstrukturen gegeben. Die nächsten Abschnitte beschreiben die Implementierung des sprachabhängigen Teiles und die Änderungen, die im sprachunabhängigem Teil vorgenommen werden mußten. Danach wird die Problematik der Anbindung von Modula-2-Code an C++-Code dargestellt, da ein Teil des sprachabhängigen Teiles von DOgMA in Modula-2 implementiert wurde. Den Abschluß bildet eine chronologische Darstellung der Vorgehensweise bei der Implementierung.

5.1 Überblick

Dieser Abschnitt gibt einen Überblick über die Implementierung von DOgMA, sowohl für die C++- als auch für die Modula-2-Version, mit besonderer Ausrichtung auf die für die Anpassung an Modula-2 relevanten Teile des Programmsystems. Zuerst wird auf die Klassenhierarchie eingegangen und dann werden die wichtigsten Datenstrukturen vorgestellt.

Modula-2-DOgMA wurde auf einer Sun-Workstation 3/60 unter dem Betriebssystem Unix und der Benutzeroberfläche SunView erstellt. Für die Implementierung wurde die Programmiersprache C++ und das Application Framework[1] ET++ verwendet.

5.1.1 Klassenhierarchie

Die Klassen von DOgMA werden in Abbildung 5.1 dargestellt, wobei die von ET++ implementierten Klassen dick umrandet sind. Die Klassen, die bei der Implementierung von Modula-2-DOgMA erstellt wurden, sind als Ovale dargestellt, alle übrigen Klassen wurden von C++-DOgMA übernommen.

Root ist die Oberklasse aller Klassen von ET++. Alle ET++-Klassen, die nicht systemspezifisch sind, werden von *Object* abgeleitet. Objekte der Klasse *EvtHdlr* können auf Ereignisse, die z.B. der Benutzer auslöst, reagieren. Die Klasse *Document* implementiert die Funktionalität zum Öffnen und Schließen einer Applikation. Die Klassen *Text*, *GapText* und *StyledText* implementieren Datenstrukturen zur Speicherung von Texten, wobei *StyledText* zusätzlich die Speicherung des Darstellungsstils erlaubt (vergleiche [WGM89]).

Mit einem Objekt der Klasse *HyperText* wird ein Textstück gespeichert, welches mit Hilfe der von ET++ ererbten Funktionalität editiert werden kann.

Die Klasse *M2HyperTextDocument* implementiert unter anderem die Benutzerschnittstelle, wobei die Klasse HyperTextDocument, die die Benutzerschnittstelle für C++-DOgMA implementiert, benutzt wird (siehe auch Abschnitt 5.3). Hier ist auch die Funktionalität zum Laden und Speichern

[1] Unter einem "Application Framework" wird eine Klassenbibliothek verstanden, die es mit vergleichsweise geringem Aufwand erlaubt, Applikationen – meist für graphische Benutzeroberflächen – zu erstellen.

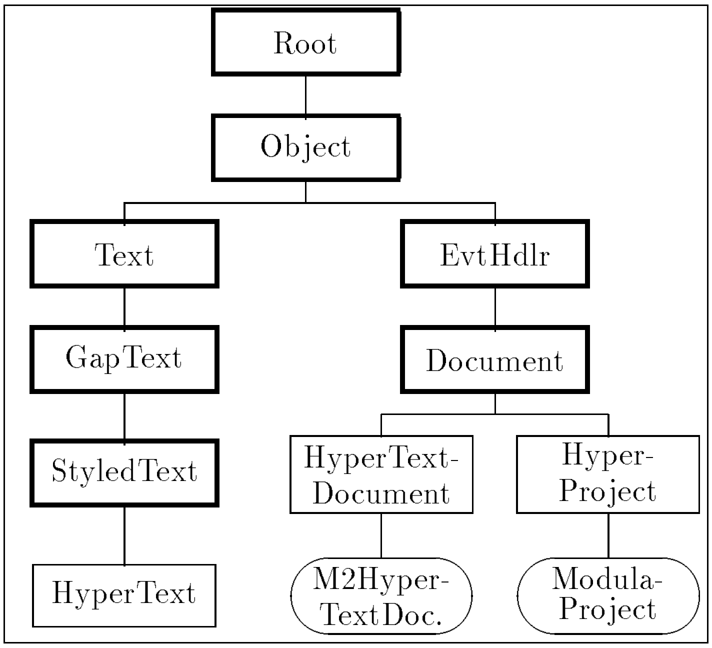

Abbildung 5.1: Klassenhierarchie von Modula-2-DOgMA

von Dateien gekapselt.

Die Klasse *ModulaProject* erbt von HyperProject die Funktionalität, um allgemeine Informationen über ein Softwaresystem erstellen und verwalten zu können. Zusätzlich stellt ModulaProject die Methoden ReadProject und ParseProject zur Verfügung, die die statische Analyse des Modula-2-Quellcodes durchführen. Die dabei gewonnenen Informationen werden mit Hilfe von Mitteilungen an Objekte der Klasse HyperText weitergegeben, sodaß diese Informationen bei der Visualisierung des Codes, z.B. um Schlüsselwörter hervorzuheben, und zum Navigieren benutzt werden können.

5.1.2 Datenstrukturen

Um die in Kapitel 4 beschriebene Funktionalität bereitstellen zu können ist eine sehr komplexe Datenstruktur notwendig. Sie besteht aus einer Anzahl von Knoten im Sinne des Hypertext-Konzeptes. Jeder Knoten enthält Verweise auf einen oder mehrere andere Knoten, sodaß ein Netz in Form eines gerichteten Graphens entsteht. Entlang der Kanten dieses Graphens kann der Benutzer navigieren, die ihm dabei zur Verfügung stehenden Möglichkeiten werden in den Auswahllisten (siehe Abschnitt 4.2.3) angegeben. Jedem dieser Knoten ist ein Text zugeordnet, der den dazugehörenden Programm- oder Dokumentationstext enthält. Jedem Text können mehrere Markierungen zugeordnet sein, die die Darstellung des Textes am Bildschirm bestimmen. Dadurch ist es möglich, ein Schlüsselwort anders darzustellen als einen Kommentar. Bei einem Text kann es sich um den Code einer Prozedur oder eines Moduls oder um ein Kapitel der Dokumentation handeln.

Die Knoten werden durch Objekte der Klasse NodeItem verwirklicht. Die Texte werden von Objekten der Klasse HyperText verwaltet, die Klasse HyperMarks implementiert die Markierungen.

5.1.2.1 HyperText

In einem HyperText wird ein in sich geschlossenes Textstück gespeichert. Dabei kann es sich um ein Modul, eine Prozedur, eine Datei oder um ein Kapitel der Dokumentation handeln. HyperText ist eine Unterklasse von StyledText (siehe Abbildung 5.1), welche von ET++ zur Verfügung gestellt wird. Die von StyledText und deren Oberklassen implementierte Funktionalität erlaubt es, einen Text zu editieren und bei der Darstellung dieses Textes verschiedene Schriftarten und Schriftstile zu verwenden. Diese Möglichkeit wird dazu benutzt, die verschiedenen Codeteile und -arten optisch besser unterscheidbar zu machen.

In ET++ wird ein Text als eine Reihe von ASCII-Zeichen abgespeichert, wobei sich im eigentlichen Text keine Stilinformationen befinden. Diese werden in einer parallel zum Text laufenden Liste abgelegt, in der sich die Information, ab welchem Zeichen welcher Darstellungsstil zu verwenden ist, befindet. In DOgMA wird zusätzlich noch in einer ähnlichen Liste die Art des Textteiles beschrieben, also ob es sich um ein Schlüsselwort, eine Bezeichnerdefinition, eine Bezeichnerverwendung oder einen Kommentar handelt. Diese Liste besteht aus Objekten der Klasse HyperMark (siehe auch Abschnitt 5.1.2.3).

Aufgrund der Einträge dieser Liste entscheidet ein Objekt der Klasse HyperText welcher Darstellungsstil für ein bestimmtes Codestück zu verwenden ist.

5.1.2.2 NodeItems

Eines der typischen Merkmale des Konzeptes von Hypertext besteht darin, daß sich der Benutzer von einem Informationsknoten zum nächsten bewegt, wenn er durch das Gesamtdokument navigiert. Diese Informationsknoten werden in DOgMA durch die Klasse NodeItems und deren Unterklassen implementiert. Die Einordnung dieser Klassen in die Klassenhierarchie ist in Abbildung 5.2 dargestellt.

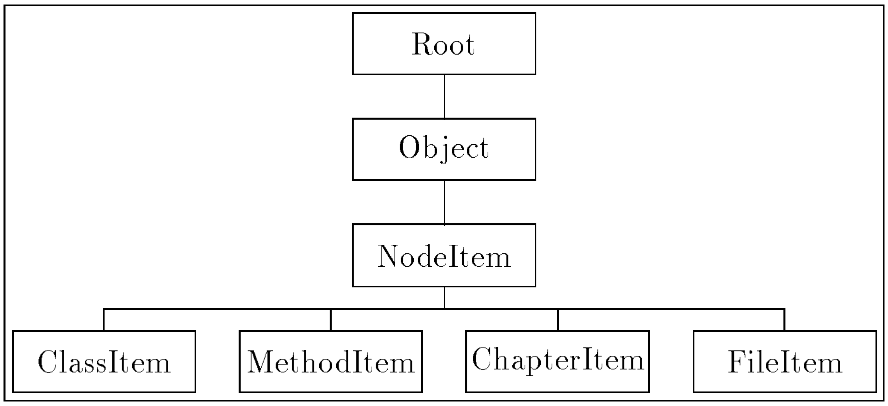

Abbildung 5.2: Klassenhierarchie der NodeItems

Die Namen der Objekte wurden von C++-DOgMA übernommen, sodaß ein ClassItem ein Modul und ein MethodItem eine Prozedur enthält. Die Anpassung der Bezeichner an Modula-2, also die Umbenennung von ClassItem in ModuleItem und von MethodItem in ProcedureItem hätte sehr große Eingriffe in viele Teile des Programmsystems bedeutet, was sich auch bei der Übernahme von Neuerungen von C++-DOgMA in Modula-2-DOgMA negativ ausgewirkt hätte. Es wäre natürlich wünschenswert, allgemeinere Namen zu finden, aber die Auswahl eines aussagekräftigen Oberbegriffes für ClassItem und ModuleItem ist, gelinde gesagt, sehr schwierig.

Zu jedem NodeItem gehört ein HyperText, der Text, der bei Auswahl des Knoten im Editor dargestellt werden soll. Weiters gehen von jedem Knoten mehrere Verbindungen zu anderen NodeItems aus. So existieren zum Beispiel in einem ClassItem Verweise auf die Datei, in der das Modul enthalten ist, auf importierte Module und auf Prozeduren, die in dem Modul implementiert werden. Diese Verbindungen werden in den Auswahllisten des Werkzeuges dargestellt, sodaß sich der Benutzer sehr einfach von einem Knoten zum nächsten bewegen kann.

Die einzelnen NodeItems haben folgende Aufgaben:

ClassItem: speichert alle Informationen, die ein Modul betreffen sowie den Programmtext, der im Modul außerhalb der Prozeduren existiert, d.h. die Prozeduren sind ausgeblendet.

MethodItem: enthält den Quellcode einer Prozedur sowie den Verweis auf das Modul, zu dem die Prozedur gehört.

ChapterItem: beinhaltet einen Teil der Dokumentation und die Verweise auf den Quellcode.

FileItem: enthält Informationen über eine Datei.

5.1.2.3 HyperMarks

Objekte der Klasse HyperMark und deren Unterklassen dienen dazu, bestimmte Stellen eines HyperTextes zu identifizieren (siehe auch Abschnitt 5.1.2.1). Aufgrund dieser Information wird das entsprechende Textstück mit der dazugehörigen, frei wählbaren Darstellungsart im Editor des Werkzeuges dargestellt. Abbildung 5.3 stellt die Einordnung der verschiedenen HyperMarks in die Objekthierarchie von Modula-2-DOgMA dar.

