2.3 DOgMA: Ein Werkzeug zur Wartung objektorientierter Softwaresysteme in C++

Johannes Sametinger
Christian Doppler Labor für Software Engineering, Institut für Wirtschaftsinformatik
Johannes Kepler Universität Linz

DOgMA wurde als Programmierumgebung für die objektorientierte Programmiersprache C++ konzipiert. Dabei wurde Wartungsaspekten ein besonderes Augenmerk geschenkt, d.h. es wurde versucht, Tätigkeiten, die in der Wartungsphase wichtig sind, besonders zu unterstützen. Das Akronym DOgMA steht für ***DO****cumentation* ***& MA****intenance*.

In Abschnitt 2.3.1 werden die bei der Entwicklung von DOgMA verfolgten Zielsetzungen erläutert. Die zugrundeliegenden Konzepte sind in Abschnitt 2.3.2 beschrieben. Abschnitt 2.3.3 zeigt, wie DOgMA benutzt wird; damit gemachte Erfahrungen kommen in Abschnitt 2.3.4 zum Ausdruck. Eine Zusammenfassung in Abschnitt 2.3.4 und Literaturangaben in 2.3.5 beenden dieses Kapitel über DOgMA.

2.3.1 Zielsetzung

Bestehende Software-Werkzeuge weisen verschiedene Mängel auf, die ihre Brauchbarkeit stark verringern. Sie decken meist nur bestimmte Teilbereiche ab, sind nicht kompatibel zu anderen Werkzeugen und haben schlechte Benutzerschnittstellen. Bei der Entwicklung von DOgMA wurde versucht, folgende Zielsetzungen im Auge zu behalten:

- *Konzentration auf das Wesentliche*
  Es sollten nicht alle denkbaren Aktivitäten der Wartung unterstützt werden. Stattdessen war das Hauptziel, mit vertretbarem Aufwand eine spürbare Kostensenkung bei Wartungstätigkeiten zu erreichen.
- *Synthese vielversprechender Konzepte*
  Die Synthese von Konzepten wie *Literate Programming* und *Hypertext* sollte neue Möglichkeiten eröffnen und wurde einer kombinierten Verwendung von einzelnen Werkzeugen für diese Konzepte vorgezogen.
- *Kompatibilität*
  Isolierte Werkzeuge sind wenig brauchbar, selbst wenn sie den gesamten Software-Lebenszyklus unterstützen. Eine einfache Integration in bestehende (Programmier-) Umgebungen und mit anderen Werkzeugen ist daher wesentlich.
- *Moderne Benutzerschnittstelle*
  Graphische Benutzerschnittstellen sind mittlerweile weit verbreitet, obwohl es immer noch Werkzeuge gibt, die ohne mehrfache Fenster oder Menüs auskommen. Für die Akzeptanz eines Werkzeuges ist eine einfache Benutzung der springende Punkt. Niemand ist gewillt, dicke Manuals zu lesen. Die Verwendung muß deshalb weitgehend intuitiv möglich sein.

Ein zentrales Problem der Software-Wartung ist das Verstehen von Programmen. Dafür muß in der Regel die meiste Zeit aufgewendet werden (Gibson 1989). Systemdokumentation kann hier am leichtesten Abhilfe bringen. Durch Unterstützung des Dokumentierens und des Verstehens bzw. Einarbeitens in fremde Programmsysteme können Wartungskosten am effektivsten reduziert werden. Zu diesem Zweck definieren wir folgende Ziele:

- *Motivation zum Schreiben von Dokumentation*
  Normalerweise verfügen nur die Programmierer selbst über genügend Wissen, um eine brauchbare Systemdokumentation zu erstellen. Sie sind aber selten dazu motiviert, dies zu tun, bzw. tun es nur mit Widerwillen. Bessere Werkzeugunterstützung für das Erstellen und die Integration mit dem Quelltext sollten sich auf die Motivation zumindest positiv auswirken.

- *Zugang zur Dokumentation*
  Dokumentation ist nur von geringer Bedeutung, wenn sie nicht leicht zugänglich ist. Der Benutzer muß darüber informiert werden, ob für bestimmte Stellen im Quelltext Dokumentation vorhanden ist und wo sie zu finden ist.

- *Konsistenz der Dokumentation*
  Dokumentation ist leider selten auf dem aktuellen Stand und enthält daher oft falsche und irreführende Informationen. Irreführende Dokumentation ist sicherlich noch schlechter als gar keine. Ein Werkzeug kann die Konsistenz einer Dokumentation nicht gewährleisten, aber zumindest sollte der Benutzer unterstützt werden, sie zu überprüfen.

- *Vollständigkeit der Dokumentation*
  Der Dokumentationsumfang ist zum Teil firmenspezifisch und variiert oft von Projekt zu Projekt. Ein Wartungswerkzeug muß in der Lage sein, den Benutzer darüber aufzuklären, welche Teile des Systems (wo) dokumentiert sind und welche Lücken sich in der Dokumentation befinden.

Gute Dokumentation ist ein Weg, um das Verstehen von Programmen zu unterstützen. Aber es gibt noch eine Fülle anderer Möglichkeiten.

- *Zugang zu Informationen*
  Einfacher Zugang zum Quelltext ist ebenso wichtig wie der zur Dokumentation. Zum Verstehen von Quelltext ist oft die Beantwortung vieler Fragen, wie *Wo wird diese Variable definiert/gesetzt/verwendet?*, notwendig. Ein Wartungswerkzeug muß einfachen Zugang zu all jenen Informationen schaffen, die zur Beantwortung solcher Fragen notwendig sind.

- *Verhindern von Seiteneffekten*
  Kleine Änderungen in einem Softwaresystem haben oft ungeahnte Konsequenzen. Die Ursache dafür ist meist mangelndes Verständnis über das zu wartende System. Hier kann wiederum einfacher und schneller Zugang zu relevanten Informationen Abhilfe schaffen.

- *Bewältigung der Komplexität*
  Es hat sich herausgestellt, daß objektorientierte Softwaresysteme meist komplexer sind als konventionelle. In der Regel existieren eine Vielzahl von Komponenten und

Beziehungen, die man ohne entsprechende Werkzeugunterstützung nur schwer nachvollziehen kann.

Aktivitäten, die während der Wartung durchgeführt werden, unterscheiden sich nicht völlig von denen in der Entwicklungsphase. Lediglich die Schwerpunkte sind unterschiedlich. So kommt dem Verstehen in der Entwicklung weniger Bedeutung zu, wenn die Entwickler von Anfang an dabei waren. Ein Wartungswerkzeug muß aber dennoch alle Aktivitäten der Software-Entwicklung unterstützen, da Wartung nicht nur aus Fehlerbehebungen besteht, sondern vielfach auch die Weiterentwicklung eines Systems bedeutet.

2.3.2 Konzepte

Das Verstehen von (fremden) Programmen ist wie bereits erwähnt eines der Hauptprobleme bei der Softwarewartung. *Hypertext* verspricht hier wesentliche Verbesserungen zu bringen. *Literate Programming* hingegen hat gezeigt, daß die Qualität sowohl von Quelltext als auch der Dokumentation durch eine Integration wesentlich verbessert werden kann. Eine Kombination von *Hypertext* und *Literate Programming* bedeutet eine noch stärkere Bindung zwischen Dokumentation und Quelltext mit der Möglichkeit, Vollständigkeit und Konsistenz sehr einfach kontrollieren zu können.

