HyperSource: Ein Hypermedia-Ansatz für Programmentwicklung und -dokumentation

Diplomarbeit

Jan Oliver Borchers
30. Mai 1995

Universität Karlsruhe
Institut für Betriebs- und Dialogsysteme
Abteilung Dialogsysteme und graphische Datenverarbeitung
Prof. Dr. A. Schmitt

Betreuer:
Prof. Dr. A. Schmitt
Dr. A. Klingert

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Zusammenfassung

Die Art und Weise, in der Entwickler heute ihre Programme entwerfen, implementieren und dokumentieren, nutzt in vielerlei Hinsicht die Möglichkeiten moderner graphischer Arbeitsplatzrechner nicht aus.

Diese Arbeit stellt einen Ansatz vor, der das *Hypermedia*-Paradigma auf die Softwareentwicklung überträgt: Entwickler schreiben ihr Programm nicht mehr als eine Sammlung linearer ASCII-Dokumente, sondern als Hypertext. Randanmerkungen und eingebundene Bilder können dann den Quelltext kommentieren, Querverweise (Hyperlinks) erleichtern die Navigation durch die verschiedenen Programmdateien und helfen, zwischen Quelltext und zugehöriger Dokumentation zu wechseln.

Diese Maßnahmen erleichtern das Entwickeln, aber auch das Lesen und Verstehen von Quelltext. Damit wird das Vertrauen in fremde Programme rationalisiert und letztlich die Wiederverwendbarkeit solcher Software verbessert.

Auch zur Erstellung von Unterlagen für das computergestützte Lernen, insbesondere im Bereich der Informatikausbildung, ist das HyperSource-Konzept gut geeignet.

Neben der Präsentation des HyperSource-Konzepts umfaßt die vorliegende Arbeit eine Umfrage, die derzeitige Mißstände in der Praxis der Programmentwicklung identifiziert, sowie eine Betrachtung existierender Entwicklungssysteme bezüglich ihrer Eignung zur Umsetzung des HyperSource-Konzepts. Ein eigenes Werkzeug wird entworfen, mit dem Programme nach dem HyperSource-Paradigma entwickelt werden können.

Die Implementierung erfolgte unter Unix & X und lieferte ein einsetzbares Werkzeug auf Basis des verbreiteten Universaleditors Emacs und unter Verwendung des HTML-Standards für Hypertextdokumente.

Schlüsselworte

Hypertext, Multimedia, Hypermedia, Dokumentation, Programmentwicklung, CASE, graphische Benutzungsschnittstellen, Editoren, SGML, HTML, Emacs

Inhaltsverzeichnis

A computer isn't some clunky old typewriter
with a television in front of it.
It's an interface where the mind and body
can connect with the universe
and move bits of it about.

— Douglas Adams, „Mostly harmless“

1 Einleitung

Die Wiederverwendung von Software ist in der Praxis auch heute noch eher die Ausnahme denn die Regel. Insbesondere im akademischen Bereich werden vielerorts Bibliotheken und Applikationen entwickelt, die von erheblichem Nutzen für andere Entwicklungen sein könnten. Selbst wenn jedoch ein potentieller Benutzer von dem Paket erfährt, wird von der Wiederverwendung oftmals abgesehen.

Der Grund dafür ist letztlich häufig mangelndes Vertrauen in das von anderen entwickelte System, weil es nicht verstanden wird: Der Leistungsumfang, die Einsatzmöglichkeiten und insbesondere die Struktur der Software sind nur selten überschaubar. So identifizierten auch Wartungsprogrammierer in einer Umfrage das Verstehen der Absicht eines Programmierers und seines Programmierstils als das Haupthindernis bei Änderungen an existierenden Programmsystemen [FjelHaml83].

Das Paradigma, Software als *Black Box* weiterverwenden zu können, ohne ihre interne Struktur zu betrachten, ist nur in den seltensten Fällen erfolgreich. Oft sollen Teile eines Systems benutzt werden, die vom ursprünglichen Entwickler gar nicht als isolierte Einheiten vorgesehen waren. Häufig besteht auch nur das Bedürfnis, ein vorliegendes Programm daraufhin zu untersuchen, wie es ein bestimmtes Problem löst, um diese Lösung auf das eigene Projekt zu übertragen.

Damit sind spätere Benutzer, aber auch Programmierer, die das System anschließend warten sollen, letztlich auf die einzige akkurate Repräsentation des Programms, den Quelltext, angewiesen, der jedoch meist kaum verständlich ist, da er

- typographisch unzureichend (u. a. in einem einzigen Zeichensatz und ohne Überschriften) gestaltet ist,
- bestenfalls textuelle, aber keine graphischen Kommentare enthält und

- keine ausreichenden Querverbindungen zur Dokumentation bietet.

Die Ursache dieser Mängel liegt in den Techniken, die Softwareentwickler heute zum Entwurf, zur Implementierung und zur Dokumentation einsetzen: Sie nutzen die Möglichkeiten moderner graphischer Workstations nur zu einem Bruchteil aus. Nach einer Untersuchung von [Boehm81] entfallen jedoch (oder gerade deshalb) bei einem Softwareprojekt typischerweise 30% des gesamten Arbeitsaufwands auf die Dokumentation des Systems.

Die Abneigung gegenüber der Erstellung von Dokumentation ist einerseits verständlich, da sie letztlich nicht zur Funktionalität des Systems beiträgt, welche ja insbesondere bei Software, die für Kunden entwickelt wird, im Vordergrund steht. Der Auftraggeber ist nicht an den Interna des Systems interessiert; er verlangt lediglich die vereinbarte Funktionalität, wie sie nach außen in Erscheinung tritt.

Doch auch im Bereich akademischer Softwareentwicklung ist das Problem mangelhafter Dokumentation und damit Versteh- und Wiederbenutzbarkeit weit verbreitet. Hier liegen die Gründe allerdings darin, daß meist der Abschluß des Projekts als wissenschaftliche Arbeit im Vordergrund steht, während die Weiterverwendung von Teilen des — oft sogar nur als exemplarischen Prototypen entwickelten — Programms selten geplant und damit angestrebt wird.

Mit der immer stärker zunehmenden Komplexität von Softwaresystemen, deren Erklärungsbedarf auch durch das Aufsetzen auf immer weiter abstrahierte Basissysteme eher zu- als abnimmt, wird jedoch allmählich deutlich, daß eine viel intensivere Wiederverwendung existierender Pakete zur Lösung der auch heute noch nicht bewältigten „Softwarekrise“ [Naur76] unabdingbar ist.

In einem dem Bundesminister für Forschung und Technologie im Oktober 1993 vorgelegten Schreiben [GI93] empfiehlt die Gesellschaft für Informatik (GI), zur Förderung der deutschen Forschung und Entwicklung in der Softwaretechnologie die Aufmerksamkeit verstärkt auch auf die Software-Wiederverwendung zu richten. Sie sei eine hochproduktive Methode, um für professionelle Anwendungsentwickler gleichzeitig den Aufwand an Neuentwicklungen zu reduzieren und die Qualität und Flexibilität von Softwareprodukten zu verbessern.

Strenge Vorschriften in Form von Dokumentationsrichtlinien bieten zwar eine Möglichkeit, derartige Qualitätsstandards zu sichern; sie setzen jedoch meines Erachtens am falschen Ende an: Wenn die Dokumentation nur erfolgt, indem beispielsweise ein vorgebenes Kommentarformular ausgefüllt wird, dann wird sie für den Entwickler weiterhin ein notwendiges Übel bleiben.

Wenn darüberhinaus keine möglichst problemlos zu bedienenden, intelligenten Werkzeuge existieren, um diese Dokumentation zu erstellen, ist die Abneigung des Entwicklers erst recht begründet, denn er muß möglicherweise für die Dokumentation immer wiederkehrende Tätigkeiten erledigen, von denen nicht einzusehen ist, weshalb sie ihm die vor ihm stehende Hochleistungs-Workstation nicht abnehmen kann.

Sind derartige Vorschriften im industriellen Bereich noch durchsetzbar, so werden sie spätestens in der akademischen Welt, in der auf ingenieursartige Qualität geschriebener Software noch weniger geachtet wird, wirkungslos.

Aus diesem Grunde muß man, will man den Qualitätsstandard von Dokumentation verbessern, dem Entwickler Werkzeuge an die Hand geben, die

- ihm durch die Integration von Bildern, Querverweisen und typographischer Information in den Quelltext ein besseres *Dokumentieren* seiner Programme erlauben,
- ihn möglichst auch bei *Entwurf und Implementierung* unterstützen, so daß für ihn selbst ein unmittelbarer Nutzen in ihrer Verwendung erkennbar ist,
- vor allem aber schnell *erlernbar*, einfach *einsetzbar* und problemlos in seine gewohnte Entwicklungsumgebung *integrierbar* sind.

Gerade der letzte Punkt dürfte von entscheidender Bedeutung sein, wie auch die Umfrage (siehe Anhang A) gezeigt hat: Werkzeuge mit begrenztem Funktionsumfang, die nicht innerhalb weniger Minuten versteh- und (zumindest in ihrer Grundfunktionalität) einsetzbar sind, werden – so scheint es wenigstens im akademischen Bereich – keine weite Akzeptanz finden. Eine Umstellung auf eine völlig neue Umgebung wird ein Entwickler, wenn überhaupt, nur akzeptieren, wenn damit für ihn offensichtliche überragende Vorteile bezüglich der Effizienz und Einfachheit seiner Arbeitsabläufe entstehen. Das Prinzip, daß Benutzer längere Antwortzeiten nur in Kauf nehmen, wenn das dafür gelieferte Ergebnis dies rechtfertigt [Schmitt83], ist in noch stärkerem Maße auf die Bereitschaft, Zeit für die Einarbeitung in ein neues System aufzuwenden, anwendbar.

Man sollte also zuerst ein solches Werkzeug zur Verfügung stellen, bevor man einen neuen Standard der Dokumentation oder gar einen Paradigmenwechsel zu einem neuen „Hyperprogramming"-Konzept fordert.

Dieses Ziel verfolgt letztlich auch diese Diplomarbeit, die nicht nur ein neues Konzept untersucht, sondern auch die Umsetzung in ein einsetzbares Werkzeug beinhaltet.

Mit einem Ansatz, der Hypermedia-Konzepte auf den Bereich des Entwurfs, der Implementierung und der Dokumentation überträgt, soll einerseits dem Entwickler ein effizienteres Arbeiten ermöglicht werden. Vor allem aber sollte durch diesen neuen Ansatz die Qualität der Dokumentation von Software verbessert werden. Dies erleichtert die Nachvollziehbarkeit von Algorithmen für andere und stellt damit auch ihr Vertrauen in die Leistungsfähigkeit und Korrektheit des Codes auf eine rationalere Basis. Damit wird hoffentlich die Entscheidung „Benutzen oder Neuschreiben" öfter zugunsten der Wiederverwendung des bereits existierenden Pakets ausfallen.

1.1 Aufgabenstellung

Ziel dieser Arbeit war es also,

- zunächst die *Probleme* zu identifizieren, die die derzeitige Praxis der Softwareentwicklung ineffizient machen,
- einen *Lösungsansatz* auf der Basis von Hypertext- und Multimediakonzepten zu erarbeiten (diese Schlagworte werden im Kapitel 3 noch genauer definiert),

- *existierende Systeme* auf ihre Eignung zur Umsetzung dieses Ansatzes zu untersuchen und schließlich
- eine eigene *Implementierung* zu schaffen, in der Aspekte des „HyperSource"-Konzepts konkret zur Programmentwicklung eingesetzt werden können.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit gliedert sich im Anschluß an diese Einführung dementsprechend in die folgenden Teile:

- Kapitel 2 schildert *typische Arbeitstechniken* in Entwurf, Implementierung und Dokumentation, insbesondere im akademischen Bereich, und ihre Nachteile. Die Ergebnisse beruhen unter anderem auf einer unter Entwicklern durchgeführten Umfrage, die jedoch keinen Anspruch auf Allgemeingültigkeit erhebt.
- Kapitel 3 definiert und beschreibt das *HyperSource-Konzept*. Es klärt grundlegende Begriffe und erläutert, wie dieser Ansatz die identifizierten Probleme lösen kann und welche Vorteile er birgt, sowohl für den Entwickler als auch für den „Wiederverwender" derartig entstandener Software.
- Kapitel 4 spezifiziert zusammengefaßt die *Anforderungen*, die an ein konkretes System gestellt werden, und die Rahmenbedingungen (Benutzer, Systemumgebungen etc.) für seinen Einsatz. Einige Qualitätskriterien werden ebenfalls festgehalten.
- Kapitel 5 beschreibt *existierende Systeme*, die einige Aspekte der HyperSource-Idee unterstützen. Spezifische Vorteile, die als Anregung für den eigenen Entwurf dienen können, werden ebenso herausgestellt wie die Gründe, weshalb keines dieser Systeme eine Eigenentwicklung ersetzen kann.
- Kapitel 6 stellt den *Systementwurf* anhand zentraler, begründeter Entwurfsentscheidungen vor und bietet eine Einführung in die Werkzeuge, Sprachen, Formate und Standards, die als Basis für das System gewählt wurden.
- Kapitel 7 beschreibt die *Implementierung*. Es enthält einen Gesamtüberblick über das System in Form eines Datenflußdiagramms, eine Modulübersicht sowie eine Darstellung der Probleme und Lösungen bei der Realisierung der einzelnen Teilfunktionen und der Benutzerschnittstelle.
- Kapitel 8 bietet einen abschließenden Überblick über das Erreichte, bewertet sowohl das abstrakte Konzept als auch das implementierte System und zeigt einige Richtungen für weitere Forschungs- und Entwicklungsarbeiten auf.
- Im Anhang schließlich finden sich die detaillierten Ergebnisse der Umfrage, ein Beispiel für den praktischen Einsatz des Systems sowie das Literaturverzeichnis.

Real programmers don't document their code.
If it was hard to write, it should be hard to read.

— Alte Entwickler-Weisheit

2 Entwurf, Implementierung und Dokumentation heute

Zu Beginn jeder Problemlösung sollte eine Untersuchung darüber stehen, welche Probleme in der Realität tatsächlich existieren und von welcher Relevanz ihre Lösung ist. Ansonsten ist mit einer praktischen Verwertbarkeit und damit mit einem Erfolg des eigenen Konzepts kaum zu rechnen.

Im vorliegenden Fall galt es also herauszufinden, welche Unzulänglichkeiten Entwickler besonders bei ihrer Arbeit stören, aber auch, welche Arbeitstechniken sich trotz allgemeiner Akzeptanz bei genauerem Hinsehen als verbesserungsbedürftig erweisen. Dies wurde durch eine Umfrage realisiert.

2.1 Umfrage

Da die geplante Implementierung im akademischen Umfeld stattfinden würde, galten als potentielle Benutzer vor allem Mitarbeiter an Universitäten und ähnlichen Einrichtungen, die Programmentwicklung betreiben, aber auch Entwickler in Unternehmen der Softwarebranche.

Als Form der Umfrage wurde ein Fragebogen gewählt, der im Anhang A wiedergegeben ist. Der Bogen wurde in den meisten Fällen von den Befragten in meinem Beisein ausgefüllt, da durch das persönliche Gespräch neben den statistischen Ergebnissen oft noch viel wichtigere individuelle Hinweise und Ideen zur Sprache kommen. Aus diesem Grund stammen die Befragten größtenteils aus der Abteilung Graphische Datenverarbeitung der Universität Karlsruhe. Dies entspricht jedoch auch der ursprünglichen Idee, die hinter diesem Projekt stand: Die Quelltexte graphisch-geometrischer Algorithmen sollten verständlicher darstellbar werden.

Die Umfrage ist mit 17 Befragten sicherlich statistisch nicht ganz gefestigt; allerdings sind die meisten für das weitere Vorgehen entscheidenden Fragen recht eindeutig beantwortet worden. Im Übrigen stellt natürlich die Umfrage nur eine von vielen Informationsquellen dar, auf Grund derer Entwurfsentscheidungen getroffen wurden.

Die Resultate der Umfrage werden im weiteren an den entsprechenden Stellen zitiert; die genauen Ergebnisse finden sich zusammengefaßt im Anhang A.

2.2 Entwurfspraxis

Beim Programmentwurf gilt allgemein, daß die klassischen Regeln der siebziger Jahre, die *Top-Down-Strategie* (eingeführt von [Wirth71] als *Schrittweise Verfeinerung* und von [SteMyeCon74] als *Structured Design*) und die *Bottom-Up-Strategie* (vorgestellt beispielsweise im *Levels of Abstraction*-Ansatz von [Dijkstra68]) nur selten eingehalten werden: Die meisten Programme werden weder rein nach der einen noch nach der anderen Methode entworfen.

Im Normalfall liegt vielmehr eine Mischung aus beiden Techniken vor, bei der während des Entwurfs zwischen den verschiedenen Abstraktionsebenen hin- und hergewechselt wird [Fairley85]. Zu demselben Ergebnis kam auch die Umfrage: Typischerweise wird innerhalb eines Projekts zu etwa 40% Top-Down entworfen, zu 30% Bottom-Up und ebenfalls zu 30% aus mittleren Abstraktionseben heraus. Diese Arbeitsweise ist so verbreitet, daß sie sogar eine eigene Bezeichnung erhielt: *Jo-Jo-* oder *Middle-Out*-Strategie.

Der Entwurf erfolgt darüberhinaus, zumindest gemäß der Umfrage, nur zu einem geringen Teil rechnergestützt: Nur 28% der Befragten nutzen ihre Workstation zur Erstellung von Entwurfsdokumenten. Entsprechend gering fällt auch die Weiterverwendung erstellter Entwurfsdokumente (als Kommentar bei der Implementierung oder in der Dokumentation) in späteren Phasen aus: Textdokumente werden zu 28% „wiederverwertet“, Skizzen gar nur zu 7%. Vor allem auf Papier erstellte Skizzen sind natürlich nur schlecht weiterzuverwenden.

Darüberhinaus muß jedoch auch bedacht werden, daß Entwurfsdokumente oftmals zum Zeitpunkt der Dokumentation bereits veraltet und inhaltlich nicht mehr korrekt sind, womit die Wiederverwendung ohnehin erschwert wird. Allerdings stellen oftmals gerade solche Unterlagen, die Fehlentscheidungen im Entwurfsprozeß dokumentieren und erklären, wertvolle Informationen für andere Entwickler dar.

2.3 Implementierungspraxis

Die überwiegende (von 94% der Befragten unter anderem verwendete) Programmiersprache ist *C*, die somit von einem zu entwerfenden Werkzeug auf jeden Fall unterstützt werden muß. Trotzdem sollte das System sprachunabhängig entworfen werden, um auch mit anderen Programmiersprachen leicht einsetzbar zu sein.

Um ein Tool zu entwickeln, das sich in existierende Umgebungen nahtlos einfügt, muß diese Umgebung zunächst identifiziert werden: Die Befragten arbeiten ausschließlich auf Unix-Workstations unter dem Fenstersystem *X*. Der *Emacs*-Editor ist der am häufigsten benutzte

Programmeditor (74%), gefolgt vom Standardeditor *vi* des Unix-Systems. Ein Versionskontrollsystem wird nur zum Teil eingesetzt, dabei überwiegend (von 41% der Befragten) das System *RCS* [Tichy85].

2.4 Dokumentationspraxis

Es stellt sich heraus, daß die Dokumention von Programmen nur dann in externen Dokumenten erfolgt, wenn dies zwingend notwendig ist: Nur 41% der Befragten erstellen routinemäßig externe Dokumentation in Form von *Manual Pages*; andere Formen sind noch seltener (siehe Anhang A). Ansonsten werden Softwareprojekte nahezu ausschließlich durch Kommentare im Programmtext dokumentiert.

Diese Kommentierung von Quelltext erfolgt im wesentlichen nach drei verschiedenen Schemata:

Abgesetzte Blöcke: Dies ist die bei weitem dominierende Art der Kommentierung: 94% der Befragten setzen sie ein. Sie wird verwendet, um beispielsweise Funktionen mit einem „Kommentar-Kopf" zu versehen, der einerseits die wesentlichen Eigenschaften der Funktion wie Name, Eingabe, Ausgabe etc. zusammenfaßt *(Informationsfunktion)*, andererseits aber auch dazu dient, den Beginn der Funktion ähnlich einer Kapitelüberschrift optisch abzusetzen *(Gestaltungsfunktion)*.

Hierzu bedienen sich die meisten Programmierer jedoch geradezu steinzeitlicher Methoden: Der Block wird bestenfalls als leerer Rahmen mit den Kopierfunktionen des Editors immer wieder eingefügt (oder sogar jedesmal von Hand erzeugt) und dann manuell ausgefüllt. Die graphische Hervorhebung erfolgt typischerweise durch das Einrahmen der Textzeilen mit untereinander ausgerichteten Kommentarzeichen. Ändert sich der Text, so muß im Normalfall die Kommentarumrahmung ebenfalls nachgebessert werden — eine zeitraubende und vom eigentlichen Implementieren ablenkende, aber leider notwendige Arbeit, da keiner der in der Befragung erwähnten oder sonst bekannten Editoren hierzu eine umfassende Unterstützung bietet.

Dieser Kommentarstil ist im folgenden an einem Funktionskopf für eine hypothetische C-Funktion zur Bestimmung des Schnittpunkts zweier Geraden veranschaulicht:

```
/**************************************************/
/* Funktionsname: intersect()                     */
/* Argumente:     Linien a,b in der Ebene         */
/* Ergebnis:      Punkt p: Schnittpunkt von a und b */
/*                oder Fehlermeldung, wenn kein   */
/*                Schnittpunkt vorhanden          */
/* Benutzt:       Funktionen vector(), point()    */
/* Autor:         Jan Borchers <job@ira.uka.de>   */
/* Geaendert:     9.5.1995                        */
/**************************************************/
```

Diese optische Hervorhebung verletzt auch typographisch die Regeln einer ästhetischen Dokumentgestaltung. Die Verwendung einer anderen Schriftgröße statt der künstlichen Umrandung würde hier helfen, den Quelltext typographisch weniger als eine abschreckende „ASCII-Wüste“ erscheinen zu lassen.

Einzelzeilen: 71% der Befragten dokumentieren ihren Quelltext, indem sie einzelne Kommentarzeilen innerhalb von Funktionen etc. einstreuen. Diese haben meist die *Informationsfunktion*, die nächsten 1–10 Zeilen Programmcode zu erklären, und gelegentlich die *Gestaltungsfunktion*, Programmzeilen innerhalb einer Funktion zu Blöcken zu gruppieren und diese Blöcke voneinander abzusetzen.

Als Beispiel folgt ein Ausschnitt aus der oben beschriebenen C-Funktion:

```
...
/* Falls kein Schnittpunkt: Fehlermeldung */
if(parallel(a,b) && (!equal(a,b)) {
  printf("Error: No intersection!");
  exit(1);
}
/* Ansonsten: Schnittpunkt zurueckgeben */
else {
...
```

Hybridzeilen: Die dritte Form der Quelltextkommentierung schließlich sind Kommentare innerhalb von Programmzeilen, die üblicherweise am rechten Ende der Zeile eingefügt werden. Die *Informationsfunktion* ist dabei normalerweise auf die nebenstehende einzelne Programmzeile beschränkt, die der Kommentar erläutert. Eine *Gestaltungsfunktion* existiert nur insofern, als solche Zeilen meist auf diejenigen Stellen in einem Programm hindeuten, in denen wichtige Operationen stattfinden, die deshalb gesondert erklärt werden. Allerdings sind die Attribute „wichtig“ und „erklärungsbedürftig“ von Programmstellen nicht immer korreliert; oft erläutert solch ein Kommentar auch nur ein nicht offensichtliches Implementierungsdetail, das aber für das Verständnis der eigentlichen Funktionalität nicht relevant ist.

Teilweise werden solche „Inline-Kommentare“ jedoch auch als *Randanmerkungen* benutzt, um größere Programmeinheiten zu beschreiben, ohne wie die Einzelzeilen den textuellen Zusammenhang des Quelltextes zu stören. Üblicherweise beginnen diese Anmerkungen dann in einer gleichbleibenden Spalte, wodurch effektiv ein zweispaltiges Layout des Programmtexts entsteht. Diese Benutzung kann soweit gehen, daß neben dem Programmtext ein zweiter lesbarer Text steht, der den Ablauf des nebenstehenden Programms „1:1“ wiedergibt:

```
...
if(parallel(a,b) && (!equal(a,b)) {   /* Falls kein Schnittpunkt: */
  printf("Error: No intersection!");  /* Fehlermeldung */
  exit(1);                            /* Programm beenden */
} else                                /* Ansonsten: */
...
```

Als problematisch erweist sich dabei die Tatsache, daß gelegentlich die 1:1-Beziehung gestört wird, weil beispielsweise eine Programmzeile mehrere Kommentarzeilen erfordert.

> Zur Ausrichtung solcher Zeilen bieten manche Programmeditoren noch funktionelle Unterstützung; ein getrennter Umbruch von Programm und Kommentar ist jedoch in keinem Falle mehr automatisch realisiert und erfordert so unter Umständen großen manuellen Editieraufwand bei Änderungen in einem der beiden Teile. Eine Unterstützung vom Quelltext unabhängiger Randanmerkungen wäre wünschenswert.

Neben diesen drei Formen der Kommentierung ist natürlich noch der Modulkopf zu erwähnen, der beispielsweise zu Beginn einer Datei eine den Funktionsköpfen vergleichbare *Informationsfunktion* erfüllt. Die *Gestaltungsfunktion* hingegen ist eher als „Titel“ der Datei gegeben. Gelegentlich allerdings werden in solchen Kommentaren sogar „ASCII-Bilder“ erstellt, in denen beispielsweise eine Baumstruktur des Programms durch Buchstabengraphiken visualisiert wird.

Ein Beispiel füer diese Art der Dokumentation zeigt der folgende authentische Auszug aus einer Header-Datei der *Inventor*-Bibliothek[1].

```
////////////////////////////////////////////////////////////////////
//    Class: SoLightKit
//
//      A parent node that manages a collection of child nodes
//      into a unit with the following structure:
//
//                            this
//                              |
//     --------------------------
//     |                        |
//     |                  "transformGroup"
//"callbackList"                |
//               -----------------------------------
//               |                |                |
//          "transform"   "light"       "iconSeparator"
//                                                 |
//                                              "icon"
//
////////////////////////////////////////////////////////////////////
```

Dies schließlich zeigt einerseits, daß ein Bedürfnis danach besteht, beispielsweise aussagekräftige Graphiken in den Quelltext einzufügen. Gleichzeitig ist diese Art der graphischen Dokumentation eine der zeitraubendsten, deren Ergebnis in keinem Verhältnis zum Aufwand bei der Erstellung, geschweige denn bei Änderungen steht. Dies ist einer der wichtigsten Widersprüche, die diese Arbeit beseitigen soll.

[1]Copyright © 1991–94 Silicon Graphics, Inc.

The inheritance from the master becomes
not only his addition to the world's record,
but for his disciples the entire scaffolding
by which they were erected.

Vannevar Bush, „Memex“ (1945)

3 Ein Hypermedia-Ansatz

3.1 Was ist Hypermedia?

Das Kunstwort *Hypermedia* ist aus der Kombination zweier anderer Schlagworte, Hypertext und Multimedia, entstanden.