Die einzelnen MarkItems bezeichnen folgende Stellen im Programm- oder Dokumentationstext:

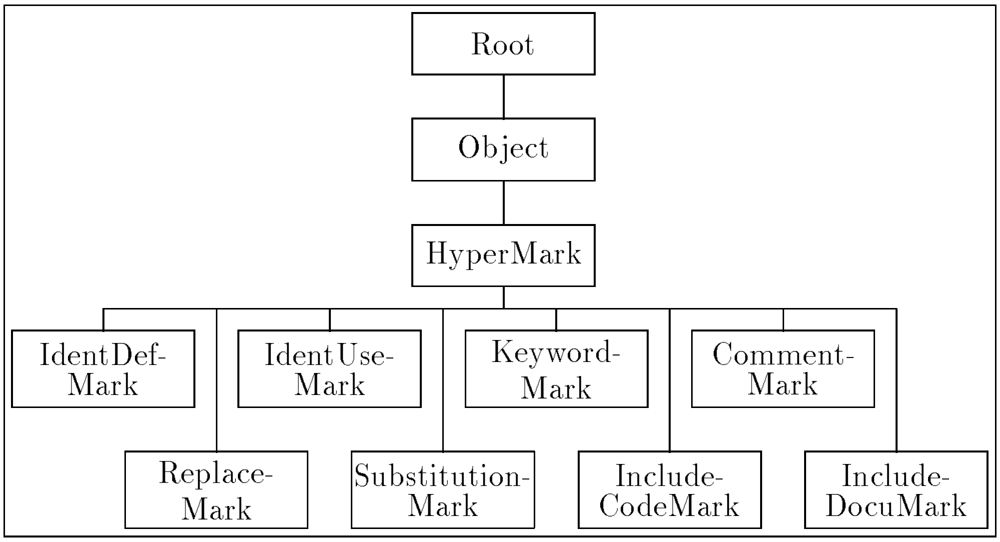

Abbildung 5.3: Klassenhierarchie der MarkItems

IdentDefMark: Definition eines Bezeichners

IdentUseMark: Verwendung eines Bezeichners

KeywordMark: ein Schlüsselwort

CommentMark: ein Kommentar

IncludeDocuMark: Verweis auf ein Stück der Dokumentation

IncludeCodeMark: Verweis auf ein Stück Quellcode

SubstitutionMark: Verweis auf einen anderen HyperText, zum Beispiel auf eine Prozedur

ReplaceMark: Verweis auf eine ersetzte Textstelle

Abbildung 5.4 zeigt die Anwendung der HyperMarks. Sie sind untereinander verkettet und jedes HyperMark zeigt auf eine Textstelle.

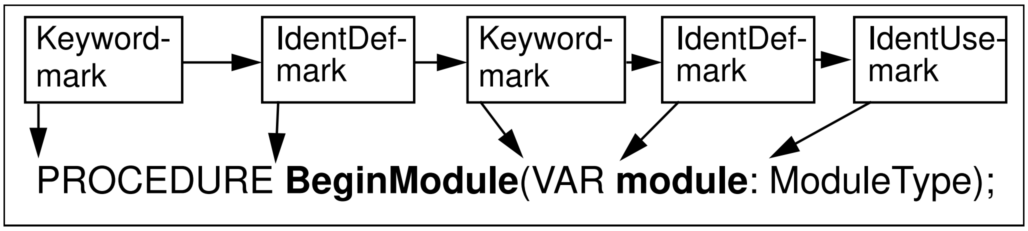

Abbildung 5.4: Anwendung von HyperMark-Objekten

5.2 Sprachabhängiger Teil

Der sprachabhängige Teil ist in der Klasse ModulaProject gekapselt, welche eine Unterklasse von HyperProject ist. Die Methoden von ModulaProject zerfallen dabei in zwei große Gruppen, die nach der jeweils hierarchisch obersten Methode benannt worden sind: ReadProject und ParseProject. Die Grundstruktur von ModulaProject und der dazugehörigen Module wird in Abbildung 5.5 dargestellt.

ReadProject (in der Graphik als Oval dargestellt) liest Dateien ein und zerlegt diese in die einzelnen Informationsbausteine wie Module, Prozeduren und Dokumentationskapitel. Damit stehen alle für das Navigieren benötigten Basisinformationen zur Verfügung. Werden darüberhinausgehend noch Informationen über die im Code enthaltenen Bezeichner gewünscht, müssen die dafür zusätzlich benötigten Information mit Hilfe von ParseProject erzeugt werden.

ParseProject erzeugt diese Informationen mit Hilfe eines Parsers, der in Modula-2 implementiert wurde (unterer Teil der Abbildung). Der Parser benutzt die Module ParseIdentifierHdlr und ParseStackHdlr, um die Bezeichner zu verwalten, und den Scanner, um den Eingabestrom aufzubereiten. Der Parser und die Methode ParseProject sind durch je ein Schnittstellenmodul für C++ und Modula-2 miteinander verbunden.

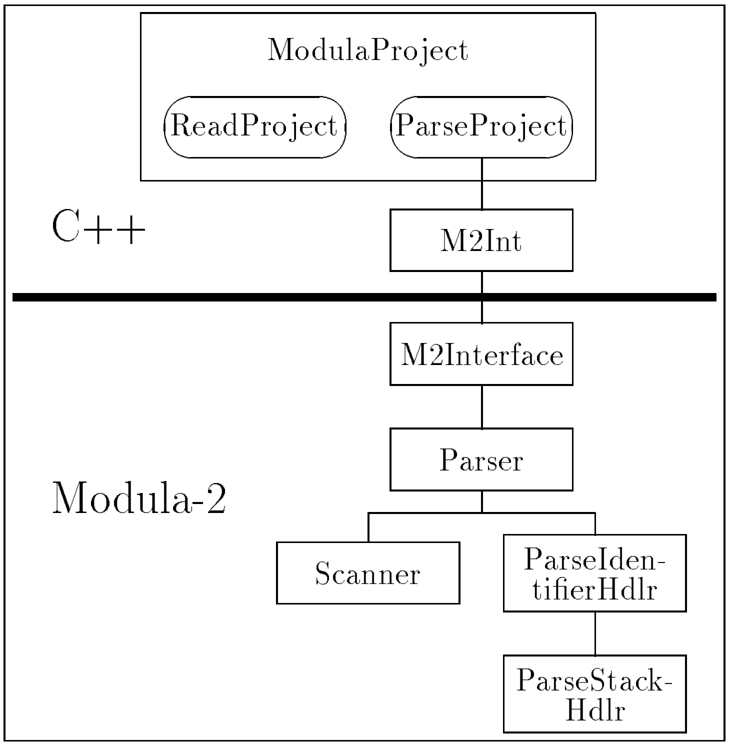

Abbildung 5.5: Grundstruktur von ModulaProject

5.2.1 ReadProject

Die Methode ReadProject stellt die Funktionalität für das Einlesen der Quelltexte und für den Aufbau der grundlegenden Datenstrukturen zur Verfügung. Dafür wird zuerst der Projektmodul eingelesen und dabei aufgrund der vorgefundenen IMPORT-Anweisungen die angegebenen Dateien eingelesen.

Beim Einlesen einer Datei wird zuerst versucht, den Typ der Datei zu bestimmen, also festzustellen, ob es sich um ein Definitions- oder Implementierungsmodul handelt. Dann wird die Datei auf IMPORT- und PROCEDURE-Anweisungen untersucht. Wenn eine IMPORT-Anweisung erkannt wird, wird die importierte Datei, wenn sie bisher noch nicht eingelesen worden ist, durch einen rekursiven Aufruf abgearbeitet. Das Erkennen einer PROCEDURE-Anweisung führt zu einer dem Modultyp entsprechenden Verarbeitung der Prozedur. Während des Einlesens der Quelltexte wird der Benutzer in einem speziellen Fenster über den Fortschritt des Vorganges unterrichtet.

ReadProject ruft zuerst die Methode ReadCode auf, die das eigentlich Einlesen und Aufbereiten der Quellcodedateien steuert. Nach dem Einlesen der Dokumentation durch die Methode ReadDocu werden die Metainformationen, z.B. über die IMPORT Hierarchie – in C++-DOgMA über Unter- und Oberklassen – aus den einzelnen Modul-Datenstrukturen aufgebaut und sortiert. Die Methode ReadProject sowie die Methoden für das Einlesen der Dokumentation wurden ohne Änderungen von C++-DOgMA übernommen.

Die Methode ReadCode implementiert die Funktionalität für das eigentliche Einlesen der Quelltexte und deren Aufteilung auf die jeweiligen Informationsbausteine. Diese Methode und alle ihr hierarchisch untergeordneten Methoden mußten für Modula-2-DOgMA angepaßt werden. Obwohl versucht worden war, die Programmstruktur von ModulaProject möglichst ähnlich der von CProject zu halten, konnte dieser Vorsatz nur sehr beschränkt in die Tat umgesetzt werden. Die Aufteilung der Funktionalität auf die einzelnen Methoden wurde in etwa beibehalten, obwohl auch hier einige Umschichtungen vorgenommen wurden, um eine bessere Konsistenz der Methoden untereinander in Bezug auf die Geschlossenheit und Einheit der Funktionalität zu erreichen.

Der interne Aufbau von ReadCode besitzt gewisse Ähnlichkeiten mit einem Parser: ReadCode holt sich jeweils ein Symbol von seinem Scanner, in diesem Fall der Methode NextToken, und durchtaucht dann in Abhängigkeit vom erkannten Zeichen einen sehr einfachen Syntaxbaum. Bei dem von NextToken geliefertem Symbol handelt es sich um das jeweils nächste, noch nicht gelesene relevante Zeichen des Eingabestromes.[2] ReadCode untersucht aufgrund dieses Zeichens, ob ein an diesem Punkt der Datei für das Einlesen relevantes Schlüsselwort im Eingabestrom enthalten ist und veranlaßt gegebenenfalls die Verarbeitung.

Die von ReadCode erkannte Syntax ist sehr einfach gehalten. ReadCode versucht zuerst zu erkennen, welche Art von Modul eingelesen wird, wobei folgende Modularten möglich sind:

[2] Unter einem relevanten Zeichen ist in diesem Zusammenhang ein Zeichen zu verstehen, welches weder Teil eines Kommentars noch einer Zeichenkette ist und sich nicht zwischen zwei Klammern befindet.

```
IMPLEMENTATION MODULE
DEFINITION MODULE
MODULE
PROJECT MODULE
```

Die Information des Modultyps dient als wichtige Information für das Aufbauen der Datenstrukturen, da diese über die Einreihung der Datei in die entsprechenden Listen und über die Verarbeitung der Prozeduren der Datei entscheidet. Wenn ReadCode keinen Modultyp erkennen kann, wird der Dateiinhalt überlesen und eine entsprechende Fehlermeldung ausgegeben. Die übrigen Dateien werden jedoch korrekt eingelesen.

Wenn der Modultyp erkannt ist, wird die Datei auf Import- und Prozeduranweisungen untersucht. Dabei werden folgende Importanweisungen behandelt:

```
IMPORT MODULE
IMPORT DOCUMENTATION
IMPORT
FROM ... IMPORT
```

wobei es sich bei IMPORT MODULE und IMPORT DOCUMENTATION um DOgMA-spezifische Anweisungen handelt.

Wenn die in der Import-Anweisung angegebene Datei existiert und noch nicht importiert worden ist, wird sie eingelesen. Dafür müssen die entsprechenden Datenstrukturen für die Dateibeschreibung sowie der HyperText erzeugt und für das Lesen vorbereitet werden. Auf die verschiedenen Arten der IMPORT-Anweisung wurde bereits in Abschnitt 4.1 eingegangen.

Schließlich müssen noch die Prozeduren verarbeitet werden. Nachdem eine Prozedur erkannt worden ist, wird der entsprechende HyperText, der nur den Text der Prozedur enthält, erzeugt und die Informationen über die Prozedur, der Name der Prozedur und zu welchem Modul die Prozedur gehört, in die globale Datenstruktur eingefügt.

Modula-2 ist in vieler Hinsicht eine sehr viel einfachere Sprache als C++, was besonders dann deutlich wird, wenn man ModulaProject und CProject miteinander vergleicht. Generell kann gesagt werden, daß der Code in ModulaProject weniger komplex und leichter verständlich ist.

5.2.2 ParseProject

Bei ParseProject war die Situation ähnlich wie bei ReadProject: ParseProject ruft die Prozeduren ParseCode und ParseDocu auf, bereitet die von den beiden Methoden gelieferten Metainformationen auf und sortiert diese. Die Prozeduren, die das Parsen der Dokumentation erledigen, konnten ebenso wie bei ReadProject ohne Änderungen übernommen werden. Bei einem Neuentwurf wäre es günstig, ReadProject, ParseProject und die die Dokumentation verarbeitenden Prozeduren entweder in die Oberklasse HyperProject hinaufzuverlagern oder die ReadCode- und ParseCode-Methoden in einer sprachabhängigen Unterklasse zu kapseln.