Im folgenden werden die Ideen von *Hypertext* und *Literate Programming* kurz vorgestellt. Anschließend werden zur Anwendung dieser Konzepte die Strukturen von Quelltext und Dokumentation, sowie deren mögliche Integration erläutert. Weiters werden die Konsequenzen dieser Integration am Umgang mit der Dokumentation (Lesen, Schreiben, Ändern) dargestellt. Abschließend werden noch weitere Möglichkeiten zur Verbesserung der Lesbarkeit erörtert.

Hypertext

Textdateien sind linear aufgebaut. Dieser lineare Aufbau ist in vielen Fällen ungünstig. Sowohl der Quelltext eines Softwaresystems als auch die Dokumentation haben eine komplexe logische Struktur mit gedanklichen Querverbindungen auch zwischen Code und Dokumentationstext. Lineare Textdateien sind völlig ungeeignet, solche Strukturen abzubilden. Abhängig von den jeweiligen Interessen der Leser gibt es eine Vielzahl von Reihenfolgen, die vorhandene Information zu studieren.

Hypertext bietet die Möglichkeit des nichtsequentiellen Lesens und Schreibens. Ein *Hypertext* besteht aus einer Reihe von Knoten. Jeder Knoten enthält eine (meist kleine) Menge an Information, üblicherweise ein Stück Text, ein Bild; aber auch Videosequenzen, akkustische Signale oder ausführbare Programme sind denkbar. Die Knoten werden durch Kanten verbunden und bilden einen gerichteten Graphen.

Navigieren durch einen *Hypertext* bedeutet das Verfolgen von Kanten. Jeder Knoten kann mehrere ausgehende Kanten haben. Deshalb gibt es eine Vielzahl von Reihenfolgen, die vorhandene Information zu *konsumieren*. Der Leser hat somit das Gefühl, daß er sich frei durch eine große Menge von Information bewegen kann (Fiderio 1988, Nielsen 1990, Smith 1988).

Der Ursprung von *Hypertext* reicht zurück in die vierziger Jahre. Damals schon hatte Vannevar Bush eine Vision über die Organisation von Informationen (Bush 1945). Er hatte aber zu der Zeit leider noch keine Möglichkeit, seine Ideen praktisch umzusetzen. Diese Ideen beeinflußten aber ca. 20 Jahre später die Arbeiten von Douglas Engelbart,

der eines der ersten *Hypertext*-Systeme entwickelte. Es sollte aber weitere 20 Jahre dauern, bis Computer schnell und billig genug wurden, um *Hypertext*-Systeme einem größeren Publikum zugänglich zu machen (Conklin 1987, Shneiderman 1989).

Nichtsequentielles Lesen und Schreiben ist sowohl für den Quelltext als auch für die Dokumentation von Softwaresystemen von besonderer Wichtigkeit. Insbesondere können dadurch aber auch Verbindungen zwischen Quelltext und Dokumentation hergestellt werden. *Hypertext* gewährleistet somit einfachen Zugang zu wichtiger Information.

Auch die bereits sehr verbreiteten Browsing-Werkzeuge können als (einfache) *Hypertext*-Werkzeuge betrachtet werden, da sie einen nichtsequentiellen Zugriff auf Informationen ermöglichen. So kann in modernen Programmierumgebungen, wie sie beispielsweise für Smalltalk gang und gäbe sind (Digitalk 1989, LaLonde 1990), von Klassen zu deren Methoden und auch zu anderen Klassen, die mit der ursprünglichen in bestimmten Beziehungen stehen, verzweigt werden. Bei Browsing-Werkzeugen werden Knoten (z.B. Klassen, Methoden) in Listen dargestellt. In unterschiedlichen Listen stehen dann Knoten, die untereinander in Beziehung stehen (z.B. Unterklassen, (geerbte) Methoden).

Literate Programming

Programme werden geschrieben, um von einem Computer ausgeführt zu werden. Man sollte sie aber schreiben, damit sie von anderen Leuten gelesen werden (können). Eine von Donald Knuth veröffentlichte Idee besagt, daß wir versuchen sollten, Leuten zu erklären, was wir vom Computer wollen, anstelle einfach die Maschine zu instruieren (Knuth 1984, Knuth 1992). Computerprogramme sollten also ebenso leserlich gestaltet werden wie herkömmliche Literatur, deshalb auch der Name *Literate Programming*. Das Hauptziel besteht nicht darin, ein ausführbares Programm zu schreiben, sondern ein Problem mit seiner Lösung mitsamt allen Annahmen, alternativen Lösungen und Entwurfsentscheidungen zu beschreiben. Das dies tatsächlich möglich ist wurde an mehreren kleinen aber auch großen Beispielen gezeigt, z.B. (Knuth 1986).

Um die Idee des *Literate Programming* auch in die Praxis umsetzen zu können entwickelte Knuth das WEB-System. Ein WEB-Programm besteht aus einer Reihe von Dokumentationskapiteln, in die der Quelltext integriert ist. Jedes Kapitel beschreibt einen bestimmten Aspekt des Software-Systems. Der Quelltext muß dabei nicht in der vom Compiler verlangten Reihenfolge notiert werden. Vielmehr wird vom WEB-System der Code aus der Dokumentation extrahiert und richtig zusammengesetzt und kann dann übersetzt und ausgeführt werden. Ebenso kann ein fertig dokumentiertes System mit automatisch generierten Querverweisen, Inhaltsverzeichnis und Indizes ausgegeben werden. Neben dem ursprünglich für die Programmiersprache Pascal entwickelten System entstanden ein Menge andere Versionen u.a. für die Sprachen C, Modula-2, Lisp, Fortran und auch C++ und Smalltalk.

Programmieren mit der Dokumentation ist ein Schritt vorwärts in Richtung besser verständlicher und wartbarer Software.

Um die Vorteile von *Hypertext* und *Literate Programming* ausnützen zu können, müssen wir sowohl im Quelltext als auch in der Dokumentation Strukturen erkennen und finden. Dann können wir ein Knotengerüst im Sinne von *Hypertext* definieren und damit auch eine Integration im Sinne von *Literate Programming* vornehmen.

Struktur des Quelltextes

Objektorientierte Softwaresysteme bestehen in der Regel aus einer Reihe von Klassen. Eine Klasse beschreibt Struktur und Verhalten einer Menge von Objekten. Eine Klasse (Unterklasse) kann Eigenschaften von anderen Klassen (Oberklassen, Basisklassen) erben. Objekte kommunizieren durch das Versenden von sog. Messages, die das Ausführen von bestimmten Operationen bewirken. Diese Operationen nennt man üblicherweise Methoden. Das Verhalten von Unterklassen kann durch das Überschreiben von Methoden der Oberklassen verändert werden.