Der Begriff *Hypertext* wurde in den sechziger Jahren von Ted Nelson geprägt. Er definierte ihn als „eine Menge von textuellem oder bildlichem Material, das so komplex vernetzt ist, daß es nicht auf einfache Weise auf Papier dargestellt oder vermittelt werden kann“ [Nelson65].

Allerdings beschrieb bereits 1945 Vannevar Bush in seiner Vision der Zukunft wissenschaftlichen Arbeitens ein persönliches Informationsmanagementwerkzeug *Memex*, das Hypertext-Eigenschaften aufweist [Bush45].

Unter *Hypertext* versteht man heute einfach ausgedrückt „die Verknüpfung von Textdokumenten durch hierarchische Relationen und/oder Verweisstrukturen“ [Schnupp92]. Hypertexte bestehen also im allgemeinen aus einer Sammlung von Textfragmenten, die über Querverweise *(Hyperlinks)* miteinander verknüpft sind. Hyperlinks werden oftmals als hervorgehobene Textstellen dargestellt. Die Präsentation von Hypertexten mit Hilfe des Computers erlaubt die automatische Verfolgung solcher Querverweise, bei graphischen Systemen beispielsweise durch Anklicken mit der Maus.

Unter dem Begriff *Multimedia* hingegen wird im allgemeinen das Konzept verstanden, verschiedene Informationsformen bzw. „Medien“ (Text, Bilder, Animationen, Ton etc.) zu einem Gesamtdokument zu verknüpfen. Dabei muß aber Hypertext-Funktionalität nicht zwangsläufig unterstützt werden [Woodhead91].

Hypermedia-Dokumente können also Fragmente *verschiedener Medien* kombinieren und gleichzeitig eine *komplexe Vernetzung* dieser Fragmente enthalten.

3.2 Hypermedia als Entwurfskonzept

Die zentrale Idee dieser Arbeit ist die Anwendung dieses Hypermedia-Gedankens auf die Welt der Programmentwicklung. In der Entwurfsphase sprechen für diesen Ansatz folgende Argumente:

- Beim Entwurf wird in der Praxis oft zwischen den einzelnen Abstraktionsebenen gewechselt. Eine Repräsentation des Projekts in der Entwurfsphase als Hypertext unterstützt daher die *natürliche Denkweise* des Menschen: Während beispielsweise die Wurzel eines „Projektbaumes“ eine abstrakte Aufgabenstellung und Gliederung in Teilaufgaben darstellt, können von diesem Startknoten aus Verweise auf andere Knoten existieren, die ihrerseits die einzelnen Teilaufgaben repräsentieren. Von dort können wiederum Verweise auf konkretere, Module darstellende Knoten zeigen, die schließlich zu den eigentlichen Funktionen führen. Querverweise verbinden zwischen Quelltext und Dokumentation. Bei bidirektionalen Verweisen ist so ein bequemes Navigieren durch das Projektdokument möglich (siehe Abb. 3.1).

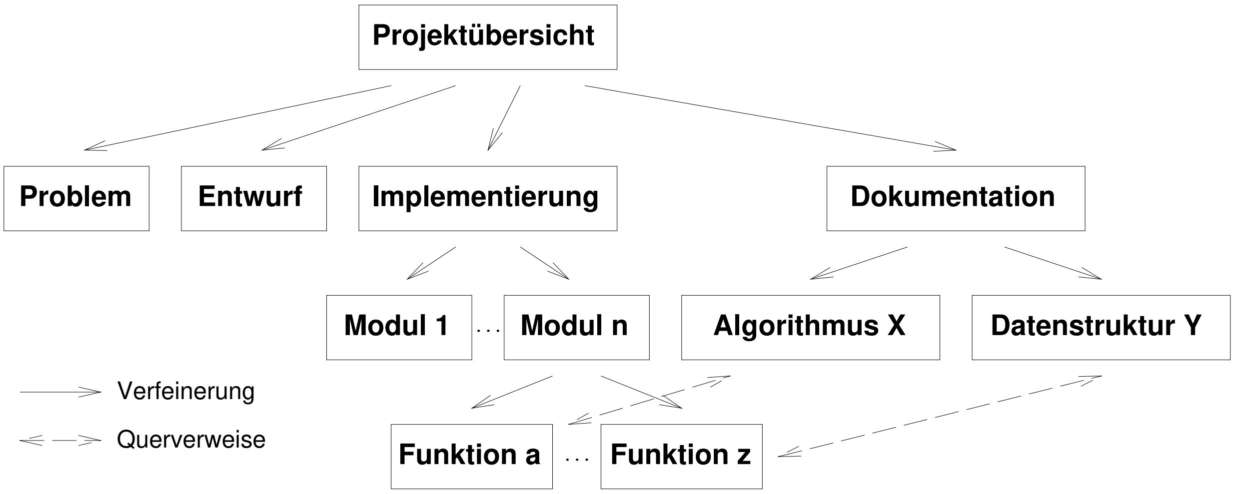


Abbildung 3.1: Beispiel für die Organisation von Projektdokumenten als Hypertext-Graph.

- Neben diesen Verweisen, die die Semantik einer Verfeinerungs- bzw. Dokumentationsrelation besitzen, sind weitere Verweistypen mit anderer Semantik, beispielsweise einer Aufrufrelation, denkbar, um verschiedene Sichten auf das Projekt zu ermöglichen.

- Die Präsentation eines entstehenden Entwurfs gegenüber anderen Projektteilnehmern wird vereinfacht, da die Struktur des Projektdokuments ein gemeinsames Durchgehen ermöglicht und sogar sinnvoll lenken kann.

- Das Hypermedia-Konzept erlaubt es außerdem, verschiedenste Dokumente — Texte, Skizzen, Graphiken, Tonaufzeichnungen etc. — in das Projektdokument einzubinden. Dies kann die Verständlichkeit erhöhen, wenn beispielsweise eine Graphik eine bestimmte Entwurfsentscheidung besser als ein Text veranschaulichen kann.

- Durch die Rechnerunterstützung können sämtliche Dokumente des Projekts erreichbar gemacht werden, so daß Zugriffszeiten auf einzelne Informationen erheblich reduziert werden.

3.3 Hypermedia als Implementierungskonzept

Vergleicht man die typische Struktur eines Programms mit den „Golden Rules of Hypertext“ [Shneiderman89], die als Kriterien dafür dienen sollen, wie gut sich ein Dokument zur Umwandlung in einen Hypertext eignet, so zeigt sich, daß Quelltexte geradezu ideale Voraussetzungen hierfür mitbringen:

- Die Informationen sollen in zahlreichen Fragmenten vorliegen. Dieser Forderung entspricht die Gliederung praktisch jedes Programms in Module, Funktionen, Klassen oder analoge Einheiten.
- Die Fragmente sollen miteinander in Beziehung stehen. Auch diese Forderung ist in jedem Falle erfüllt, beispielsweise durch Klassenhierarchien, Benutzungs- oder Aufrufbeziehungen.
- Der Benutzer ist zu einem bestimmten Zeitpunkt immer nur an einem kleinen Teil der Fragmente interessiert. Dies ist bei Programmtexten ebenfalls notwendigerweise gegeben, da bei der Betrachtung zu großer Codemengen zwangsläufig die Übersicht verlorengeht. Während beispielsweise der Benutzer, das heißt der potentielle Wiederverwender eines Softwarepakets, einen bestimmten Algorithmus genauer studiert, ist er nicht an den Datei-Ein-/Ausgaberoutinen interessiert, die in einem anderen Modul definiert sind. Nur in einer Grobübersicht könnte er Interesse daran haben, von beiden Modulen eine Repräsentation zu sehen, die auch die Zusammenhänge zwischen ihnen aufzeigt.

Aus diesem Grund kann das Hypertext-Konzept dem Entwickler eine sinnvolle Unterstützung bei der Navigation durch seine Quelltexte bieten: Er kann beispielsweise von einem Funktionsaufruf zu dessen Definition springen, um nachzusehen, wie er die Funktion zuvor implementiert hat. Es wäre sogar denkbar, daß die Ausformulierung der Funktion noch gar nicht existiert und beim Verfolgen des Verweises von ihrer Benutzung aus automatisch ein leerer Funktionsrahmen generiert wird, den der Entwickler dann mit der Implementierung ausfüllen kann.

3.4 Hypermedia als Dokumentationskonzept

Der wohl größte Vorteil dieses Ansatzes aber findet sich in der Dokumentationsarbeit: Hier kann der Programmierer seinen Quelltext nicht nur mit textuellen Kommentaren, sondern mit beliebigen Anmerkungen wie beispielsweise eingebundenen Bildern versehen, was die Lesbarkeit und Verständlichkeit des Quelltexts erhöht.

Dabei können auch textuelle Kommentare als eigene Objekte behandelt werden, wodurch die geschilderten Probleme mit normalen Kommentarzeilen beim Editieren des Programms beseitigt werden.

Auch Querverweise zwischen Quelltext und externer Dokumentation sind möglich und erleichtern das Verständnis eines Programms. Das klassische Problem der Inkonsistenz von Programm und Dokumentation kann damit reduziert werden, denn statt Informationen in der externen Dokumentation zu duplizieren, können sie als Verweis auf die Originalinformationen im Quelltext angelegt werden.

Die Möglichkeit, einen Programmtext wie einen gewöhnlichen Text mit typographischen Mitteln optisch zu strukturieren, erhöht ebenfalls die Les- und damit Verstehbarkeit des Programms. Beispielsweise zeigen [OmanCook90] in einer empirischen Studie, daß die Verwendung eines buchähnlichen Formats für Quelltexte zu besserem und schnellerem Erfassen der Programmstruktur führt.

Eine Untergliederung in verschiedene Programmabschnitte durch Überschriften in einer größeren Schrift beispielsweise kann helfen, den Quellcode wie jeden Text übersichtlich und in seiner Struktur schneller erfaßbar zu machen [Siemoneit89].

[ChaFisKra91] zeigen theoretisch und experimentell, daß eine Mischung verschiedener Medien besser geeignet ist, um komplexe Sachverhalte, insbesondere in Gruppenarbeit, zu formulieren. So werden beispielsweise Sprachanmerkungen für abstrakte, globale, „High-Level"-Kommentare gegenüber textuellen Anmerkungen bevorzugt.

The cheapest, fastest, and most reliable components of a computer system are those that aren't there.

— Gordon Bell

4 Anforderungsspezifikation

Nach der Vorstellung der HyperSource-Idee im letzten Kapitel wird nun eine Anforderungsspezifikation nach [Fairley85] aufgestellt, die beschreibt, welche Funktionalität ein HyperSource-System unter welchen Randbedingungen erbringen muß. Der Funktionsumfang wird nach Prioritäten klassifiziert. Erst anschließend können im folgenden Kapitel existierende Systeme auf ihre Eignung hin untersucht werden.

Die meisten gestellten Anforderungen werden durch die Ergebnisse der Umfrage mitbestimmt, die daher an den entsprechenden Stellen zitiert wird.

4.1 Problemdefinition

Programmentwickler schreiben ihre Quelltexte derzeit als lineare Dokumente, die keine Strukturinformationen enthalten, wodurch die Navigation in ihnen und zwischen Quelltext und Dokumentation für Entwickler und spätere Leser umständlich ist.

Quelltexte enthalten keine Formatinformation und lediglich textuelle Kommentare, was Programme zu schwer erfaßbaren „ASCII-Wüsten“ macht.

Kommentare werden vom eigentlichen Quelltext strukturell nicht unterschieden, wodurch es zu Kollisionen zwischen Programmtext und Kommentar bezüglich des Layouts und damit schlechter Lesbarkeit kommt.

Diese Arbeitsweisen sind vor allem darin begründet, daß für lineare, rein textuelle Dokumente ein breites Angebot an gut kooperierenden Werkzeugen existiert, die im Laufe der Jahrzehnte ein beachtliches Maß an Komfort innerhalb der Möglichkeiten des zugrundeliegenden einfachen Formats geschaffen haben. Solch Standard stellt natürlich gleichzeitig einen Hemmschuh des Fortschritts auf seinem Gebiet dar; ein neues System wird nur erfolgreich sein können, wenn es sich in den existierenden Standard möglichst nahtlos einfügt.

4.2 Systemziel

Es ist offensichtlich, daß eine Diplomarbeit bezüglich des Umfangs einer Implementierung nicht mit professionellen CASE-Umgebungen konkurrieren kann. Zu viele Arbeitsabläufe der Software-Entwicklung, wie die Versionshaltung, das Arbeiten an Projekten im Team und die damit verbundenen Konsistenzprobleme etc. lassen sich durch intelligent konzipierte Umgebungen und Funktionen vereinfachen (siehe z. B. [Simon93]). Daher sollte sich die Implementierung auf einige Konzepte beschränken, diese jedoch tatsächlich für den allgemeinen Gebrauch einsetzbar implementieren.

Dieser Weg nutzt darüberhinaus die Tatsache, daß viele typische CASE-Funktionen bereits in der Umgebung des Entwicklers vorhanden sind und benutzt werden. Es ist damit sogar von Vorteil, wenn die Implementierung sich möglichst nahtlos in ein verbreitetes Rahmensystem einfügt und das neue Paradigma in die Denkweisen der Benutzer einfließen läst, ohne daß auf ihrer Seite sofort zu große, grundsätzliche Umgewöhnung und Umstellungen erforderlich sind.

Es soll daher ein System entwickelt werden, mit dem Programme als Hypertexte erstellt werden können, die Querverweise innerhalb des Codes und zur Dokumentation enthalten. (Querverweise wurden auf einer Skala von 1 (unnütz) bis 5 (sehr nützlich) im Mittel mit 4,1 bewertet.)

Ein Programmtext soll mit typographischen Mitteln optisch zu strukturieren sein. Textuelle Kommentare sollen so einfügbar sein, daß sie nicht mit dem normalen Editieren des Quelltextes kollidieren. Der Quelltext soll graphische Anmerkungen, eventuell auch mathematische Formeln, direkt enthalten können (das Einbinden von Bildern wurde ebenfalls als sehr nützlich (4,1) befunden).

4.3 Randbedingungen

Das System sollte sich in existierende Standard-Entwicklungsumgebungen einfügen und verbreitete Werkzeuge integrieren, statt ihre Funktionalität zu replizieren (die Bereitschaft zum Wechsel zu einem neuen Editor war unter den Befragten relativ gering). Der Benutzer darf nur mit einem Minimum an neu zu erlernenden Bedienungskonzepten belastet werden (eine Erlernbarkeit in wenigen Minuten wurde von vielen Befragten informell gefordert).

Obwohl 94% der Befragten *C* als Programmiersprache verwenden, sollte das System nicht auf die Verwendung mit einer speziellen Programmiersprache ausgelegt sein. Trotzdem sollte eine spezielle Editierunterstützung für verschiedene Sprachen weiter möglich sein.

Eine Darstellung nach dem WYSIWYG-Prinzip ist wünschenswert, da sie den Benutzer nicht damit belastet, die Befehle einer Formatierungssprache wie TEX oder HTML erlernen zu müssen.

Dokumente sollten ein standardisiertes Format verwenden, um die Formatierung und Strukturierung auch ohne das hier vorgestellte Werkzeug mit geeigneten Programmen nutzen zu können.

Schließlich sollte das Format transparent (ASCII-lesbar) sein, um notfalls auch mit einem einfachen Texteditor gelesen werden zu können.[1]

4.4 Funktionalität

Die Funktionen des Systems wurden nach ihrer Priorität geordnet:

Notwendig:

- Modus zum Editieren von Programmtexten, der programmiersprachenspezifische Unterstützung bietet
- Einfügen von Bildern, Randanmerkungen und Überschriften in den Quelltext
- Einfügen und Verfolgen von Hyperlinks innerhalb der Programmtexte und auf externe Dokumentation
- Ablage der Dokumente mit allen Formatinformationen in einem Standardformat
- Erzeugung von compilierbarem „Roh-Programmtext" aus Dokumenten

Erstrebenswert:

- Anzeige von Bildern und Benutzbarkeit von Hyperlinks beim Editieren (56% der Befragten bevorzugen einen WYSIWYG-Editor)
- Aufruf externer Editor-/Viewer-Programme für Graphiken etc. per Mausklick aus dem System heraus, um den natürlichen Arbeitsfluß nicht zu unterbrechen
- Kombinierbarkeit mit anderen, vorhandenen Werkzeugen der Entwicklungsumgebung

Denkbar:

- Formatiertes Ausdrucken des Quelltexts
- Andere Ausgabeformate für Quelltext
- Direkte Schnittstelle zu Versionsverwaltung und Compiler
- Graphische Übersicht über Projektdokumente (von 53% der Befragten als Wunsch geäußert)
- Eingabemöglichkeit für mathematische Formeln

[1] Gerade in der Unix-Welt wird jeglichen Datei-Formaten, die versuchen, etwas vor dem Entwickler und seinem *vi*-Editor zu verbergen, traditionell mit großer Skepsis begegnet — zu Recht, denn eine der Stärken des UNIX-Systems ist die universelle Verwendbarkeit und Kombinierbarkeit seiner vielen Werkzeuge durch das einheitliche ASCII-Dateiformat.

4.5 Benutzercharakterisierung

Das System soll die Entwicklung von Programmen vor allem im akademischen Bereich unterstützen, da hier größere Bereitschaft zu erwarten ist, experimentelle Ansätze auszuprobieren. Benutzer sind also typischerweise Studenten und Mitarbeiter aus der Informatik, aber auch anderen Fachrichtungen, die Programme entwerfen und implementieren.

Zunächst wird ein Einsatz im lokalen Institut angestrebt, in dem vornehmlich an Problemen aus der Computergraphik gearbeitet wird. Aus diesem Grund ist die Unterstützung graphischer Kommentare von besonderer Bedeutung.

4.6 Entwicklungs- und Laufzeitumgebung

Das System soll unter Unix/X11R5 auf Workstations von Silicon Graphics und Sun Microsystems entwickelt werden. Dieselbe Umgebung ist auch für den Einsatz des Systems gegeben.

*The computer programs that are truly beautiful,
useful, and profitable must be readable by people.*

— Donald Knuth, „Literate Programming“

5 Existierende Systeme

Bevor ein eigenes System entworfen werden kann, das die im Kapitel 4 gestellten Anforderungen erfüllt, sollen zunächst andere bereits existierende Systeme vorgestellt werden. Ihre Eignung für die gestellte Aufgabe, das Editieren nach dem HyperSource-Paradigma, wird untersucht. Auch wenn sich keines der Systeme als voll geeignet erwiesen hat, finden sich viele Ansätze, die Anregungen für den eigenen Entwurf enthalten.

Soweit nicht anders angegeben, sind Implementierungen für den Einsatz unter Unix/X bestimmt.

5.1 Literate Programming

Dieser bereits seit etwa einem Jahrzehnt existierende Ansatz verfolgt ähnliche Ziele wie das HyperSource-Konzept: Beim Literate Programming soll der Entwickler eher so programmieren, als wolle er das Programm einem anderen Menschen erklären, statt es sofort und ausschließlich „computergerecht“ einzugeben. Donald E. Knuth, der neben einigem anderen auch als „Vater des Literate Programming“ gelten darf, beschreibt seine Idee so:

> „Instead of imagining that our task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.“ [Knuth84]

Er geht davon aus, daß es für den Programmierer in diesem Falle natürlicher ist, Programm und Dokumentation gemeinsam zu entwickeln, als beide Aufgaben getrennt zu erledigen. Aus diesem Grund wird beim Literate Programming nur mit einem Dokument gearbeitet, das Quelltext und Dokumentation enthält. Anschließend erzeugen verschiedene Übersetzer aus dieser Datei zum einen compilierbaren Code, zum anderen formatierte, lesbare Dokumentation.

Der Vorteil dieser Arbeitsweise besteht vor allem darin, daß sich der Entwickler mehr an seinen eigenen Denkprozessen orientieren kann, statt sich streng nach der syntaktisch gegebenen Reihenfolge richten zu müssen, die die Programmiersprache vorschreibt. So kann er beispielsweise zunächst eine umgangssprachliche Beschreibung einer Funktion erstellen (die anschließend in die Dokumentation aufgenommen wird) und dabei wichtige Abschnitte des Quelltexts „zitieren“, die für das Verständnis der Funktion notwendig sind. Der weniger interessante Teil des Codes kann hingegen in einem Anhang zusammengefaßt werden. Bei der Erzeugung des compilierbaren Codes werden die derart verteilten Codefragmente dann mit Hilfe von Modulbezeichnern in der richtigen Reihenfolge zusammengesetzt.

Literate Programming kann somit als eine Art Weiterentwicklung des „Strukturierten Programmierens“ der siebziger Jahre verstanden werden: Während dort der Top-Down- oder Bottom-Up-Ansatz zu wählen war, kann beim Literate Programming zwischen den verschiedenen Abstraktionsebenen gewechselt werden; eine Vorgehensweise, die bereits in Kapitel 3 als natürlich und damit effizienter identifiziert wurde.

Eine ausführlichere Darstellung über die Idee des Literate Programming findet sich in [Knuth92].

5.1.1 WEB und seine Varianten

Mit dem System WEB [Knuth83], das Knuth während seiner zweiten Implementierung des TeX-Satzsystems entwickelte, setzte er seine Idee des Literate Programming in ein benutzbares System um. Das WEB-System vereint die Programmiersprache PASCAL und das Satzsystem TeX. Inzwischen wurde das System unter dem Namen CWEB[1] für die Verwendung mit der Programmiersprache C weiterentwickelt [Knuth93b]; diese Variante soll im folgenden beschrieben werden.

CWEB besteht aus zwei Programmen: `cweave` und `ctangle`. Ein Programmierer, der ein CWEB-Programm schreibt, arbeitet pro Modul mit einer einzigen CWEB-Datei `modul.w`. Diese Datei besteht zu einem großen Teil aus TeX-Code, der als Dokumentation dient, und aus Abschnitten, die C-Quelltext darstellen. Der Aufruf `cweave modul.w` erzeugt eine Ausgabedatei `modul.tex`, die wiederum mit TeX in eine typographisch ansprechend gesetzte Dokumentation umgewandelt werden kann, in der Aspekte wie Seitenlayout, Einrückung, verschiedene Hervorhebungen und die Verwendung mathematischer Symbole berücksichtigt sind. Der Aufruf `ctangle modul.w` hingegen erzeugt eine Datei `modul.c`, die dann übersetzt werden kann, um ein ausführbares Programm zu erhalten.

Neben dieser Möglichkeit der leichteren Dokumentation bietet CWEB jedoch auch den Vorteil, daß der Programmtext umgeordnet werden kann, so daß ein größeres Programm in kleinen, zusammenhängenden Blöcken verständlicher präsentierbar wird.

Zu diesem Zweck kann eine CWEB-Datei aus mehreren weitgehend abgeschlossenen Abschnitten bestehen, die jeweils drei Unterabschnitte enthalten:

- Einen TeX-Teil, der den Abschnitt erläutert,

[1] URL: http://heplibw3.slac.stanford.edu/FIND/FREEHEP/NAME/CWEB/FULL

- einen Teil, in dem C-Makros definiert werden, die die Lesbarkeit des Quelltextes verbessern, sowie
- einen C-Teil, der einen Teil des zu erstellenden Programms darstellt. Der Code sollte im allgemeinen nur zehn bis zwanzig Zeilen umfassen, also in etwa den Umfang einer üblichen Funktion besitzen.

Abschnitte können mit Bezeichnern versehen und damit an beliebigen anderen Stellen referenziert bzw. im C-Code benutzt werden.

Das Umschalten zwischen den verschiedenen Abschnitten, die Verwendung von Variablennamen in der Dokumentation sowie viele weitere Funktionen werden von CWEB über eine Reihe von Kontrollbefehlen, die üblicherweise mit dem speziellen Zeichen „`@`“ beginnen, ebenfalls unterstützt.

Insgesamt bietet das CWEB-System sehr gute Möglichkeiten, Quelltext und Dokumentation aus einer einzigen Datei zu erzeugen. Durch das Literate Programming kann der Entwurfs- und Implementierungsprozeß vereinfacht werden. Es findet sich jedoch keine Unterstützung, um Programme interaktiv zu lesen; eine Darstellung als Online-Hypertext ist nicht vorgesehen.

Aus dem WEB-System sind viele andere Werkzeuge für das Literate Programming hervorgegangen; einige interessante Ansätze werden nun vorgestellt.

Norman Ramsey[2] entwickelte NOWEB[3] als eine vereinfachte Version des CWEB-Systems, die mit einem Bruchteil der Kontrollsequenzen des Originals auskommt, jedoch nicht die „Pretty-Printing“-Funktionalität von CWEB bietet.

NOWEB sowie zwei weitere Pakete, NUWEB und FUNNELWEB (beide finden sich im Literate Programming Archive, siehe unten), stellen die am weitesten verbreiteten WEB-Varianten dar. FUNNELWEB bietet dabei sogar Unabhängigkeit von der verwendeten Programmiersprache.

5.1.2 CLiP

Dieses System[4] erweitert die WEB-Idee auf beliebige Programmiersprachen und Textverarbeitungen. Dazu durchsucht es den Quelltext nach Kommentaren mit einem bestimmten, definierbaren Format und führt die darin enthaltenen Pseudoanweisungen aus. Dieser Ansatz bietet den Vorteil, daß Quelltexte auch mit herkömmlichen Textverarbeitungen geschrieben werden können.

Das System wurde so entworfen, daß auch eine Anbindung an ein Hypertext-System möglich ist, allerdings wurde diese Kombination in der Praxis bislang noch nicht untersucht.

[2] Email: norman@bellcore.com
[3] URL: ftp://ftp.dante.de/tex-archive/web/noweb
[4] URL: http://heplibw3.slac.stanford.edu/FIND/FREEHEP/NAME/CLIP/FULL

5.1.3 Fold2Web

Das Werkzeug Fold2Web[5] ist besonders geeignet, um fremde Programme im Nachhinein zu strukturieren und damit besser zu verstehen: Es geht davon aus, daß ein Texteditor verwendet wird, der einen „folding mode“ unterstützt, d.h. mit dem es beispielsweise möglich ist, von einer Datei nur die Funktionsdefinitionen zu betrachten und den restlichen Text „einzufalten“. Fold2Web nutzt anschließend diese Strukturinformation, um den Quelltext in WEB-Abschnitte zu unterteilen, die dann kommentiert werden können.

Dieser Ansatz ermöglicht eine interessante alternative Arbeitsweise: Ein Programm kann zunächst als reiner Quelltext implementiert und die WEB-Struktur erst dann hinzugefügt werden, wenn das Programm in einer stabileren Version vorliegt. Der Overhead, der durch umfangreichere Änderungen an bereits dokumentiertem Code im allgemeinen entsteht, kann dadurch vermindert werden.

Allerdings ist das System auf den Einsatz unter MS-DOS und die Verwendung der WEB-Variante NUWEB spezialisiert, was einen breiten Einsatz im akademischen Bereich verhindert.

Weitere Informationen zum Literate Programming finden sich im „Literate Programming Archive (LPA)“[6] sowie im „Comprehensive TeX Archive Network (CTAN)“[7].