Die Methode ParseCode steuert das Parsen eines Programmsystemes. Sie initialisiert den Modula-2-Teil, also den Parser, und veranlaßt das Parsen jeder Datei des Programmsystemes. Der eigentliche Parser wurde in Modula-2 implementiert, wobei Teile des SCT-Modula-2-Compilers (siehe [Bis90]) verwendet wurden. Die ursprüngliche Absicht, den gesamten SCT-Modula-2-Compiler für DOgMA anzupassen scheiterte daran, daß sich der SCT-Parser nur mit unverhältnismäßig hohem Aufwand aus dem Gesamtsystem hätte herauslösen lassen. Für die Entscheidung, den Parser selbst zu implementieren, gaben folgende Gründe den Ausschlag:

- Ein speziell für DOgMA angefertigter Parser muß nur die Fehler in der Eingabedatei erkennen können, die das weitere Einlesen unmöglich machen. Typinkompatibilitäten, falsch geschachtelte Klammern und viele andere Fehler können aber ignoriert werden, ohne daß dadurch ein konsistentes Arbeiten mit den erzeugten Informationen unmöglich gemacht wird. Dadurch können mit dem Werkzeug selbst auf dieser Ebene der Informationsgewinnung syntaktisch sehr fehlerhafte Programme sinnvoll bearbeitet werden.

- Ein selbst geschriebene Parser ist vom Aufbau her gesehen einfacher und muß weniger Überprüfungen des Codes vornehmen, deswegen sollte er auch schneller sein.
- Als Alternative zur Implementierung des Parsers besteht die Möglichkeit, den SCT-Parser anzupassen. Bei beiden Alternativen wurde der zeitliche Aufwand für die Realisierung als in etwa gleich angesehen.

Für die Entscheidung, den Parser in Modula-2 und nicht in C++ zu entwickeln, waren folgende Gründe wichtig:

- Ein Vorteil der Verwendung von C++ wäre die Möglichkeit, die von ET++ angebotenen Datenstrukturen, wie z.B. Collections (siehe [WGM89]), verwenden zu können. Dieser Vorteil kommt für DOgMA aber insoferne nicht zum Tragen, als sich die benötigten, teilweise *sehr* komplexen Datenstrukturen nicht aus den oben erwähnten vordefinierten Klassen mit vernünftigem Aufwand ableiten lassen.
- Modula-2 verfügt über eine sehr viel strengere Typprüfung als C++. Diese ist speziell bei der Verwendung einer Vielzahl von verschiedenen Zeigertypen, wie es bei diesem Problem der Fall war, von großem Vorteil. Mit der weniger rigiden Typprüfung für Zeiger in C++ hätten viele Fehler nicht schon bei der Übersetzung, sondern erst viel später und nach einer schwierigeren Fehlersuche korrigert werden können.
- Die Entwicklungszeiten, speziell der Übersetzung-Binden-Ausführung Zyklus, sind für Modula-2-Programme kürzer als für C++-Programme.
- Die Modula-2-Schnittstelle zu C++ war zu diesem Zeitpunkt schon vorhanden.
- Für die Entwicklung des Modula-2-Teiles konnte eine Zwischenversion von Modula-2-DOgMA benutzt und praktisch erprobt werden.

Der Parser von Modula-2-DOgMA unterscheidet sich von einem Parser, der für einen Compiler verwendet wird, hauptsächlich dadurch, daß er lediglich einige wenige Informationen über die zu verarbeitenden Dateien bereitstellen muß. Bei der Implementierung wurde darauf geachtet, daß wirklich nur die benötigten Informationen erzeugt werden und daß so wenige Überprüfungen der Korrektheit des Quellcodes wie möglich durchgeführt werden. Folgende Informationen werden von DOgMA benötigt:

Bezeichnerdefinition: Anfangs- und Endposition einer Bezeichnerdefinition in der Programmdatei

Bezeichnerverwendung: Anfangs- und Endposition einer Bezeichnerverwendung und Verweis auf die dazugehörige Bezeichnerdefinition

Kommentare: Anfangs- und Endposition eines Kommentares

Schlüsselworte: Anfangs- und Endposition eines Schlüsselwortes

Um diese Informationen erzeugen zu können ist speziell für die Bezeichnerverarbeitung ein beträchtlicher interner Aufwand erforderlich.

Nach den während der Ausarbeitung gewonnenen Erfahrungen beinhalten ReadProject, ParseProject und die Dokumentations-Methoden keine sprachspezifischen Eigenheiten und sollten daher besser durch Vererbung als durch Kopieren des Quellcodes, wie es bei Modula-2-DOgMA noch der Fall ist, wiederverwendet werden.

Die folgenden Abschnitte beschreiben die einzelnen Komponenten des Parsers.

5.2.2.1 Schnittstelle zu C++

Den grundlegenden Aufbau einer Schnittstelle zwischen Modula-2 und C++ beschreibt Abschnitt 5.4, wo alle Techniken, die bei der Implementierung benutzt wurden, detailliert dargestellt werden. Die Erstellung der Schnittstelle gestaltet sich insoferne als aufwendig, als Einträge in vielen Dateien aufeinander abgestimmt werden müssen. Dank der strengen Modula-2-Typprüfung lassen sich die dabei auftretenden Fehler relativ schnell aufspüren und korrigieren.

5.2.2.2 Scanner

Der Scanner ließt Zeichen für Zeichen den Eingabestrom, der den Quellcode enthält, bis er ein Symbol der Programmiersprache erkannt hat. Dieses Symbol wird dem Parser zur weiteren Bearbeitung übergeben. Wenn der Parser dieses Symbol verarbeitet hat, fordert er beim Scanner das nächste Symbol an.

Der Scanner und einige dem Scanner untergeordnete Module wurden dem SCT-System entnommen (vergleiche [Bis90]). Um ihn für DOgMA verwendbar zu machen waren nur wenige Eingriffe notwendig, wobei die wichtigsten Änderungen auf die andere Struktur des zu verarbeitenden Eingabestroms zurückzuführen sind. Der ursprüngliche Scanner verarbeitete einen einzelnen, linearen Eingabestrom, wie ihn z.B. eine Datei liefert, die zeichenweise gelesen wird. Im Fall von DOgMA wird vom HyperText zwar auch zeichenweise gelesen, ein HyperText muß aber nicht den ganzen zu verarbeitenden Text enthalten. So sind z.B. die Implementierungen der Prozeduren jeweils in einem separatem HyperText abgespeichert. Um den Scanner für DOgMA verwendbar zu machen, mußten einerseits die eigentlichen Zugriffsfunktionen auf den Eingabestrom angepaßt werden und andererseits der Scanner dahingehend erweitert weden, daß die verschiedenen HyperTexte in der richtigen Reihenfolge zur Verarbeitung herangezogen werden. Weitere Änderungen bezogen sich die Bezeichnerverwaltung sowie darauf, daß dem C++-Teil die Position von Schlüsselwörtern und Kommentaren bekanntgegeben werden muß.

5.2.2.3 Parser

Die vom Scanner erkannten Symbole werden vom Parser weiterverarbeitet. Dabei wird rekursiv der Syntaxbaum bearbeitet und immer dann, wenn eine Bezeichnerdefinition oder -verwendung erkannt worden ist, der Bezeichnerverwalter aufgerufen. Der strukturelle Aufbau des Parsers lehnt sich eng an die EBNF-Darstellung der Syntax von Modula-2 in [Wir91] an, die Namen der Prozeduren wurden ebenfalls analog dazu gewählt. Die Fehlerbehandlung wurde bewußt einfach gehalten: wird ein Fehler erkannt, wird die Bearbeitung der Datei abgebrochen und die Zeilennummer, in der der Fehler auftrat, an den C++-Teil übergeben, der eine entsprechende Fehlermeldung ausgibt. Danach werden die restlichen Dateien normal abgearbeitet.

5.2.2.4 Bezeichnerverwalter

Der Bezeichnerverwalter ist der mit Abstand komplexeste Teil des gesamten Modula-2-Teils. Er besteht aus zwei Modulen, die je einen abstrakten Datentyp implementieren: ParseIdentifierHdlr und ParseStackHdlr. ParseIdentifierHdlr enthält die Daten sämtlicher Bezeichner aller bisher verarbeiteten Dateien und benutzt zu deren Verwaltung ParseStackHdlr, der die Bezeichner eines einzelnen Moduls verarbeitet. Der Großteil der Funktionalität wurde in ParseStackHdlr implementiert, der ausschließlich von ParseIdentifierHdlr benutzt wird. ParseIdentifierHdlr implementiert den abstrakten Datentyp ParseStack, dessen Aufbau eine gewisse Ähnlichkeit mit einem (wohlstrukturiertem) "Drahtverhau" bestehend aus verschiedenen Zeigern hat. Er speichert für jeden Bezeichner die Stelle der Definition, den Datentyp, wie der Bezeichner qualifiziert werden kann beziehungsweise werden muß, den Gültigkeitsbereich des Bezeichners und viele weitere Informationen, die für eine korrekte Identifizierung im Programmkontext notwendig sind. Die Datenstruktur selbst besteht aus mehreren einfach verketteten Listen, die untereinander und in sich selbst mit verschiedenen Zeiger-Querverweisen verbunden sind. Die beiden Module wurden so gestaltet, daß sie auch für andere, hinreichend ähnliche Sprachen verwendet werden können, wenn der Parser und der Scanner entsprechen ausgetauscht werden. Wenn die entsprechenden Informationen über die Defintion oder die Verwendung eines Bezeichners vorliegen, werden diese dem C++-Teiles mitgeteilt.

5.3 Sprachunabhängiger Teil

Neben den Änderungen in ModulaProject mußten auch Anpassungen in einem Teil des Programmsystems vorgenommen werden, der nicht explizit als sprachabhängig deklariert worden war: bei der Definition der Benutzerschnittstelle. Die Funktionalität, die den in diesem Kontext relevanten Teil der Oberfläche beschreibt, ist in der Klasse HyperTextDocument gekapselt. Die für die Anpassung an Modula-2-DOgMA notwendigen Änderungen wurden in einer eigenen Unterklasse von HyperTextDocument mit Namen M2HyperTextDocument vorgenommen.

Diese Änderungen waren von geringer Schwierigkeit und beschränkten sich im allgemeinen auf die Anpassung von anzuzeigenden Zeichenketten. So mußten zum Beispiel die Bezeichnungen einiger Menüeinträge angepaßt werden, da nunmehr anstelle von Methoden Prozeduren und anstelle von Klassen Module verarbeitet werden. Manche Menüpunkte wurden ersatzlos gestrichen, so unter anderem die Anzeige von Unterklassen, während die Anzeige von Oberklassen in die Anzeige von importierten Modulen umfunktioniert wurde.

Weitere Änderungen betrafen die Menüpunkte, mit deren Hilfe neue Programmbausteine in das System eingefügt werden können. Diese wurden so abgeändert, daß gültige Modula-2-Prozeduren beziehungsweise Module erstellt werden. Die Benutzung und die Funktionsweise dieser Menüpunkte wurde unter Abschnitt 4.2.4.2 beschrieben.

Schließlich mußte noch die Dialogbox, die die Informationen über einen bestimmten Bezeichner anzeigt, angepaßt werden. Die Box wird in C++-DOgMA durch die Klasse RemarkDialog implementiert, für Modula-2-DOgMA wurde davon die Klasse M2RemarkDialog abgeleitet, wobei aber lediglich eine Methode zu überschreiben war.

Generell kann festgestellt werden, daß im Falle von HyperTextDocument die Kapselung von sprachabhängiger Funktionalität weniger gut geglückt ist als bei CProject bzw. ModulaProject – HyperTextDocument wurde offensichtlich nicht im Hinblick auf ein einfaches Ableiten von Unterklasssen entwickelt. So existiert z.B. eine kleine Privatklasse in der Datei HyperTextDocument.c – MyPulldownbar – die einen Pulldownbar implementiert. Da diese Klasse zwangsläufig in einer Unterklasse benötigt wird, mußte die Definition von MyPulldownbar in HyperTextDocument.h aufgenommen werden. Die Änderungen bezüglich der Methoden, die die Ereignisverarbeitung durchführen, lassen sich durch Überschreiben und Verwenden der Methoden, die von der Oberklasse HyperTextDocument vererbt werden, elegant durchführen. Diese Vorgehensweise versagt aber bei den Methoden, die das Aussehen der Oberfläche definieren, also z.B. die Menüstruktur sowie die PopUp-Menüs. Hierbei mußten mehrere hundert Zeilen Code kopiert werden, um einige Zeichenketten ändern zu können.

Bei einem Neuentwurf von HyperTextDocument wäre es günstiger, die ziemlich großen Definitionsmethoden in mehrere kleinere zu zerlegen, die jeweils einen Teil der Schnittstelle modellieren. So könnte jedes PopUp der Menüleiste in einer eigenen Methode definiert werden, also beispielsweise je eine für das File- und das Editmenü. Eine analoge Strategie sollte auch für die übrigen Bausteine der Benutzerschnittstelle verfolgt werden. Dadurch besteht dann die Möglichkeit, die Vorteile der Wiederverwendbarkeit der Methoden durch Vererbung besser zu nutzen, sodaß zum Beispiel für die Anpassung des Menüs nur noch das Überschreiben zweier kleiner Methoden notwendig ist. Dies würde sich insoferne auch auf die Wartung von DOgMA selbst günstig auswirken, als eine Änderung der Benutzerschnittstelle momentan noch eine Neuimplementierung von M2HyperTextDocument nach sich zieht, wie der Autor am Ende der Implementierung von Modula-2-DOgMA erfahren mußte.