Ein objektorientiertes Softwaresystem besteht demnach aus Klassen und Methoden mit folgenden Beziehungen:

- Eine Klasse kann Ober- und/oder Unterklasse von anderen Klassen sein.
- Eine Klasse hat eine Menge von Methoden.
- Eine Methode gehört zu einer bestimmten Klasse.
- Eine Methode kann in einer Unterklasse überschrieben sein.

Abbildung 2.3-1 zeigt Beziehungen zwischen Klassen und Methoden.

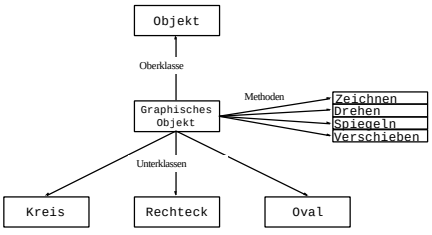

Abb. 2.3-1 Beziehungen zwischen Klassen und Methoden

Man kann sich mit der Strukturierung auf dieser Ebene zufriedengeben, oder aber auch Klassen und Methoden weiter zergliedern und auf Ebene der Bezeichner Beziehungen in Betracht ziehen:

- Ein Bezeichner ist in einer Klasse oder Methode definiert.
- Die Verwendung eines Bezeichners steht in Beziehung mit dessen Definition und anderen Verwendungen dieses Bezeichners.
- Ein Kommentar enthält möglicherweise eine Kurzbeschreibung eines Bezeichners, z.B. Beschreibung einer Klasse, Methode oder Variablen. Solche Kommentare stehen mit allen Verwendungen eines Bezeichners in Verbindung.

Struktur von in C++ implementierten Systemen

Softwaresysteme werden in bestimmten Programmiersprachen implementiert. Es stellt sich die Frage, ob die Sprache einen wesentlichen Einfluß auf die Struktur eines Softwaresystems hat. C++ ist keine reine objektorientierte Programmiersprache sondern eine objektorientierte Erweiterung der Sprache C (Stroustrup 1991). Man spricht daher auch von einer hybriden Sprache. Ein in C++ implementiertes Softwaresystem besteht aus einer Menge von Dateien, welche Klassendefinitionen, Methodenimplementierungen und globale Deklarationen enthalten. Globale Deklarationen können in mehreren Klassen und Methoden sichtbar sein.

In C++ gibt es keine Vorschriften und Regeln darüber, wie die Teile eines Systems auf Dateien verteilt werden müssen. Eine Datei kann somit mehr als eine Klassendefinition

enthalten, und eine Klassendefinition mit ihren dazugehörigen Methodenimplementierungen kann über mehrere Dateien verstreut sein.

Um eine bestimmte Klasse verwenden zu können, muß die Datei mit der Klassen definition inkludiert werden. Dies geschieht mit einer speziellen Preprozessor-Anweisung. Neben der oben erwähnten Vererbungsbeziehung zwischen Ober- und Unterklassen existiert somit noch eine weitere wichtige Beziehung zwischen Dateien eines in C++ geschriebenen Softwaresystems, die *include*-Beziehung. Um alle notwendigen Infomationen über eine Klasse zu eruieren, muß man sehr oft mehrere Dateien, die direkt oder indirekt inkludiert werden, inspizieren. Folgende Beziehungen sind von Bedeutung:

- Eine Klassendefinition *steht in* einer Datei.
- Eine Klasse *erbt von* einer anderen Klasse.
- Eine Methode *steht in* einer Datei.
- Eine Methode *gehört zu* einer bestimmten Klasse.
- Eine Methode *wird* in einer Unterklasse *überschrieben*.
- Eine Datei *wird* von anderen Dateien *inkludiert*.

Ein Bezeichner kann nicht nur in einer Klasse oder einer Methode definiert werden, sondern auch global in einer Datei und kann somit auch in mehreren Klassen verwendet werden. Der von der Sprache C übernommene Mechanismus, Dateien zu inkludieren, sowie ein fehlender Mechansimus, um Dateien und Klassen zuzuordnen, erschwert den Prozeß des Verstehens von C++-Programmen beträchtlich.

In C++-Programmen können natürlich auch konventionelle Strukturen wie Funktionen oder globale Deklarationen, die von C übernommen wurden, vorkommen. Ein Software-Werkzeug muß in der Lage sein, alle vorhandenen Strukturen in entsprechender Weise dem Benutzer darzustellen und damit den Einarbeitungsprozeß nach Möglichkeit zu verringern.

Benutzerdefinierte Strukturen

Die bisher beschriebenen Strukturen werden wesentlich von der Syntax der Programmiersprache beeinflußt. Einheiten wie Methoden müssen weiter zergliedert werden können, um den logischen Aufbau besser darzustellen. Dies ist beispielsweise durch die Zusammenfassung und kurze Beschreibung von Textteilen möglich. Ein Werkzeug kann solche Textteile durch ihre Beschreibung (Überschrift) ersetzen und damit Details verbergen und auf Verlangen wieder zugänglich machen. Der Programmtext einer Methode, die einen Menübalken erstellt, kann beispielsweise in verschiedene Textgruppen für die einzelnen Menüs unterteilt werden.

Abbildung 2.3-2 zeigt eine C++-Methode zum Erzeugen von sechs Menüs für einen Menübalken. Der Quelltext für die einzelnen Menüs ist verborgen.

Bei der Anwendung des Prinzips der schrittweisen Verfeinerung können somit die verschiedenen Entwicklungsschritte aufgehoben werden. Üblicherweise gehen diese wichtigen Informationen für die Wartungsphase verloren. Mit einer einfachen Zuordnung von Beschreibungstexten zu Textteilen aber können sie erhalten werden.

```
PullDownBar *HyperTextDocument::CreateMenuBar()
{
    ObjList *list= new ObjList;
    File Menu
    Edit Menu
    Project Menu
    Text Menu
    Identifier Menu
    Goodies Menu
    return new PullDownBar(this, list);
}
```

Abb. 2.3-2 Verborgene Textteile

Struktur der Dokumentation

Ein Softwaresystem besteht nicht nur aus Quelltext, auch die Dokumentation ist ein wesentlicher Bestandteil. Leider bestehen für den Aufbau der Dokumentation keine formalen Kriterien, wie dies für den Quelltext der Fall ist. Eine gut strukturierte Dokumentation ist aber wesentlich für ein schnelles Zurechtfinden. Wir wollen daher eine einfache Struktur für unsere Zwecke definieren:

> Die Dokumentation besteht aus einer Reihe von Kapiteln. Diese sind ähnlich organisiert wie Klassen, d.h. es gibt eine hierarchische Struktur ähnlich der Vererbungshierarchie. Ein Kapitel selbst besteht aus einem Titel, Dokumentationstext (purer ASCII-Text) und Programmtext (d.h. Verbindungen zu bestimmten Teilen im Quelltext).

Die Kapitel der Dokumentation bilden einen Baum:

- Ein Kapitel kann mehrere Unterkapitel haben, aber höchstens ein Oberkapitel.
- Zusätzlich existiert zwischen Kapiteln mit gleichem Oberkapitel eine Ordnungsrelation.