Allen vorgestellten Ansätzen bleibt jedoch das Prinzip der Trennung zwischen Dokumenterstellung, Übersetzung und Vorschau gemeinsam, die durch die Verwendung eines Satzsystems wie TEX bedingt ist. WYSIWYG-Systeme lassen sich mit diesem Prinzip nicht realisieren.

5.2 Literate Programming und Hypertext

Es gibt bereits mehrere Ansätze, die Idee des Literate Programming um Hypertext-Funktionalität zu erweitern.[8] Dabei können zwei grundsätzlich verschiedene Wege gegangen werden: Zum einen kann von einem Hypertext-System ausgegangen und dieses um Satzfunktionen und die Verwaltung von Quelltext und Dokumentation ergänzt werden. Die zweite Möglichkeit ist, Literate-Programming-Werkzeuge mit einer Hypertext-Unterstützung zu versehen. Aufgrund der hohen Komplexität von Satzsystemen wie beispielsweise TEX wird in den meisten Fällen der zweite Weg gewählt.

5.2.1 WEB-Erweiterungen

Inzwischen wurde in die meisten WEB-Varianten eine Hypertext-Unterstützung integriert, meist in der Form einer automatischen Erzeugung von Dokumenten im HTML-Format. Das oben erwähnte NOWEB-Paket beispielsweise realisiert das Konzept eines HTML-Backends,

[5] URL: ftp://kirk.ti1.tu-harburg.de:pub/fold2web
[6] URL: ftp://ftp.th-darmstadt.de/programming/literate-programming
[7] URL: http://www.dante.de/dante/Ctan.html
[8] URL: http://info.desy.de:80/www/LitProg/HTML.html

so daß NOWEB-Programme mit einem WWW-Browser betrachtet werden können.

5.2.2 HyperTEX

Ein Beispiel für die Erweiterung eines Satzsystems um Hypertext-Funktionen ist das HyperTEX-Konzept.[9] Es basiert auf der Idee, in TEX-Dokumente zusätzliche `\special`-Kommandos einzufügen, die als Hypertext-Links dienen. Diese werden nur von entsprechend erweiterten TEX-Makropaketen bzw. DVI-Viewern verstanden. Auf diese Weise kann eine große Zahl von Links, beispielsweise zwischen Kapiteln und Abschnitten oder bei normalen Querverweisen, automatisch erzeugt werden, so daß vorhandene TEX-Texte leicht in HyperTEX umgewandelt werden können.

Eine Kombination dieses Systems mit einem der obigen Literate-Programming-Werkzeuge wäre eine Möglichkeit, beide Konzepte zu vereinen.

5.2.3 LPW

LPW[10] *(Literate Programming Workshop)* ist eine Literate-Programming-Umgebung für den Apple Macintosh. Sie ergänzt die dortige Standard-Entwicklungsumgebung MPW (Macintosh Programmer's Workshop) um einen WYSIWYG-Editor mit Hypertext-Funktionalität, Extraktionsfunktionen für Dokumentation und Sourcecode sowie ein Dokumentmanagementsystem.

Das System bietet jedoch keine Plattformunabhängigkeit, da es auf die Verwendung in Verbindung mit einem speziellen Entwicklungssystem zugeschnitten ist, das nur auf Macintosh-Systemen verfügbar ist. Darüberhinaus basiert es nicht auf einem Standardformat für strukturierte Dokumente wie HTML.

5.2.4 Standard-Textverarbeitungen

Die bisherigen Systeme gehen von der Verwendung eines Programmeditors zur Eingabe aus. Es gibt jedoch auch Ansätze, Standard-Textverarbeitungen zu verwenden, in denen Quelltexte und Dokumentation in einer Datei gemeinsam erstellt werden, wobei die unterschiedlichen Blöcke durch Absatzformate identifiziert werden, was auch ein getrenntes Abspeichern ermöglicht. Allerdings sind Standard-Textverarbeitungen zum Schreiben von Programm-Quelltexten nicht sehr geeignet. Beispiele für diesen Ansatz sind CU_HTML[11] oder ANT_HTML[12], Sammlungen von Templates für die Textverarbeitung „Word“ von Microsoft, die das Erstellen von HTML-Dokumenten erleichtert, existierende HTML-Quelltexte allerdings nicht einlesen können, was die Übernahme vorhandener Dokumente erschwert. Ein weiterer Nachteil ist die fehlende Verfügbarkeit für Unix-Systeme.

[9] URL: http://xxx.lanl.gov/hypertex/
[10] URL: ftp://ftp.apple.com:/pub/literate.prog
[11] URL: ftp://ftp.cuhk.hk/pub/www/windows/util/cu_html.zip
[12] URL: ftp://ftp.einet.net/EINet/pc/

5.3 Ansätze für komplette CASE-Umgebungen

Die Idee, Hypertext-Konzepte in CASE-Werkzeuge zu integrieren, ist nicht neu. [Shneiderman92] erwähnt als eines der Einsatzgebiete für Hypertextsysteme den Bereich der Programmentwicklung und -dokumentation.

Da eine Datenbank den Kern eines jeden CASE-Tools darstellt, bauen auch die meisten Ansätze, eine komplette Entwicklungsumgebung mit Hypermedia-Unterstützung zu entwerfen, auf einem Datenbanksystem auf.

5.3.1 Neptune

Das Problem, ein System zu entwerfen, das die Hypertext-informationen applikationsunabhängig speichert, wurde im Projekt *Neptune* [Bigelow88] durch eine Schichtenarchitektur realisiert: Die unterste Schicht bildet ein transaktionsbasierter Dienstgeber, die *Hypertext Abstract Machine (HAM)*, auf der Applikationen und die Benutzerschnittstelle aufsetzen. Dieser Ansatz erlaubt auch eine Versionshaltung mit Knoten und Verweisen als Einheiten.

5.3.2 HyperCode

Das *HyperCode*-Projekt[13] des MIT verfolgt ein der vorliegenden Arbeit sehr verwandtes Ziel: Hypertext-Ansätze sollen für die Programmentwicklung nutzbar gemacht werden. Programmtext soll mit direkten Querverweisen zu anderen Projektdokumenten versehen werden:

> „HyperCode brings HyperText technology to program source code. Program source code is richly annotated with direct links to relevant information including definition sites, program genealogy, quality annotations, and documentation.“ [DeHBroEsl94]

Entstanden ist dieser Ansatz im Rahmen des Projekts „Global Cooperative Computing“, in dem Programmtexte selbst als verteilte Dokumente gehalten werden sollen, an die Benutzer jederzeit eigene Anmerkungen oder Verbesserungen anheften können. Auch in diesem Projekt wird als eine Motivation die schlechte Lesbarkeit und damit Wiederverwendbarkeit von Software genannt:

> „Complicated dependencies within the code make it impractical to reuse portions of contributed software without undertaking significant effort to understand the context required for the code to operate.“ [DeHBroEsl94]

Das Projekt hat jedoch bislang noch kein universell einsetzbares, verfügbares Produkt hervorgebracht.

[13] URL: http://www.ai.mit.edu/projects/transit/tn105/tn105.html

5.3.3 Gwydion

Dieses Projekt[14] [Fahlman94] hat die Erstellung einer Software-Entwicklungsumgebung für die von Apple Computer entwickelte neue dynamische, objektorientierte Programmiersprache *Dylan*[15] [Apple92] zum Ziel. Eines der Schlüsselkonzepte dabei ist auch hier die Idee, Programme durch Hypertext-Funktionalität lesbarer zu machen:

> „A program should no longer be thought of as a linear string of ASCII characters with (maybe) a few supporting documents. The Gwydion environment will deal in *hypercode*, analogous to hypertext. A program in Gwydion will be a complex linked data structure stored in an object-oriented database. This data structure will contain source code linked with documentation and all the other information needed to support the software system throughout its entire life-cycle." [Fahlman94]

Allerdings erlaubt die Festlegung auf die Programmiersprache Dylan keine universelle Nutzung der Gwydion-Entwicklungsumgebung mit beliebigen Sprachen.

5.3.4 Andere Ansätze

[BroCze90] präsentieren ein Hypertext-System für das Literate Programming, das auf WEB und einer relationalen Datenbank basiert. Navigationsoperationen im Dokument werden auf Anfragen an die Datenbank abgebildet. Dadurch sind Links mit unterschiedlicher Semantik und eigenen Operatoren möglich, die auch vom Benutzer erweitert werden können. Verschiedene Darstellungsmöglichkeiten für Texte, Indices und graphische Übersichten werden diskutiert.

Ein auf C++ zugeschnittenes Werkzeug für das Literate Programming haben [SamPom92] vorgestellt. Es bietet eine moderne Benutzerschnittstelle und komfortable Funktionen zur Navigation durch den Quelltext. Das System wurde bei einigen Projekten bereits eingesetzt und hat nach Aussage der Autoren zu einer deutlichen Senkung des Aufwands in der Wartung dieser Softwareprojekte geführt. Die Implementierung wurde in einen sprachunabhängigen Hypertextbrowser, einen sprachabhängigen Quelltextparser und einen einfachen Dokumentationsparser aufgegliedert.

5.4 Automatische Dokumentation

Neben der Umstellung auf eine gänzlich neue Art zu programmieren, wie es beim Literate Programming notwendig ist, oder der Einführung einer komplett neuen Entwicklungsumgebung gibt es auch eine Reihe interessanter praktischer Ansätze, die sich als nützlich für die teilweise Automatisierung der Dokumentation erwiesen haben. Ihnen ist gemeinsam, daß aus einer Metabeschreibung der Dokumentation erst die eigentliche Dokumentation in einem oder mehreren verschiedenen Formaten generiert wird.

[14] URL: http://legend.gwydion.cs.cmu.edu:8001/gwydion/index.html
[15] http://www.cambridge.apple.com/dylan/dylan.html

5.4.1 Texinfo

Das Texinfo-System [ChaSta93] wird von der Free Software Foundation zur Dokumentation sämtlicher Pakete ihres GNU-Projekts verwendet. Es ist ein Dokumentationssystem, mit dem aus einer einzigen Dokumentationsdatei sowohl eine Online-Hypertext-Version als auch eine druckbare Version der Dokumentation erzeugt werden kann. Dies senkt erheblich den Aufwand, wenn bei der Dokumentation beide Versionen erstellt werden sollen.

Eine Texinfo-Datei ist im wesentlichen eine TEX-Datei, die einige spezielle TEX-Befehle zur Definition der Dokumentstruktur (Querverweise etc.) enthält. Die dazu benötigten TEX-Definitionen sind in einer Makrodatei `texinfo.tex` zusammengefaßt, die am Anfang jedes Texinfo-Dokuments eingebunden wird. Um aus einer Texinfo-Datei ein gedrucktes Dokument zu erzeugen, wird sie direkt mit dem TEX-Satzprogramm übersetzt, das eine DVI-Datei erzeugt, die anschliessend wie gewohnt gesetzt und ausgedruckt werden kann. Alternativ kann die Datei auch mit einem `texi2roff`-Übersetzer in das Format gewandelt werden, das die UNIX-Satzprogramme `troff` und `nroff` verwenden.

Die Online-Dokumentation hingegen wird aus der Texinfo-Datei durch das `makeinfo`-Programm erzeugt; das Ergebnis ist eine *Info*-Datei mit Knoten, Menus, Querverweisen und Indizes, die mit einem separaten Programm des GNU-Projekts oder aus dem Emacs-Editor heraus gelesen werden kann.

Die Universalität des Texinfo-Systems ist gleichzeitig sein Nachteil: Da das Dokument sowohl auf einer Vielzahl von Druckern als auch auf einer Vielzahl von Terminals angezeigt werden können muß, suchte man beim Texinfo-Format den kleinsten gemeinsamen Nenner. Die Online-Version einer Texinfo-Datei kann beispielsweise *keine Graphiken* enthalten. Auch können im Text nur wenige Sonderzeichen eingefügt und kein exaktes Layout spezifiziert werden.

Die Schaffung eines Metaformats für Online-Hypertexte und gedruckte Dokumentation ist jedoch mit dem Texinfo-System sehr gut und insbesondere für den realen Einsatz brauchbar gelöst. So wird beispielsweise das „Emacs LISP Reference Manual“ [StaLewLibWin94] im Texinfo-Format ausgeliefert, aus dem bei der Installation eine Online-Hilfe und bei Bedarf ein ausdruckbares Handbuch mit rund 700 Seiten erzeugt werden.

5.4.2 OSE

Die Entwicklungs- und Dokumentationsumgebung OSE basiert auf dem Konzept, in den Quelltext Kommentare in einem festgelegten Format einzufügen, die dann von einer Reihe von Übersetzerprogrammen über ein gemeinsames Zwischenformat in verschiedene Ausgabeformate zur Dokumentation und Programmentwicklung gewandelt werden. Die Verwendung eines Zwischenformats birgt dabei den Vorteil, daß die sonst übliche kombinatorische Explosion der Anzahl benötigter Wandlerprogramme bei Hinzufügen weiterer Dokumentformate vermieden wird.

Das System ist jedoch auf die Verwendung der C^{++}-Programmiersprache beschränkt, und stellt *keine Hypertextfunktionalität* oder graphische Gestaltungsmöglichkeiten der Dokumentation oder des Quelltexts zur Verfügung. Trotzdem ist es dazu geeignet, beispielsweise die

Generierung von Manual Pages aus einem Quelltext zu automatisieren. Auch fügt sich das Werkzeug sehr gut in die typische Unix-Entwicklungsumgebung ein, was seinen Einsatz erst praktikabel macht.

5.5 Hypermedia-Tools

Auf einem anderen Gebiet sind Systeme entstanden, die auf die Erzeugung multimedialer Dokumente, also die Integration beispielsweise von Bildern, Animationen, Tonaufnahmen etc. in ein Dokument spezialisiert sind. Da im allgemeinen auch eine Navigation durch diese Dokumente über Querverweise unterstützt wird, lohnt es sich, hier nach Ansätzen für die Umsetzung der HyperSource-Idee zu suchen.

5.5.1 Klassische Autorensysteme

Für die Erstellung von Hypermedia-Dokumenten existiert bereits eine ganze Reihe hochentwickelter Werkzeuge, die sogenannten Autorensysteme (Beispiele hierzu wären Asymetrix' *ToolBook* oder das *HyperCard*-System der Firma Apple Computer). Sie bieten zwar die Möglichkeit, multimediale, vernetzte Dokumente zu erstellen, sind aber in keinem Fall zum Schreiben von Programmtexten geeignet, da sie die dazu übliche zusätzliche Funktionalität (automatisches, sprachabhängiges Einrücken und Formatieren, Aufruf von Übersetzern etc.) nicht bieten.

Obwohl sie also die Erstellung qualitativ sehr hochwertiger und ansprechender dokumentierter Programmtexte erlauben, wird durch die fehlende Einbindung in Entwicklungsumgebungen (proprietäre Dateiformate, fehlende Integration von CASE-Wekzeugen etc.) der erforderliche Aufwand insbesondere bei größeren Softwareprojekten untragbar.

5.5.2 HTML-basierte Werkzeuge

Dieser Umstand könnte sich bessern, wenn Werkzeuge betrachtet werden, die auf dem Hypertext-Dokumentformat *HTML* basieren, das im *World-Wide Web* verwendet wird (siehe Kapitel 6). Das World-Wide Web besteht zu weiten Teilen aus Dokumenten von Autoren, die im wissenschaftlichen Bereich oder bei Unternehmen arbeiten, in denen auf Workstations mit Internet-Anbindung entwickelt oder allgemein gearbeitet wird. Entwickler verwenden also diese Umgebung zum Schreiben von Dokumenten und zur Programmentwicklung.

HTML-Browser

Es gibt inzwischen eine große Zahl von Programmen, mit denen Dokumente des World-Wide Web graphisch betrachtet, aber nicht editiert werden können (sog. *Browser*). Sie wurden trotzdem untersucht, da sie teilweise Anregungen für den eigenen Entwurf liefern können, aber auch als potentielle Viewer für das noch zu entwerfende System in Betracht kommen könnten. [16]

Der wohl bekannteste Browser ist *NCSA Mosaic*[17], der durch sein sorgfältiges Layout HTML-Seiten ein sehr ästhetisches Aussehen gibt, was zu einem Großteil den Erfolg des WWW bewirkte, so daß für viele Benutzer „Mosaic" und „World-Wide Web" Synonyme darstellen.

Der *Netscape Navigator*[18] hat diese Spitzenposition inzwischen übernommen (im Mai 1995 nach unabhängigen Schätzungen über 75% Marktanteil), insbesondere auf Grund seiner frühen Multithreading-Fähigkeit, durch die das Navigieren in Dokumenten bereits möglich ist, bevor diese komplett übertragen sind, und durch ein hierarchisches Hotlist-Konzept, mit dem interessante Seiten für den privaten Gebrauch baumartig angeordnet werden können. Aber auch ganz pragmatische proprietäre Erweiterungen von HTML 2.0, wie blinkende Textpassagen oder die Möglichkeit, Hintergrundbilder für Dokumente zu spezifizieren, sowie Netscapes schnelle Anpassungen an den zur Zeit entstehenden Standard *HTML 3.0* haben zu seiner Verbreitung beigetragen.

Arena[19] ist ein Browser der W^{3}C, des World-Wide Web Consortiums, der insbesondere eine direkte Umsetzung der HTML3-Spezifikation darstellt, ansonsten jedoch eher spartanisch ausgestattet ist. Allerdings ist für die fernere Zukunft ein Editor-Modus geplant.

Die in HTML fehlende Möglichkeit dynamischer Dokumente, die Animationen und ähnliche Abläufe lokal ausführen können, wurde mit dem Browser *HotJava*[20] von Sun Microsystems gelöst. Er ist ein Beispiel für den Einsatz von *Java*, einer neuentwickelten, C^{++}-ähnlichen Programmiersprache, die jedoch verschiedene Vorzüge (Architekturunabhängigkeit durch Erzeugung von Bytecode, der zur Laufzeit interpretiert wird, automatische Garbage Collection, Standardbibliotheken mit Unterstützung von TCP/IP-Protokollen wie HTTP und FTP sowie in die Sprache integriertes Multithreading) für die Entwicklung insbesondere verteilter Applikationen birgt.[21]

Der *Midas*-Browser des Stanford Linear Acceleration Center[22] bietet eine komfortablere Behandlung von Links durch ein Popup-Menu, das bei längerem Anklicken eines Links verschiedene Operationen auf diesem erlaubt, und ist in der Lage, eine Vielzahl von Formaten (z.B. PostScript) bereits integriert darzustellen. Daneben ist sein Konzept hierarchischer, speicherbarer History-Listen interessant.

Viola[23] war der erste WWW-Client für das Fenstersystem X und bietet bereits eine eigene Erweiterung von HTML zur zweispaltigen Darstellung, eine Fähigkeit, die auch in der HyperSource-Implementierung von Bedeutung ist. Tatsächlich ist der Browser nur ein Teil eines kompletten Toolkits zur Entwicklung interaktiver Applikationen, das sogar das Integrie-

[16] Interessanterweise existieren in diesem Bereich fast ausschließlich frei erhältliche Programme; die folgenden Browser sind allesamt kostenfrei verfügbar.

[17] URL: http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-about.html

[18] URL: http://home.netscape.com/

[19] URL: http://www.w3.org/hypertext/WWW/Arena/Status

[20] URL: http://java.sun.com/

[21] Letzte Meldung (23.5.95): Netscape wird Java in sein Produkt integrieren.
(URL: http://home.netscape.com/newsref/pr/newsrelease25.html)

[22] URL: http://www-midas.slac.stanford.edu/midas_latest/overview.html

[23] URL: ftp://ftp.ora.com/pub/www/viola/

ren ganzer Programme mit eigener Benutzerschnittstelle in eine Dokumentseite ermöglicht.

HTML-Editoren

Während heute kaum noch jemand WWW-Seiten ohne einen graphischen Browser (und erst recht nicht in einem Texteditor im HTML-Quellformat) betrachtet, wird das Verfassen von HTML-Dokumenten derzeit noch zu einem großen Teil mit Standard-Texteditoren, eventuell erweitert um einen speziellen Modus, der das einfache Einfügen von HTML-Tags ermöglicht, erledigt [DecRan94]. Erst in letzter Zeit sind mehrere Editoren speziell für dieses Dokumentformat entwickelt worden. Grundsätzlich lassen sich diese HTML-Editoren analog zu Textverarbeitungen in zwei Klassen einteilen:

- Editoren, die nach dem *WYSIWYG*-Prinzip arbeiten, bei denen also das HTML-Dokument so angezeigt wird, wie es sich später auch dem Leser präsentieren könnte,
- Systeme, die auf dem *Edit+Preview*-Konzept beruhen, bei denen also das HTML-Dokument zunächst unformatiert bearbeitet wird und erst anschließend, beispielsweise mit einem HTML-Browser, betrachtet werden kann.

Hierbei muß festgehalten werden, daß echtes WYSIWYG für HTML-Dokumente gar nicht definiert sein kann [Weber95], da diese Dokumente per definitionem lediglich strukturelle, aber keine Layout-Formatinformation enthalten (siehe Kapitel 6). Unterschiedliche Browser werden also ein und dasselbe Dokument auf verschiedene Weise darstellen; unter Umständen sogar lediglich in textbasierter Form! Es kann also nur darauf ankommen, eine *mögliche* und wahrscheinliche spätere Sicht des Dokuments zu erzeugen.

Durch die zusätzliche Funktionalität eines Hypertext-Systems entsteht jedoch noch ein weiteres Klassifizierungsmerkmal:

- Bei *Online-Editoren* erfolgt das Editieren „im Netz“. Erzeugte Hyperlinks sind beispielsweise sofort aktiv und führen den Autor tatsächlich zum Zieldokument, wenn er sie verfolgt. Damit einher geht das Problem, zwei Operationen auf einem Hyperlink (Editieren und Verfolgen) auf eine intuitive Art nebeneinander anzubieten.
- *Offline-Editoren* besitzen dieses Problem nicht: Bei ihnen ist ein Querverweis nicht aktiv, solange das Dokument editiert wird. Erst in einem speziellen Modus oder in einem externen Previewer werden die Links benutzbar.

Es gibt auch Ansätze, beide Funktionsweisen zu verbinden, so daß zwei verschiedene Editiermodi (Online und Offline) existieren. Dies widerspricht allerdings der Grundforderung des Mensch-Maschine-Dialogs nach Benutzerschnittstellen mit einem Minimum an verschiedenen Modi, die sich möglichst auf ein bestimmtes Objekt beziehen und nur von begrenzter Dauer sein sollen (siehe die Diskussion über den Entwurf von *Smalltalk* [Tesler81]).

Einer der ersten dieser Editoren war *tkWWW*, ein frei verfügbares System, das auf dem Tcl/tk-System aufsetzt. Er ist sogar Browser und Editor in einem, also ein „Online-Editor“, birgt jedoch wie alle Spezialeditoren den Nachteil, daß seine Bedienung erst gesondert erlernt werden muß, was viele potentielle Benutzer scheuen, da für sie das Schreiben von HTML-Dokumenten nur eine von vielen Textverarbeitungstätigkeiten darstellt. Er verbirgt jedoch die HTML-Anweisungen komplett vor dem Benutzer und bietet so in gewissem Sinne „echtes“ WYSISYG. Andererseits gestaltet der Benutzer sein Dokument zwar mit verschiedenen Attributen, doch erst beim Abspeichern wird (stets korrekter, aber nicht unbedingt dem vorher Gesehenen entsprechender) HTML-Code erzeugt.

SoftQuads *HoTMetaL*[24] (frei verfügbar) bzw. *HoTMetaL Pro* (kommerziell) ist der wohl erfolgreichste WYSIWYG-Editor für HTML-Dokumente. Besonderheiten sind die Möglichkeit, die HTML-Tags in einem Dokument als Icons ein- oder auszublenden sowie automatisches Überprüfen der Dokument-Syntax, wodurch allerdings Probleme beim Einlesen existierender Dokumente entstehen können. Auch bietet HoTMetaL zwar bereits umfangreichere Editierfunktionen, ist jedoch von der Funktionalität her weder mit Universaleditoren noch mit Standard-Textverarbeitungen vergleichbar. Dabei bieten beide Bereiche erprobte Konzepte, die ein Dokumenteditor enthalten sollte.

5.6 Erweiterungen von Standardeditoren

Praktisch allen bislang vorgestellten Ansätzen (die Erweiterungen von Standard-Textverarbeitungen ausgenommen) ist ein Nachteil gemein: Sofern sie überhaupt zur Erstellung von Programmtexten geeignet sind, stellen sie hierzu eine neue Umgebung zur Verfügung. Für das Schreiben gewöhnlicher Hypermedia-Dokumente mag das Erlernen eines weiteren Textsystems noch akzeptabel sein, da die meisten Textverarbeitungen eine vergleichbare Funktionalität zur Verfügung stellen, die in weiten Teilen leicht nachgebildet werden kann. Entwickler jedoch, die Programme schreiben müssen, haben sich im Laufe der Jahrzehnte hochkomplexe Umgebungen konstruiert, die verschiedenste Abläufe des Entwicklungs- und Dokumentationsprozesses erleichtern. Als Beispiel seien hier nur die Versionshaltung, das automatische Einrücken von Quelltext bei der Eingabe und die Einbindung von Übersetzern, Debuggern und anderen Programmen erwähnt.

Bei der Arbeit mit einem System zum Schreiben von HyperSource-Code müssen diese Funktionen weiterhin benutzbar sein. Dazu bestehen für das System drei Möglichkeiten: Es kann

- die Funktionalität selbst zur Verfügung stellen,
- existierende Tools integrieren und anbieten oder
- sich in eine bestehende Umgebung einfügen und nur die spezielle HyperSource-Funktionalität hinzufügen.

Nur die beiden letzten Ansätze ergeben vom softwaretechnischen Standpunkt her einen Sinn. Daher werden im folgenden Standard-Editoren im Unix/X-Bereich auf ihre Verwendbarkeit bzw. Erweiterbarkeit zu einem HyperSource-Editorsystem untersucht.

[24] URL: http://www.sq.com/

5.6.1 htmltext

Der Editor *htmltext*[25] basiert auf dem *Andrew Toolkit*, einer Alternative zum Standard-Xt-Toolkit für X. Dadurch stehen Standardfunktionen zur Textbearbeitung und Programmentwicklung zur Verfügung, die auch über vielen Entwicklern bekannte Kürzel, die sich an den Emacs-Editor anlehnen, erreicht werden können. Allerdings erfordert die volle Funktionalität die Installation des wenig verbreiteten Andrew Toolkits.

5.6.2 Symposia

Symposia[26] basiert auf *GRIF*, einem universellen SGML-Editor, entwickelt am INRIA und im Vertrieb der GRIF S.A. (Frankreich). Er wurde vor kurzem mit dem Tool *SIGMA* um dynamische Such- und Linkfunktionen erweitert. Hervorzuheben ist sein statistischer Ansatz zum automatischen Erzeugen und Auffinden dynamischer Links, der auf Ähnlichkeitsvergleichen beruht. Es handelt sich um einen WYSIWYG-Online-Editor, der sogar das Speichern von Seiten auf entfernten HTTP-Servern ermöglicht, die das HTTP-PUT-Protokoll unterstützen. Wie bei HoTMetaL existieren eine freie sowie eine kommerzielle Vesion. Allerdings bleibt das Problem, daß es sich nicht um einen Standard-Programmeditor handelt, eine Verwendung für Implementierungsarbeiten also unkomfortabel bleibt.