5.4 Schnittstelle C++ – Modula-2

Wie unter Abschnitt 5.2.2 beschrieben, mußte bei der Implementierung von Modula-2-DOgMA an das Werkzeug, welches in C++ implementiert worden war, ein Teil hinzugebunden werden, der in Modula-2 programmiert worden ist. Dieser Abschnitt beschreibt die verschiedenen Möglichkeiten, diese Anbindung durchzuführen und die Funktionalität der jeweils anderen Sprache zu benutzen.

5.4.1 Modula-2-Aufrufe in C++

Ein direkter Aufruf von Modula-2-Prozeduren von C++ aus ist nicht möglich, der Grund dafür wird in Abschnitt 5.4.4 dargestellt. Modula-2-Code kann nur von C aus aufgerufen werden, dieser C-Code wiederum kann von C++ verwendet werden. Der dabei zu verwendende Bezeichner ergibt sich nach folgendem Schema:

```
Modulname$Prozedurname
```

So wird die Modula-2-Funktion

```
PROCEDURE GetInt(): INTEGER;
```

von C aus folgendermaßen aufgerufen:

```
int getInt()
{
  return ModInt$GetInt();
}
```

Um eine wohldefinierte Schnittstelle zu erhalten, empfielht es sich, alle Modula-2-Aufrufe in einer C-Datei zu kapseln. Diese kann dann mit den in [Wei89] beschriebenen Techniken in das C++-System eingefügt werden.

5.4.2 C++-Aufrufe in Modula-2

Der Aufruf von C++-Funktionen von Modula-2-Code aus ist etwas schwieriger als der umgekehrte Fall. Es existieren drei Möglichkeiten. Sie werden im folgenden auf ihre Eignung zum Einsatz für die Anbindung an C++ untersucht.

5.4.2.1 CCALL

Die einfachste Möglichkeit ist die Benutzung der Modula-2-Funktion CCALL, die vom Modul SYSTEM exportiert wird. Die Funktion hat folgendes Aufrufmuster:

```
CCALL(functionname {, parameter});
```

wobei functionname der Name der gewünschten C-Funktion ist. Der Rückgabewert entspricht dem Rückgabewert der C-Funktion, kann aber ignoriert werden, d.h. CCALL kann auch wie eine Prozedur verwendet werden. Der folgende Code ruft die C-Funktion printf auf, die die an sie übergebenen Parameter auf den Ausgabestrom schreibt und die Anzahl der ausgegebenen Zeichen als Funktionswert zurückgibt:

```
length := CCALL("printf", "This is an integer: \%d",
                12345);
```

Der Aufruf von C-Funktionen durch CCALL hat aber mehrere gravierende Nachteile:

- Es können *alle* C-Funktionen an jeder Programmstelle ohne Einschränkungen benutzt werden, sodaß keine eindeutig abgegrenzte Schnittstelle existiert.
- Die fehlende Schnittstellendefinition ist aber nicht nur wegen der fehlenden Übersichtlichkeit und Klarheit des Programmaufbaus ein großes Manko. Da der Modula-2-Compiler keinerlei Information über die aufzurufenden C-Funktionen besitzt, können bei der Übersetzung weder die Anzahl noch die Art der übergebenen Parameter überprüft werden. Wenn die angegebenen Parameter bezüglich Anzahl oder Typ nicht mit der aufgerufenen C-Funktion übereinstimmen, treten die daraus entstehenden Fehler erst bei der Ausführung des Programmes in Erscheinung.
- Es können nur C-Funktionen angesprochen werden, nicht jedoch Methoden von C++-Objekten.

Aufgrund der fehlenden Schnittstellenbeschreibung sollte die Verwendung von CCALL-Aufrufen auch für den Fall, daß C-Funktionen angesprochen werden sollen, unterlassen werden.

5.4.2.2 DEFINITION FOR C MODULE

Einen Teil dieser Nachteile beseitigt der "DEFINITION FOR C MODULE". Dieser gleicht, bis auf das "FOR C" im Titel, einem normalen Modula-2-Definitionsmodul, wobei aber kein Implementierungsmodul existiert, da sich die angegebenen Funktionen im C-Teil des Systems befinden.

Folgender Modul, der aus [Sun86] entnommen wurde, zeigt einen "DEFINITION FOR C MODULE", der die C-Funktion "printf" für Modula-2 verfügbar macht.

```
DEFINITION FOR C MODULE CIOexample;

  PROCEDURE printf ( format:ARRAY OF CHAR; ..);

END CIOexample.
```

"printf" übernimmt als Parameter einen String und beliebig viele weitere Parameter, die im "DEFINITION FOR C MODULE" durch zwei Punkte (..) dargestellt werden.

Aber auch diese Art, C-Funktionen aufzurufen ist mit Mängeln behaftet:

- Mit dem "DEFINITION FOR C MODULE" existiert eine Schnittstellenbeschreibung, sodaß die C-Aufrufe im übrigen Programm gegen diese Schnittstelle geprüft werden können, die Schnittstellenbeschreibung selbst bleibt aber ungeprüft.
- Der Nachteil, daß nur C-Funktionen, nicht aber C++-Methoden angesprochen werden können, besteht weiter.

Der "DEFINITION FOR C" MODULE stellt, unter Berücksichtigung der oben angeführten Mängel, eine passable Möglichkeit zum Aufruf von C-Funktionen dar.

5.4.2.3 Prozedurvariablen

Somit verbleibt nur die dritte Alternative, die Verwendung von Prozedurvariablen, die etwas komplizierter zu handhaben ist, dafür aber die Möglichkeit bietet, C++-Methoden anzusprechen.

Als erster Schritt müssen "Prozedurschablonen" erstellt werden. Für jede C++-Methode wird sowohl im C- als auch im Modula-2-Teil je ein entsprechender Prozedurtyp definiert.

Als Beispiel soll das Objekt "TestObj" dienen, welches eine Methode namens printValue besitzt, die einen übergebenen integer-Wert ausdruckt. In der C-Datei, die die Schnittstelle implementiert, muß die folgende Typdefinition durchgeführt werden:

```
typedef void (Object::*PrintValueProc)(int);
```

womit eine Methode namens PrintValueProc definiert wird, die einen Parameter vom Typ int und keinen Rückgabewert hat. Auf der Modulaseite steht dementsprechend in dem Schnittstellenmodul folgende Deklaration:

```
TYPE PrintValueProc = PROCEDURE(ADDRESS, INTEGER);
```

wobei ADDRESS für einen Zeiger auf das Objekt, zu dem die Funktion gehört, steht und INTEGER für den zu übergebenden Parameter.

Um eine solche Prozedurvariable benutzen zu können, werden einerseits der Zeiger auf das Objekt, zu der die Methode gehört, und andererseits ein Zeiger auf die Methode selbst benötigt. Beim Aufruf der Modula-2-Prozedur, die PrintValueProc benutzt, der in diesem Fall aus C++ heraus über die Prozedur ITintPrintInt erfolgt, werden diese Informationen übergeben. Die Prozedur hat folgende Definition:

```
extern void ITIntPrintInt(
              void *obj,
              PrintValueProc myPrintValueProc,
              int value);
```

Dabei ist das erste Argument ein Zeiger auf das Objekt, zu dem die Methode gehört, das zweite Argument ein Zeiger auf die Methode selbst und das dritte

Argument der auszugebende Integerwert. Der Aufruf dieser Prozedur hat folgendermaßen zu erfolgen:

```
ITintPrintInt(&myObj, &TestObj::printValue, value);
```

Die entsprechende Modula-2-Prozedur hat folgende Schnittstelle:

```
PROCEDURE ModIntPrintInt(
            obj: ADDRESS;
            myPrintValueProc: PrintValueProc;
            value: INTEGER);
```

Diese Prozedur enthält nur den Aufruf der C++-Methode:

```
myPrintValueProc(obj, value);
```

Wenn ein und dieselbe Methode ein und desselben Objektes wiederholt aufgerufen wird, ist es sinnvoller, den Zeiger auf das Objekt und die entsprechende Prozedurvariable nur einmal mit einer eigenen Initialisierungsprozedur zu übergeben, wobei deren Parameter dann von globalen Variablen im Modula-2-Teil gespeichert werden müssen.

Auf diese Weise ist es möglich, alle benötigten Aufrufe mit einer exakt definierten Schnittstelle zur Verfügung zu stellen. Der Nachteil dieser Variante besteht in dem großen Schreibaufwand, der notwendig ist, um diese Schnittstelle zu realisieren und neue Funktionen in diese einzufügen.

5.4.3 Integration

Um das aus C++- und Modula-2-Code bestehende Programmsystem zu übersetzen benötigt man zusätzlich noch ein entsprechendes makefile. Das makefile dient als Steuerungsdatei für das Unix-Kommando make. Dieses benutzt die Informationen des makefile, um die Übersetzung und das Binden von Programmsystemen durchzuführen. In diesem makefile müssen auch alle Dateien, die eine besondere Behandlung benötigen, aufgeführt werden.

In unserem Fall sind solche Dateien die Schnittstellenmodule, da diese mit dem C- und nicht mit dem C++-Compiler zu übersetzen sind (siehe Abschnitt 5.4.4). Weiters muß für jedes einzelne Modula-2-Modul genau die Übersetzungsanweisung und die Module, von denen das zu übersetzende Modul abhängt, angegeben werden. Das Erstellen dieser Einträge ohne Werkzeugunterstützung ist zwar für ein kleines System noch einfach, in einem größeren System kann es aber durch falsch eingetragene oder veraltete Abhängigkeiten zu interessanten Fehlern oder Programmabstürzen kommen. Für die C/C++-Dateien sind diese detaillierten Angaben nicht erforderlich.

Für ET++-Applikationen wird zum Binden üblicherweise eine Kommandoprozedur namens "etld" verwendet. Diese Kommandoprozedur kann jedoch für gemischte Programmsysteme wie Modula-2-DOgMA nicht ohne Änderungen verwendet werden. Folgende Anpassungen waren vorzunehmen:

- Der Modula-2-Teil muß zu Beginn der Programmausführung initialisiert werden. Dazu muß in der main-Funktion des C++-Teiles eine Funktion namens m2_init aufgerufen werden, die während der Übersetzung vom Modula-2-Compiler (m2c) erzeugt und vom Modula-2-Binder (m2l)[3] eingebunden wird. Aus diesem Grund muß das *gesamte* Programmsystem vom Modula-2-Binder gebunden werden, weshalb nicht das Standard-Bindungsverfahren, wie es durch etld implementiert wird, verwendet werden kann, da dieses den Standard-Unix-Binder "ld" benutzt.

[3] m2l selbst wird allerdings nie direkt, sondern immer über m2c aufgerufen

- In dem modifizierten etld muß angegeben werden, welches Modula-2-Modul als Schnittstellenmodul zum C++-Code hin zu verwenden ist. Dies bedeutet, daß für jede Applikation eine eigene Version des modifizierten etld's erstellt werden muß.

- Schließlich müssen noch die beim Binden zu verwendenden Bibliotheken angegeben werden. Dieser Punkt ist insoferne kritisch, da die Reihenfolge, in der die Bibliotheken angegeben werden, beim Binden eine wichtige Rolle spielt. Schon ein geringes Abändern dieser Reihenfolge kann entweder zu interessanten Fehlermeldungen oder – allerdings bedauerlicherweise seltener – zu einem richtig gebundenem Programm führen. Nach den während der Implementierung gemachten Erfahrungen waren beim Binden folgende Bibliotheken in dieser Reihenfolge zu verwenden:

```
edittools.a libtools.a libsuntool.a
libsunwindow.a libpixrect.a libm.a UtiLib.a
libl.a -Bstatic libC.a libm.a libc.a
```

5.4.4 Weiterführende Erläuterungen

Wie unter Abschnitt 5.4.1 ausgeführt worden ist, kann Modula-2-Code nicht direkt von C++-Code aus angesprochen werden. Der Grund dafür liegt in der Weise, wie Modula-2-Prozeduren, wenn sie als Objektcode vorliegen, identifiziert werden.

Bei der Übersetzung wird aus Prozedurname und Modulname nach folgendem Schema das Symbol erzeugt, unter dem der Objekt-Code angesprochen werden kann:

```
MODULE ModuleName;

PROCEDURE ProcedureName;
```

ergibt das Symbol

```
_ModuleName$ProcedureName
```

Die Generierung von Funktionssymbolen erfolgt analog. Das führende Unterstreichungszeichen (_) wird vom C/C++-Compiler automatisch erzeugt, es wird auch bei den für den C/C++-Code erzeugten Bezeichnern angefügt. Der Grund, warum nun der C-Compiler verwendet werden muß ist der, daß in C das Dollarzeichen ($) ein legaler Bestandteil des Bezeichners ist, nicht jedoch in C++. Daß die Schnittstellendatei mit dem C-Compiler übersetzt werden muß, muß in der make-Datei (siehe auch Abschnitt 5.4.3) entsprechend eingetragen werden.