Abbildung 2.3-3 zeigt die Beziehung des Kapitels 1.2, welches ein Ober- und drei Unterkapiteln hat und auch mit seinem Vorgänger- und Nachfolgerkapitel in Beziehung steht. (Üblicherweise unterscheidet man zwischen Kapiteln und Abschnitten. Für die hier definierte Dokumentationsstruktur ist diese Unterscheidung jedoch nicht von Relevanz.)

Wie der Quelltext soll auch die Dokumentation in Dateien abgelegt werden, die ein oder mehrere Kapitel enthalten können. Das Ablegen jedes Kapitels in separaten Dateien würde zu einer unnötig großen Zahl von Dateien führen.

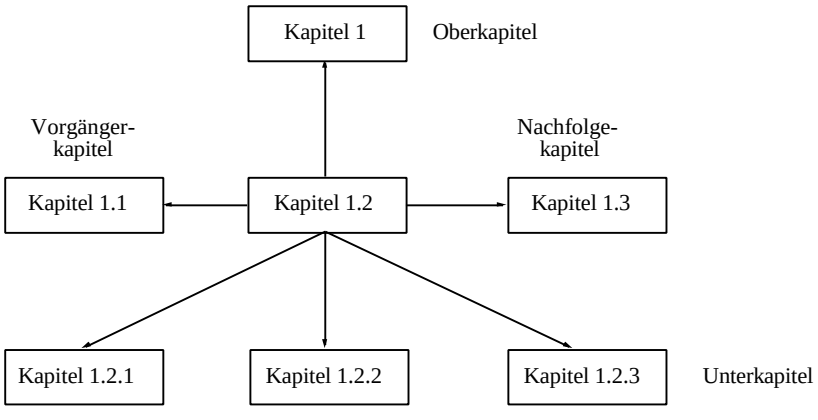

Abb. 2.3-3 Dokumentationsstruktur

Integration von Quelltext und Dokumentation

In Sinne des *Literate Programming* muß es eine Verbindung zwischen Dokumentation und Quelltext geben, damit für das Verständnis wichtige Informationen wie Entwurfsentscheidungen auch vom Quelltext aus leicht zugänglich sind. Zu diesem Zweck definieren wir, wie diese Integration stattfinden soll. Insbesondere wollen wir auch eine Einbindung von einzelnen Bezeichnern vornehmen, da diese sehr häufig in Dokumentationstext vorkommen und auch dabei Querverbindungen zum Quelltext erhalten bleiben sollen. Demnach gibt es zwei Möglichkeiten, Quelltext und Dokumentation zu vereinen:

1) einfache Bezeichner
2) eine beliebige Anzahl aufeinanderfolgender Zeilen im Quelltext

Einfache Bezeichner sollen an beliebiger Stelle in der Dokumentation stehen dürfen. Mehrere Quelltextzeilen sind vom Dokumentationstext klar getrennt, siehe Abb. 2.3-4. Zur besseren Lesbarkeit sind Quelltext-Bezeichner im Text fett und mit anderer Schrift dargestellt.

These menu entries for copying and pasting an identifier are enabled only when the source code has been parsed (i.e., **`alreadyParsed`** is set to TRUE) and when the current mark (**`cm`**) or the last mark copied is an identifier, respectively. The method **`CopiesIdent`** yields this information.

```
if (cm && cm->CopiesIdent())
   identMenu->EnableItem(cIdentCopy);
if (GetClipMark() && GetClipMark()->CopiesIdent())
   identMenu->EnableItem(cIdentPaste);
```

Finally, whenever one of our menu entries is selected by the user, we gain control in the method **`DoMenuCommand`**.

Abb. 2.3-4 Integration von Quelltext und Dokumentation

Hinsichtlich der Integration von Quelltext und Dokumentation darf es keine wesentlichen Beschränkungen geben. Ein Kapitel kann eine einzelne Klasse beschreiben, eine Methode, eine Funktion, aber auch einen Sachverhalt, dessen Implementierung über mehrere Klassen und/oder Methoden verstreut ist. (Dies ist bei objekt orientierten Systemen typischerweise der Fall.)

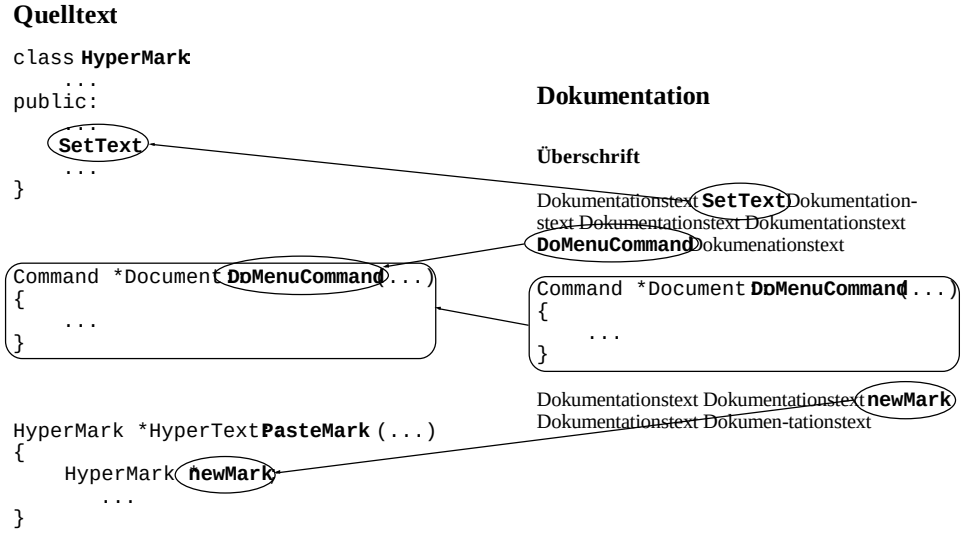

Abb. 2.3-5

Verbindungen zwischen Quelltext und Dokumentation

Abbildung 2.3-5 zeigt den schematischen Aufbau eines Kapitels. Es besteht aus einer Überschrift, Text mit mehreren Bezeichnern aus dem Quelltext, sowie einer Methode.

Durch die vorgestellte Integration von Quelltext und Dokumentation ergibt sich ein anderer Umgang mit der Dokumentation. Wir wollen daher im folgenden darstellen, wie wir uns die Arbeit mit der Dokumentation vorstellen, siehe dazu auch (Sametinger 1992a).

Das Schreiben von Dokumentation

Dokumentation sollte sowohl vor als auch während der Implementierung geschrieben werden. Dies geschieht allerdings sehr selten. Einer der Gründe dafür ist, daß sich die Struktur und der Quelltext eines Softwaresystems während der Entwicklung häufig ändern. Dies bedingt eine entsprechende Änderung der Dokumentation, was ein wesentlicher Mehraufwand gegenüber einer einmaligen Dokumentation nach der Implementierung zu sein scheint.

Ein Werkzeug soll keine keine bestimmte Vorgehensweise erzwingen sondern nach Möglichkeit alle Fälle unterstützen. So kann Dokumentation vor der Entwicklung (z.B. Entwurfsentscheidungen), parallel mit dem Quelltext und/oder im nachhinein (auch für ein bereits bestehendes System) erstellt werden.