5.6.3 vi

Der *vi* (siehe z. B. [Gulbins84]) ist der Standardeditor des Betriebssystems UNIX. Er ist auf jedem UNIX-System verfügbar, jedoch nicht programmier- bzw. erweiterbar und nicht graphikfähig. Daher ist er als HyperSource-Editor ungeeignet. Es ist jedoch zu bedenken, daß viele Entwickler den Editor verwenden und daher an einem System interessiert sind, das sich (ggf. in einem speziellen Modus) ebenso bedienen läßt.

5.6.4 GNU Emacs

Dies ist der verbreitetste programmierbare Editor im UNIX-Bereich. Er ist als LISP-Maschine konzipiert und daher extrem flexibel und erweiterbar. Es existieren zahlreiche Zusatzpakete, mit denen der Emacs praktisch zu einer kompletten, kostenfreien Entwicklungsumgebung wird. Allerdings ist die verbreitetste Variante, der GNU-Emacs der Free Software Foundation, nicht in der Lage, Bilder darzustellen.

5.6.5 XEmacs

Der *XEmacs*[27] löst dieses Problem: Er stellt eine Reimplementierung des Emacs als echte X-Applikation dar, die auf dem Xt-Toolkit aufsetzt, ansonsten aber zum GNU Emacs kompatibel ist. Er verfügt dadurch jedoch insbesondere über die Fähigkeit, Graphiken in Form

[25] URL: http://www.cs.city.ac.uk/homes/njw/htmltext/htmltext.html
[26] URLs: http://www.inria.fr/CeBIT/1Dynamics-deu.html und http://symposia.inria.fr/
[27] URL: http://xemacs.cs.uiuc.edu

von X-Pixmaps darzustellen, was ihn als Basis für einen WYSIWYG-HyperSource-Editor geeignet macht. So existiert für diesen Editor bereits ein graphischer HTML-Browser-Modus *w3*.

Eine genauere Beschreibung des XEmacs und der Programmiersprache Emacs-LISP findet sich im Kapitel 6.

5.7 Zusammenfassung der Ansätze und ihrer Eigenschaften

Als Überblick wurden die oben beschriebenen Systeme hier noch einmal zusammengefaßt, soweit sie zum Editieren von HyperSource-Dokumenten im Prinzip geeignet sind. Für jedes System wurde tabellarisch aufgeführt, ob:

- Hypertexte mit interaktiven Querverweisen erzeugt werden können,
- Graphiken in Dokumente einzufügen sind,
- Programmiersprachen vom System beim Editieren unterstützt werden,
- Quelltext und Dokumentation gemäß der Idee des Literate Programming in einer Datei gemeinsam entwickelt werden können,
- das WYSIWYG-Prinzip umgesetzt wurde,
- das System mit einem Standard-Text- oder Editorsystem verwendbar ist,
- eine Implementierung des Systems unter Unix/X existiert.

System	Hypertext	Graphik	Sprachen	Lit. Prog.	WYSIWYG	Standardumg.	Unix/X
WEB	–	+	Pascal	+	–	+	+
CWEB	–	+	C	+	–	+	+
CLiP	–	+	beliebig	+	(+)[1]	+	–
Fold2Web	–	+	beliebig	+	–	+	–
FunnelWeb	–	+	beliebig	+	–	+	+
NuWEB	–	+	beliebig	+	–	+	+
NoWEB	+	+	beliebig	+	–	+	+
HyperTeX	+	+	–	–	–	+	+
LPW	+	+	–	+	+	–	–
CU_HTML	+	+	–	–	+	+	–
ANT_HTML	+	+	–	–	+	+	–
Neptune	+	+	beliebig	–	(+)[2]	+	+
HyperCode	+	+	beliebig	+	?	?	?
Gwydion	+	+	Dylan	+	?	?	?
[BroCze90][3]	+	+	C	+	–	–	–
[SamPom92][3]	+	-	C^{++}	+	–	–	+
Texinfo	+	–	–	–	–	+	+
OSE	–	–	C^{++}	–	–	+	+
Toolbook	+	+	–	–	+	–	–
tkWWW	+	+	–	–	+	–	+
HoTMetaL	+	+	–	–	+	–	+
GRIF+SIGMA	+	+	–	–	+	(+)[4]	+
vi	–	–	beliebig	–	–	+	+
GNU Emacs	+	–	beliebig	–	–	+	+
XEmacs	+	+	beliebig	–	+	+	+

In der Tabelle bedeutet:

+	Anforderung erfüllt;
–	Anforderung nicht erfüllt;
(+)	Anforderung teilweise erfüllt, siehe Anmerkung;
?	Status noch nicht bekannt.

Anmerkungen:

1. Je nach verwendetem Dokumentationssystem.
2. Je nach verwendeter Editor-Applikation.
3. Diese Systeme tragen keinen eigenen Namen, daher hier Referenzen auf die Veröffentlichungen.
4. Universaleditor, aber kommerziell und daher nicht sehr verbreitet.

Eine aktuelle Zusammenfassung von Werkzeugen speziell für das Schreiben von HTML-Dokumenten findet sich in [Paoli95].

Contrary to popular belief, the world is not ASCII.

— aus der XEmacs-Dokumentation

6 Entwurf

6.1 Entwurfsentscheidungen

Hier zunächst ein Überblick über die wichtigsten getroffenen Entwurfsentscheidungen mit kurzen Begründungen:

Basiseditor XEmacs: Als Ausgangspunkt für das zu entwickelnde System war ein Standardeditor gesucht, der die üblichen Programmiertätigkeiten bereits umfassend unterstützt und gut erweiterbar ist. Da ein WYSIWYG-System entworfen werden sollte, wurde auf Grund seiner Graphikfähigkeiten der *XEmacs*, eine Variante des bekannten GNU Emacs, ausgewählt. Er wird im Anschluß an diese Übersicht beschrieben.

System als Emacs-Zusatzmodus: Unabhängigkeit von der Programmiersprache, mit der das HyperSource-System verwendet wird, war ein weiteres Entwurfsziel. Daher wurde entschieden, das System in Emacs-Lisp als einen *Zusatzmodus* (Minor Mode) des Emacs zu entwerfen, der unabhängig vom aktuellen *Hauptmodus* (Major Mode) verwendbar ist.

Dateiformat HTML: Entscheidend war, daß die im Ansatz beschriebene Format- und Hypertextinformation nicht nur dem Entwickler mit einem Spezialeditor, sondern auch jedem späteren Benutzer des Codes zugute kommen sollte. Daher mußten diese Informationen in den Quelltext integriert, gewissermaßen also eine Art „Rich Source Format“ entwickelt werden, das zu einem weitverbreiteten Standard kompatibel ist. Hier bietet sich die *Hypertext Markup Language (HTML)* an, die sich seit der Entstehung des *World-Wide Web*[1] [Berners-Lee94] vor einigen Jahren zum Standard für strukturierte Hypertext-Dokumente entwickelt hat und in diesem Kapitel ebenfalls dargestellt wird. Sie bietet im Gegensatz zu Ansätzen wie beispielsweise dem *Rich Text Format (RTF)* von Microsoft den besonderen Vorteil, das HTML-Dateien reine Textdateien sind, die auch für Menschen lesbar sind.

[1] URL: http://www.w3.org/

6.2 Verwendete Werkzeuge, Sprachen, Formate und Standards

In diesem Abschnitt werden die verschiedenen Systeme und Standards kurz vorgestellt, die im Entwurf ausgewählt wurden.

6.2.1 Der Editor XEmacs und die Sprache Emacs-Lisp

Dieser Editor stellt eine Reimplementierung des GNU Emacs dar. Der GNU Emacs [Stallman81] ist ein verbreiteter Bildschirmeditor mit folgenden Eigenschaften:

Anpaßbar: Die zur Bedienung nötigen Tastaturkürzel können frei belegt werden, und das Verhalten der Kommandos paßt sich an die Art des editierten Textes an. Beispielsweise können automatisch Kommentarzeichen erzeugt werden, die der jeweiligen Programmiersprache entsprechen („`/* */`“ in C, „`;`“ in Lisp etc.).

Erweiterbar: Das System kann jederzeit durch neue, in Emacs-Lisp geschriebene, interpretierte Routinen erweitert werden, ohne daß dazu eine Neuübersetzung des Editors nötig wäre. Pakete existieren zur Unterstützung und Integration aller typischen Arbeiten eines Unix-Entwicklers: Schreiben von Quelltexten in verschiedensten Programmiersprachen, Compilieren, Debugging, Versionskontrolle, Erstellen von Dokumentation mit TeX, Lesen und Schreiben von E-Mail und News, FTP etc. Der Vorteil ist, daß eine einheitliche Grundbedienung (Bewegen des Cursors, Löschen von Text u. ä.) für alle Arbeiten existiert, was dem Entwickler die Inhomogenität verschiedener Werkzeuge erspart.

Selbstdokumentierend: Kontextsensitive Hilfe steht jederzeit zur Verfügung. Die Dokumentation jeder Emacs-Funktion erfolgt im Quelltext in einem speziellen Format, das gleichzeitig als Hilfe für den Benutzer verwendet werden kann.

Der *XEmacs* ist zum GNU Emacs auf Lisp-Ebene abwärtskompatibel, setzt jedoch im Gegensatz zu seinem Vorbild konsequent auf dem Xt-Toolkit des Fenstersystems X auf. Dadurch stellt er eine einheitliche Ereignisbehandlung zur Verfügung und bietet eine Reihe weiterer Funktionen insbesondere im graphischen Bereich. Im weiteren wird vom Emacs-Editor gesprochen, wenn die beschriebenen Eigenschaften für beide Varianten zutreffen.

Systemüberblick

Der grundsätzliche Aufbau des Emacs-Editors ist in Abb. 6.1 wiedergegeben.

Aus der Sicht eines Programmierers präsentiert sich der Emacs- Editor als eine Architektur aus 5 Schichten: Die Schnittstelle zum Betriebssystem sowie der Lisp-Interpreter sind in C implementiert. Diese Ebene ist für den Programmierer jedoch durch eine Lisp-Schnittstelle verdeckt, die die Grundfunktionalität eines Universaleditors bietet. Der Emacs-Basiseditor wirkt damit selbst wie eine initial geladene Lisp-Funktionsbibliothek.

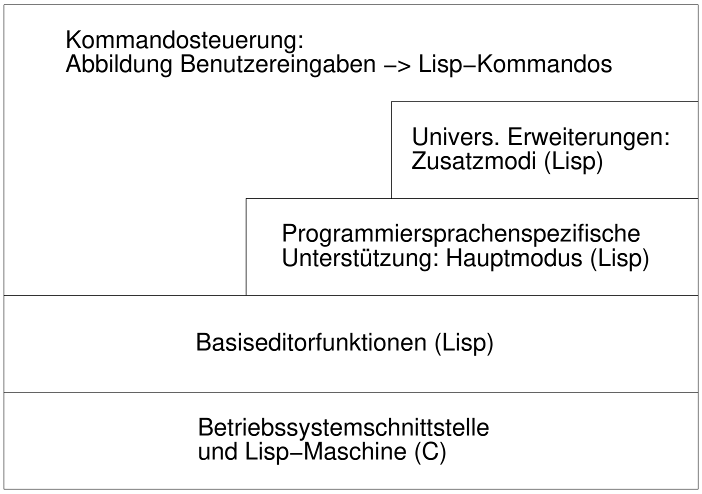


Abbildung 6.1: Architektur des XEmacs-Editors

Über dieser Schicht findet sich die aufgabenabhängige Unterstützung durch einen *Hauptmodus* sowie gegebenenfalls mehrere *Zusatzmodi* (s.u.). Der Benutzer sieht von alledem nichts, da alle seine Tastatureingaben in der zuoberst liegenden Kommandosteuerungsschicht über eine Hierarchie von Tastaturtabellen *(Keymaps)*, die vom aktuellen Haupt- und Zusatzmodus abhängen können, auf Lisp-Kommandos abgebildet werden.

Trotz dieser durchlässigen Schichtung ist die Sicht des Systems für den Emacs-Programmierer homogen: Alle Lisp-Funktionen des Editor-Systems sind einheitlich aufrufbar. Ihre Dokumentation ist stets auf die gleiche Weise über eine integrierte Hilfe abrufbar, was die Implementierungsarbeit in Lisp sehr erleichtert.

Im folgenden werden die für diese Arbeit wesentlichen Konzepte des XEmacs in Richtung zunehmender Abstraktion und Spezialisierung vorgestellt.

Emacs-Lisp

Emacs-Lisp ist nicht nur eine Sammlung von Editierfunktionen, sondern eine vollwertige Programmiersprache. Es ist eine Variante der funktionalen Programmiersprache *Lisp* (LISt Processing Language), die in den 50er Jahren am MIT für Forschungsprojekte der Künstlichen Intelligenz entwickelt wurde. Im Vergleich zum heutigen Standard *Common Lisp* wurden allerdings einige Spracheigenschaften vereinfacht, um den Speicherbedarf des Editorsystems zu senken, und es wurden vordefinierte Datentypen und Funktionen beispielsweise zur komfortablen Behandlung von Zeichenketten und Textpuffern hinzugefügt.

Für eine umfassende Einführung in Emacs-Lisp ist das *Emacs Lisp Reference Manual* [StaLewLibWin94] zu empfehlen. Hier werden nur die für diese Arbeit relevanten Aspekte der Sprache und des Editors beschrieben.

Folgende Eigenschaften charakterisieren zunächst allgemein die Sprache Emacs-Lisp:

Funktionales Paradigma: Ein Lisp-„Programm“ ist, ebenso wie eine Lisp-Funktion, eine

Folge von Lisp-Ausdrücken, die nacheinander ausgewertet werden. Der Wert des letzten Ausdrucks bestimmt den Rückgabewert der Funktion; Ein- und Ausgaben oder Variablenmanipulationen sind nur als Seiteneffekte bestimmter Funktionen möglich.

Interpretierbarkeit: Das Emacs-System basiert auf einem Lisp-Interpreter, der Funktionsdefinitionen interaktiv oder aus Quelltextdateien einlesen und ausführen kann. Dies erleichtert wie bei allen interpretierten Sprachen die Fehlersuche, da der aktuelle Zustand des gesamten Systems stets interaktiv abfragbar ist.

Daneben können fertige Pakete jedoch auch durch einen (selbst in Emacs-Lisp geschriebenen) Compiler, der Teil des Emacs-Editors ist, in schneller ausführbaren Byte-Code übersetzt werden. Die Geschwindigkeit von C wird dabei zwar trotzdem nicht erreicht, allerdings sind die beim Editieren üblicherweise anfallenden Berechnungen meist einfach genug, um ohne merkliche Verzögerung ausführbar zu bleiben.

Dynamische Erweiterbarkeit: Eine Erweiterung des Emacs auf Ebene der C-Routinen ist zwar möglich, aber zu vermeiden, da sie die Neuübersetzung des Editors erforderlich macht und damit die Verwendung des Pakets für andere Benutzer des Editors erheblich erschwert. Dagegen können in Lisp geschriebene Erweiterungen sogar ohne einen Neustart des Editors jederzeit dynamisch hinzugeladen werden und stehen sofort zur Verfügung.

Selbst-typisierende Objekte: In den meisten Programmiersprachen muß der Programmierer den Typ einer Variablen angeben. Dieser Typ ist dann dem Compiler bekannt, aber in den Daten selbst nicht repräsentiert. In Emacs-Lisp hingegen gibt es keine Typdeklarationen; eine Variable kann Werte beliebigen Typs aufnehmen. Dies bedeutet jedoch nicht, daß keine Typüberprüfung stattfindet: Der aktuelle Typ wird in der Variablen vermerkt und bei Zugriffsoperationen überprüft. Er kann mit verschiedenen Prädikaten vom Programmierer ermittelt werden.

Unbegrenzter Gültigkeitsbereich: Das Emacs-Lisp-System besitzt einen globalen Namensraum. Dies bedeutet, daß sämtliche *Symbole*, die an einer beliebigen Stelle, beispielsweise innerhalb einer Funktion, definiert werden, im Normalfall an jeder anderen Stelle (auch in einer anderen, unabhängigen Funktion) sichtbar sind. Es gibt spezielle Funktionen, um eine lokale Bindung von Namen (beispielsweise für lokale Variablen) zu erreichen. Aus diesem Grund hat sich die Konvention gebildet, alle Funktions- und Variablennamen, die das Paket nach außen zur Verfügung stellt, mit einem einheitlichen Präfix (beispielsweise `hypersrc-`) zu versehen, um Namenskollisionen mit anderen Erweiterungen zu vermeiden.

Dynamische Bindung: Trotz unbegrenzter Gültigkeit kann ein bereits gebundenes Symbol beispielsweise innerhalb einer Funktion F_1 durch einen speziellen Ausdruck eine lokale, andere Bindung erhalten, die die globale Bindung verdeckt. Diese Bindung bleibt aber während der gesamten Laufzeit der Funktion bestehen, so daß ihr Wert in einer anderen, innerhalb von F_1 aufgerufenen Funktion F_2 sichtbar bleibt. Dieser und der vorhergehende Punkt unterscheiden Lisp grundlegend von den meisten anderen Programmiersprachen wie beispielsweise C, in denen der Gültigkeitsbereich über die textuelle Anordnung des Quellcodes begrenzt wird und die Bindung statisch erfolgt.

Symbole

Das *Symbol* ist das grundlegendste Konzept von Lisp. Es ist ein Lisp-Objekt, das vier Komponenten enthält:

- einen eindeutigen *Namen*, der das Symbol für die textuelle Ein- und Ausgabe repräsentiert,
- einen *Variablenwert*, der den aktuellen Wert des Symbols als Variable enthält und für die Evaluation von Ausdrücken verwendet wird,
- einen *Funktionswert*, der die Funktionsdefinition für das Symbol enthält und benutzt wird, wenn das Symbol als Funktion in einem Ausdruck verwendet wird. Ein Symbol kann also sowohl eine Funktion als auch eine Variable repräsentieren, ohne daß es zu einem Namenskonflikt kommt. Beide Werte können auch undefiniert (`void`) sein.
- eine Liste von Attributen (*Property List*), die Eigenschaften des Symbols als „Name-Wert"-Paare speichern kann. Damit kann praktisch jedes Symbol als eine Sammlung von Daten (ähnlich einer Struktur in C) verwendet werden.

Datentypen

Emacs-Lisp stellt eine Reihe *primitiver Datentypen* zur Verfügung, darunter Ganzzahlen, Fließkommazahlen, Symbole, „Cons"-Zellen (ein Datentyp, der 2 Zeiger enthält und die Grundeinheit von Listen darstellt), Zeichen, Strings, Vektoren und Funktionen sowie speziell für Textverarbeitung benötigte Typen wie Textpuffer und Textfenster. Aber auch Datentypen für Prozesse, Streams und fenstersystemspezifische Ressourcen, beispielsweise Zeichensätze, sind vorhanden.

Dateien, Textpuffer und Textfenster

Für das Verständnis des Emacs-Editors ist es wichtig, den Unterschied zwischen diesen drei Konzepten zu kennen.

Eine *Datei* ist ein Objekt, das permanent unter einem eindeutigen Namen auf einem Medium gespeichert ist. Ein *Textpuffer* ist ein Emacs-Lisp-Objekt, das innerhalb des Emacs-Editorsystems existiert und Text und weitere Informationen enthalten kann. Ein Textpuffer kann mit einer Datei in einer *Visited*-Beziehung stehen, das heißt, sein ursprünglicher Inhalt entstand durch Einlesen der Datei, und nach Änderungen im Textpuffer soll er später voraussichtlich wieder in diese Datei zurückgeschrieben werden. Ein Textpuffer kann jedoch auch keine solche Beziehung besitzen, etwa wenn sein Inhalt automatisch oder aus der Ausgabe eines Programms erzeugt wurde. Ein Beispiel für solch einen Fall wäre ein Textpuffer, der die Statusmeldungen eines externen Übersetzerlaufs aufnimmt, um sie dem Entwickler innerhalb der Emacs-Umgebung zugänglich zu machen.

Eine Datei kann von mehreren Textpuffern gleichzeitig „besucht" werden; ein Textpuffer jedoch kann nur höchstens eine Datei „besuchen". Im ersteren Fall werden konkurrierende Änderungen über einen Zeitstempel-Mechanismus aufgelöst.

Ein *Textfenster* schließlich ist ebenfalls ein Emacs-Lisp-Objekt, das einen Teil des gesamten Emacs-Fensters (im Sinne des Fenstersystems) repräsentiert. Durch mehrere Textfenster, die nach dem Kachelungsprinzip über- und nebeneinander dargestellt werden können, ist es möglich, gleichzeitig den Inhalt mehrerer Textpuffer zu betrachten und zu editieren. Ein Textfenster stellt dabei immer irgendeinen der existierenden Textpuffer dar; ein Textpuffer kann jedoch gleichzeitig in mehreren Textfenstern betrachtet und editiert werden, um beispielsweise den Anfang und das Ende einer langen Datei simultan bearbeiten zu können.

Relationstheoretisch gesehen bilden Dateien, Textpuffer und Textfenster also eine {0,1}:{1,n}:{0,m}-Beziehung.

Haupt- und Zusatzmodi

Um das Editieren zu einem bestimmten Zweck zu erleichtern, befindet sich jeder Textpuffer zu einem Zeitpunkt in genau einem von mehreren möglichen *Hauptmodi* (Major Modes). Der Modus wird im allgemeinen über die Endung der besuchten Datei ermittelt, kann aber auch explizit gewählt werden. Programmiersprachen-Modi „kennen“ die Syntax der jeweiligen Programmiersprache und bieten so beispielsweise automatisches, sinnvolles Einrücken von Programmzeilen oder Vervollständigung nur teilweise ausgeschriebener Schlüsselworte der jeweiligen Sprache über eine Tastenkombination, was den Schreibaufwand reduziert.

Andere Modi dienen nicht dem Schreiben von Programmen, sondern bespielsweise dem Lesen von E-Mail. In diesen Fällen wird der anfängliche Textpufferinhalt aus einer Datei nicht einfach eingelesen, sondern konstruiert, und kann nicht beliebig editiert, sondern nur über bestimmte Tastaturkürzel („Lösche diese Nachricht“ u. ä.) verändert werden. Beim Abspeichern erfolgt die Rückwandlung des Textpufferinhalts in das entsprechende Dateiformat.

Während Hauptmodi sich also gegenseitig ausschließen, da immer nur ein Modus zur Zeit aktiviert sein kann, ist dies bei *Zusatzmodi* nicht der Fall: Sie haben die Aufgabe, zusätzliche Funktionalität zur Verfügung zu stellen, die in verschiedenen Hauptmodi einen Sinn ergibt. Ein Beispiel ist der „Auto-Fill“-Zusatzmodus, der beim Eingeben von Text beim Erreichen einer bestimmten Spalte automatisch einen Zeilenumbruch durchführt, wie man es von Textverarbeitungen her kennt. Zusatzmodi sind allerdings im allgemeinen wesentlich komplexer als Hauptmodi zu implementieren, da sie kaum Annahmen über ihre Umgebung machen können, wenn sie mit verschiedensten Haupt- und anderen Zusatzmodi funktionieren sollen.

Graphische Eigenschaften

Dieser Teil bezieht sich ausschließlich auf den *XEmacs*-Editor, der über weitreichendere Fähigkeiten in diesem Bereich verfügt.

Die Grundlage des Darstellungssubsystems bildet das *Glyph*, das ein graphisches Zeichen, also einen Buchstaben oder auch ein Rasterbild repräsentiert. Im einfachsten Fall befindet sich in einem Textpuffer lediglich Text, der in einem einheitlichen Zeichensatz als Folge von Glyphen dargestellt wird.

Zur besseren Übersicht ist es allerdings wünschenswert, einzelne Textpassagen in einem anderen Zeichensatz oder in einer anderen Farbe darzustellen oder beispielsweise Bilder im Textpuffer anzuzeigen. Dies ist durch folgende Konzepte möglich:

- Jedes Zeichen wird in einem bestimmten *Face* dargestellt. Dies ist eine mit einem Namen versehene Sammlung graphischer Attribute (Zeichensatz, Vorder- und Hintergrundfarbe etc.). Verschiedene Namen und ihr Anwendungsbereich sind bereits vordefiniert; so wird normaler Text im `default`-Face dargestellt, während die Statuszeile des Editors das `modeline`-Face verwendet.
- Um einen beliebigen Textabschnitt anders darzustellen, als es das für ihn zuständige Face spezifiziert, kann ein *Extent* definiert werden. Dies ist ein Lisp-Objekt, das eine Start- und Endposition für den darzustellenden Text sowie eine Reihe von Attributen enthalten kann, darunter ein eigenes Face. Extents können sich überlappen; solange sich die Spezifikationen nicht widersprechen, werden die Attribute gemischt, ansonsten löst eine Prioritätsregelung den Konflikt auf. Extents können sogar ein eigenes *Start- oder Endglyph* (also Bild oder Text) besitzen. Dieses Glyph kann über ein wählbares *Layout* auf verschiedene Weisen angezeigt werden, beispielsweise direkt am Anfang bzw. Ende des Extents oder im Rand neben dem Textpuffer.
- *Annotations* schließlich basieren auf dem Konzept von Extents, bieten aber daneben noch die Möglichkeit, Ereignisse, die stattfinden, während sich der Mauszeiger über dem Start- bzw. Endglyph befindet, gesondert zu behandeln. Ihre Anfangs- und Endposition als Extent überdeckt normalerweise nur ein Zeichen des Textpuffers, an dem sie damit unsichtbar „angeheftet" sind. Sichtbar werden sie erst durch das Start- bzw. Endglyph, das als Text oder Bild üblicherweise im Rand neben dem Textpuffer dargestellt wird. Sie bieten also im Prinzip eine Möglichkeit, Randanmerkungen an einzelnen Stellen im Textpuffer anzubringen. Allerdings ist noch keine Benutzerschnittstelle implementiert worden, über die der Benutzer diese Funktionalität zur Dokumentation seiner Programmtexte einsetzen könnte.
- Graphiken innerhalb des Textpuffers sind durch das Einfügen von Glyphen des Typs *Pixmap* möglich, die aus Rasterbildern im *XPM*-Format [LeHors91] erzeugt werden. Auch hier existiert aber keine Schnittstelle zum interaktiven Einsatz durch den Benutzer.

Weitere Informationen zu den für das HyperSource-System relevanten Eigenschaften des XEmacs-Editors finden sich in der Beschreibung der Implementierung im Kapitel 7.

6.2.2 Die Hypertext Markup Language HTML

HTML ist das Format, in dem die Dokumente des *World-Wide Web* erstellt werden. Es basiert auf dem *SGML*-Format, das daher zunächst kurz vorgestellt werden soll.

SGML

Ein Hauptproblem beim Austausch elektronischer Dokumente zwischen verschiedenen Hard- und Softwareumgebungen ist die Inkompatibilität der Dokumentformate. Daher entstand in den sechziger Jahren unter der Federführung der amerikanischen Graphic Communications Association (GCA) erstmals die Idee, die spezifische Formatierung und die inhaltliche Struktur von Dokumenten konzeptuell zu trennen.