5.4.5 Beispiel

Im Zuge der Implementierung von Modula-2-DOgMA wurde ein Beispielsystem für die verschiedensten Arten von Aufrufen zwischen den Sprachen Modula-2 und C++ erstellt. Der Grund dafür war der Wunsch, die Schnittstelle ohne Behinderung durch Seiteneffekte, die vom übrigen Code des Werkzeugs verursacht werden, und ohne die langen Übersetzungszeiten untersuchen zu können. Dieses Testsystem soll auch anderen als Einstiegshilfe in diese doch nicht ganz alltägliche Problematik dienen. Eine exakte Beschreibung der Vorgehensweise und der Bedeutung der einzelnen Anweisungen wurde als Kommentare in die entsprechenden Dateien eingefügt.

Bei der Erstellung des Testsystemes wurde versucht, alle möglichen und sinnvollen Aufrufarten zu verwenden. Die Anbindung von C++-Code an Modula-2-Code, d.h. die Hauptfunktion (analog zum C/C++ main) liegt im Modula-2-Code, wurde nicht untersucht.

Das Testsystem besteht aus folgenden Dateien:

ITmain.c ist die Hauptdatei mit der main-Funktion und einem C++-Objekt, welches vom Modula-2-Teil aus benutzt wird.

ITint.h enthält die Deklaration der Aufrufprozeduren von ITint.c sowie die Deklaration der Prozedurvariablen, die die aufzurufenden C++-Methoden beschreiben. Sowohl ITint.c als auch ITint.h sind C-Dateien.

ITint.c implementiert die eigentlichen Aufrufe der Modula-2-Prozeduren.

ModInt.def implementiert die Schnittstellenbeschreibung auf der Modula-2-Seite.

ModInt.mod implementiert die Modula-2-Prozeduren.

makefile enthält die Informationen zur Generierung sowohl des C++- als auch des Modula-2-Teiles.

myetld dient zum Binden und wird vom makefile aufgerufen.

5.4.6 Abschließende Bemerkungen

Wenn man Modula-2-Code von C++ aus aufrufen will, sollte man sich vor der Erstellung der Schnittstelle folgende Fragen stellen:

- Wird diese Schnittstelle wirklich benötigt?
  Wie schon unter Abschnitt 5.2.2 ausgeführt worden ist, wäre bei Modula-2-DOgMA eine Implementierung der Funktionalität des Modula-2-Teiles in C++ zeitlich wesentlich günstiger gewesen, besonders wenn auch der zeitliche Aufwand für die Meisterung der Schnittstelle in Betracht gezogen wird. Aber diese Frage kann oft, wie z.B. in diesem Fall, erst nach Beendigung der Arbeit beantwortet werden. Einen weiteren Grund, auf eine solche Schnittstelle zu verzichten, geben Sun Microsystems zu Beginn des Kapitels 13 mit dem Titel "Using Modula-2 with Other Languages" der Modula-2-Dokumentation [Sun86]:

  > Caution:

> You may encounter difficulties in porting programs written using the features in this chapter to other Modula-2 environments, *including those on future Sun systems*. Encapsulate the use of these features carefully to allow for convenient future program maintenance.[4]

- Wie soll die Schnittstelle gestaltet werden?
  Möglichst einfach.
  Darunter soll in diesem Zusammenhang einerseits die Verwendung möglichst einfacher Datentypen und andererseits eine überschaubare Anzahl von Schnittstellenprozeduren verstanden werden. Dies ist besonders insoferne wichtig, als sich die Schnittstellenprozeduren in *einem* Modul befinden müssen, welcher bei vielen Prozeduren unübersichtlich wird.

- Wie soll die Schnittstelle gekapselt werden?
  Auf der Modula-2-Seite muß man ein Modul erstellen, das sämtliche Schnittstellen-Prozeduren, auf die der C-Teil zugreifen soll, enthält. Dies ist, neben der logischen Kapselung, schon deshalb notwendig, da beim Binden dieses eine Modul angegeben werden muß. Die Aufrufe dieser Schnittstellenprozeduren dagegen können über mehrere Modula-2-Module verteilt werden.

  Auf der C++-Seite erstellt man zuerst das in reinem C zu implementierende Schnittstellenmodul. Über dieses kann dann bei Bedarf noch eine C++-Klasse "herumgebaut" werden. Dies wurde für DOgMA jedoch unterlassen: die Schnittstelle wird nur von einer einzigen Klasse verwendet und die Wahrscheinlichkeit, daß diese Schnittstelle jemals von einer anderen Klasse benutzt wird, dürfte nicht sehr hoch sein. Weiters existiert kein Potential, eine (sinnvolle) abstrakte Oberklasse für solche Schnittstellen zu erzeugen, da diese zu applikationsspezifisch sind.

[4] Hervorhebung durch den Autor

5.5 Zeitlicher Ablauf der Implementierung

Dieser Abschnitt beschreibt chronologisch, wie bei der Implementierung vorgegangen wurde. Bis auf die ersten beiden Phasen basieren die Ausführungen auf der Auswertung eines Arbeitstagebuches, welches während der Implementierung mit Hilfe einer einfachen Kommandoprozedur und eines Editors geführt wurde.

Phase 1: Einführendes Literaturstudium

Die erste Phase diente dem theoretischen Kennenlernen des Betriebssystems Unix, der Programmiersprachen C und C++, der objektorientierter Programmierung und Application Frameworks.

Phase 2: Einarbeiten in Unix

Dem theoretischen Einlesen folgte das praktische Einarbeiten am Rechner mit Unix und der Benutzeroberfläche SunView. Unix bereitet jemandem, der schon lange mit MS-DOS arbeitet und dadurch einen Großteil der Befehle und die allgemeine Vorgangsweise kennt, keine Schwierigkeiten.

Umstellungsschwierigkeiten könnten sich wegen der Benutzeroberfläche und durch die Verwendung einer Drei-Tastenmaus ergeben, sowie durch die Tatsache, daß ein Verschieben der Maus den Eingabestrom in ein anderes Fenster umleitet, ohne daß dieses Fenster explizit aktiviert werden muß.

Während der praktischen Arbeit stellte sich heraus, daß die Kombination aus graphischer Benutzeroberfläche und Befehlseingabe durch Kommandozeilenschnittstelle sehr flexibel ist und durch einen erfahrenen Benutzer wesentlich besser an die spezifischen Bedürfnisse angepaßt werden kann als eine rein graphische Oberfläche. Für einen "naiven" Benutzer ist allerdings eine rein graphische Benutzeroberfläche, wie sie zum Beispiel vom Macintosh angeboten wird, sicher besser geeignet.

Phase 3: Kennenlernen von DOgMA und C++

Die Bedienung von DOgMA ist intuitiv leicht verständlich und bereitete bei der Einarbeitung keine Schwierigkeiten.

Größere Schwierigkeiten hingegen bereitete die Suche nach der für die vorzunehmenden Anpassungen interessanten Schnittstelle zwischen dem sprachabhängigen und dem sprachunabhängigen Teil. Zuerst ist das Einlesen in ein Programmsystem mit vielen, umfangreichen Klassen sehr verwirrend – ohne gute Kenntnisse über den Aufbau und die Verwendung von Application Frameworks und ohne zusätzliche Erklärungen ist der Code kaum verständlich.

Die ersten C++-Versuche waren von den üblichen Schwierigkeiten begleitet: kryptische Fehlermeldungen, fehlende Pfadangaben, unerklärliche Abstürze, ...

Phase 4: Abändern von ReadProject

Zuerst wurden kleine Veränderungen vorgenommen, zum Beispiel das Anpassen der About-Meldung, und im Zuge dieser Änderungen die Dateien, die die Übersetzung und Bindung steuern an Modula-2-DOgMA angepaßt.

Nachdem der Arbeitsbereich, die Klasse ModulaProject – vormals CProjekt – isoliert worden war, wurde Methode für Methode zuerst bezüglich ihrer Aufgabe und Funktionsweise analysiert und dann für Modula-2 abgeändert. Dabei waren bei der Schnittstelle der Klasse und den Datenstrukturen keine Anpassungen notwendig, lediglich im internen Aufbau mußten Änderungen vorgenommen werden.

Das erste Ziel bestand darin, eine einzelne Datei korrekt einlesen zu können. Dazu war es hauptsächlich notwendig, die in der Datei enthaltenen Prozeduren zu erkennen und richtig zu verarbeiten. Um dies zu erreichen mußten die Kommentare korrekt abgearbeitet, d.h. überlesen, und nur die signifikanten Zeichen behandelt werden. Danach konnten Beginn und Ende einer bestimmten Prozedur erkannt und die so erhaltene Prozedur in die entsprechende Datenstruktur einfügt werden (siehe auch Abschnitt 5.2.1).

Anschließend mußte dafür gesorgt werden, daß die einzelnen Modultypen richtig erkannt und verarbeitet werden. MODULE und IMPLEMENTATION MODULE werden dabei gleich behandelt, während ein Definitionsmodul eine andere Behandlung der Prozeduren notwendig macht, da in diesem Fall nur der Verweis auf eine Prozedur, nicht aber der Prozedur(programm)-text vorliegt.

Nun konnte eine einzelne Datei korrekt eingelesen werden und die letzte Aufgabe dieser Phase bestand darin, die Importanweisungen richtig zu verarbeiten. Hierbei entstand auch das Projektmodul und die verschiedenen Formen der Importe (siehe Abschnitt 4.1)

Bei allen zuvor angeführten Änderungen kann gesagt werden, daß sie großteils zu einer Vereinfachung des Codes in Bezug auf Komplexität führten, was auf die – nicht nur für diese Zwecke – besser geeignete Sprachstruktur von Modula-2 zurückzuführen ist.

Bei der praktischen Anwendung von C++-DOgMA offenbarten sich einige Fehler, so verhielt sich das Werkzeug beim Anlegen neuer Hypertexte instabil, die für die neue Version korrigiert wurden.

Phase 5: Anpassen der Oberfläche

Hierbei mußte hauptsächlich die Klasse HyperTextDocument angepaßt werden. Zuerst wurden die Änderungen versuchsweise direkt im Quelltext von HyperTextDocument vorgenommen, aufgrund der dabei gewonnenen Erkenntnisse wurde eine eigene Unterklasse von HyperTextDocument namens M2HyperTextDocument erstellt. Bei der Implementierung dieser Klasse stellte sich heraus, daß sich die Vererbung nicht immer gut ausnützen ließ, da teilweise große Teile unnötig kopiert werden mußten, damit die notwendigen Änderungen vorgenommen werden konnten. Die Änderungsvorschläge wurden im Abschnitt 5.3 aufgeführt.

Weiters mußten die ebenfalls in M2HyperTextDocument enthaltenen Prozeduren InsertNewProcedure und InsertNewFile angepaßt werden, die neue Hypertexte in das allgemeine Datensystem einfügen und diese mit einem durch Umgebungs-Variablen einstellbaren Standardtext versehen.

Phase 6: Schnittstelle C++ – Modula-2

Um nicht durch Seiteneffekte unnötigt abgelenkt zu werden, wurde ein einfaches Beispielprogramm erstellt, das auch anderen als Anleitung bei der Meisterung der Schnittstelle dienen soll. Das Übersetzen des Programmes bereitete keine Schwierigkeiten – nachdem der Modula-2-Compiler mehrmals neu installiert worden war – wohl aber das Binden. Durch die vereinten Kräfte der Herren Dr. Johannes Sametinger, Dipl.-Ing. Rainer Weinreich und Dipl.-Ing. Stefan Schiffer konnten in vielen Versuchen nach und nach alle Fehler ausgemerzt werden. Die Hauptfehlerquelle stellten die Bibliotheken dar, die sich entweder an der falschen Stelle befanden, in der falschen Reihenfolge angesprochen wurden oder sich schlußendlich doch als die falschen Bibliotheken herausstellten. Schließlich konnten die für die Übersetzung notwendigen Programmdateien und Bibliotheken an einer Stelle zusammengezogen werden.

Nun konnte mit der Erstellung des Beispielprogrammsystem begonnen werden, welches alle möglichen Aufrufrichtungen (von C++ nach Modula-2 und umgekehrt) und alle Arten der Parameterübergabe (VAL, VAR und als Funktionswert) sowie eine umfangreiche Kommentierung umfaßt (siehe auch Abschnitt 5.4.5).