Der Quelltext für eine bestimmte Aufgabe ist insbesondere bei objektorientierten Systemen oft über mehrere Stellen (Klassen und Methoden, und sog. Dateien) verstreut. Um Dokumentation für ein System zu schreiben, wollen wir einfach ein neues Kapitel anlegen, einfachen Dokumentationstext schreiben und einzelne Bezeichner sowie größere Textteile aus dem Quelltext einfügen (kopieren). Dabei spielt es keine Rolle, ob der Quelltext schon existiert oder zum Zeitpunkt der Dokumentationserstellung erst entwickelt wird.

Das Ändern von Dokumentation

Übliche Editiermöglichkeiten erlauben beliebige (explizite) Änderungen in der Dokumentation. Von wesentlicher Bedeutung sind Änderungen, die implizit durch die vorhandenen Querverbindungen durchgeführt werden und so automatisch Konsistenz zwischen Quelltext und Dokumentation gewährleisten:

- Wird der Name eines Bezeichners im Quelltext geändert, dann werden automatisch alle Vorkommnisse dieses Bezeichners in Quelltext und Dokumentation umgeändert.
- Werden andere Änderungen im Quelltext gemacht, dann werden aufgrund der Verbindungen zwischen Quelltext und Dokumentation diese Änderungen ebenfalls in der Dokumentation automatisch nachvollzogen.
- Werden Teile des Quelltextes gelöscht, die auch in der Dokumentation vorkommen, dann wird an den entsprechenden Stellen ein Vermerk angezeigt, welcher Quelltextteil an diesen Stellen eingefügt war und nun nicht mehr vorhanden ist.

Beispiel-Dokumentation

Angenommen es soll neue Funktionalität in ein bestehendes Softwaresystem eingefügt werden. Dazu ist überlicherweise eine Änderung an mehreren Stellen im Quelltext notwendig. Diese Änderungen bzw. Erweiterungen wollen wir in einem Kapitel der Dokumentation beschreiben. Wir können sowohl Quelltext als auch Dokumentation parallel

entwickeln. Dazu müssen wir aber von Anfang an genau wissen, was wir zu tun haben. Wir schlagen vor, zuerst den Quelltext einzufügen (vorausgesetzt es ist nicht zu umfangreich) und zu testen. Erst dann schreiben wir die Dokumentation und integrieren den vorhin geschriebenen Quelltext. Ein Beispiel für ein solches Kapitel ist in Abb. 2.3-6 zu sehen. Dieses Beispiel soll einen Eindruck über den möglichen Aufbau eines Kapitels vermitteln. Dessen volles inhaltliches Verständnis ist ohne Wissen über das dabei erweiterte System allerdings nicht möglich. Zur besseren Lesbarkeit wurden verschiedene Schriftarten für Bezeichner verwendet (siehe dazu später).

Das Lesen von Dokumentation

Es gibt zwei grundsätzliche Möglichkeiten, durch ein System zu navigieren. Man kann entweder den Quelltext inspizieren und für weitere Informationen bei der Dokumentation nachschauen, falls welche vorhanden ist. Man kann aber auch die Dokumentation lesen und in den Quelltext verzweigen, falls die Information in der Dokumentation nicht detailliert genug ist. Man darf aber nicht vergessen, daß ein Werkzeug zwar ein komfortables Arbeiten ermöglichen kann, dies aber keine Garantie für eine hochwertige und vollständige Dokumentation ist.

Es gibt mehrere Möglichkeiten, von der Dokumentation in den Quelltext zu verzweigen:

- Von einem Bezeichner in der Dokumentation kann man zu dessen Definition verzweigen. Die Definition eines Bezeichners befindet sich immer im Quelltext.
- Weiters kann man auch zu jener Stelle im Quelltext verzweigen, von der ein größerer Textteil in die Dokumentation übernommen wurde.
- Bezeichner können in der Dokumentation ebenso wie im Quelltext hervorgehoben werden, sodaß die Vorkommnisse von Bezeichnern nicht nur im Quelltext sondern auch in der Dokumentation sehr leicht gefunden werden können.

Um vom Quelltext in die Dokumentation zu verzweigen, ist die Information notwendig, ob zu einem bestimmten Teil Dokumentation vorhanden ist, d.h. ob ein Quelltext-Teil oder einzelner Bezeichner irgendwo in der Dokumentation integriert ist.

Verbesserung der Lesbarkeit

Lesbarkeit von Quelltext und Dokumentation haben einen wesentlichen Einfluß auf das Verstehen eines Softwaresystems. Ein Werkzeug kann inhaltlich keine Änderungen vornehmen, aber optisch können durchaus Verbesserungen an der Lesbarkeit vorgenommen werden. Eine sehr einfache Möglichkeit dazu ist die Verwendung von verschiedenen Schriftarten für verschiedene syntaktische Elemente wie Kommentare, Schlüsselwörter, Bezeichnerdefinitionen. Letzteres ist insbesondere für eine Sprache wie C++ von Vorteil, da die kryptische Syntax das Erkennen der definierten Bezeichner in Deklarationen oft erschwert.

In der Klassendefinition in Abb. 2.3-7 sind Instanzvariablen und Methodennamen auf einen Blick erkennbar. Desweiteren sind die auskommentierten Methoden sofort als solche zu sehen.

Copy/Paste of Identifiers

If the user wants to copy an identifier, we just remember the identifier (i.e., the current mark) in a global variable by calling the method **SetClipMark** .

```
case cIdentCopy:
   if (textView->GetCurrentMark())
      project->SetClipMark(textView->GetCurrentMark());
   break;
case cIdentPaste:
   ...
   break;
```

If the identifier is to be pasted ,we send the text the message **PasteMark** and provide the mark of the identifier as first parameter. (We get this global variable again by calling **GetClipMark**). Afterwards we call **SetText**. This guarantees that the styles are set correctly (especially for the one identifier or text we just pasted). Finally, the identifier is selected.

```
HyperMark *HyperText::PasteMark (HyperMark *m,int *from,int *to)
{
   DoDelayChanges dc(this);
   HyperMark *newMark;
   newMark= m->PasteMark(this,from,to);
   //marks are already updated correctly in PasteMark
   SetChanged();
   return newMark;
}
```

We just call the method **PasteMark** of the **HyperMark** that we get as our first parameter. **PasteMark** is a virtual method of the class **HyperMark** which is overwritten in the marks for identifiers (i.e., in **IdentDefMark** and **IdentUseMark**). In **HyperMark** we do not have to do anything:

```
virtual HyperMark *PasteMark (class HyperText *, int *, int *)
{ return 0; }
```

If the use of an identifier has to paste itself, it delegates this task to its definition:

```
HyperMark *IdentUseMark::PasteMark (HyperText *ht, int *from, int *to)
{
   return identDef?identDef->PasteMark(ht,from,to):0;
}
```

The definition finally does the pasting:

```
HyperMark *IdentDefMark::PasteMark (HyperText *ht, int *from, int *to)
{
   TextPtr t= hText->Save(pos,pos+len);
   ht->Paste(t,*from,*to);
   *to= *from+len;
   ...
}
```

Two cases have to be distinguished: In a documentation text we generate a **DocuIdentMark** and in a source text we generate a usual identifier use, i.e., an **IdentUseMark**. In the documentation we must not forget to insert our special character (**cHyperTextChar**), so we can recognize these identifiers when writing the documentation text to a file.