Das *GenCode*-Projekt der GCA stellte fest, daß eine spezifische, prozedurale Codierung der Formatierung von Nachteil ist. Stattdessen sollte die *logische Struktur* eines Dokuments mit einer generischen, deskriptiven Codierung erfaßt werden. Statt also beispielsweise Festlegungen vorzunehmen wie „Setze das folgende Wort in Times, 18 Punkt, fett", um den Effekt einer Überschrift zu erreichen, sollte besser die Formatieranweisung selbst lauten „Setze das folgende Wort als Überschrift". Die Umsetzung solcher Anweisungen in eine konkrete Formatierung kann dann das darstellende System je nach seinen Möglichkeiten vornehmen, weil ihm die logische Struktur des Dokuments bekannt ist.

1969 entwickelten Charles Goldfarb und einige andere Mitarbeiter bei IBM die *Generalized Markup Language (GML)*, in der diese Konzepte umgesetzt wurden. Diese Sprache wurde in den siebziger Jahren weiterentwickelt und schließlich 1986 von der ISO als *SGML* (Standard Generalized Markup Language) verabschiedet.

Die Dokumentstruktur wird in SGML über *Markup* beschrieben: Innerhalb des Dokumenttextes finden sich spezielle Bezeichner (*Tags)*, die den Text strukturieren. Durch Anfangs- und End-Tags wird der Text in eine hierarchische Verschachtelung von *Elementen* zerlegt, die jeweils einer gemeinsamen Struktur (beispielsweise einer Überschrift) angehören. Elemente können dabei über *Attribute* mit weiteren Eigenschaften versehen werden.

Dabei definiert SGML nicht eine feste Menge von Elementen wie „Hauptüberschrift", „Unterüberschrift" etc., sondern die Definition der verwendbaren Elemente ist selbst als Vorspann in das SGML-Dokument integriert. Ferner wird festgelegt, welche anderen Elemente innerhalb eines Elements vorkommen dürfen *(Inhaltsmodell).*

Schließlich können in einem SGML-Dokument noch *Entitäten* definiert werden, die als Platzhalter für Textteile verwendet werden können.

Ein SGML-Dokument besteht daher üblicherweise aus drei Teilen:

1. **SGML-Deklaration** mit Festlegung des zu verwendenden Zeichensatzes und der SGML-Sonderzeichen,

2. **DTD** (Dokumenttypdefinition) mit den zu verwendenden Tagelementen, Attributen und Entitäten,

3. **Dokumentinstanz** mit dem eigentlichen Inhalt des Dokuments, in der die zuvor definierten Elemente zur logischen Strukturierung verwendet werden.

Die DTD wird im allgemeinen für eine ganze Klasse von Dokumenten (beispielsweise Projektberichte oder Aktennotizen) einheitlich gewählt und daher nicht in jedem Dokument repliziert, sondern meist ausgelagert und durch eine spezielle Anweisung vom SGML-Parser während des Einlesens des Dokuments eingebunden.

Als *Sonderzeichen* zur Kennzeichnung von Markup verwendet SGML standardmäßig folgende Zeichen, die jedoch in der SGML-Deklaration auch umdefiniert werden können:

- `<! ... >` fassen eine Markup-Deklaration ein,
- `<? ... >` fassen eine Verarbeitungsanweisung ein,
- `< ... >` fassen eine Markup-Anweisung vom Typ Start-Tag ein,
- `</ ... >` fassen eine Markup-Anweisung vom Typ End-Tag ein,
- `& ... ;` fassen eine Referenz auf eine Entität ein.

Da die meisten Dokumente die DTD nicht selbst enthalten, sind Start- und End-Tags die häufigsten Formen von Markup in typischen SGML-Dokumenten.

SGML ist nicht nur die Grundlage für HTML, sondern wird in vielen Institutionen und Großunternehmen benutzt, um einheitliche Dokumentformate beispielsweise für interne Berichte zu definieren. Die Sprache wurde hier nur kurz vorgestellt; das Standardwerk zu diesem Thema ist *The SGML Handbook* [Goldfarb94], während [Herwijnen94] eine praktische Einführung in SGML bietet.

HTML

SGML kann als eine „Meta-Sprache" betrachtet werden, denn erst durch die DTD wird eine konkrete Menge von Tags definiert, die in einem Dokument verwendet werden können. HTML ist letztlich nichts anderes als eine solche DTD, natürlich mit einer begleitenden semantischen Konvention, die festlegt, welche *Bedeutung* die einzelnen Elemente haben sollen, wie sie also sinnvollerweise einzusetzen sind. Die DTD definiert bestimmte Tags, um beispielsweise Überschriften, Absätze und Aufzählungen, aber auch Hypertext-Links und integrierte Bilder zu repräsentieren.

HTML wurde 1990 von Tim Berners-Lee am CERN für das World-Wide-Web-Projekt als eine einheitliche Dokumentbeschreibungssprache für Hypertextdokumente mit integrierten Graphiken entwickelt [Berners-Lee94]. Durch die überaus rasche Ausdehnung des World-Wide Web ist HTML zum de-facto-Standard für die Beschreibung hypertextartig strukturierter, verteilter Dokumente geworden.

Derzeit wird HTML in der Version 2.0 von den meisten HTML-Browsern (siehe Kapitel 5) verstanden. Inzwischen liegt eine Spezifikation von HTML in der Version 3.0 vor. Eine wesentliche Neuerung war die Einführung von Tags zur Definition von Tabellen, die wiederum fast beliebige andere Elemente in ihren Zellen enthalten können, wodurch sie auch zur Definition eines Seitenlayouts geeignet sind. Diese Eigenschaft war für das zu entwickelnde HyperSource-System von Bedeutung, denn die Darstellung von textuellen oder graphischen Randanmerkungen erfordert ein zweispaltiges Layout. Daher wird im folgenden der Umfang von HTML in der Version 3.0 skizziert.

HTML 3.0 stellt unter anderem Tags für folgende Strukturen zur Verfügung:

- Überschriften in verschiedenen Abstufungen,
- Absätze,

- Hypertext-Links auf beliebige andere Dokumente als Ganzes oder auf einzelne Stellen in anderen HTML-Dokumenten,
- Logische (z. B. `<EM>` für „Emphasize“, „Hervorheben“) und typographische (z. B. `<IT>` für „Italics“, „Kursivschrift“) Elemente (logische sind zwecks Portabilität zu bevorzugen),
- Integrierte Bilder (im GIF-Dateiformat),
- Numerierte und unnumerierte Listen,
- Tabellen,
- Formeln,
- Vorformatierter Text,
- Zitate,
- Formulare, die interaktiv ausgefüllt werden können.

Um einen Eindruck von HTML zu erhalten, ist vermutlich ein Beispiel am geeignetsten. Deshalb ist im folgenden eine einfache Beispieldatei wiedergegeben.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.0//EN//">
<HTML>
<HEAD>
<TITLE>Beispielseite</TITLE>
<!-- Changed by: Jan Borchers, 21-May-1995 -->
<!-- Ein Kommentar -->
</HEAD>

<BODY>
<H1> Gro&szlig;e &Uuml;berschrift </H1>
<H2> Kleinere &Uuml;berschrift </H2>

<P> Dies ist ein Absatz mit normalem Flie&szlig;text. Er wird
automatisch umgebrochen.


<P> <IMG SRC="ball.gif"> Am Beginn dieses Absatzes steht ein
GIF-Bild. Hier wird etwas <em>hervorgehoben</em>.  Anschlie&szlig;end folgt
eine horizontale Trennlinie.  <HR>

<P> Und hier ein Verweis auf die Homepage von
<A HREF="http://www.sgi.com">Silicon Graphics</A>.

</BODY>
</HTML>
```

Durch die erste Zeile wird dem SGML-Parser mitgeteilt, daß das vorliegende Dokument eine externe DTD verwendet. Das restliche Dokument besteht stets aus einem `HTML`-Element, das sich in einen Vorspann (`HEAD`) und den auf der Seite darzustellenden Textrumpf (`BODY`) gliedert. Der Vorspann enthält hingegen einen Titel, der nicht auf der Seite dargestellt wird, sondern als Kurzinformation für das Dokument dient. Auf diese Weise genügt es beispielsweise zur Suche oft, den Vorspann einzulesen.

`H1` und `H2` sind Überschriftselemente, `P` markiert einen neuen Absatz (hier ist kein End-Tag notwendig), und `IMG` spezifiziert ein einzubindendes Bild.

Das Ergebnis der Darstellung dieser Seite auf einem WWW-Browser ist in Abb. 6.2 zu sehen.

Abbildung 6.2: Darstellung obiger HTML-Seite durch einen WWW-Client (hier Netscape).

HTML soll hier nicht erschöpfend behandelt werden. Die Beschreibung der Implementierung in Kapitel 7 stellt diejenigen Elemente von HTML 3.0 noch näher vor, die für das HyperSource-System relevant sind.

Ansonsten ist das (allerdings unleserliche) Referenzdokument die *HTML 3.0 DTD*[2], zu der jedoch auch eine verständlichere Sprachspezifikation[3] existiert. Daneben gibt es im WWW verschiedene Einführungen in HTML. Die wichtigsten Informationen finden sich auf dem Server des *World-Wide Web Consortium*[4]. Gedruckte Informationen wie beispielsweise [DecRan94] sind hier weniger zu empfehlen, da sie bei Erscheinen meist bereits veraltet sind.

[2] http://www.w3.org/hypertext/WWW/MarkUp/html3-dtd.txt
[3] URL: http://www.hpl.hp.co.uk/people/dsr/html/CoverPage.html
[4] URL: http://www.w3.org/

Use the SOURCE, Luke!
— Programmierer-Weisheit, frei nach dem Film „Star Wars“

7 Implementierung

7.1 Datenflußdiagramm

Zur ersten Übersicht über die Implementierung betrachte man das Datenflußdiagramm in Abb. 7.1. Sie macht am Beispiel einer C-Quelldatei deutlich, welche verschiedenen Zustände eine mit dem HyperSource-System entwickelte Programmdatei durchläuft.

- Zu Beginn (Punkt 1) sei eine bereits existierende oder noch leere Quelltextdatei `bsp.c` gegeben. Diese wird mit dem XEmacs-Programmeditor geöffnet. Dabei wird wie gewöhnlich automatische derjenige Hauptmodus gewählt, der für das Editieren von C-Programmen vorgesehen ist.
- Als nächstes wird der HyperSource-Modus als Zusatzmodus aktiviert (Punkt 2). Der textuelle Inhalt des Textpuffers ändert sich nicht; es wird lediglich ein Rand für Anmerkungen und Bilder eingeblendet. Ferner ändert sich der Name des Textpuffers in *bsp.c.hs*, um zu signalisieren, daß die Datei beim Abspeichern im HyperSource-Format gesichert wird. Ein Menü mit den wichtigsten Benutzerbefehlen wird ebenfalls eingerichtet.
- Nun kann einerseits der eigentliche Quelltext im Textpuffer mit den gewohnten Kommandos editiert werden, zusätzlich stehen aber Formatier- und Kommentieranweisungen zur Verfügung, mit denen im Quelltext Überschriften und Querverweise sowie am Rand Anmerkungen, Bilder und Formeln eingefügt werden können (Punkt 3).
- Das normale Sichern des Programmtexts speichert dann den Text mit den eingebrachten Formatierungen und Kommentaren im HTML-3.0-Dateiformat (Punkt 4) als `.c.hs`-Datei. Um hingegen compilierbaren Quelltext zu erhalten, ist auch ein Abspeichern (Exportieren) ohne Formatinformation als `.c`-Datei möglich (Punkt 5). Diese Datei führt anschließend mit den üblichen Schritten (Übersetzen und Binden) zu einem ausführbaren Programm (Punkt 6).

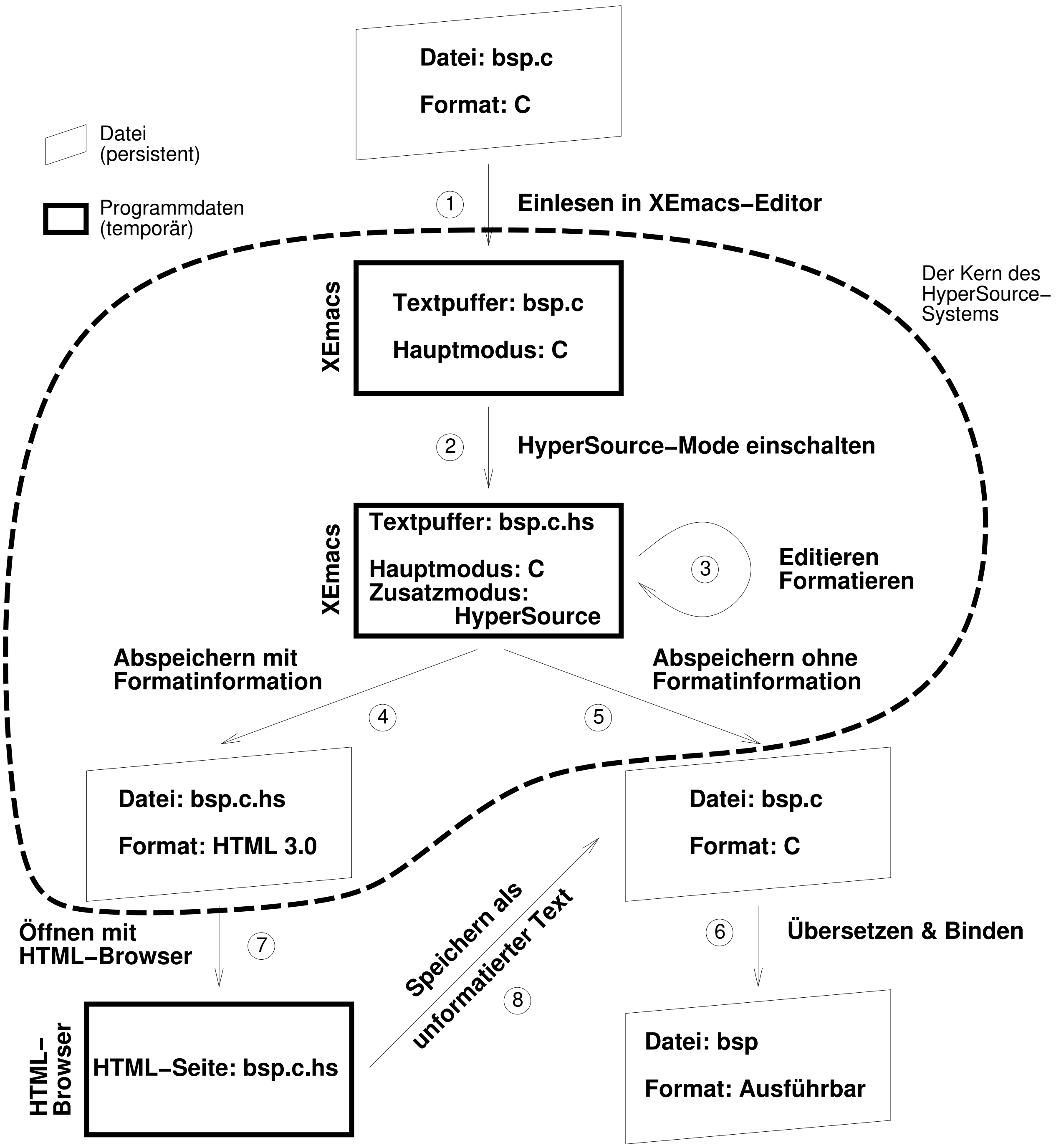


Abbildung 7.1: Datenfluss im HyperSource-System am Beispiel eines C-Quelltextes.

- Die formatierte Version kann aber auch — beispielsweise von einer anderen Person, die den Quelltext weiterverwenden, warten oder aus anderen Gründen verstehen und benutzen möchte, jedoch nicht über das HyperSource-System verfügt — mit einem HTML-Browser (siehe Kapitel 5) geöffnet werden (Punkt 7). Sie stellt sich dort genauso dar wie zuvor im HyperSource-Programmeditor; es handelt sich also um ein

WYSIWYG-System.

- Der Quelltext muß aber für andere nicht nur gut lesbar, sondern auch noch compilierbar sein. Dies ist möglich, indem der Benutzer den HTML-Browser anweist, die vorliegende Datei als reinen Text ohne Formatinformationen zu sichern. Dadurch werden die HTML-spezifischen Anweisungen aus dem Text entfernt. Das HyperSource-Format ist so konzipiert, daß diese Wandlung den ursprünglichen, compilierbaren Code wiederherstellt (Punkt 8). Damit steht auch anderen Personen, die die HyperSource-Umgebung nicht verwenden, eine Weiterverwendung des Quelltexts offen.

7.2 Modulübersicht

Das System wurde als eine Sammlung von Emacs-Lisp-Modulen implementiert (siehe Abb. 7.2). Sie sind im folgenden kurz beschrieben:

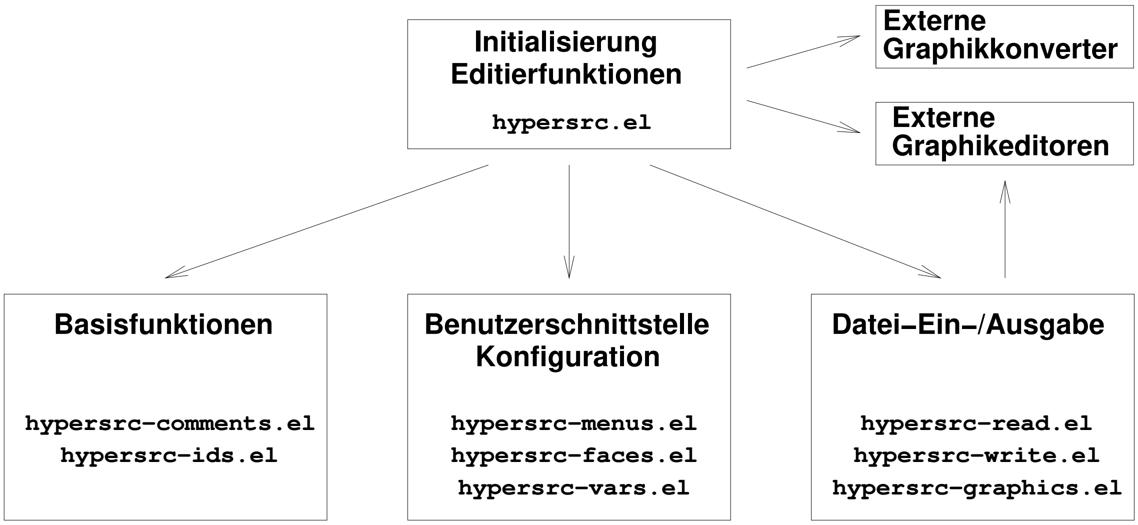


Abbildung 7.2: Übersicht über die Emacs-Lisp-Module des HyperSource-Modus

hypersrc.el: Dies ist das Hauptmodul des Systems. Es enthält den Initialisierungscode und sämtliche Editierfunktionen (Einfügen von Randkommentaren, Querverweisen etc.).

hypersrc-write.el: Die Funktionen in diesem Modul dienen dazu, einen im XEmacs formatierten Quelltext als HTML-3.0-Datei abzuspeichern.

hypersrc-read.el: Hier findet sich die Parsing-Funktionalität, mit der abgespeicherte HyperSource-Dateien wieder eingelesen werden. HTML-Anweisungen werden in interne Formatierungsinformationen umgewandelt.

hypersrc-graphics.el: Bilder können in einer Vielzahl von Formaten vorliegen. In diesem Modul finden sich generische Konverter, die den passenden Graphikkonverter bestimmen und auf das Originalbild anwenden, um eine im XEmacs bzw. in HTML einbindbare Darstellung zu erhalten.

hypersrc-menus.el: Dieses Modul definiert das dem Benutzer zur Verfügung stehende HyperSource-Funktionsmenü und Routinen zu seiner Verwaltung.

hypersrc-comments.el: Zugriffsfunktionen, um je nach Programmiersprache korrekte Kommentarzeichen zu erhalten, sind hier zusammengefaßt.

hypersrc-ids.el: Querverweise, die auf einzelne Stellen in anderen Dateien verweisen, müssen ihr Ziel eindeutig und invariant gegenüber Änderungen innerhalb der Zieldatei identifizieren können. Dieses Modul erzeugt dazu persistent eindeutige Namen für die Ziele von Querverweisen.

hypersrc-faces.el: Hier sind die vom HyperSource-Modus verwendeten graphischen Attribute (Zeichensätze, Farben etc.) festgelegt.

hypersrc-vars.el: Alle sonstigen Variablen, die vom Benutzer des Systems zur Konfiguration verändert werden können (beispielsweise die Namen externer Graphikprogramme), sind in diesem Modul zusammengefaßt.

Das HyperSource-System kann externe Graphikprogramme benutzen, um dem Benutzer ein integriertes Editieren von Bildern zu ermöglichen. Dabei werden externe Standard-Graphikkonverter verwendet, um die Bilder in ein für das System lesbares Format zu wandeln.

Da die verfügbaren Konverter und Graphikprogramme nicht alle konsistent in ihrer Bedienung sind, ihre Integration in das HyperSource-System jedoch über eine einheitliche Schnittstelle erfolgen sollte, wurden für die nicht-konformen Programme Hüllen in Form von Shellskripten entwickelt, die die Übergabe von Argumenten und bei den Konvertern die Ausgabe der Ergebnisdatei vereinheitlichen.[1]

Anschließend an diese Übersicht werden nun die einzelnen Funktionen des HyperSource-Systems und ihre Implementierung vorgestellt. Dabei werden vor allem Implementierungsentscheidungen aufgezeigt und begründet, die nicht offensichtlich waren.

7.3 Initialisierung und Start des HyperSource-Modus

Der HyperSource-Modus kann auf zwei Arten aktiviert werden:

- Während des Editierens einer normalen Programmdatei kann der Modus mit einem gezielten Kommando (`M-x hypersrc-mode`) eingeschaltet werden. Daraufhin wird der Name des Textpuffers um die Endung `.hs` ergänzt, der linke Rand eingerichtet, das HyperSource-Menü installiert sowie die modifizierten Routinen zum Lesen und Schreiben von HyperSource-Dateien (s. u.) eingehängt.

[1] Das Postscript-Zeichenprogramm *idraw* kann beispielsweise aus unerfindlichen Gründen nicht mit einer neuen Datei als Argument aufgerufen werden. Ein Skript *idraw.sh* beseitigt diesen Mangel, indem es in diesem Falle vor dem Aufruf des eigentlichen Programms eine leere Datei erzeugt. Der HyperSource-Benutzer wird dadurch von dieser Inkonsistenz verschont.

- Der Modus wird aber auch automatisch aktiviert, wenn eine HyperSource-Datei geladen wird. In diesem Fall finden dieselben Installationen wie oben statt, zusätzlich aber wird natürlich der Inhalt der zu öffnenden Datei sofort über die Parsing-Routine eingelesen, so daß die HTML-Anweisungen in Format- und Kommentarobjekte gewandelt werden.

Der zweite Punkt warf einige Probleme auf: Aufgrund der Dateiendung wird im Emacs der Hauptmodus ausgewählt. Es gibt zwar auch Möglichkeiten, diese Zuordnung aufzuheben und andere Einstellungen automatisch beim Laden einer Datei vorzunehmen, dies erfordert aber das Einfügen Emacs-spezifischer Anweisungen in die Datei. Dies wurde verworfen, um das Dateiformat nicht mit Anweisungen für einen speziellen Editor zu vermischen.

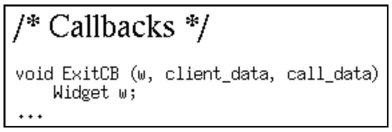
Stattdessen wurde ein Weg gewählt, durch den die existierenden Hauptmodi weiterhin für ihre normalen Dateien, aber auch für entsprechende HyperSource-Dateien aktiviert werden. Der HyperSource-Zusatzmodus wird dann je nach Art der Datei (normal oder HyperSource) automatisch hinzugeschaltet. Dies wird durch folgende Schritte möglich:

- In der XEmacs-internen Tabelle `auto-mode-alist`, die Dateiendungen auf Hauptmodi abbildet, wurde für alle gefundenen Endungen ein weiterer Eintrag hinzugefügt, der denselben Hauptmodus auch für Dateien startet, die zusätzlich über die Endung `.hs` (für HyperSource) verfügen.

  So findet sich in dieser Tabelle beispielsweise ein Eintrag, der `.c`-Dateien im `c-mode` startet. Beim Laden des HyperSource-Pakets wird daher ein weiterer Eintrag in die Tabelle eingefügt, so daß auch `.c.hs`-Dateien im `c-mode` geöffnet werden.

- Für alle Modi, in denen der HyperSource-Mode benutzt werden soll, müssen nun nur noch zwei Zeilen in der Konfigurationsdatei `hypersrc-vars.el` ergänzt werden: Eine, die den HyperSource-Modus in den *Hook* des jeweiligen Modus, einer Liste von Funktionen, die beim Start des Modus ausgeführt werden, einträgt, und die Angabe, welche Kommentarbegrenzer die jeweilige Sprache verwendet. Wie diese Eintragungen aussehen, kann in genannter Datei oder der Online-Hilfsdokumentation der Variablen `hypersrc-mode-comment-alist` nachgesehen werden. Für die Sprachen C, C^{++}, Objective C und Pascal sind diese Eintragungen bereits vorhanden.

- Beim Laden einer Datei führt ein so präparierter Hauptmodus dann unter anderem die Routine `hypersrc-loading-mode` aus. Diese untersucht den Quelltext, um festzustellen, ob es sich um eine HyperSource-Datei handelt. Wenn ja, wird der HyperSource-Modus wie beschrieben eingeschaltet und die Datei entsprechend geladen und transformiert; ansonsten bleibt diese Routine inaktiv, und der Hauptmodus arbeitet unverändert.



7.4 Überschriften

Um einen Quelltext lesbarer zu gestalten, war zunächst die Möglichkeit zu schaffen, Teile des Textes typographisch hervorzuheben. Auf diese Weise soll dem Entwickler das Umrahmen von Text mit Kästchen aus Kommentarzeichen oder ähnlichem erspart bleiben.

Der XEmacs-Editor bietet zwar die Möglichkeit, Abschnitte des Textpuffers in verschiedenen Zeichensätzen, Schriftschnitten und Farben darzustellen, allerdings wird diese Funktionalität nur von Paketen genutzt, um bestimmte Passagen (beispielsweise die Titelzeile einer E-Mail) automatisch hervorzuheben. Für den Benutzer existieren keine Kommandos, um solche Hervorhebungen selbst interaktiv vorzunehmen.

Dies ist verständlich, da solche Formatinformationen ja ohnehin nicht mit dem Text abgespeichert werden können, solange der Inhalt des Textpuffers 1:1 als ASCII-Datei gesichert wird. Durch die Einführung eines mächtigeren Dateiformats (HTML) jedoch kann jetzt die Formatierung bewahrt bleiben.

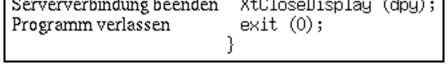
Daher wurde die Funktion `hypersrc-edit-create-header-from-region` implementiert, die für den aktuell markierten Bereich des Textpuffers einen Extent erzeugt, der diesen Abschnitt in einem anderen, größeren Zeichensatz darstellt. Die Größe kann vom Benutzer gewählt werden. Die Anzahl der Alternativen (6) entspricht der Anzahl verfügbarer Überschriftsstufen in HTML.

Wichtig ist zu erwähnen, daß durch das Erzeugen einer Überschrift der Inhalt des Textpuffers *nicht* verändert wird. Es wird lediglich ein *Extent* (siehe Seite 41) als internes Lisp-Objekt erzeugt, der dafür sorgt, daß der überdeckte Textabschnitt durch den Editor zukünftig mit den gewählten Attributen dargestellt wird. Beim Abspeichern muß dieser Extent abgearbeitet und in einzufügende HTML-Anweisungen umgesetzt werden, sonst ist er mit dem Schließen der Datei verloren.

Ein Vorteil von Extents als Attributierungsobjekten ist, daß sie ihre Anfangs- und Endposition automatisch mitführen, wenn sich der Inhalt des Textpuffers durch Editiervorgänge ändert. Wird also vor der Überschrift Text eingefügt, so verschiebt sich der Extent mit dem zu markierenden Überschriftstext einheitlich nach hinten. Es kann sogar der Text der Überschrift, also innerhalb des Extents selbst verändert werden; in diesem Falle paßt der Extent seine Größe automatisch an. Dies erleichtert das Editieren, da Überschriften weiterhin wie der restliche Text bearbeitet werden können. Auch beim Kopieren von Textblöcken wird die Extentinformation mitkopiert.

Ferner wurden Funktionen implementiert, um erzeugte Überschriften in ihrer Größe zu verändern oder auch zu entfernen (ohne dabei den Text der Überschrift zu löschen).

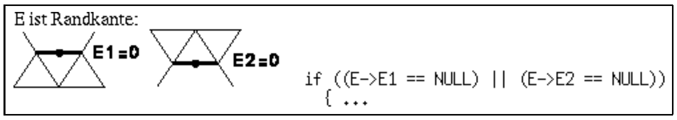
7.5 Textuelle Randanmerkungen



Eine weitere Forderung lautete, neben dem Quelltext abgesetzte Anmerkungen anbringen zu können, die mit dem normalen Editieren des Programms nicht interferieren. Diese Anmerkungen wurden mit Hilfe von *Annotations* (siehe S. 41) implementiert.

Die Randanmerkungen werden am Beginn der aktuellen Programmzeile angeheftet, und der für die Anmerkung eingegebene Text wird als Start-Glyph verwendet, um damit eine Annotation zu erzeugen, so daß der Text neben der aktuellen Zeile angezeigt wird. Dafür wird ein verkleinerter Zeichensatz verwendet, wie es in der klassischen Typographie für Marginalien empfohlen wird (siehe z. B. [Siemoneit89]).

Über einen getrennten Menüpunkt kann der linke Rand interaktiv in seiner Breite verändert werden.

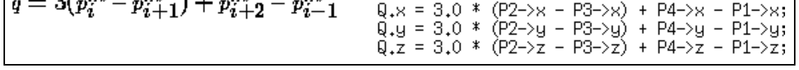


7.6 Bilder

Bilder wurden ebenfalls als Randanmerkungen mit Hilfe von Annotations implementiert. Ein zu bewältigendes Problem dabei war die Vielfalt der Graphikprogramme und Dateiformate, die der Entwickler möglicherweise einsetzen will. Innerhalb des Textpuffers können nur verhältnismäßig kleine Bilder sinnvoll untergebracht werden; es sollte aber auch möglich sein, beliebig große Bilder als Dokumentation mit dem Quelltext zu verbinden. Ferner sollte das Editieren dieser Bilder mit verschiedenen Graphikprogrammen direkt aus dem Editor heraus möglich sein.

Zur Lösung dieses Konflikts wurde ein mehrstufiges Bildkonzept entwickelt:

- Das *Originalbild* ist eine Graphikdatei, die in einem beliebigen Format vorliegen darf. Das HyperSource-System bestimmt das passende Graphikprogramm über eine erweiterbare Tabelle in Abhängigkeit von der Dateiendung. So enthält diese Tabelle beispielsweise einen Eintrag, in dem für `.ps`-Dateien das Programm `idraw` als Editor festgelegt wird. Der Benutzer arbeitet also stets nur mit dem Originalbild.
- Aus diesem Bild wird dann über das „Portable Pixmap Format (PPM)“ als Zwischenformat ein *Inline-Bild* im „X Pixmap Format (XPM)“ erzeugt und so skaliert, daß es in den dafür vorgesehenen Rand des XEmacs-Textpuffers hineinpaßt. Für diese Wandlung enthält die oben erwähnte Tabelle für jedes Dateiformat den Namen einer Konvertierroutine, die aus dem Bild ein PPM-Bild erzeugt. Diese Konverter sind für alle gängigen Bildformate verfügbar; die meisten sind im `netpbm`-Paket[2] zusammengefaßt. Das Inline-Bild dient gleichzeitig als „Vorschau“ für das eventuell detailreichere Original und als Querverweis, um dieses Original zu editieren.
- Beim Abspeichern wird das Originalbild hingegen über das PPM-Format in das HTML-konforme GIF-Format konvertiert, so daß es auch bei Betrachtung mit einem HTML-Browser als Inline-Bild dargestellt werden kann. Wiederum ist das Inline-Bild dabei Vorschau und aktiver Verweis auf das Original zugleich.



7.7 Formeln

Die Stärke des Konzepts von Original- und Inline-Bild zeigt sich, wenn man betrachtet, wie das System das Einfügen mathematischer Formeln ermöglicht:

[2] URL: ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM

1. Als „Originalbild" dient in diesem Falle eine Textdatei mit der Endung `.fml`, die beliebigen LaTeX-formatierten Text enthalten kann:

```
B\'ezier-Kurve:\\
$ {\bf p}(x)=\sum_{i=0}^{n-1}{\bf b}_i B^n_i(x) $
```

2. Der eigens als Shellskript geschriebene `textoppm`-Konverter ergänzt diese Datei nun zunächst um einen Standard-LaTeX-Dokumentrahmen, damit auf die üblichen LaTeX-Kommandos zurückgegriffen werden kann:

```
\documentstyle[12pt]{article}
\thispagestyle{empty}
\begin{document}
B\'ezier-Kurve:\\
$ {\bf p}(x)=\sum_{i=0}^{n-1}{\bf b}_i B^n_i(x) $
\end{document}
```

3. Diese Datei wird dann mit LaTeX in eine DVI-Datei und mit `dvips` in eine PostScript-Datei gewandelt. Anschließend erfolgt eine Konvertierung in das PPM-Format sowie eine Skalierung und ggf. Quantisierung, um überflüssige Antialiasing-Graustufen aus der Skalierung zu reduzieren. Nach der Wandlung in das XPM-Format mit einem weiteren Konverter erscheint dann die Formel im Randbereich des HyperSource-Textpuffers:

Bézier-Kurve:
$\mathbf{p}(x) = \sum_{i=0}^{n-1} \mathbf{b}_i B_i^n(x)$

Auf diese Weise sind TeX-Formeln nur ein besonderer Fall von Bildern, mit einem eigenen Konverter nach PPM und einem eigenen Editor (nämlich dem XEmacs selbst).

Übrigens definiert der Standard HTML 3.0 auch eine eigene Formelsyntax; diese ist jedoch nicht voll zum TeX-Format kompatibel, und es existiert außer dem experimentellen *Arena*-Browser keine Applikation, die dieses Format darstellen kann. Ihre Verwendung hätte außerdem eine Sonderbehandlung von Formeln erfordert, da sie sich dann nicht mehr in das Konzept von Originaldatei und Inline-Bild eingefügt hätte. In Zukunft ist aber denkbar, daß ein Konverter verfügbar wird, der HTML-Formeln in Graphiken oder TeX-Formeln in HTML-Formeln umwandelt. Mit solchen Werkzeugen wäre dann eine Sonderbehandlung von Formeln realisierbar, die den Vorteil einer verbesserten graphischen Darstellungsqualität bei der Betrachtung mit einem HTML-Browser mit sich brächte.