Phase 7: ParseProject – der C++ Teil

Der nächste Schritt bestand nun darin, dem Parser die Module in der richtigen Reihenfolge vorzugelegen, sodaß die importierten Module rechtzeitig – also vor der Benutzung ihrer Bezeichner – geparst werden.

Um dies zu erreichen, wurde zuerst eine Methode implementiert, die die Modulhierarchie ausdruckt, die dann soweit angepaßt wurde, daß sie jeden Modulnamen nur einmal ausdruckte – jedes Modul muß genau einmal geparst werden.

Phase 8: ParseProject – der Scanner

Zuerst wurden die einzelnen Dateien des SCT-Systems gesichtet, da dessen Compiler isoliert und für Modula-2-DOgMA verwendet werden sollte. Es stellte sich heraus, daß es unmöglich war, die für das Werkzeug interessanten Dateien einfach herauszutrennen, da der Parser eng mit der Codegenerierung verzahnt ist. Deshalb wurde zuerst versucht, den Scanner alleine an DOgMA anzuhängen. Dieser war relativ einfach zu isolieren und benutzt nur wenige andere Module. Aufgrund der Abhängigkeiten zwischen den Modulen wurde das makefile erstellt, da das Kommando make Modula-2-Dateien im Gegensatz zu C- und C++-Dateien nicht automatisch abarbeiten kann (siehe auch Abschnitt 5.4.3).

Die Hauptaufgabe dieser Phase bestand einerseits darin, den Scanner-Code zum C++-Code zu binden und andererseits dafür zu sorgen, daß der Scanner alle Symbole eines Moduls korrekt erkennt und an den noch zu implementierenden Parser liefert.

Der Einbau der C++- und Modula-2-Schnittstelle erfolgte zuerst ohne Probleme. Als jedoch nach einer längeren Unterbrechung die Arbeit wieder aufgenommen wurde, gab es eine böse Überaschung, da inzwischen das Dateisystem abgeändert worden war, sodaß mehrere Dateien bzw. Verzeichnisse nicht gefunden wurden, was ein Binden unmöglich machte. Nachdem nun wirklich alle benötigten Bibliotheken an einer Stelle versammelt waren, mußten nur mehr kleinere Programmierfehler – wobei die C-Fehlermeldungen entweder keine Hilfe darstellten oder überhaupt nicht erfolgten – und fehlende Initialisierungen behoben werden, bis der Scanner probeweise angesprochen werden konnte.

Der Scanner mußte nun so angepaßt werden, daß er in der Lage war, das Modul selbsttätig zu re-linearisieren, also die Aufteilung der Modul-Datei in einzelne Hypertexte rückgängig zu machen. Nachdem der Scanner die Datei korrekt in die einzelnen Symbole zerlegen konnte, wurde die Behandlung der Kommentare und Schlüsselwörter implementiert, was keine Schwierigkeiten bereitete.

Phase 9: ParseProject – der Parser

Zuerst wurde versucht, den Parser ebenso wie den Scanner zu isolieren und dann in das Gesamtsystem einzufügen. Da dieser aber sehr eng mit den anderen Modulen verwoben war, drängte sich die Alternative auf, den Parser selbst zu implementieren. Die Gründe, die dann den Ausschlag zugunsten der Selbstimplementierung gaben, sind in Abschnitt 5.2.2 beschrieben.

Zuerst wurde der Teil implementiert, der für die Behandlung der Bezeichner notwendig war, da dieser ebenfalls für das richtige Erkennen der Kommentare und der Schlüsselwörter benötigt wurde.

Dann wurde der eigentlichen Parser erstellt, wobei die EBNF-Darstellung von Modula-2 (vergleiche [Wir91]) als Vorlage für die Grobstruktur herangezogen wurde. Dabei wurden einige Fehler bei dem Aufspalten der Module und deren Prozeduren in die jeweiligen HyperTexte bzw. deren Zuordnung untereinander sichtbar, die sich jedoch relativ einfach beheben ließen. Die weitere Ausgestaltung des Parsers und der Bezeichnerverwaltung wurde durch immer wieder neu zu Bewußtsein kommende Feinheiten von Modula-2 – Zeigerdereferenzierungen, damit verbunden die Frage der Datentypen, Vorwärtzreferenzen (Prozeduren und bei der Zeigerdeklaration), Qualifizierungen, Felder, WITH-Anweisungen, Exporte von Enumerationsbezeichnern, ... – immer komplexer, wirklich hartnäckige Probleme traten aber nicht auf.

Phase 10: Abschlußarbeiten

Nun wurde noch der sprachabhängige Teil in eine neue DOgMA-Version integriert. Die für diese Aufgabe relevanten Änderungen zwischen den beiden DOgMA-Versionen – die Namen von zwei Bezeichnern hatten sich geändert – waren gering, sodaß die endgültige Version schnell fertiggestellt werden konnte.

Die neue DOgMA-Version bietet die interessante Möglichkeit, zwischen implementierten und nicht implementierten Prozeduren bei der Anzeige in den Auswahllisten zu unterscheiden. Bei Modula-2-DOgMA können nun also die Prozeduren, für die die dazugehörige Implementierung nicht geladen wurden, schnell identifiziert werden. Ansonsten waren dieselben Änderungen durchzuführen, die in Phase 5 beschrieben wurde.

Weiters wurde die Klasse M2RemarkDialog eingefügt, die die Anzeige der Dialogbox, die die Informationen über einen Bezeichner anzeigt, an Modula-2 anpaßt.

Schließlich wurde die Dokumentationsfunktion aktiviert, was wiederum ohne Schwierigkeiten durchgeführt werden konnte.

Kapitel 6

Bewertung von DOgMA

Dieses Kapitel beschäftigt sich mit der Bewertung des Werkzeuges DOgMA. Nach einer allgemeinen Würdigung wird auf die Punkte eingegangen, bei denen sich während der praktischen Arbeit Potentiale zur Verbesserung der Leistungsfähigkeit gezeigt haben. Der letzte Abschnitt beschäftigt sich mit der weiteren Entwicklung des Werkzeuges.

6.1 Würdigung

DOgMA ist sowohl in der C++- als auch in der Modula-2-Version ein gutes Werkzeug zur Unterstützung der täglichen Programmier- und Wartungsaufgaben. Es sollen nun einige besonders bemerkenswerte Punkte nochmals herausgestrichen werden.

- DOgMA vereint die beiden Konzepte Hypertext und Literate Programming. Jedes für sich bietet eine Verbesserung des Verständnisses bestehenden Quellcodes. Ihre Vereinigung bringt zusätzliche Synergieeffekte (vergleiche Abschnitt 2.3).

- DOgMA hilft dem Benutzer, sich das Programmsystem auf eine sehr intuitive und von zum Beispiel eingebauten Hilfesystemen bekannte Art und Weise zu erschließen. Dazu trägt wesentlich das Herabbrechen der teilweise großen Dateien in die kleineren, leichter verständlichen Informationseinheiten der Prozeduren sowie die Gruppierung dieser kleineren Bausteine nach logischen Kriterien bei.

- DOgMA verbessert die Information über ein Programmsystem durch gezielte Informationen über jeden im Programm vorkommenden Bezeichner. Auch dies erleichtert den Einstieg in ein fremdes Programm beträchtlich. Weiters stellt dieser Mechanismus ein wertvolles Hilfsmittel dar, wenn Bezeichner geändert werden sollen oder sich die Frage stellt, ob ein Bezeichner überhaupt noch verwendet wird.

- DOgMA ist einfach zu bedienen. Durch Verwendung einer modernen Benutzeroberfläche, deren Bedienung sich ähnlich wie die der meisten Programme unter einer modernen graphischen Benutzeroberfläche gestaltet, kann die Einarbeitungsphase mit dem Werkzeug auf ein Minimum reduziert werden.

- DOgMA erlaubt es dem Benutzer frei zu entscheiden, ob er die Dokumentation vor, parallel oder nach der Quellcodeentwicklung erstellt. Dadurch werden zu große Eingriffe in das Programmierverhalten vermieden. Ebenso ist es möglich, nicht mit DOgMA erstellte Programmsysteme nachträglich zu dokumentieren.

- DOgMA unterstützt den Benutzer bei der Erstellung der Dokumentation durch die verschiedenen Möglichkeiten, Quellcode in der Dokumentation zu verwenden. Dieser Quellcode wird automatisch nachgeführt, sodaß in der Dokumentation immer der aktuell gültige Code angeführt wird.

- DOgMA benutzt kein eigenes Format für die Quellcodedateien. Dadurch bleibt der Quellcode ohne Änderungen auch für andere Werkzeuge, wie zum Beispiel Compiler und Testhilfen, verwendbar. Ein weiterer Vorteil dieser Vorgehensweise liegt darin, daß bereits bestehende, nicht mit DOgMA erstellte Programmsysteme mit dem Werkzeug bearbeitet werden können. Lediglich das Projektmodul (siehe auch Abschnitt 4.1) muß erstellt werden.

- DOgMA erleichtert die Erstellung eines Programmsystemes nach dem Konzept des Literate Programming dadurch, daß weder Kommandos zur Formatierung der Dokumentation noch zur Verbindung von Code und Dokumentation eingegeben werden müssen. Der Benutzer muß also nicht zwei zusätzliche Programmiersprachen erlernen, sondern kann den gewünschten Effekt durch Anwendung von Vorgehensweisen, die von jeder Textverarbeitung her bekannt sind, erreichen. Durch den Wegfall dieser Programmiersprachen fällt natürlich auch die Möglichkeit weg, daß der Benutzer Fehler bei der Anwendung dieser Sprachen macht, was sich zusätzlich günstig auf den zeitlichen Aufwand auswirkt.

- DOgMA bietet im Gegensatz zum WEB-System jederzeit den aktuellen Stand sowohl der Dokumentation als auch des Codes an, ohne daß irgendeine Anstrengung seitens des Benutzers notwendig wäre.

- DOgMA erzeugt einen Großteil der Hypertext-Verbindungen automatisch, die übrigen Verbindungen können durch von Textverarbeitungen bekannten Vorgehensweisen vom Benutzer einfach und transparent hergestellt werden.

6.2 Verbesserungsvorschläge

Während der praktischen Verwendung von DOgMA offenbarten sich einige Ansatzpunkte für Verbesserungen. Es soll aber nachdrücklich darauf hingewiesen werden, daß DOgMA auch ohne die Realisierung dieser Vorschläge ein gutes und ausgesprochen wertvolles Werkzeug für den Programmieralltag darstellt.

Die Vorschläge zu Modifikationen, die weniger die Benutzung von DOgMA als vielmehr den internen Aufbau betreffen, wurden in Abschnitt 5.2.2 und 5.3 beschrieben.

6.2.1 Realisierte Modifikationen

Bei der praktischen Arbeit mit den beiden DOgMA-Versionen offenbarten sich einige Möglichkeiten, die Benutzbarkeit des Werkzeuges zu verbessern. Einige dieser Vorschläge wurden schon aufgegriffen und sind bereits in der endgültigen DOgMA-Version für Modula-2 implementiert worden. Die wichtigsten dieser Änderungen sind folgende:

Menü bei Betätigung der rechten Maustaste

Bei den Sun- und Sparcworkstations werden Mäuse mit drei Tasten benutzt. Der Autor verwendet zwei verschiedene Benutzeroberflächen, SunView und das auf NeWS (Network Window System) basierende OpenWindows, die sich in Bezug auf die Zuordnung der Funktionen auf die Maustasten etwas unterscheiden. Die nachfolgend gemachten Aussagen beziehen sich auf die Benutzeroberfläche SunView, die während der Implementierung verwendet wurde. Hier dient die linke Maustaste zum Selektieren und Aktivieren von Objekten. Die mittlere Maustaste wird für spezielle Aufgaben benutzt, so zum Beispiel zum Ändern der Größe und der Position der Fenster. Die rechte Maustaste aktiviert ein zu der Applikation, in dessen Fenster sich der Mauszeiger befindet, gehöriges PopUp-Menü. Bei SunView-Applikationen ist es üblich, in einem solchen Menü die Funktionalität anzubieten, die beim Arbeiten mit der Applikation selbst benötigt wird.

In der ersten Version von DOgMA fehlte dieses Menü, alle Menüpunkte wurden wie zum Beispiel bei Windows für MS-DOS oder bei der Macintosh-Oberfläche im Applikationsmenü angeboten. Dies stellte bei der Benutzung von DOgMA einen Bruch der Erwartungskonformität bezüglich der Bedienung dar, der sich mit steigender Gewöhnung an die Benutzeroberfläche SunView immer unangenehmer bemerkbar machte. Deshalb wurde vom Autor der Vorschlag gemacht, die am häufigsten benutzten Menüpunkte in einem solchen, über die rechte Maustaste zu aktivierenden Menü zur Verfügung zu stellen.