Abb. 2.3-6 Beispielkapitel

```
class HyperMark: public Mark {     //abstract class
     class HyperText    *hText;
public:
     HyperMark(int pos, int len, class HyperText *ht);
     ~HyperMark();
     virtual void SetText(class HyperText *ht) { hText= ht; }
     class HyperText *GetText() { return hText; }
     virtual void SetString(char *str);
     virtual char *GetString(); /*
     virtual void SetStyle(TextRunArray *styles);
     virtual Style *MyStyle();
     HyperMark *GetLastLink() { return lastLink; }
     virtual bool HasSameDefinition(HyperMark *mp);  */
     virtual void UpdateUses();
     …
};
```

Abb. 2.3-7 Verschiedene Schriftarten in einer Klassendefinition

Verschiedene Schriftarten sind auch für die Dokumentation anwendbar, um Quelltextbezeichner im Dokumentationstext zu erkennen, um nicht auflösbare Verweise in den Quelltext zu finden und natürlich in den Quelltextteilen der Dokumentation genauso wie im Quelltext selbst.

Weiters können verschiedene Schriftarten nicht nur syntaktische sondern auch semantische Unterschiede zum Ausdruck bringen. So ist eine unterschiedliche Schriftart für globale und lokale Variablen sehr hilfreich, um beispielsweise Seiteneffekte in Funktionen und/oder Methoden zu finden. Auch kann auf diese Weise das Vorkommen einzelner Bezeichner sowohl im Quelltext als auch in der Dokumentation sehr schnell aufgespürt werden. Abbildung 2.3-8 zeigt eine Methode, in der Instanzvariablen der dazugehörigen Klasse fett und lokale Variable zusätzlich kursiv dargestellt sind. Zur besseren Unterscheidung können auch unterschiedliche Farben eingesetzt werden.

```
void IdentUseMark::DoObserve(ObjPtr,void *what)
{
   //ident def has changed or we have to use another definition
   if ((int)what==cMasterCopyChanged) {
      if (!copy) return;
      int defPos, defLen;
      Lock();
      identDef->GetTextPosition(defPos,defLen);
      if (len!=defLen)
        hText->Paste (copy, pos, pos+len);
      else
        hText->StyledText::Paste (copy, pos, pos+len);
      len= defLen;
      Unlock();
   } else if ((int)what==cDefinitionChanged) {
      identDef->RemoveDependent(this);
      identDef= identDef->NewDefinition();
      identDef->AddDependent(this);
   } else if ((int)what==cHighlightIdent) {
      if (identDef->highlight) hText->GetNodeItem()->IncCount();
      else hText->GetNodeItem()->DecCount();
      //styles are updated automatically when a text is displayed
   }
}
```

Abb. 2.3-8 Hervorgehobene Bezeichner in einer Methode

2.3.3 Benutzung

Die Benutzerschnittstelle basiert auf modernen graphischen Fenster- und Menütechniken. Einfache Benutzung war eines der Hauptziele. Es wurde weiters versucht, die Benutzerschnittstelle weitgehend unabhängig von der unterstützten Programmiersprache C++ zu gestalten. (Es wurde auch eine DOgMA-Version für Modula-2 erstellt.)

Dieses Kapitel bringt zunächst eine Erläuterung über den Aufbau der graphischen Benutzerschnittstelle. Anschließend werden die verschiedenen Möglichkeiten zum Navigieren erläutert. Weitere nützliche Kommandos, insbesondere in Zusammenhang im Bezeichnern (z.B. Informationen einholen, Umbenennen, Vorkommnisse eruieren), runden dieses Kapitel ab. Eine detailliertere Beschreibung der Benutzerschnittstelle ist in (Sametinger 1992b) enthalten.

Überblick

Abbildung 2.3-9 zeigt die Benutzerschnittstelle, die aus einem Menübalken, einer Informationsbox, zwei Auswahllisten und einem Editor besteht.

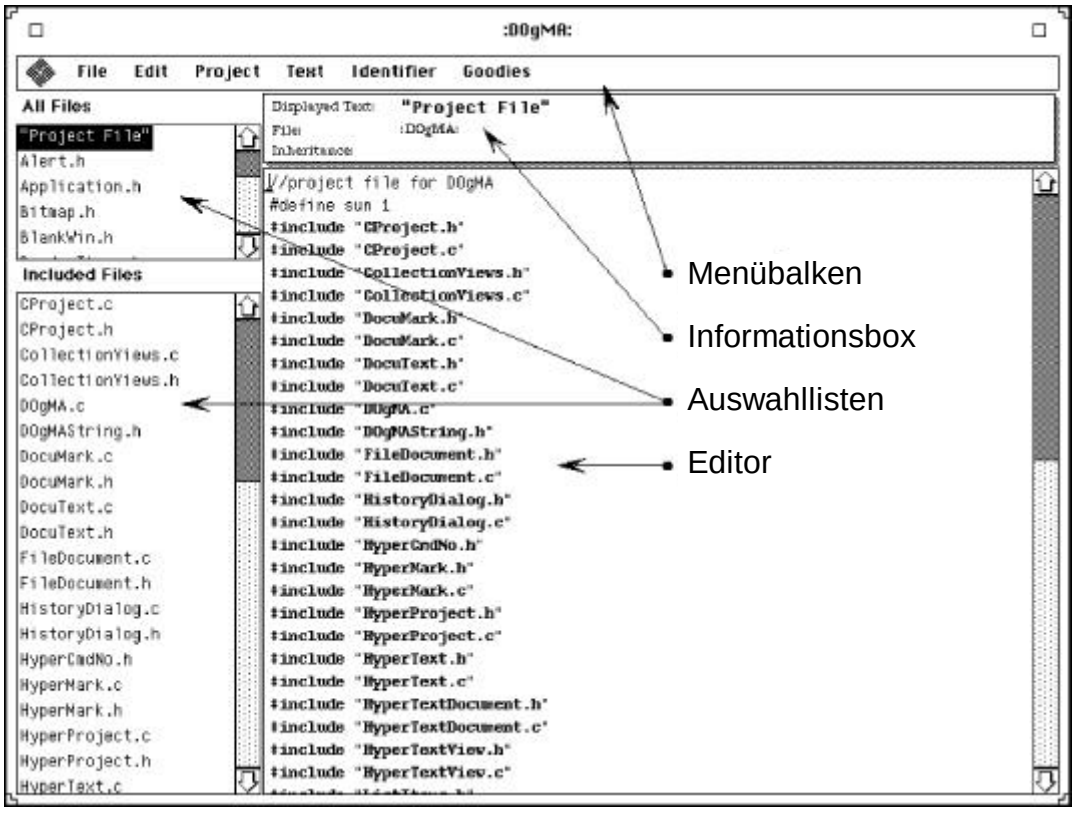

Abb. 2.3-9 Benutzerschnittstelle

Der Menübalken

Es werden folgende sechs Menüs angeboten:

- Das ***File-Menü*** enthält Menüeinträge zum Laden und Speichern von Dateien.
- Das ***Edit-Menü*** bietet neben Menüeinträgen zum einfachen Editieren von Text (z.B. cut, copy, paste) auch die Möglichkeit, Klassen und Methoden anzulegen, bzw. zu löschen.
- Das ***Project-Menü*** offeriert Kommandos, die auf ein gesamtes Projekt anwendbar sind, z.B. öffnen, sichern, schließen eines Projekts.