```
P->new = 1;
P      = PointTyp_Get(N);
E->R   = P;
```

7.8 Querverweise

Neben den einzufügenden Randbildern lautete eine weitere wichtige Forderung, es solle möglich sein, innerhalb des Quelltextes und vom Quelltext zur Dokumentation Querverweise zu erzeugen. Die Quelle eines HyperSource-Verweises ist stets ein zuvor markierter Bereich in einem HyperSource-Textpuffer. Der Bereich wird mit einem speziellen Extent markiert, der beim Speichern in einen HTML-Querverweis umgewandelt wird. Die Quelldatei wird vom Benutzer interaktiv mit Hilfe der bekannten Standardfunktionen des XEmacs zur Dateiauswahl bestimmt.

Für das Ziel eines Verweises sind zwei Fälle zu unterscheiden:

1. Der Verweis zeigt auf ein weiteres HyperSource-Dokument. In diesem Fall kann der Benutzer dort anschließend einen zweiten Bereich auswählen, der als Ziel des Verweises identifiziert wird. In der Zieldatei wird dieser Bereich mit einem eindeutigen Bezeichner als Markierung versehen, so daß auch bei Änderungen in der Zieldatei der Querverweis intakt bleibt.
2. Der Verweis zeigt auf ein beliebiges anderes Dokument. In diesem Fall kann nicht auf eine exakte Stelle im Zieldokument verwiesen werden, da dazu das Anbringen einer Markierung im Zieldokument notwendig wäre, was auf Grund des fehlenden Wissens über das Format des Dokuments nicht möglich ist (von einer externen Speicherung der Verweise wurde aus Gründen des Verwaltungsaufwandes abgesehen).

Eine naheliegende Erweiterung ist eine automatische Erzeugung gewisser Querverweise, beispielsweise vom Aufruf einer Funktion zu ihrer Definition. Dazu wurde der *etags*-Mechanismus des Emacs, der in ähnlicher Form als *ctags*-Werkzeug auch für den vi-Editor existiert, in die Oberfläche des HyperSource-Systems integriert.

Zur Benutzung dieses Mechanismus muß lediglich das externe Programm `etags`, das Teil der Emacs-Distribution ist, ausgeführt werden. Es durchsucht die angegebenen Quelldateien und legt eine Datei `TAGS` an, in der die Dateien und Positionen sämtlicher gefundenen Funktions- und Typdefinitionen abgelegt sind. Das `etags`-Programm kennt die Syntax der verbreitetsten Programmiersprachen.

Anschließend kann durch einen einfachen Befehl vom Aufruf zur Definition jeder Funktion gesprungen werden, die in einer der durchsuchten Dateien definiert ist.

Allerdings bildet die derzeitige Version des HyperSource-Systems diese „impliziten Querverweise“, die extern in der TAGS-Datei spezifiziert sind, noch nicht auf entsprechende HTML-Anweisungen ab, so daß sie nur bei Verwendung des (X)Emacs-Editors benutzbar sind.

7.9 Wandlung nach HTML

Die Formatierung und Ergänzung des Quelltextes mit Randanmerkungen, Bildern und Querverweisen muß persistent gemacht werden, indem der Quelltext in einem „mächtigeren“ Format als ASCII (hier HTML) abgespeichert wird. Dies erfordert einen recht tiefen Eingriff in das XEmacs-System, da sämtliche Routinen davon ausgehen, daß durch das Speichern der textuelle Inhalt des Textpuffers und die „besuchte“ Datei „identisch“ werden sollen. Diese Annahme fließt in Funktionen wie beispielsweise das Auflösen von Konflikten bei konkurrierenden Änderungen und an vielen anderen Stellen ein. Es gibt zwar gewisse Modi (beispielsweise zum Lesen von E-Mail), die Dateien nicht 1:1 einlesen, sondern interpretieren; diese Modi bieten aber allesamt kein normales Editieren, sondern nur begrenzte, kontrollierte Änderungsmöglichkeiten über einzelne Tastaturkürzel, die bestimmte Operationen auslösen.

Im vorliegenden Fall jedoch galt es, einen Zusatzmodus zu implementieren, der mit den üblichen Hauptmodi zum Editieren von Programmtexten (die ja von einer Äquivalenz zwischen Datei und Textpuffer ausgehen) zusammenarbeitet. Die normale Dateisicherungsroutine wurde daher — für den jeweiligen Hauptmodus transparent — durch eine eigene Funktion ersetzt, die den Textpuffer auf Formatierungen, Randanmerkungen, Bilder und Querverweise in Form verschiedener Extents durchsucht und an diesen Stellen in den zu sichernden Zeichenstrom HTML-Kommandosequenzen einfügt.

Hierzu wurde eine Abbildung der verschiedenen Formate und Anmerkungen auf HTML-Konstrukte definiert. Ferner wurde eine Filterfunktion vorgeschaltet, die alle im Programm eventuell vorkommenden HTML-Sonderzeichen (also <, > und &) durch die entsprechenden HTML-Entitäten (siehe Seite 42) zur Darstellung dieser Zeichen ersetzt. Dadurch kann der Programmtext beliebige Zeichen enthalten, ohne daß es zu einem Konflikt mit der HTML-Sprachdefinition kommen kann.

Ein besonderes Problem stellte die *Abbildung von Randanmerkungen und -bildern* dar: HTML 2.0 bot keine Möglichkeiten, ein solches Layout durch Elemente zu erzielen. Daher mußte HTML 3.0 als Dokumentformat gewählt werden.

Doch auch in dieser Version bietet die Sprache natürlich keine expliziten Seitenlayout-Befehle, da HTML ja als SGML-Derivat nur die logische Dokumentstruktur wiedergeben soll. Allerdings besteht die Möglichkeit, *Tabellen* zu definieren, die wiederum beliebige andere Elemente enthalten dürfen. Diese Fähigkeit wurde genutzt, indem das gesamte Quelltext-Dokument als zweispaltige Tabelle gesetzt wurde, in deren linker Spalte sich die Randanmerkungen und in deren rechter Spalte sich der eigentliche, fest formatierte Quelltext befinden.

Im einzelnen wurden die folgenden Abbildungen definiert:

HyperSource-Objekt	HTML-Element	HTML-Tags
Normaler Quelltext	Vorformatierter Text	<PRE>...</PRE>
Überschrift	Überschrift	<H1>...</H1> bis <H6>...</H6>
Randanmerkung Text	Zelle in neuer Tab.-Zeile	<TR><TD>...
Randanmerkung Bild	–“– mit Bild und Link	<TD><A HREF=...><IMG SRC=...></A>
Querverweis (Quelle)	Hypertext-Link	<A HREF=...>...</A>
Querverweis (Ziel)	HTML-Bezeichner	<A ID=...>...</A>
<, >, &	HTML-Sonderzeichen	< , > , &

Bei der Abbildung von Bildern ist anzumerken, daß wie beim Erzeugen eines XPM-Inline-Bildes eine automatische Wandlung des Originalbildes, hier jedoch in das GIF-Format, angestoßen wird, um ein HTML-kompatibles Inline-Bild zu erhalten.

Die so generierte Datei ist somit in einem Format, das eine echte Untermenge von HTML 3.0 darstellt: Jede HyperSource-Datei ist eine HTML-3.0-Datei und kann von jeder zu dieser Sprache konformen Applikation gelesen und angezeigt werden. Umgekehrt ist jedoch nicht jedes HTML-3.0-Dokument eine gültige HyperSource-Datei; zum Sprachumfang von HTML 3.0 gehören eine Reihe weiterer Konstrukte, die von der im folgenden beschriebenen Einleseroutine nicht umgesetzt werden.

Zur Veranschaulichung ist im Anhang B ein Beispielquelltext wiedergegeben, der mit dem HyperSource-System bearbeitet wurde.

7.10 Einlesen von HTML

Das Einlesen einer generierten Datei ist naturgemäß ein deutlich komplexeres Problem als das Erzeugen solcher Dateien.[3] Im vorliegenden Fall ergibt sich aber eine gewisse Vereinfachung dadurch, daß die entscheidenden Teile (die HTML-Kommandos) in den einzulesenden Dateien nicht manuell, sondern automatisch erzeugt werden und somit von einer einheitlichen, vorhersagbaren Struktur sind.

Auch hier wurde, wie beim Speichern von Dokumenten, eine eigene Routine in das XEmacs-System eingebracht, die die Standard-Laderoutine im Falle von HyperSource-Dateien ersetzt. Dadurch ist der Wandlungsvorgang für den Benutzer des HyperSource-Systems transparent; der XEmacs wirkt wie ein WYSIWYG-Editor für HyperSource-Dateien. Der Benutzer braucht insbesondere nicht zu wissen, wie die verschiedenen HTML-Tags heißen oder zu verwenden sind. Für ihn existieren lediglich die Konzepte „Überschrift“, „Randbild“ etc.

Die Laderoutine liest den Dateiinhalt zunächst in einen temporären Textpuffer und durchsucht ihn dort nach HTML-Anweisungen. Anschließend wird der eigentliche vom Benutzer zu editierende Textpuffer aus diesem Zwischenpuffer gefüllt, wobei entsprechend den gefundenen HTML-Tags die nötigen Lisp-Objekte (Extents, Annotations, Pixmaps) erzeugt werden. Schließlich werden noch vorhandene HTML-Entitäten in die entsprechenden Einzelzeichen umgewandelt; aus einem `&` in der HyperSource-Datei wird also im Textpuffer beispielsweise wieder das `&`, das vielleicht im C-Quelltext als Adreßoperator benutzt worden war.

Der implementierte Parser stellt eines der komplexesten Module des Systems dar. Er besteht aus einer Sammlung von Routinen, die einzelne Aufgaben wie das Auffinden des nächsten HTML-Tags oder das Ersetzen der Entitäten übernehmen.

[3] „Es ist auch ungleich einfacher, Zahnpasta aus einer Tube herauszudrücken, als diese wieder hineinzubekommen.“ (A. Waibel)

7.11 Export compilierbaren Quellcodes

Der Programmtext muß aber auch ohne Formatinformation abspeicherbar sein, da die HTML-Sequenzen eine Übersetzung der formatierten HyperSource-Datei verhindern. Dazu wurde eine eigene Funktion zur Verfügung gestellt, mit der die zum Speichern installierte Konvertierungsroutine temporär deaktiviert wird, um wie in einem normalen Textpuffer lediglich dessen textuellen Inhalt unverändert zu sichern. Eine solche Datei enthält keine HTML-Sequenzen, Bilder, Randanmerkungen, Überschriftsformatanweisungen oder Querverweise. Sie kann somit an einen Übersetzer übergeben werden, um das ausführbare Programm zu erzeugen.

Ein Vorteil des implementierten Systems ist, daß durch die HyperSource-spezifischen Operationen der textuelle Inhalt des Textpuffers *nicht* verändert wird. Dies bedeutet, daß beispielsweise Fehlermeldungen des Übersetzers weiterhin auf die korrekte Zeilennummer im Textpuffer verweisen! Der klassische Entwicklungszyklus — Editieren, Übersetzen, Starten, Korrigieren — wird also nicht behindert.

7.12 Sprachunabhängigkeit

Eine wichtige Forderung war die Unabhängigkeit von der Programmiersprache, mit der der HyperSource-Modus verwendet werden soll. Diese Unabhängigkeit wurde erreicht, da für die Verwendung einer neuen Programmiersprache lediglich die Bedingung erfüllt sein muß, daß die Sprache öffnende und schließende Kommentarbegrenzer zur Verfügung stellt (siehe auch Seite 65). Diese Begrenzer werden benötigt, um Randanmerkungen im HTML-Dateiformat vom Quelltext abzutrennen, da sonst der ASCII-Export aus einem Browser heraus keinen compilierbaren Quellcode erzeugen würde. Ansonsten kann der HyperSource-Zusatzmodus mit Hauptmodi für jede beliebige Programmiersprache kombiniert werden.

7.13 Benutzerschnittstelle

Entscheidendes Kriterium für den Entwurf der Benutzeroberfläche war die Minimierung des Lern- und Bedienaufwands für den typischen Benutzer. Dies bedeutete im einzelnen:

- Rasche Erlernbarkeit durch eine „schlanke“ Benutzerschnittstelle mit nicht mehr als dem notwendigen Minimum an neuen Konzepten und Befehlen,
- Nutzung des vorhandenen Wissens über die Bedienung des Editorsystems durch Rückgriff auf bereits existierende Bedienkonzepte für Standardaufgaben (wie Dateiauswahl, Wechsel zwischen Textpuffern etc.).

Die erste Forderung wurde durch eine Menüschnittstelle umgesetzt: Für den Benutzer präsentiert sich das HyperSource-System nach der Aktivierung durch eine einzige, einfache Schnittstelle in Form eines Auswahlmenüs. Es ist als *Popup*-Menu verfügbar und gleichzeitig (da die Existenz von Popup-Menüs in einer Benutzeroberfläche mangels graphischer

Repräsentation nie offensichtlich ist) als *Pulldown*-Menü über einen Eintrag in der XEmacs-Menüzeile. Daneben wurden sämtliche Befehle mit Tastaturkürzeln versehen, die so gewählt wurden, daß sie sich an verbreitete XEmacs-Konventionen anlehnen und nicht mit Kürzeln der Hauptmodi kollidieren (die Tastenkombination „Control-C“, gefolgt von einer weiteren Kontrollsequenz, ist im XEmacs für die Belegung durch Zusatzmodi reserviert).

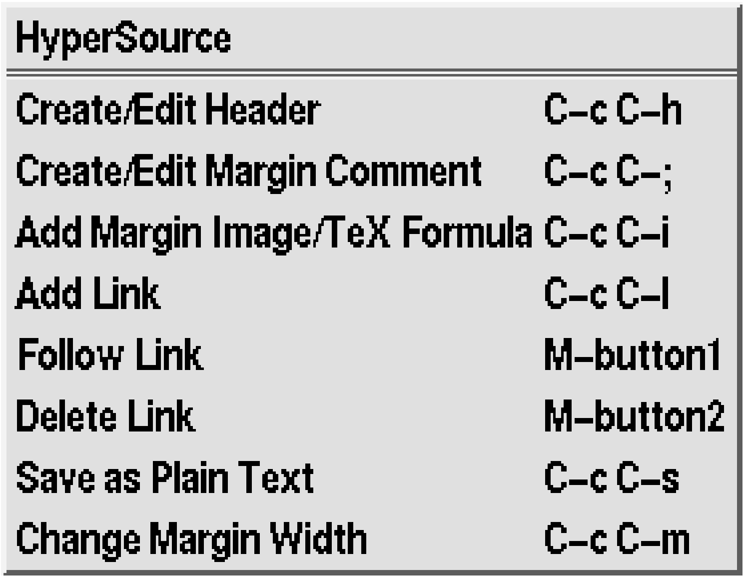


Abbildung 7.3: Das HyperSource-Menü

Auf diesem Menü, das in Abb. 7.3 wiedergegeben ist, befinden sich sämtliche Zusatzfunktionen, die dem Benutzer im HyperSource-Modus zur Verfügung stehen:

Überschriften: Nach dem Markieren eines Textbereiches mit den üblichen XEmacs-Kommandos kann eine Überschrift aus dem markierten Text erzeugt werden. Dabei kann, wie bereits erwähnt, aus 6 verschiedenen Größen ausgewählt werden, die den Gliederungsstufen <H1> bis <H6> von HTML entsprechen. Die Auswahl erfolgt interaktiv im *Minipuffer*, einem Textbereich in der Benutzeroberfläche des Emacs, der für derartige Eingaben vorgesehen ist.

Wird eine bereits vorhandene Überschrift markiert, so kann mit demselben Menüeintrag ihre Größe verändert oder die Überschriftsmarkierung gelöscht werden. Die Auswahl der einzelnen Aktion erfolgt über denselben Minipuffer-Mechanismus.

Textuelle Randanmerkungen: Durch Auswahl dieses Menüpunkts kann in der aktuellen Programmzeile eine Marginalie erzeugt werden. Der Text des Kommentars wird daraufhin im *Minipuffer* eingegeben. Um den Text einer Randanmerkung später erneut zu editieren oder zu löschen, kann er mit der Maus oder per Tastaturkürzel selektiert und wieder im Minipuffer editiert werden. Die normalen Editierkommandos („Gehe zum Beginn der aktuellen Zeile“ etc.) ignorieren den Kommentar, was das Editieren des eigentlichen Programmtextes wie gefordert erleichtert.

Randbilder und Formeln: Um ein Bild in den Rand des Quelltextes einzufügen, wählt der Entwickler den entsprechenden Menüpunkt aus. Er kann daraufhin den Namen einer Bilddatei (des „Originalbilds“) im Minipuffer angeben und wird dabei von den üblichen Emacs-Hilfsfunktionen (Namenskomplettierung, Anzeige von Verzeichnisinhalten etc.) unterstützt.

Existiert das Originalbild bereits, so wird aus ihm direkt über die verschiedenen Konverter ein Inline-Bild berechnet und im HyperSource-Textpuffer angezeigt. Ansonsten wird das entsprechende Graphikprogramm als asynchroner Subprozess aus dem XEmacs heraus mit einer neuen Bilddatei dieses Namens gestartet.

Der Benutzer kann nun im Graphikprogramm sein Bild wie gewohnt erstellen, der Programmeditor ist aber währenddessen nicht blockiert; der Benutzer kann also gleichzeitig auch weiter im Programmtext arbeiten. Dies ist wichtig, da der Inhalt einer dokumentierenden Skizze im allgemeinen vom gerade geschriebenen Quelltext abhängt, der auf diese Weise betracht- und editierbar bleibt.

Wird das Graphikprogramm beendet, so detektiert das HyperSource-System die Beendigung dieses Prozesses und erzeugt aus dem neuen Originalbild nunmehr die Inline-Version zur Anzeige im Textpuffer.

Das Inline-Bild kann anschließend jederzeit mit der Maus angewählt werden, woraufhin über den Minipuffer angeboten wird, das Bild zu editieren oder aus dem Textpuffer zu löschen.

Das Einfügen marginaler Formeln funktioniert ebenso, mit der Ausnahme, daß statt eines externen Graphikprogramms ein XEmacs-Textpuffer für die Eingabe der textuellen Formel geöffnet wird.

Querverweise: Der Benutzer markiert einen Textbereich als Ausgangspunkt eines Querverweises und erhält dann über den Menüeintrag die Möglichkeit, im Minipuffer die Zieldatei anzugeben. Bei HyperSource-Dateien wird diese Datei geöffnet, und der Benutzer kann das Ziel markieren und mit einer vorher angegebenen Tastenkombination die Erstellung des Querverweises beenden. Daraufhin werden Quelle und Ziel als Extents in den jeweiligen Dateien eingetragen, so daß sie beim Abspeichern in den Text eingefügt werden. Die Querverweise sind sofort benutzbar; dazu wurden neben dem Menüeintrag auch Maus-basierte Kürzel definiert, um einen Verweis einfach verfolgen zu können. Beim Löschen wird auch der Eintrag in der Zieldatei, sofern vorhanden, mit entfernt.

Ist die Zieldatei keine HyperSource-Datei, so genügt die Angabe ihres Namens bereits zur Erzeugung des Links, der dann nur auf die Datei als Ganzes verweisen kann.

ASCII-Export: Der Quelltext kann wie oben beschrieben über den vorletzten Menüeintrag ohne Formatinformation gesichert werden. Diese „maschinenlesbar exportierte“ Datei ist nur für die Übersetzung und ähnliche Schritte nötig. Die eigentliche Hauptdatei, auf der beispielsweise auch eine Versionskontrolle arbeiten würde, ist die HyperSource-Datei im HTML-Format.

Randeinstellung: Ein letzter Menüpunkt erlaubt die individuelle Einstellung des linken Dokumentrandes, der für Randanmerkungen und Bilder benutzt wird. Diese Einstellung wird in der HyperSource-Datei persistent als Breite der linken Spalte der HTML-Tabelle gespeichert.

Hilfe-Funktionen: Eine umfassende Hilfe speziell für den HyperSource-Modus steht jederzeit über die Standard-Emacs-Funktion `describe-mode (C-h m)` zur Verfügung. Die Funktionen des HyperSource-Modus wurden ferner so geschrieben, daß ihre Dokumentation über die eingebaute Emacs-Hilfefunktion zugreifbar ist.

Damit ist die Beschreibung der Implementierung abgeschlossen. Für detaillierte Informationen sei auf den Quelltext verwiesen, der über den ftp-Server des Instituts[4] erhältlich ist.

[4] URL: ftp://ninive.ira.uka.de/pub/hypersrc/

Don't let HTML become the Fortran of the Web.
— Alan Kay, Apple Computer

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit wurde ein neues Paradigma für die Softwareentwicklung untersucht: das *HyperSource*-Konzept, nach dem Programm-Quelltexte nicht mehr als lineare Texte ohne Strukturierungs- und Formatierungsbestandteile, sondern als Hypermedia-Dokumente mit hervorgehobenen Überschriften, textuellen Randanmerkungen, integrierten graphischen Skizzen und Querverweisen sowohl innerhalb als auch zwischen Quelltext und Dokumentation erstellt werden. Dabei sollte auch ein reales System entworfen und implementiert werden, um die Umsetzbarkeit und den Nutzen dieses Ansatzes in der Praxis überprüfen zu können.

Zu Beginn der Arbeit stand eine empirische Untersuchung in der Form einer *Umfrage* unter Softwareentwicklern aus dem akademischen Bereich, die einen Einblick in die derzeitige Praxis in Entwurf, Implementierung und Dokumentation gab. Sie zeigte vor allem, daß die derzeit tatsächlich eingesetzten Techniken in allen drei genannten Phasen die Möglichkeiten moderner graphikfähiger Hard- und Software nicht ausnutzen. Insbesondere die Kommentierung von Quelltexten erfolgt praktisch ohne jede Unterstützung durch die Entwicklungsumgebung. Quelltexte werden entweder nur sporadisch und unübersichtlich kommentiert, oder es wird mit viel manuellem Aufwand versucht, mit den begrenzten Möglichkeiten des ASCII-Formats so etwas wie ein „Layout" oder gar Graphiken im Quelltext zu schaffen. Das Resultat ist: Sourcecode ist ästhetisch und typographisch wenig ansprechend, schlecht lesbar und damit schwer nachvollziehbar. Die Navigation durch eine Sammlung von Quelltexten wird nicht durch dem Programmtext inhärente Strukturinformationen unterstützt.

Der vorgeschlagene *Hypermedia-Ansatz* bietet Lösungen für diese Probleme: Im Entwurf kann der Entwickler über ein Hypertext-Netz von Dokumenten, die das Projekt in verschiedenen Detailstufen beschreiben, einfacher zwischen den verschiedenen Abstraktionsebenen wechseln, was dem typischen, natürlichen Vorgehen entspricht. Bei der Implementierung wird die Navigation durch den Quelltext durch Hypertext-Links erleichtert, und die konzeptuelle Unterscheidung von Quelltext und Kommentaren erleichtert die Dokumentation des

Codes. Den größten Vorteil aber bringt dieser Ansatz für spätere Benutzer des Programmtextes: Ihnen präsentieren sich Sourcecode und Dokumentation als ein Netz von Dokumenten, innerhalb derer einfach navigiert werden kann, was die Übersicht erleichtert, und multimediale Kommentare innerhalb des Quelltexts erleichtern das Lesen. Programmtexte werden besser nachvollziehbar und dadurch besser wiederverwendbar, ein wichtiger ökonomischer Aspekt aus Sicht des Software-Engineering.

In einer anschließenden *Anforderungsspezifikation* wurde die Funktionalität des zu entwickelnden HyperSource-Entwicklungssystems abgesteckt. Als primäre Ziele wurden die Möglichkeiten des Einfügens von Bildern, Randanmerkungen, Überschriften und Querverweisen in einen Programmtext definiert. Randbedingung war, möglichst bestehende Werkzeuge zu einem System zu integrieren, das wenig Lernaufwand erfordert, von der Programmiersprache der Projektdateien unabhängig ist und sich nahtlos in bestehende Standardumgebungen einfügt.

Vor einer Eigenentwicklung stand natürlich eine umfassende *Untersuchung existierender Systeme*, um festzustellen, welche Schritte in Richtung eines Hypermedia-Ansatzes zur Programmentwicklung bereits gemacht wurden. 25 Systeme wurden untersucht; es zeigte sich jedoch, daß keines alle gestellten Anforderungen erfüllte. Die verfügbaren Systeme waren stets nur entweder zur Hypermedia-Dokumenterstellung *oder* zur Programmentwicklung geeignet; lediglich bei einigen aktuellen Projekten fanden sich bereits Ansätze, ähnlich wie in dieser Arbeit beide Gebiete zu vereinigen. Trotzdem lieferte diese Untersuchung nützliche Anregungen für den eigenen Entwurf.

Der *Entwurf* legte dann einige Eckdaten des eigenen Systems fest. So wurde entschieden, das HyperSource-System als eine Erweiterung des XEmacs-Editors zu konzipieren, da dieser Universaleditor zum weitverbreiteten GNU-Emacs kompatibel und ebenso kostenlos und erweiterbar, zusätzlich aber graphikfähig ist. Als Dateiformat für HyperSource-Dokumente wurde die Hypertext Markup Language HTML in der Version 3.0 gewählt, für die durch die rasche Verbreitung des World-Wide Web zahlreiche Anwendungen und Werkzeuge existieren.

Anschließend erfolgte die *Implementierung* des Systems nach dem WYSIWYG-Prinzip als Zusatzmodus des XEmacs-Editors. Die Erweiterung wurde so integriert, daß die normalen umfangreichen Editierfunktionen nicht beeinträchtigt werden. Sämtliche Zusatzfunktionen, die in der Anforderungsspezifikation als primäre Ziele aufgeführt waren (siehe oben), wurden über ein Extra-Menü zur Verfügung gestellt. Die Systemroutinen des Editors zum Speichern und Laden wurden so erweitert, daß die zusätzliche Struktur- und Formatinformationen innerhalb des HyperSource-Textpuffers in HTML-Elemente und umgekehrt übersetzt werden. Die Verwendung beliebiger Bildformate wurde über einen flexiblen, dynamisch erweiterbaren Konvertiermechanismus realisiert, der mit dem Konzept eines Originaldokuments und eines daraus berechneten Inline-Bildes arbeitet. Dadurch wurde auch eine Erweiterung auf das Einfügen mathematischer Formeln einfach möglich.

8.2 Bewertung

Das entwickelte HyperSource-System erfüllt die in der Anforderungsspezifikation gestellten Forderungen. Es ist ein Werkzeug, in dem die Anwendung des HyperSource-Paradigmas auf den Bereich der Programmentwicklung in vielerlei Hinsicht realisiert wurde, und das praktisch eingesetzt werden kann, um Quelltexte verständlicher und besser wiederverwendbar zu gestalten. Mehrere Mitarbeiter des Instituts und einige externe Entwickler haben das System bereits getestet. Es kann zur Kommentierung vorhandener oder neu zu schreibender Software verwendet werden. Besonders graphische Algorithmen, aber auch andere Arten von Programmen profitieren von der Möglichkeit, ihren Quelltext mit Inline-Bildern zu versehen. Einsatzmöglichkeiten sind aber auch beispielsweise Praktikumsversuche, in denen die Aufgabenstellung, der Programmrahmen und Lösungshinweise als Netzwerk von HTML-Dokumenten erstellt werden können. Die Korrektur von Programmieraufgaben könnte direkt im Quelltext in Form von Randanmerkungen erfolgen, ohne den funktionellen Teil des Programmtexts dadurch zu beeinflussen. Auch lehrbuchähnliche Sammlungen von Algorithmen sind mit dem HyperSource-System einfacher zu erstellen.

Es hat sich jedoch auch gezeigt, daß für den problemlosen Einsatz des Systems durch Dritte noch eine gewisse Zeit notwendig sein wird, bis das Paket soweit abgerundet ist, daß bei der Installation keine Probleme mehr zu erwarten sind. Diese Reifungsphase ist für Erstentwicklungen typisch und kann nur durch realen Einsatz an unterschiedlichen Orten durchlaufen werden. Insbesondere die Integration externer Graphikprogramme und -konverter ist oft problematisch, da auf dieser hohen Abstraktionsebene noch kaum Standards in der Unix-Welt existieren.

Die Antwortzeiten des Systems sind zufriedenstellend, mit Ausnahme der Verzögerungen von mehreren Sekunden, die das Konvertieren von Bildern über die externen Umwandlungsprogramme beinhaltet. Hier wäre eine asynchrone Konvertierung denkbar, um den Benutzer in dieser Zeit im Editor weiterarbeiten zu lassen.

Auch von der Funktionalität her bieten sich noch viele Erweiterungen und Verbesserungen an, die auf Grund der beschränkten Zeit nicht umgesetzt werden konnten. Dazu gehört beispielsweise die automatische Speicherung von Querverweisen zwischen Funktionsaufrufen und -definitionen, die letztlich zu einer notwendigen semantischen Unterscheidung verschiedener Typen von Querverweisen führt. Textuelle Randanmerkungen können derzeit noch nicht mehrere Zeilen umfassen und Bilder noch nicht von mehreren Zeilen Text umflossen werden, was durch das gegenwärtige Zeilenkonzept des XEmacs bedingt ist.

Da die aktuelle Version des XEmacs nur einen linken, aber keinen rechten Rand für Marginalien zur Verfügung stellt, werden Randanmerkungen links neben den Quelltext gesetzt. Dies erscheint mir zwar optisch auch sinnvoller, da Quelltexte links einen einheitlicheren Rand als rechts bieten, die Kommentare müssen jedoch deshalb im HTML-Dateiformat auf beiden Seiten von Kommentarzeichen eingeschlossen sein. Einige Sprachen wie Emacs-Lisp stellen nur Kommentarbeginnzeichen zur Verfügung; die verbleibenden Zeichen bis zum Zeilenende werden vom Übersetzer ignoriert. In diesen Sprachen können textuelle Randanmerkungen nur so in das Dokument eingefügt werden, daß nach ihnen kein Quelltext mehr in derselben Zeile steht. Diese Einschränkung wiegt zwar nicht schwer, aber ein alternatives Anbieten eines rechten Kommentarrandes (der in der kommenden Version des XEmacs unterstützt sein wird) wäre eine denkbare Erweiterung.

Auch wäre das Anlegen von Verweisen auf bestimmte Stellen in Dokumenten auch für andere Dokumenttypen neben HyperSource-Dateien wünschenswert und für HTML-Textdokumente sogar ohne externe Verweistabellen zu realisieren.

Letztlich könnte über die Integration vieler weiterer HTML-Fähigkeiten wie das Erstellen von Tabellen, Listen etc. nachgedacht werden. Allerdings ist fraglich, ob diese Eigenschaften zur Kommentierung von Quelltexten wirklich notwendig und sinnvoll sind. Meist ist gerade die Beschränkung auf eine bestimmte Funktionalität, zusammen mit einer nahtlosen Integration, ein Vorteil, wenn es um die Akzeptanz einer Erweiterung für ein bereits so mächtiges Entwicklungssystem geht, wie es der Emacs unter Unix und X darstellt. So wäre es beispielsweise sinnlos, das HyperSource-System um Konzepte der Versionshaltung zu ergänzen, da diese Aufgabe bereits von anderen Werkzeugen wie RCS erledigt wird, für die auch eine Schnittstelle zum XEmacs als Zusatzmodus existiert.

8.3 Ausblick

Als Ergebnis kann festgehalten werden, daß das HyperSource-Konzept einen vielversprechenden Ansatz für die Verbesserung der Lesbarkeit und damit Wiederverwendbarkeit von Software darstellt. Das implementierte HyperSource-System hat dabei als Prüfstein der praktischen Umsetzung dazu beigetragen, dieses neue Paradigma auf seine Einsetzbarkeit hin zu durchleuchten. Seine besondere Qualität gewinnt das hier entworfene System aber auch aus der Tatsache, daß es im Gegensatz zu anderen vorhandenen Ansätzen auf inzwischen weitverbreiteten Werkzeugen und Formaten wie dem XEmacs-Editor und insbesondere dem HTML-Format für Hypermedia-Dokumente aufsetzt.

Die Anstrengungen auch der großen Institutionen der Softwareentwicklung (MIT, Apple) in diese Richtung unterstützen die Vermutung, daß in einigen Jahren die Programmentwicklung im Vergleich zu heute einen radikalen Wandel hin zu einer vernetzten, multimedialen Arbeitsweise erfahren wird, die hier als HyperSource-Paradigma bezeichnet worden ist. Ein einfaches Werkzeug dazu wurde durch diese Arbeit bereitgestellt. Die „Revolution“ sei hiermit ausgerufen.

Glaube einer Statistik nur, wenn Du sie selbst gefälscht hast.

— Winston Churchill

A Umfrage

A.1 Fragebogen

Der Fragebogen gliederte sich gemäß der Aufgabenstellung in Fragen zur Entwurfspraxis, Implementierungspraxis, Dokumentationspraxis sowie Präferenzen bezüglich eines neu zu entwickelnden Systems. Sämtliche Fragen sind im folgenden wiedergegeben:

Fragebogen zum Thema „Programmentwicklung und -dokumentation“
Jan Borchers (job@ira.uka.de)

Name (für Rückfragen):

Programmentwurf

Wie entwerfen Sie Softwareprojekte mittlerer Größe (etwa 1 Monat Implementierungsaufwand)? [auf Papier oder rechnergestützt; wieviel % Top-Down/Bottom-Up/Middle-Out]

Inwieweit verwenden Sie Ihre Entwurfsdokumente für die Implementierung oder Dokumentation?
[Texte aus Entwurfsfiles übernehmen oder neutippen, Skizzen neuzeichnen oder einscannen]

Implementierung

Welche Programmiersprache und ggf. -umgebung verwenden Sie? [C,C++,WEB,CWEB]

Welchen Programmeditor verwenden Sie? [Emacs/vi/andere]

Benutzen Sie eine Versionsverwaltung? [SCCS/RCS/CVS]

Dokumentation

Wie kommentieren Sie Ihren Sourcecode? [in Programmzeilen, in Extrazeilen, in Extrablöcken, in Manpages, in TeX-Dokumenten, in GNU-Info-Files]

Verwenden Sie Formatierungshilfen für die Kommentare im Sourcecode?
[spez. Emacs-Mode, WEB]

Wären Bilder als Dokumentation Ihres Sourcecodes nützlich? [1=unnütz...5=sehr nützlich]

Wären Hyperlinks zum Lesen Ihres Sourcecodes nützlich? [1=unnütz...5=sehr nützlich]

Wären Sie bereit, einen neuen Editor mit einfacher Standardfunktionalität zu verwenden, der zusätzlich obige Features (Graphiken, Hyperlinks) unterstützt?
[1=nicht bereit...5=jederzeit bereit]

Würden Sie einen WYSIWYG-Editor oder ein System mit Editor + Viewer bevorzugen?

Würden Sie die Ablage von Sourcecode und Dokumentation in einem File (mit Extraktionsmöglichkeit z. B. des Sourcecodes) oder in verschiedenen Files (reiner Sourcecode + Hyperlink-Information + Graphiken) bevorzugen?

Sonstiges

Was wären für Sie andere wichtige Eigenschaften eines Hypermedia-Systems zur Programmentwicklung und -dokumentation?

A.2 Befragte Personen

Befragt wurden 17 Personen an der Fakultät für Informatik an der Universität Karlsruhe, darunter 10 Mitarbeiter und 5 Studenten am Institut für Betriebs- und Dialogsysteme. Die Befragten beschäftigen sich vor allem mit Problemen der Computergraphik, der geometrischen Datenverarbeitung sowie graphischen Benutzerschnittstellen. Es finden sich unter den Befragten jedoch auch 2 Mitarbeiter aus dem Gebiet des Compilerbaus.

A.3 Ergebnisse

Angegeben sind bei numerischen Antworten stets die Mittelwerte der Antworten sämtlicher Befragten. Bei Fragen mit nicht numerischen Antworten sind die prozentual häufigsten Antworten aufgeführt, wobei Mehrfachnennungen möglich waren.

Entwurf

Entwurfsstil	40% Top-Down, 30% Bottom-Up, 30% Middle-Out[1]
Entwerfe rechnergestützt	28%
Verwende Entwurfstexte weiter	28%
Verwende Entwurfsskizzen weiter	7%

Implementierung

Benutzte Programmiersprache	C (94%), C^{++} (41%)
Verwende Literate-Programming-Tool	6%
Benutzter Editor	Emacs (74%), vi (35%)
Verwende Versionshaltung	53% (RCS 41%, andere 12%)

Dokumentation

Extrablöcke	94%
Extrazeilen	71%
in Programmzeilen	59%
Manual Pages	41%
TEX	29%
Info-Files	6%
Formatierungshilfen	29% (Emacs-Mode 18%, andere 11%)

Für neues System

Bilder im Quelltext	4,1 (Skala: 1=unnütz…5=sehr nützlich)
Hyperlinks im Quelltext	4,1 (Skala: 1=unnütz…5=sehr nützlich)
Neuer Editor	2,6 (Skala: 1=nicht…5=jederzeit akzeptierbar)
Bevorzuge WYSIWYG	56%
Bevorzuge Ablage in 1 File	52%
Häufigster Wunsch	Graphische Übersicht (53%)

[1] Dies sind die mittleren Werte über alle Antworten. Die einzelnen Angaben waren hier stark gestreut.

Entscheidend ist, was hinten rauskommt.

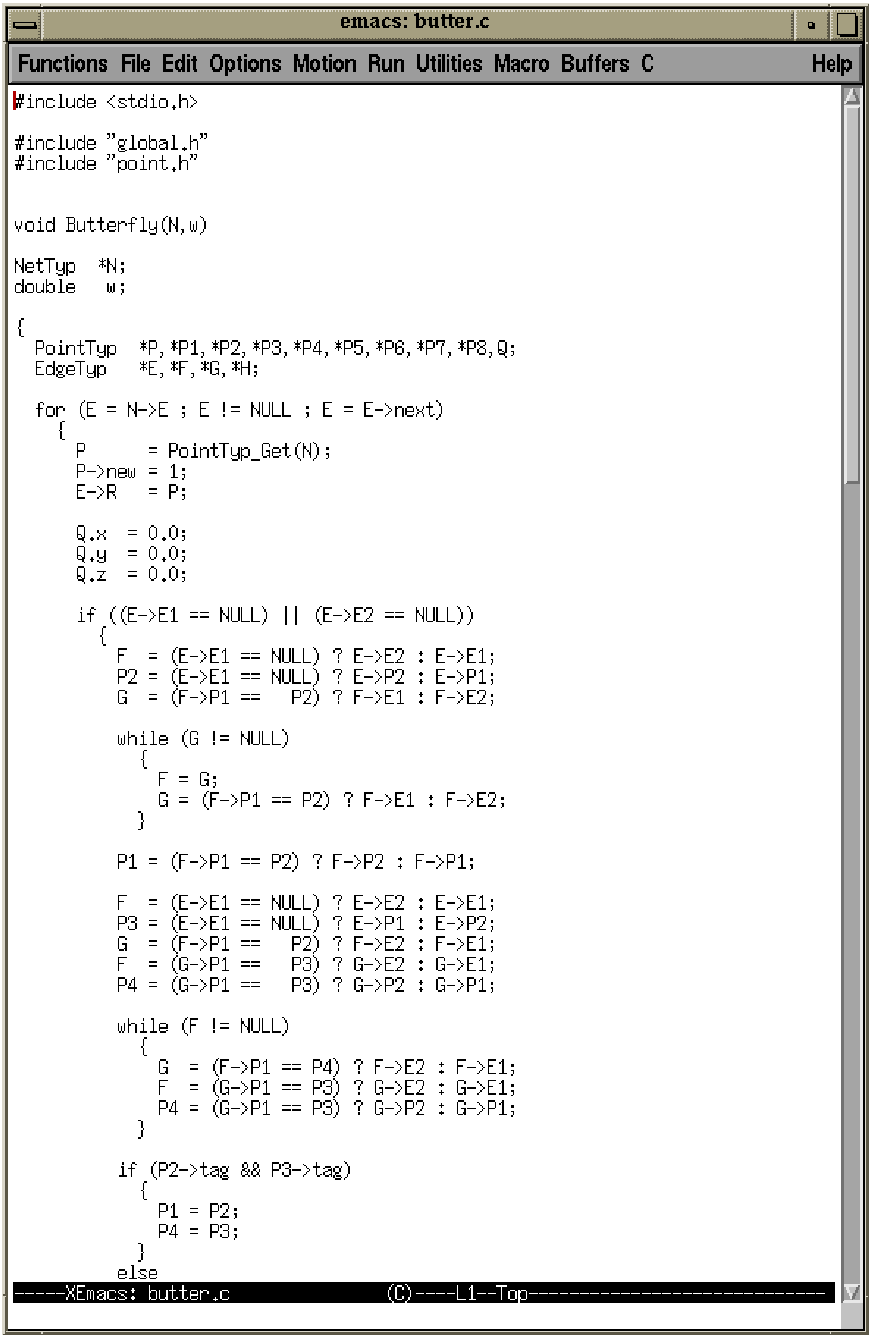
— Bundeskanzler Dr. Helmut Kohl

B Beispielanwendung des HyperSource-Systems

Als Beispiel für den Einsatz des HyperSource-Systems dient ein kürzlich entwickelter Algorithmus (die Grundlagen finden sich in [Kobbelt95]) aus dem Bereich der geometrischen Datenverarbeitung. Das Verfahren dient zur Verfeinerung von Dreiecksnetzen und verwendet dabei einige existierende Algorithmen zur Unterteilung, deren Ergebnisse mit zusätzlichen Maßnahmen anschließend verbessert werden sollen. Aus diesem System wurde hier exemplarisch der Algorithmus zur Bestimmung des nächsten Unterteilungspunktes nach dem *Butterfly-Verfahren* [DynGreLev90] herausgegriffen.

Abbildung B.1 zeigt, wie sich der Beginn dieses Quelltextes dem Entwickler bzw. Leser normalerweise im XEmacs-Editor darstellt.

Der dieser Funktion zugrundeliegende Algorithmus ist so nur schwer nachvollziehbar. Dies ließe sich durch textuelle Kommentare jedoch nur zum Teil verbessern. In Abb. B.2 ist daher stattdessen derselbe Text mit dem HyperSource-System um Überschriften, Randanmerkungen, Formeln, Graphiken und Querverweise ergänzt worden. Dargestellt sind wieder der Beginn sowie ein weiterer Ausschnitt des Quelltexts, der den Einsatz von Formeln zeigt.



emacs: butter.c

Functions File Edit Options Motion Run Utilities Macro Buffers C Help

```
#include <stdio.h>

#include "global.h"
#include "point.h"


void Butterfly(N,w)

NetTyp  *N;
double   w;

{
  PointTyp  *P,*P1,*P2,*P3,*P4,*P5,*P6,*P7,*P8,Q;
  EdgeTyp   *E,*F,*G,*H;

  for (E = N->E ; E != NULL ; E = E->next)
    {
      P      = PointTyp_Get(N);
      P->new = 1;
      E->R   = P;

      Q.x  = 0.0;
      Q.y  = 0.0;
      Q.z  = 0.0;

      if ((E->E1 == NULL) || (E->E2 == NULL))
        {
          F  = (E->E1 == NULL) ? E->E2 : E->E1;
          P2 = (E->E1 == NULL) ? E->P2 : E->P1;
          G  = (F->P1 ==   P2) ? F->E1 : F->E2;

          while (G != NULL)
            {
              F = G;
              G = (F->P1 == P2) ? F->E1 : F->E2;
            }

          P1 = (F->P1 == P2) ? F->P2 : F->P1;

          F  = (E->E1 == NULL) ? E->E2 : E->E1;
          P3 = (E->E1 == NULL) ? E->P1 : E->P2;
          G  = (F->P1 ==   P2) ? F->E2 : F->E1;
          F  = (G->P1 ==   P3) ? G->E2 : G->E1;
          P4 = (G->P1 ==   P3) ? G->P2 : G->P1;

          while (F != NULL)
            {
              G  = (F->P1 == P4) ? F->E2 : F->E1;
              F  = (G->P1 == P3) ? G->E2 : G->E1;
              P4 = (G->P1 == P3) ? G->P2 : G->P1;
            }

          if (P2->tag && P3->tag)
            {
              P1 = P2;
              P4 = P3;
            }
          else
-----XEmacs: butter.c          (C)----L1--Top----------------------------
```

Abbildung B.1: Der Beginn des Butterfly-Algorithmus als normaler C-Quelltext im XEmacs-Editor.

Um zu sehen, wie diese Kommentare und Strukturierungen in HTML-Elemente umgesetzt werden, betrachte man die im folgenden abgedruckte HTML-Datei `butter.c.hs`, wie sie das HyperSource-System beim Abspeichern des bearbeiteten Textpuffers generiert. Interessant sind zu Beginn die HTML-Präambel und der Anfang des Tabellenelements sowie Überschriften, Randanmerkungen und Bilder. In der Mitte findet sich der Einsatz von Formeln, und das Ende zeigt den Abschluß des HTML-Dokuments.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.0//EN//">
<!-- HyperSource 1.0 File Format -->
<HTML>
<HEAD>
<TITLE>/home/mit4/job/da/examples/snake/butter.c.hs</TITLE>
</HEAD>

<BODY>
<TABLE COLSPEC="R37 L80">
<TR VALIGN=BASELINE>
<TD>
<TD CLASS=SOURCE>
<PRE></PRE>
<H1>/* butter.c */
</H1>
<PRE>
/*
</PRE>
<H3>Implementierung des Butterfly-Algorithmus fuer geschlossene Flaechen und
Uebertragung auf offene Flaechen. (An den Raendern: 4-Punkt-Verfahren)
</H3>
<PRE>*/

</PRE>
<H3>/* Leif Kobbelt, 17.04.93, 23.02.94 */
</H3>
<PRE>



</PRE>
<H2>/* Includes */
</H2>
<PRE>

#include &lt;<A HREF="/usr/include/stdio.h">stdio.h</A>&gt;

#include "<A HREF="global.h.hs# [1]">global.h</A>"
#include "<A HREF="point.h.hs# [1]">point.h</A>"
</PRE>
<TR VALIGN=BASELINE>
<TD CLASS=IMAGE><A HREF="butterfly.ps"><IMG SRC="butterfly-inline.gif" ALT=""></A>
<TD CLASS=SOURCE>
<PRE>/* Lokale Numerierung der Punkte */



</PRE>
<H2>/* Butterfly-Algorithmus */
</H2>
<PRE>

/* F"ur jede Kante E des Netzes N wird ein neuer Punkt E-&gt;R berechnet. */
/* Siehe auch <A HREF="html/butterfly.html">Dokumentation</A> des Butterfly-Algorithmus. */

void Butterfly(N,w)

<A HREF="global.h.hs#NetTyp[2]">NetTyp</A>  *N;
```

```
double   w;

{
</PRE>
<TR VALIGN=BASELINE>
<TD CLASS=COMMENT NOWRAP>
<SMALL>/* Entsprechen Pkten d. Skizze */</SMALL>
<TD CLASS=SOURCE>
<PRE>  <A HREF="global.h.hs#PointTyp[1]">PointTyp</A>  *P,*P1,*P2,*P3,*P4,*P5,*P6,*P7,*P8,Q;
  <A HREF="global.h.hs#EdgeTyp[2]">EdgeTyp</A>   *E,*F,*G,*H;

</PRE>

.
.
.

<TR VALIGN=BASELINE>
<TD CLASS=COMMENT NOWRAP>
<SMALL>/* Kubische Interpolation mit Ecke P(i): */</SMALL>
<TD CLASS=SOURCE>
<PRE>             if (P2-&gt;tag)
                {
</PRE>
<TR VALIGN=BASELINE>
<TD CLASS=IMAGE><A HREF="q-p2.fml"><IMG SRC="q-p2-inline.gif" ALT=""></A>
<TD CLASS=SOURCE>
<PRE>                 Q.x = 3.0 * (P2-&gt;x - P3-&gt;x) + P4-&gt;x - P1-&gt;x;
                  Q.y = 3.0 * (P2-&gt;y - P3-&gt;y) + P4-&gt;y - P1-&gt;y;
                  Q.z = 3.0 * (P2-&gt;z - P3-&gt;z) + P4-&gt;z - P1-&gt;z;
                }

.
.
.

          P-&gt;z =   0.5     * (P4-&gt;z + P5-&gt;z)
                 + 2.0 * w * (P2-&gt;z + P7-&gt;z)
                 -       w * (P1-&gt;z + P3-&gt;z + P6-&gt;z + P8-&gt;z + Q.z);
        }
    }
}
</PRE>
</TABLE>
</BODY>
</HTML>
```

Die WYSIWYG-Eigenschaften des HyperSource-Systems zeigen sich, wenn man Abb. B.4 und B.5, in denen zu sehen ist, wie der HTML-Browser *Netscape* die gespeicherte Datei `butter.c.hs` darstellt, mit den vorhergehenden Bildern des XEmacs-HyperSource-Textpuffers vergleicht. Obwohl sich die konkrete Darstellung von HTML-Dateien natürlich von Applikation zu Applikation stark unterscheiden kann, ist die Art, in der sie im XEmacs und in Netscape angezeigt wird, eine typische Formatierung für graphische HTML-Applikationen.