In der neuen DOgMA-Version existiert nun ein solches Menü, welches die Menüpunkte für die Editierfunktionen und für die Dateisicherung anbietet. Für eine genaue Beschreibung siehe Abschnitt 4.2.5.

Virtuelle Klassendefinition

Die Anregung zu dieser Erweiterung von DOgMA basiert auf den Erfahrungen des Autors bei der Einarbeitung in objektorientierte Programmsysteme. Bei der Programmierung in Modula-2 existiert für einen Modul *eine* Schnittstelle, die die außerhalb des Modules verwendbare Funktionalität beschreibt. In der objektorientierten Programmierung ist diese Schnittstelle auf mehrere Stellen im Programmtext verteilt, da für das Verständnis einer Klasse auch die Klassen in die Betrachtung einbezogen werden müssen, die Funktionalität an die zu betrachtende Klasse vererben. Zusätzlich muß noch darauf geachtet werden, daß bei den Methoden, die, teilweise mehrfach, überschrieben werden, die richtige Methode betrachtet wird. Diese Fragmentierung bereitet besonders in der Einarbeitungsphase Schwierigkeiten.

Dem soll durch die Einführung der "virtuellen Klassendefinition" abgeholfen werden. Wenn nun im Hypertextfenster eine Klasse ausgewählt wird, werden nicht nur die in der Klasse implementierten Methoden angezeigt, sondern auch alle von den Oberklassen ererbten Methoden. Zur leichteren Unterscheidung werden diese Methoden in der Auswahlliste kursiv angezeigt (siehe 4.2.3). Wenn eine solche Methode ausgewählt wird, wird die jeweils der Klasse hierarchisch nächsthöhere Überschreibung der Methode angezeigt, wobei die Möglichkeit besteht, auch die durch hierarchisch höher gelegene Klassen vorgenommenen Modifikationen einfach zu begutachten.

Nach den Erfahrungen des Autors stellt diese Erweiterung eine Erleichterung in der besonders für die Wartung wichtigen Einarbeitungsphase dar. Es stellt sich auch die Frage, ob es nicht sinnvoll wäre, diese Art der Schnittstellenbeschreibung zu einem Teil der Sprachdefinition einer objektorientierten Programmiersprache zu machen.

Dieser Mechanismus der virtuellen Klassendefinition wird in DOgMA für Modula-2 dafür benutzt, um die Prozeduren, zu denen die Implementierung geladen worden ist, von denen zu unterscheiden, wo nur der Prozedurkopf bekannt ist.

6.2.2 Nicht realisierte Modifikationen

Neben diesen, bereits in die aktuelle DOgMA-Version eingeflossenen Modifikationen könnten folgende Veränderungen die Benutzbarkeit von DOgMA noch zusätzlich verbessern.

"Click and Branch"

Eines der Kernkonzepte von Hypertext besteht darin, daß durch Selektieren eines Objektes zu dessen textuellen oder graphischen Repräsentation verzweigt werden kann. Dies ist in DOgMA im Editor jedoch nicht direkt möglich. Um zu der Definition eines Bezeichner, im Falle einer Prozedur zu deren Programmkörper, zu verzweigen, muß zuerst der Bezeichner und danach der entsprechende Menüpunkte ausgewählt werden.

Vom Standpunkt des Bedienungskomfort wäre es angenehmer, daß durch zweimaliges Aktivieren eines Bezeichners das Informationsfenster (siehe Abbildung 4.20) angezeigt wird. Dieses Fenster sollte die Möglichkeit bieten, entweder zum Programmtext oder zu der entsprechenden Definition zu verzweigen.

Eine vergleichbare Funktionalität wurde für das Hypertextsystem Hyper-Ties (siehe [SK89]) implementiert und ist ebenso in den meisten Hypertext-Hilfesystemen zu finden. Der Einbau dieser Funktionalität erlaubt ein noch einfacheres, intuitiveres Navigieren im angezeigten Text.

Flexibles Einfügen neuer Prozeduren/Methoden

Wenn eine neue Prozedur durch Aufruf des Menüpunktes "New Procedure" in eine Datei eingefügt wird, dann wird diese neue Prozedur beim Abspeichern der Datei an deren Ende angehängt. Es wäre wünschenswert, beim Einfügen eine Stelle in der Datei angeben zu können, wo die Prozedur eingefügt werden soll, da man ansonsten die Datei mit einem normalen Texteditor nachbearbeiten muß.

Eine einfache Umgehungsmöglichkeit ist die, auf die Verwendung des Menüpunktes "New Procedure" zu verzichten und die Prozedur direkt in den Programmtext der entsprechenden Datei einzufügen. Diese Vorgehensweise hat aber den Nachteil, daß diese Prozedur erst beim nächsten Einlesen des Programmsystemes in die Auswahlliste der Prozeduren aufgenommen wird und kann daher nur als Notlösung angesehen werden.

Automatisches Anpassen der Anzeige der Prozedur- und Methodenköpfe

Wenn eine Methode oder Prozedur ausgewählt wird, kann nicht nur der Körper der Prozedur/Methode, sondern auch deren Kopf editiert werden. Änderungen im Prozedurkopf werden zwar richtig abgespeichert, die Anzeige des Prozedurkopfes in der Datei, in der die Prozedur implementiert wurde, sowie in den Auswahllisten wird aber erst beim nächsten Einlesen der Datei nachgeführt, sodaß optisch ein inkonsistenter Zustand entsteht. Durch ein Anpassen des Prozedurkopfes jeweils nach Verlassen einer Prozedur dürfte sich diese Unregelmäßigkeit vergleichsweise leicht beheben lassen.

Graphische Visualisierung

Die Benutzung der vorhandenen Graphikfähigkeiten zur besseren Visualisierung des Programmtextes beschränkt sich momentan auf die Verwendung unterschiedlicher Darstellungsstile zur Kennzeichnung verschiedener Codearten.

Da gerade eine graphische Aufbereitung der komplexen Zusammenhänge in einem Programmsystem sehr viel zum besseren Verständnis beitragen kann, sollten folgende Graphiken, zum Beispiel in Baumform, erstellt werden können:

- Hierarchiedarstellung von Klassen beziehungsweise Importgraph der Module: Die für diese Graphik notwendigen Daten sind im Datensystem vorhanden und können einfach extrahiert werden.
- Aufrufgraph von Methoden/Prozeduren: Die dafür benötigten Informationen sind nach Ausführung des Kommandos "Parse Project" verfügbar, obwohl zu deren Aufbereitung ein gewisser Aufwand notwendig sein dürfte.
- Inhaltsverzeichnis der Dokumentation: Die Dokumentation ist hierarchisch strukturiert, sodaß sich ebenfalls eine Baumdarstellung als Mittel der Visualisierung anbietet.

In einer weiteren Ausbaustufe sollte die Funktionalität der angezeigten Graphiken dahingehend erweitert werden, daß ein Anklicken der jeweiligen Baumknoten zu einem Verzweigen zu dem dazugehörigen Hypertext führt – ähnlich wie in der ET++ Entwicklungsumgebung (siehe [WGM89]). Ebenfalls sinnvoll erscheint die dort vorgeschlagene Möglichkeit, selektiv das Aussehen des Baumes durch Ein- beziehungsweise Ausblenden von Zweigen – ähnlich wie zum Beispiel bei der Gliederungsfunktion einer Textverarbeitung – zu beeinflussen.

Stärkere Orientierung an Benutzeroberfläche

DOgMA wurde nach den Standards von ET++ implemeniert. Die von ET++ angebotene Benutzeroberfläche gleicht aber weder in Benutzung noch in Aussehen derer von SunView, der Benutzeroberfläche, für die DOgMA ursprünglich implementiert wurde. Diese Unstimmigkeiten sind zwar nicht der Implementierung von DOgMA an sich anzulasten, machen sich aber besonders beim instensiven Arbeiten störend bemerkbar – so sind in DOgMA teilweise andere Mausknöpfe zur Aktivierung von bestimmten Funktionen vorgesehen als in SunView. Eine neuere Version von ET++ erhebt aber den Anspruch, dieses Problem wenigstens für die Benutzeroberflächen basierend auf Motif gelöst zu haben (siehe auch [WGM89]).

Eine Möglichkeit, diese unschöne Zweigleisigkeit zu bereinigen wäre die, alle wichtigen Werkzeuge mit der ET++-Oberfläche anzubieten. Da ein Dateieditor schon vorhanden ist, fehlt nur noch ein Kommandoprozessor, um ein konsistentes Arbeiten zu ermöglichen.

Stilinformationen

DOgMA bietet die Möglichkeit, zur besseren Visualisierung des Quellcodes für verschiedene Codearten, z.B. Kommentare, Schlüsselworte und so fort, unterschiedliche Darstellungsstile zu verwenden. Die Standardeinstellung dieser Stile kann vom Benutzer frei verändert werden (siehe Abschnitt 4.2.4.3). Dabei fehlt allerdings die Möglichkeit, diese Einstellungen abzuspeichern, sodaß der Benutzer bei jedem Aufruf des Werkzeuges die Stileinstellungen immer wieder neu seinen Wünschen entsprechend angeben muß, wenn er sich nicht mit der Standardeinstellung begnügt.

Das Abspeichern dieser Informationen in einer eigenen Datei oder deren Einstellung über Umgebungsvariablen sollte keine Probleme oder größeren Implementierungsaufwand mit sich bringen.

Einbindung von Graphik in die Dokumentation

Ein Punkt, der beim WEB-System kritisiert wird, ist der der fehlenden Möglichkeit, Graphiken einzubinden (vergleiche [Tun89] und [BC90]). Ein

Teil der Möglichkeiten wurde im Abschnitt "Graphische Visualisierung" dargestellt, wo hauptsächlich auf die von DOgMA automatisch erstellbaren Graphiken eingegangen wurde. Zusätzlich könnte aber die Qualität der Dokumentation durch direkt in diese eingefügte Graphiken deutlich verbessert werden.

Im Programmpaket von ET++ ist als Beispielprogramm ein einfaches Zeichenprogramm enthalten (siehe [WGM89]). Dieses könnte als hinreichend gute Ausgangsbasis dienen, um damit erzeugte Graphiken in die Dokumentation einbinden zu können. Die angebotene Funktionalität reicht aus, um verschiedene Hierarchien, Datenflüsse oder sonstige Informationen in geeigneter Weise graphisch aufzubereiten.

Zusätzlich wäre noch die Möglichkeit wünschenswert, Rastergraphiken in die Dokumentation einbinden zu können. Diese Rastergraphiken entstehen in diesem Zusammenhang meist dann, wenn ein Teil des Bildschirmes in eine Datei geschrieben wird. Auf diese Weise sind alle Abbildungen des Kapitels 4 entstanden. Die Aufnahme solcher Bilder in die Dokumentation könnte dazu dienen, Zustände der Applikation nicht nur von innen heraus verbal zu beschreiben, sondern zusätzlich durch Einfügen einer solchen Graphik auch aus der Sicht des Benutzers der Applikation zu verdeutlichen.

Ausdrucke

DOgMA bietet gegenwärtig keine Möglichkeit, das Programmsystem und die dazugehörige Dokumentation auszudrucken. Dies ist bei den Quellcodedateien insoferne nicht sehr störend, als diese mit den herkömmlichen Betriebssystemkommandos hinreichend gut ausgedruckt werden können. Die Dokumentation hingegen enthält die Verweise auf den Quellcode und auf andere Abschnitte der Dokumentation zwar in einer für Menschen lesbaren Form, die das Verständnis durch die Notwendigkeit, zwei Dokumente – einerseits die Dokumentation und andererseits den dazugehörigen Quellcode – parallel zu lesen sicher nicht fördert. In dieser Beziehung wäre es wünschenswert, diese Querverweise aufzulösen, um einen Darstellungsweise ähnlich des

WEB-Systemes (siehe [Knu86]) zu erhalten.

Die Darstellungsweise der Dokumentation auf Papier ist der Darstellungsweise von DOgMA am Computer in Bezug auf die Verständlichkeit und der Flexibilität der Navigierung im Dokument deutlich unterlegen, manchmal kann aber auch eine Dokumentation auf Papier notwendig sein – und sei es nur, weil ein Auftraggeber zusätzlich auf einer solchen besteht oder auch aus Sicherheitsüberlegungen.

Diese Möglichkeit des Ausdruckens der Dokumentation hätte sich auch auf die Erstellung dieser Diplomarbeit insoferne günstig ausgewirkt, als zumindest die Kapitel 4 und 5 mit Hilfe von DOgMA hätten erstellt werden können. Dadurch wäre auch ein umfangreiches Beispiel der Benutzung der Dokumentationsfunktion entstanden. Da sich aber die von DOgMA erstellte Dokumentation nicht für die Erstellung der Diplomarbeit verwenden ließ, mußte diese Vorgehensweise aus Zeitgründen unterbleiben.