- Das ***Text-Menü*** bietet allgemeine *Hypertext*-Kommandos.
- Das ***Identifier-Menü*** bietet *Hypertext*-Kommandos auf Bezeichner-Ebene.
- Das ***Goodies-Menü*** enthält einige zusätzliche, nützliche Kommandos.

Die Informationsbox

Die Informationsbox stellt wichtige Informationen über den gerade dargestellten Text (Quelltext oder Dokumentation) dar. Dies soll verhindern, daß sich der Benutzer in einem komplexen Informationsnetz verliert. Folgende Informationen werden dargestellt:

- Name des gerade (im Editor) angezeigten Textes
- Dateiname (inklusive Verzeichnispfad)
- Vererbungspfad einer Klasse oder Methode

Die Auswahllisten

Klassifizierung und Gruppierung sind wichtige Hilfsmittel zur Bewältigung der Komplexität bei großen Systemen. Smalltalk-Systeme bieten in der Regel die Möglichkeit, Klassen und Methoden beliebigen Kategorien zuzuordnen. Wir verwenden eine einfachere Variante und stellen Klassen, Kapitel und Dateien in der oberen Auswahlliste dar. Über der Liste ist jeweils ersichtlich, was gerade in der Liste angezeigt wird. In der unteren Auswahlliste werden Klassen, Kapitel, Dateien und Methoden angezeigt, die mit dem jeweils oben ausgewählten in einer bestimmten Verbin dung stehen. Der Text über der unteren Auswahlliste zeigt die jeweils dargestellte Relation an.

Um andere Kategorien oder Beziehungen auszuwählen, stehen Popup-Menüs zur Verfügung. In der oberen Liste stehen Dateien, Klassen, Funktionen und Dokumentationskapitel zur Auswahl.

In der unteren Auswahlliste stehen folgende Beziehungen zur Verfügung:

- für ***Klassen***:
  Oberklassen, Unterklassen, Methoden, für Klienten/Erben/Freunde sichtbare Methoden
- für ***Kapitel***:
  Oberkapitel, Unterkapitel, Schwesterkapitel
- für ***Dateien***:
  inkludierte Dateien, Ersetzungen (d.h. Klassen und Methodenimplementierungen, die in ihnen enthalten sind)

Wird aus einer der Auswahllisten ein Eintrag selektiert, so wird der entsprechende Text im Editor angezeigt, die Informationsbox sowie die untere Auswahlliste aktualisiert.

Der Editor

Der Editor dient zum Bearbeiten und Anzeigen von Quell- und Dokumentationstext. Er bietet die üblichen Editiermöglichkeiten wie cut/copy/paste und unterstützt ein für C++ wichtiges einfaches Zuordnen von öffnenden und schließenden Klammern.

Navigier-Möglichkeiten

Bequemes Navigieren durch eine Unmenge von Informationen ist besonders für den Wartungsprozeß von immenser Wichtigkeit. DOgMA bietet eine Reihe von Möglichkeiten zum Navigieren in Quelltext und Dokumentation.

Einfaches Navigieren

Durch die Auswahllisten lassen sich alle Stellen eines Software systems erreichen. Auf einfache Art und Weise kann man damit durch Klassen, Dateien und Kapiteln navigieren.

Navigieren mit der Informationsbox

Zum Verständnis von Programmstücken sind jene Teile von enormer Wichtigkeit, die mit diesem Programmstück in enger Beziehung stehen. Bei objektorientierten Systemen sind das insbesondere die Oberklassen und die überschriebenen Methoden. In DOgMA wird diese Information sehr bequem durch die Informationsbox angeboten, da sämtliche darin angzeigten Klassenangaben des Vererbungspfades ausgewählt werden können. Dadurch kann man sehr schnell die Vererbungshierarchie entlangwandern und kommt mit einem einfachen Mausklick wieder zum Ausgangspunkt zurück.

Diese Möglichkeit ist insbesondere praktisch, da dieser Mechanismus nicht nur auf Klassen- sondern auch auf Methodenebene eingesetzt werden kann. Dabei sind zusätzlich Klassen, die eine bestimmte Methode implementieren, fett dargestellt. Dadurch ist auf einen Blick ersichtlich, welche Klassen eine bestimmte Methode implementieren, und es genügt wiederum ein einfacher Mausklick um zu einer überschriebenen Methode zu gelangen. In Abbildung 2.3-10 sind die Klassen IdentUseMak und Object fett dargestellt, da sie die Methode DoObserve implementieren.

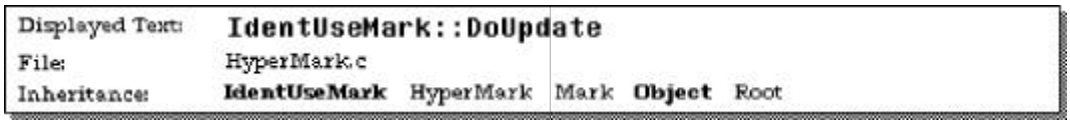

Abb. 2.3-10 Informationsbox

In der (hybriden) Sprache C++ sind nicht nur Klassen und Methoden von besonderem Interesse, sondern auch Dateien, da sie (wie in C) globale Deklarationen enthalten können, die in Klassen und Methoden verwendet werden können. Deshalb ist der soeben beschriebene Mechanismus für Klassen und Methoden auch für Dateien anwendbar. Es genügt die Auswahl des Dateinamens in der Informationsbox, um zu jener Datei zu gelangen, in der die aktuelle Klasse oder Methode enthalten ist.

Navigieren auf Bezeichnerebene

Der Benutzer kann sowohl im Quelltext als auch in der Dokumentation einen beliebigen Bezeichner auswählen und zur Definition desselben verzweigen. Weiters ist es möglich, alle Verwendungen schrittweise zu inspizieren.

Navigiergeschichte

In komplexen Informationsstrukturen besteht die Gefahr, daß sich Benutzer darin verirren. Die Navigiergeschichte ist in solchen Fällen meist hilfreich, um die Orientierung wieder zu finden. Abbildung 2.3-11 zeigt den von DOgMA zur Verfügung gestellten Dialog mit der Navigiergeschichte. Darin werden die letzten besuchten Stellen des Systems angezeigt und können mit einem Mausklick wieder erreicht werden.