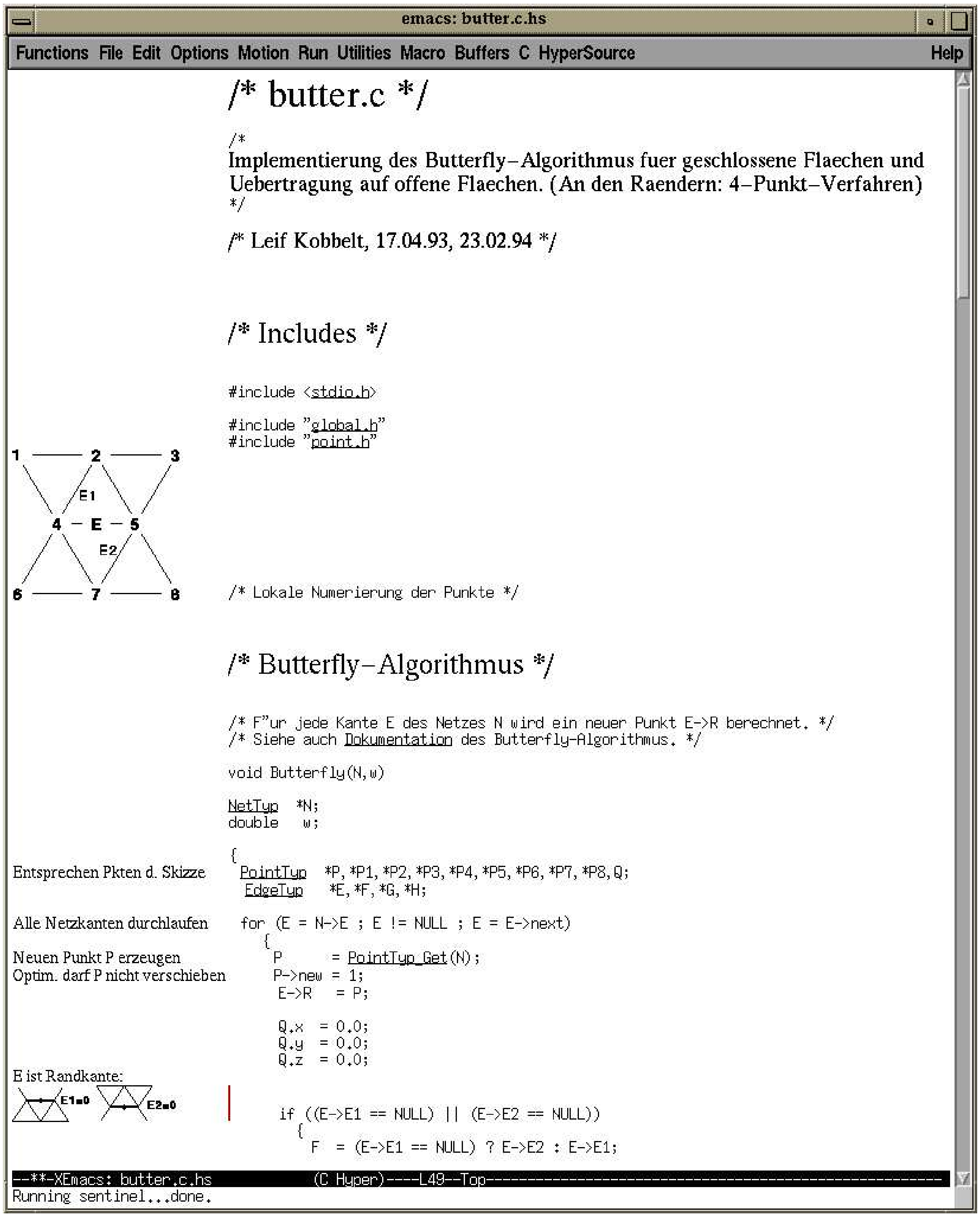


Abbildung B.2: Derselbe Quelltext bei Verwendung des HyperSource-Modus. Überschriften sind optisch hervorgehoben, der Randbereich enthält textuelle und graphische Kommentare, über Querverweise (unterstrichen) kann navigiert werden.

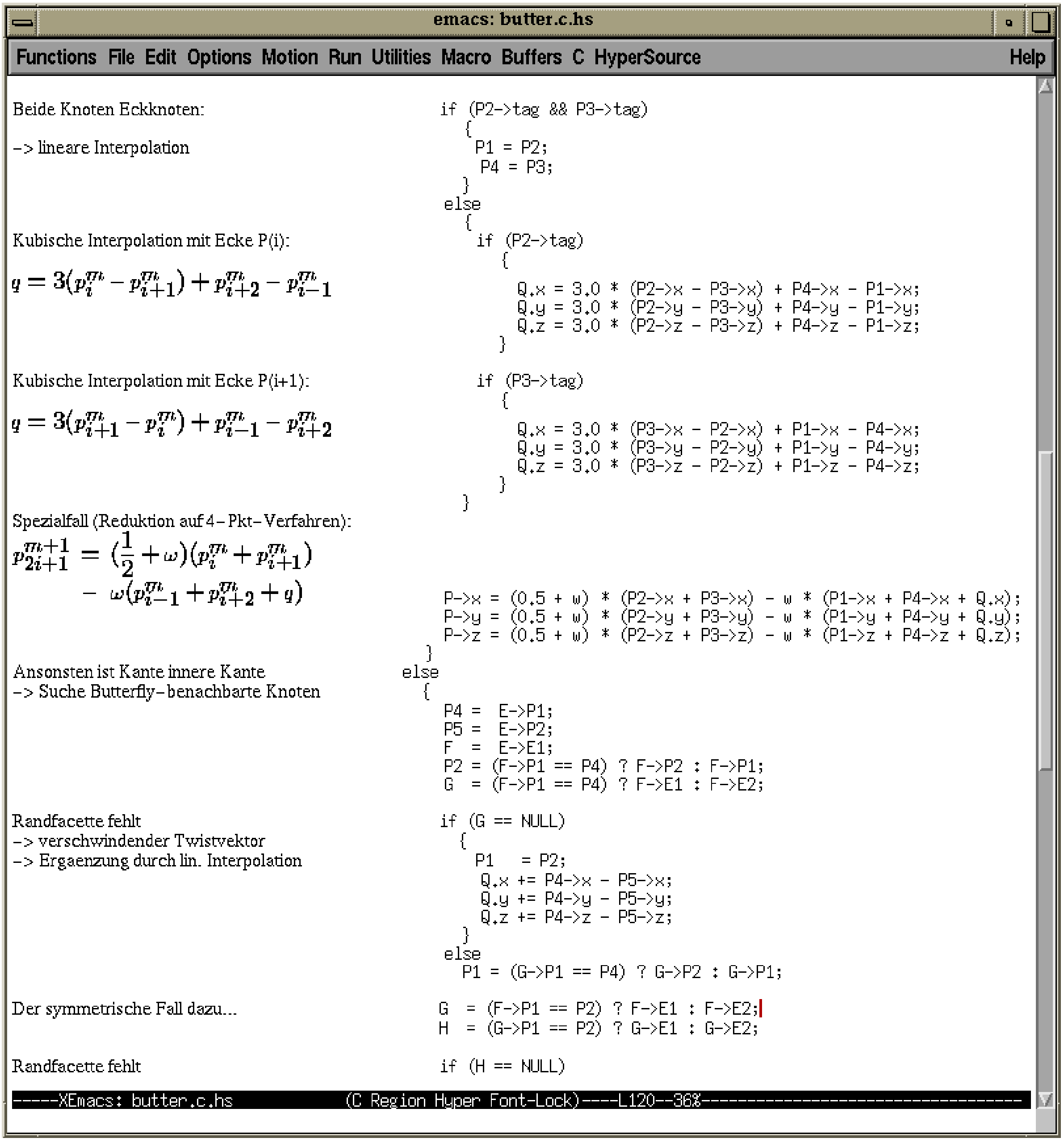


Abbildung B.3: Ein anderer Abschnitt des Quelltextes. Typographisch korrekt gesetzte Formeln sind schneller zu erfassen und machen den Programmcode lesbarer.

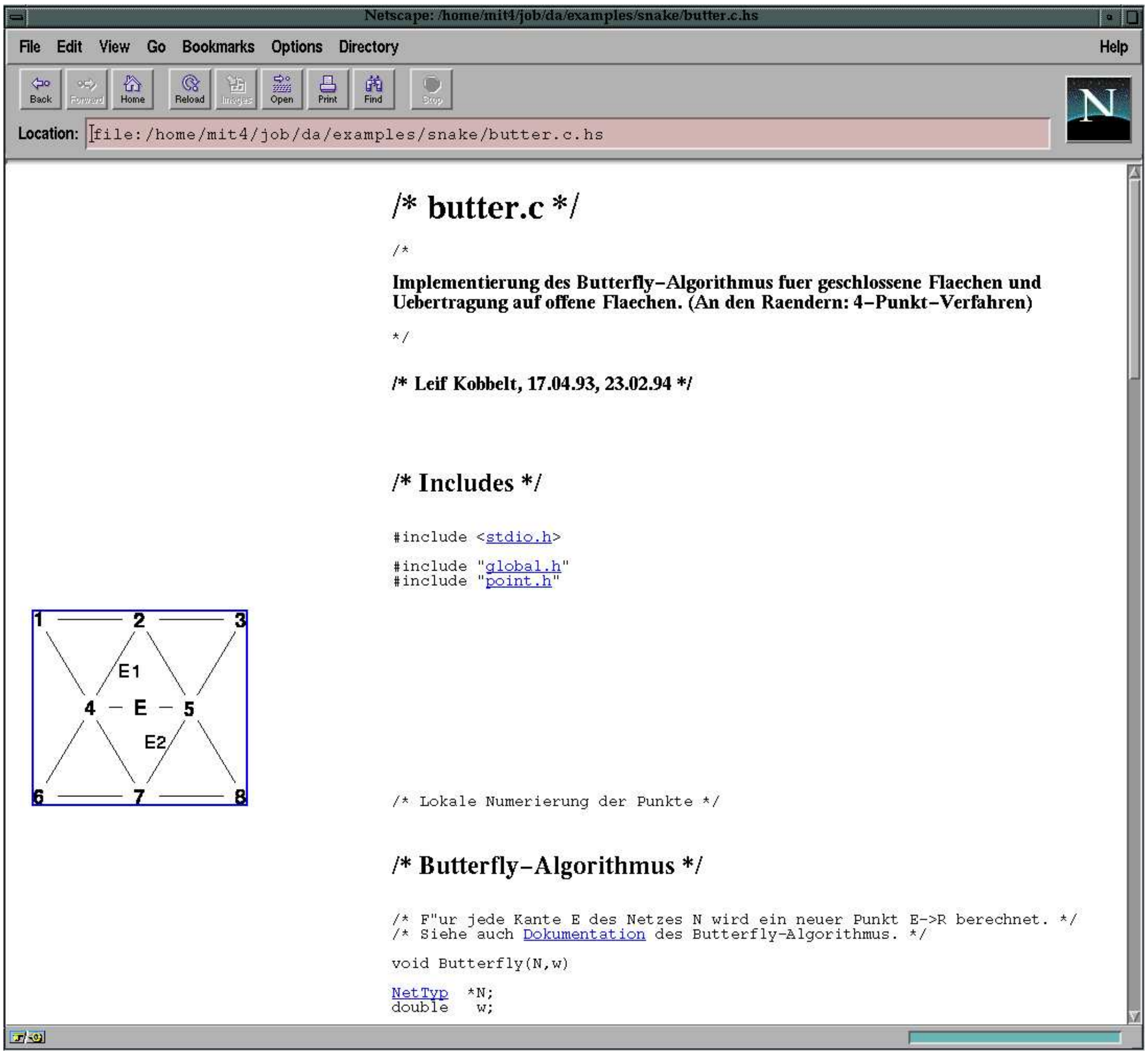


Abbildung B.4: Der Beginn der Datei `butter.c.hs`, geöffnet mit dem HTML-Browser *Netscape*. Die Darstellung unterscheidet sich kaum von jener des XEmacs im HyperSource-Modus (WYSIWYG-Prinzip).

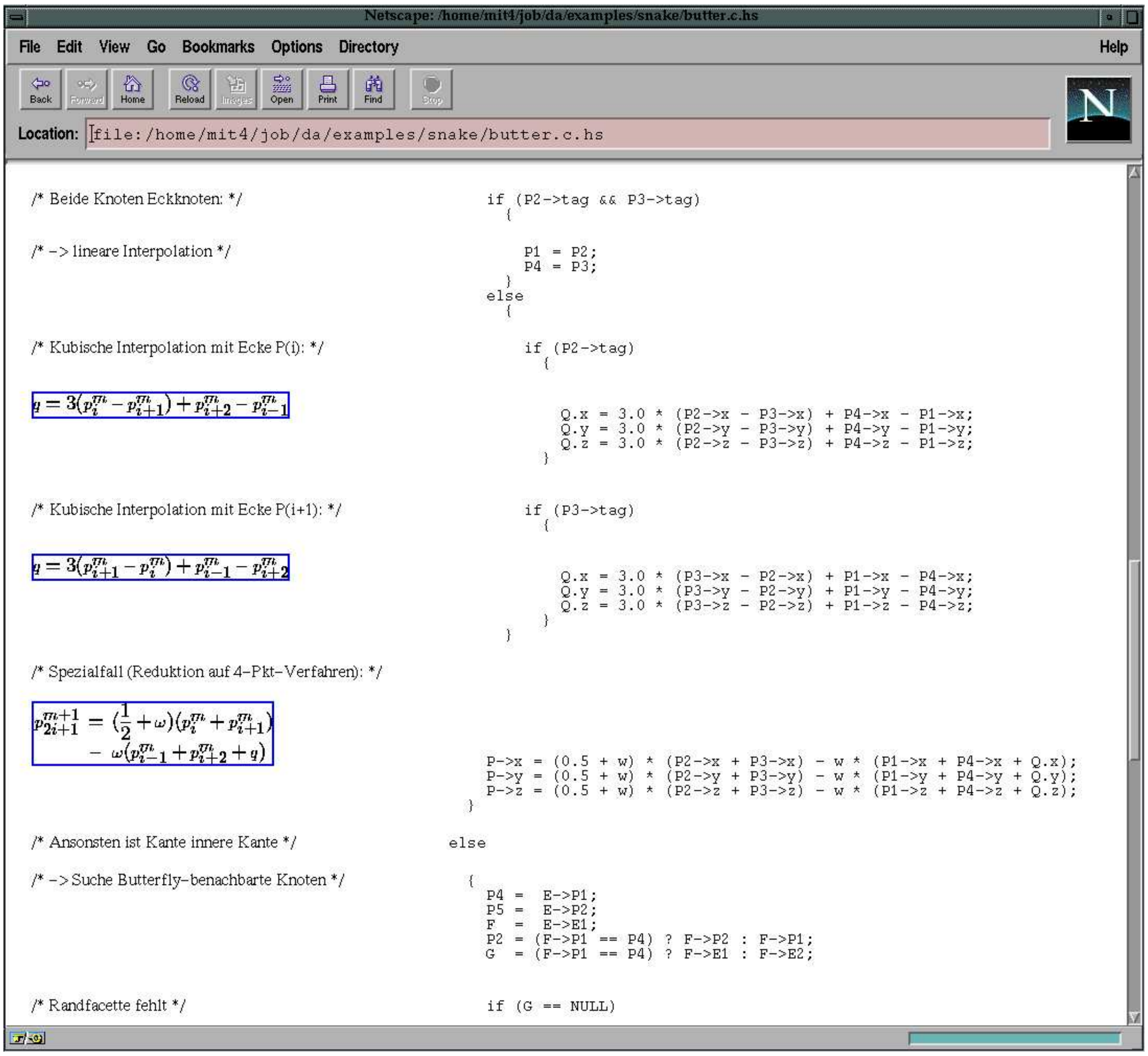


Abbildung B.5: Die Darstellung von Randanmerkungen und Formeln des HyperSource-Dokuments `butter.c.hs` im Netscape-Browser.

Literaturverzeichnis

[Apple92] *Dylan Reference Manual*, Apple Computer, Inc., Cupertino, CA, 1992.

[Berners-Lee94] Berners-Lee, Tim et al.: *The World-Wide Web*, in: CACM Vol. 37 No. 8, August 1994.

[Bigelow88] Bigelow, James: *Hypertext and CASE*, in: IEEE Software, March 1988.

[Boehm81] Boehm, B.: *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[BroCze90] Brown, M.; Czejdo, B.: *A Hypertext for Literate Programming*, Lecture Notes in Computer Science, Vol 468, 1990, S. 250–259.

[Bush45] Bush, Vannevar: *As we may think*, in: Atlantic Monthly, Vol. 76, No. 1, Juli 1945, S. 101–108.

[ChaFisKra91] Chalfonte, Barbara E.; Fish, Robert S.; Kraut, Robert E.: *Expressive Richness: A Comparison of Speech and Text as Media for Revision*, in: Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems, 1991, S. 21–26.

[ChaSta93] Chassell, Robert J.; Stallman, Richard M.: *Texinfo — The GNU Documentation Format*, Second Editition, Free Software Foundation, Cambridge, MA 1993.

[DecRan94] December, John; Randall, Neil: *The World Wide Web Unleashed*, Sams Publishing, Indianapolis, IN 1994.

[DeHBroEsl94] DeHon, Andre; Brown, Jeremy; Eslick, Ian: Global Cooperative Computing, Transit Note #105, Artificial Intelligence Laboratory, MIT, Cambridge, MA, May 1994.

[Dijkstra68] Dijkstra, E.: *The Structure of the 'THE' Multiprogramming System*, in: Communications of the ACM, Vol. 11, No. 6, May 1968.

[Dumpleton94] Dumpleton, Graham: *OSE Tools User Guide*, Dumpleton Software Consulting Pty Limited, Parramatta, Australien.

[DynGreLev90] Dyn, N.; Gregory, J.; Levin, D.: *A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control*, in: ACM Transactions on Graphics, No. 9, 1990, S. 160–169.

[Fahlman94] Fahlman, Scott E.: *Gwydion: An Integrated Software Environment for Evolutionary Software Development and Maintenance*, white paper, Carnegie-Mellon University, Pittsburgh, PA, March 1994. (http://legend.gwydion.cs.cmu.edu/gwydion/gwydion-overview.html)

[Fairley85] Fairley, Richard E.: *Software Engineering Concepts*, McGraw-Hill, Singapore 1985.

[FjelHaml83] Fjelstad, R. K.; Hamlen, W. T.: *Applications program maintenance study*, in: Parikh, G.; Zvegintzov, N. (Hrsg.): *Tutorial on Software Maintenance*, IEEE/CS Press, Silver Spring, MD 1983, S. 13–27.

[GI93] Gesellschaft für Informatik e.V. (GI): *Handlungs- und Verbundprojektvorschläge zur Förderung von Forschung und Entwicklung in der Softwaretechnologie*, in: Informatik — Forschung und Entwicklung, Band 9, Heft 1, 1994, S. 36–51.

[Goldfarb94] Goldfarb, Charles F.: *The SGML Handbook*, Oxford University Press, Oxford 1994.

[Gulbins84] Gulbins, Jürgen: *UNIX*, Springer, Berlin 1984.

[GulKah92] Gulbins, Jürgen; Kahrmann, Christine: *Mut zur Typographie*, Springer-Verlag, Berlin, Heidelberg 1992.

[Herwijnen94] Herwijnen, Eric van: *Practical SGML*, Kluwer Academic Publishers, Boston, Dordrecht 1994.

[Kobbelt95] Kobbelt, Leif: *Iterative Erzeugung glatter Interpolanten*, Dissertation der Univ. Karlsruhe / Verlag Shaker, Aachen, 1995.

[Knuth83] Knuth, Donald E.: *The WEB System of Structured Documentation*, Addison-Wesley, Reading, MA, 1983.

[Knuth84] Knuth, Donald E.: *Literate Programming*, in: The Computer Journal, Vol. 27, No. 2, S. 97–111, Mai 1984.

[Knuth92] Knuth, Donald E.: *Literate Programming*, CLSI Lecture Notes Number 27, Leland Stanford Junior University, 1992.

[Knuth93a] Knuth, Donald E.; *Interview der Computer Literacy Bookshops, Inc.*, 1993. (http://www.clbooks.com/nbb/knuth.html)

[Knuth93b] Knuth, Donald E.; Silvio Levy: *The CWEB System of Structured Documentation for C and C++, Version 3.0*, Addison-Wesley, Reading, MA 1994.

[LeHors91] Le Hors, Arnaud: *XPM — The X PixMap Format*, Groupe Bull / INRIA, Frankreich 1991. (http://avahi.inria.fr/pub/xpm/xpm-3-paper.ps.gz)

[Naur76] Naur, P. et al: *Software Engineering: Concepts and Techniques*, Petrocelli/Charter, New York 1976.

[Nelson65] Nelson, Ted: *A File Structure for the Complex, the Changing, and the Intermediate*, in: Proceedings of the ACM 20th National Conference, 1965.

[OmanCook90] Oman, Paul W.; Cook, Curtus: *Typographical style is more than cosmetic*, in: CACM, Vol. 33, No. 5, S. 506-520, Mai 1990.

[Paoli95] Paoli, Jean: *Authoring Methods for the WWW*, Tutorial Proceedings, Third International WWW Conference, Darmstadt 1995. (Auch unter http://www.grif.fr/en/tutorial/tutorial.html)

[Raggett95] Raggett, Dave: *HTML 3.0 Document Type Definition*, W3O, Geneva 1995.
(http://www.w3.org/hypertext/WWW/MarkUp/html3-dtd.txt)

[SamPom92] Sametinger, J.; Pomberger, G.: *A hypertext system for literate C++ programming*, in: Journal of Object-Oriented Programming, Vol. 4, No. 8, Januar 1992, S. 24-29.

[Sassoon93] Sassoon, Rosemary (Ed.): *Computers and Typography*, Intellect Books, Oxford, England 1993.

[Schmitt83] Schmitt, A.: *Dialogsysteme — Kommunikative Schnittstellen, Software-Ergonomie und Systemgestaltung*, BI, Mannheim 1983.

[Schnupp92] Schnupp, Peter: *Hypertext*, Oldenbourg-Verlag, München 1992.

[Shneiderman89] Shneiderman, Ben: *Reflections on authoring, editing, and managing hypertext*, in: The Society of Text, Barrett, E. (Hrsg.), MIT Press, Cambridge, MA 1989.

[Shneiderman92] Shneiderman, Ben: *Designing the User Interface*, Second Edition, Addison-Wesley, MA 1992.

[Simon93] Simon, Alan R.: *CASE*, Van Nostrand Reinhold, New York 1993

[Siemoneit89] Siemoneit, Manfred: *Typographisches Gestalten*, 2. Auflage, Polygraph, Frankfurt/M. 1989.

[Stallman81] Stallman, Richard M.: *EMACS — The Extensible, Customizable, Self-Documenting Display Editor*, Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland, Oregon, Juni 1981, S. 147–156.

[StaLewLibWin94] Stallman, Richard M.; Lewis, Bill; LaLiberte, Dan; Wing, Ben: *GNU Emacs Lisp Reference Manual*, Second Edition, Revised for Lucid Emacs, Free Software Foundation, Cambridge, MA 1994. (ftp://cs.uiuc.edu/pub/xemacs/docs/lispref.dvi.gz)

[SteMyeCon74] Stevens, W.; Myers, G.; Constantine, L.: *Structured Design*, in: IBM Systems Journal, Vol. 13, No. 2, 1974.

[Tesler81] Tesler, L.: *The Smalltalk Environment*, in: Byte 6(8), August 1981, S. 90–147.

[Tichy85] Tichy, Walter F.: *RCS — A System for Version Control*, in: Software — Practice & Experience, Vol. 15, No. 7, Juli 1985, S. 637–654.

[Weber95] Weber, Kate: *Chapter 6, in which Pooh proposes improvements to Web authoring tools, having seen said tools for the Unix platform*, in: Proceedings of the Third International WWW Conference, Computer Networks and ISDN Systems, Vol. 27, No. 6, Elsevier, Amsterdam, April 1995. (http://www.elsevier.nl/)

[Wirth71] Wirth, N.: *Program Development by Stepwise Refinement*, in: Communications of the ACM, Vol. 14, No. 4, April 1971.

[Woodhead91] Woodhead, Nigel: *Hypertext and Hypermedia*, Addislon Wesley, Wokingham 1991.