Lesezeichen

Manche Hypertextsysteme, wie zum Beispiel das Hilfesystem "Answerbook" auf Unix (vergleiche [Bro92]), bieten dem Benutzer die Möglichkeit, gewisse Textstellen mit einem "Lesezeichen" zu versehen. Dieses Lesezeichen besteht aus einem frei definierbarem Text, mit dessen Hilfe der Benutzer unmittelbar zu dem damit verbundenem Textstück gelangen kann.

Dieser Mechanismus könnte auch für DOgMA eine wichtige Erweiterung darstellen, da nunmehr der Benutzer unmittelbar die Stellen, an denen er arbeitet oder die Stelle von wichtigen Definitionen ansteuern kann, ohne durch das restliche Programmsystem navigieren zu müssen.

6.3 Tendenzen der Weiterentwicklung von DOgMA

Einer der Nachteile von DOgMA in der jetzigen Implementierung besteht einerseits darin, daß sich der gesamte Quellcode des Programmsystems und die gesamte Dokumentation im Hauptspeicher befinden und daß andererseits das Laden des Systemes einiges an Zeit benötigt. Dieses Problem wird in einem Prototyp einer neuen DOgMA-Version insoferne gelöst, als *einmal* eine Datenbank mit den benötigten Informationen aufgebaut wird, die dann beim Einlesen benutzt wird. Dadurch muß auch nicht mehr das gesamte Programmsystem im Hauptspeicher vorhanden sein – es wird der entsprechende Text bei Bedarf sehr schnell nachgeladen. Die Wartezeit beim Arbeitsbeginn beschränkt sich darauf, die Datenbank einzulesen. Der dadurch erreichte Geschwindigkeitsvorteil ist beträchtlich, wird aber mit einem einschneidendem Verlust an Funktionalität erkauft.

Die Datenbank enthält nur die Informationen, die nach ReadProject (siehe Kapitel 4 und Abschnitt 5.2.1) zur Verfügung stehen, die Informationen, die durch ParseProject (vergleiche Kapitel 4 und Abschnitt 5.2.2) erzeugt werden, werden nicht generiert, da deren Bereitstellung eine inkrementelle Übersetzung des Programmsystemes erforderlich machen würde, was noch nicht implementiert worden ist. Als Konsequenz des Geschwindigkeitsvorteiles stehen also keinerlei Informationen über die Bezeichner des Programmsystems mehr zur Verfügung. Nach den Erfahrungen des Autors sind aber gerade diese Informationen bei der Einarbeitungsphase von nicht zu unterschätzendem Wert.

Die Gewinne an Geschwindigkeit sind zwar hoch, wiegen aber obigen Nachteil nach Meinung des Autors *bei weitem* nicht auf – besonders wenn man die Rate, in der die Rechnerleistung steigt, in Betracht zieht und die Tatsache, daß das Werkzeug auf einem schon angejährtem Computer erstellt wurde[1]. Auch das Argument der Einsparung von Hauptspeicher ist für Unix-Systeme

[1] Für Messungen an einer neueren Maschine vergleiche [Sam91], Seite 83, auf einer Sun SPARC station 1+ dauert mit der alten DOgMA-Version das Einlesen und Parsen zusammen nur noch 150 Sekunden – für ca. 19000 Zeilen Code. Dieser Zeitaufwand ist

nur wenig beachtlich, da bei diesen Systemen die Möglichkeit besteht, durch Auslagern von Hauptspeicher auf eine Festplatte einen sehr großen virtuellen Hauptspeicher zur Verfügung zu stellen.

---

durchaus vertretbar, insbesonders da ein Projekt nur selten eingelesen werden muß.

Kapitel 7

Resümee

In diesem Kapitel werden die Schwerpunkte, die in Abschnitt 1.2 beschrieben wurden, noch einmal aus der Sicht des Autors nach Fertigstellung der Implementierung rekapituliert.

Die eigentlichen Änderungsarbeiten an DOgMA verliefen im allgemeinen ohne große Schwierigkeiten, besonders in dem Teil, der expilzit als sprachabhängig gekennzeichnet worden war. Die Änderungen in den anderen Teilen des Werkzeuges zeigten, daß gewisse Teile noch einmal in Bezug auf eine bessere Trennung der sprachabhängigen und sprachunabhängigen Teile überarbeitet werden sollten (siehe Abschnitt 5.3).

Die Erfahrungen, die während der praktischen Arbeit sowohl mit C++-DOgMA als auch mit Modula-2-DOgMA gemacht wurden, sind durchwegs sehr positiver Natur. Ungeachtet der vorgeschlagenen Funktionserweiterungen (siehe Abschnitt 6.2) stellt DOgMA schon jetzt ein gutes Werkzeug dar, welches der Autor bei seiner täglichen Arbeit nicht mehr missen möchte.

DOgMA leistet in zweierlei Hinsicht einen entscheidenden Beitrag zur Erleichterung der Wartungsarbeit: Einerseits wird das Verständnis bestehender Programmsysteme durch die Umsetzung des Hypertext-Konzeptes deutlich verbessert und andererseits die Erstellung einer brauchbaren Dokumentation ohne allzugroßen Aufwand oder zu starke Einschränkung des gewohnten Arbeitsstiles des Programmierers unterstützt.

Einer der Schwerpunkte war die Frage nach dem Einfluß der objektorientierten Programmierung auf die Wartung. Nach Meinung des Autors beeinflußt der objektorientierte Ansatz die Wartungstätigkeit aus folgenden Gründen positiv:

- Der objektorientierte Ansatz bringt einen noch höheren Grad an Strukturierung des Programmsystems mit sich als z.B. der modulorientierte. Durch dieses mehr an Struktur kann der Verständnisprozeß entscheident erleichtert werden.
- Andererseits bringt der höhere Strukturierungsgrad die Möglichkeit, Potentiale zur Wiederverwendung von Code in höherem Ausmaß zu erkennen und zu nutzen (siehe [WP88]). Dadurch kann das zu verstehende Programmvolumen deutlich reduziert werden, was zu einer Zeitersparnis führt, da weniger Code gelesen und verstanden werden muß. Es wird also nicht nur der Code wiederverwendet, sondern auch die Erfahrungen des Bearbeiters.

Diesen Vorteilen steht allerdings der Nachteil des teilweise hohen Einarbeitungsaufwandes, z.B. in ein Application Framework, gegenüber, der geleistet werden muß, bevor mit der Analyse des eigentlichen Applikationscodes begonnen werden kann (vergleiche [WGM89]).

Zum Abschluß der Arbeit soll noch auf einige Punkte eingegangen werden, die für die persönliche Motivation des Autors während der Erstellung der Arbeit wichtig waren.

Einer der wichtigsten Gründe, als Thema diese Arbeit zu wählen, waren für den Autor die Möglichkeit, sich intensiv Kenntnisse im Bereich der objektorientierten Programmierung aneignen zu können. Daß als Programmiersprache für das Projekt C++ gewählt worden war, war insoferne ein zusätzlicher Anreiz, als sich diese Sprache zu der am weitesten verbreiteten objektorientierten Programmiersprache zu entwickeln scheint.

Ein anderer interessanter Aspekt des Themas war die Verwendung des Application Frameworks ET++ bei der Erstellung des Werkzeuges. Schon seit Beginn der Programmiertätigkeit beschäftigte sich der Autor besonders mit der Gestaltung der Benutzerschnittstelle, sodaß ein Application Framework, mit dessen Hilfe mit vergleichsweise geringem Aufwand eine qualitativ hochwertige Benutzeroberfläche in einer graphischen Umgebung erstellt werden kann, besonders interessant ist. Leider beschränkten sich die "Kontakte" zu ET++ auf geringfügige Modifikationen einzelner Klassen, sodaß die Erwartungen des Autors bezüglich der praktischen Anwendung von ET++ im Rahmen der Arbeit selbst nicht befriedigt wurden.

Ein weiterer Grund für die Wahl dieses Themas bestand für den Autor in der Möglichkeit, ein großes, fremdes System warten zu können. Während und neben der Ausbildung wurden hauptsächlich kleinere bis mittlere Programme selbst entwickelt, Wartungsaktivitäten, die in der betrieblichen Praxis jedoch den Hauptanteil der Aktivitäten darstellen, wurden kaum durchgeführt. Das Diplomarbeitsthema bot die Möglichkeit, auf diesem wichtigen Sektor erste Erfahrungen zu sammeln.

Abbildungsverzeichnis

Literaturverzeichnis

[AU79] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Masachusetts, 1979.

[BC90] Marcus Brown and Bart Childs. An interactive environment for literate programming. *Structured Programming*, 11(1):11–25, 1990.

[BG92] Judy M. Bishop and Kevin M. Gregson. Literate programming and the LIPED environment. *Structured Programming*, 13(1):23–34, 1992.

[Bis90] Walter R. Bischofberger. *Prototyping-Oriented Incremental Software Development – Paradigms, Methods, Tools and Implications*. Dissertation, Johannes Kepler Universität, Linz, 1990.

[Bla89] Günther Blaschek. Implementation of objects in modula-2. *Structured Programming*, 10(3):147–155, 1989.

[Bro92] James Brook. *Using the Answerbooks*. Sun Microsystems, Inc., Mountain View, Kalifornien, 1992.

[EN89] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Reading, Masachusetts, 1989.

[GS89] Virginia R. Gibson and James A. Senn. System structure and software maintenance performance. *Communications of the ACM*, 32(3):347–358, 1989.

[Kaa88] Christian Kaare. *Unix - Eine Einführung*. Friedrich Vieweg & Sohn, Braunschweig, Wiesbaden, 1988.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Knu86] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley, Reading, Masachusetts, 1986.

[KR83] Brian W. Kernigan and Dennis M. Ritchie. *Programmieren in C*. Carl Hanser, München, Wien, 1983.

[Leh89] Franz Lehner. *Nutzung und Wartung von Software - Das Anwendungssystem-Management*. Carl Hanser, München, Wien, 1989.

[Lin89a] Chuck Lins. A first look at literate programming. *Structured Programming*, 10(1):60–62, 1989.

[Lin89b] Chuck Lins. An introduction to literate programming. *Structured Programming*, 10(2):107–112, 1989.

[Mit89] Josef Mittendorfer. *Objektorientierte Programmierung mit C++ und Smalltalk*. Addison-Wesley, Reading, Masachusetts, 1989.

[Nie90] Jakob Nielsen. The art of navigating through hypertext. *Communications of the ACM*, 33(3):296–310, 1990.

[Pom87] Gustav Pomberger. *Softwaretechnik und Modula-2*. Carl Hanser, München, Wien, 2. bearbeitete Auflage, 1987.

[PPS92] Gustav Pomberger, Thomas Prückler, and Johannes Sametinger. Documenting and maintaining Modula-2 programs with DOgMA. In *Proceedings of the Second European Conference on Modula-2*, Leicester, England, 1992.

[Rec90] Peter Rechenberg. Programming languages as thought models. *Structured Programming*, 11(3):105–115, 1990.

[Sam90] Johannes Sametinger. A tool for the maintenance of C++ programs. In *Proceedings of the Conference on Software Maintenance*, pages 54–59, San Diego, California, 1990.

[Sam91] Johannes Sametinger. A tool for the documentation & maintenance of software systems. Report, Institut für Wirtschaftsinformatik und Organisationsforschung, Johannes Kepler Universität, Linz, 1991.

[Sch86] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, New Jersey, 1986.

[SK89] Ben Sneiderman and Greg Kearsley. *Hypertext Hands-on: An Introduction to a New Way of Organizing and Accessing Information.* Addison-Wesley, Reading, Massachusetts, 1989.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Masachusetts, second edition, 1991.

[Sun86] Sun Microsystems, Inc., Mountain View, Kalifornien. *Using Modula-2 on the Sun Workstation*, 1986.

[SW88] John B. Smith and Stephen F. Weiss. Hypertext. *Communications of the ACM*, 31(7):816–820, 1988.

[Tun89] Sho-Huan Tung. A structured method for literate programming. *Structured Programming*, 10(2):113–120, 1989.

[Wei89] Rainer Weinreich. *Prototypingorientierte Spezifikation eines Werkzeuges zur Unterstützung des prototypingorientierten Softwarekonstruktionsprozesses.* Diplomarbeit, Johannes Kepler Universität, Linz, 1989.

[WGM89] André Weinand, Erich Gamma, and Rudolf Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[Wir86] Nicklaus Wirth. *Compilerbau.* B. G. Teubner, Stuttgart, 1986.

[Wir91] Nicklaus Wirth. *Programmieren in Modula-2.* Springer, Berlin, Heidelberg, New York, Tokyo, 4. Auflage, 1991.

[WP88] Richard S. Wiener and Lewis J. Pinson. *An Introduction to Object-Oriented Programming and C++.* Addison-Wesley, Reading, Masachusetts, 1988.