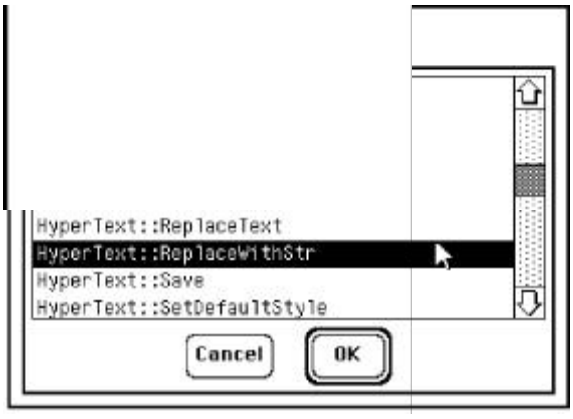

Abb. 2.3-11 Navigiergeschichte

Navigieren in der Dokumentation

Bei Inspektion des Quelltextes wird der Benutzer darüber informiert, ob dazu Dokumentation vorhanden ist, und er kann durch Auswahl eines Menübefehls dorthin verzweigen. Ebenso kann in der Dokumentation von Quelltext-Bezeichnern zu deren Definition verzweigt werden (siehe oben), bzw. von Codeteilen zur entsprechenden Stelle im Quelltext. Weiters können auf einfache Weise die Vorkommnisse eines Quelltext-Bezeichners in der Dokumentation eruiert werden (siehe unten).

Editier-Möglichkeiten

Neben den herkömmlichen Editier-Möglichkeiten (cut/copy/paste) ist es möglich, Klassen, Methoden etc. einzufügen, zu löschen, danach zu suchen, auf bestimmte Zeilen zu positionieren (für Compiler-Fehlermeldungen) etc.

Zum Einfügen von Klassen und Methoden wird von DOgMA eine Textschablone erstellt, die nur mehr auszufüllen ist. Diese Schablonen können beliebig vordefiniert werden und erleichtern so das Erstellen von Quelltext erheblich.

Globale Schriftarten

Um die Lesbarkeit zu erhöhen kann für verschiedene syntaktische Elemente und Teile der Dokumentation eine globale Schriftart definiert werden, und zwar für:

- Schlüsselwörter, Kommentare, Definitionen und Verwendungen von Bezeichnern, Ersetzungen,
- Überschriften in der Dokumentation, Programmbezeichner im Dokumentationstext, nicht mehr vorhandene, in der Dokumentation referenzierte Quelltextteile.

Für in der Dokumentation integrierten Quelltext werden dieselben Schriftarten verwendet wie direkt im Quelltext. Abbildung 2.3-12 zeigt den Dialog zum Einstellen der Schriftarten.

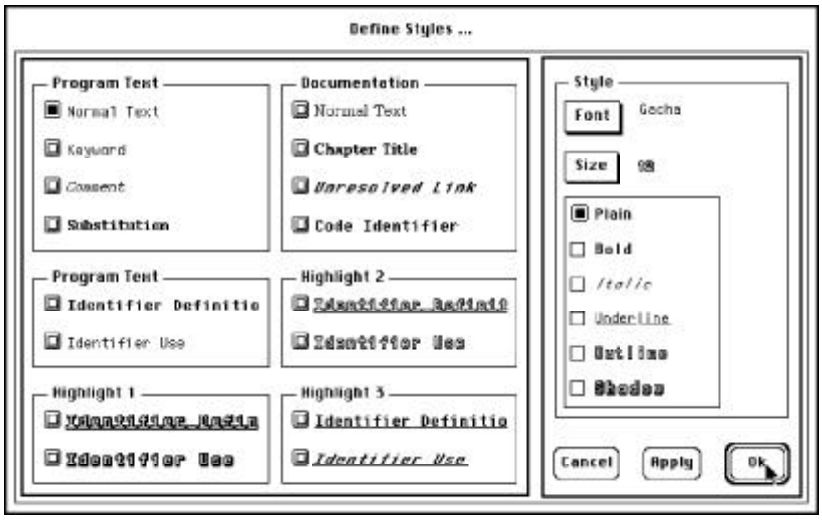

Abb. 2.3-12 Globale Schriftarten

Bestimmte Bezeichner können hervorgehoben werden (*highlight*), damit sie sowohl im Quelltext als auch in der Dokumentation besser zu erkennen sind. Dies erlaubt ein sehr einfaches Auffinden von globalen Variablen, bestimmten Methodenaufrufen, etc., da zusätzlich in den Auswahllisten bei Dateien, Klassen, Methoden und Kapiteln die Anzahl der Vorkommnisse eingeblendet werden (siehe Abb. 2.3-13).

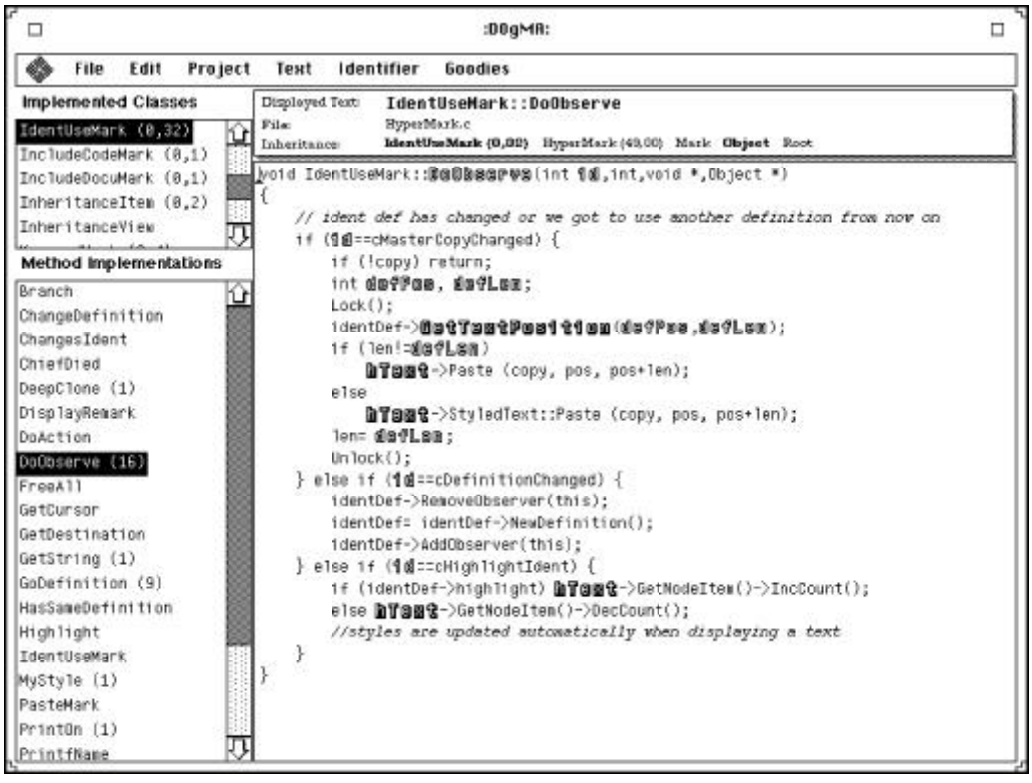

Abb. 2.3-13 Hervorgehobene Bezeichner

Umbenennen von Bezeichnern

Das Umbenennen von Bezeichnern ist mit herkömmlichen Texteditoren kein leichtes Unterfangen. Zunächst ist es nicht immer einfach, alle Vorkommnisse eines Bezeichners zu eruieren, da diese über mehrere Dateien verstreut sein können. Zudem kann es andere Bezeichner in anderen Gültigkeitsbereichen mit gleichem Namen geben.

Konsistenz mit der Dokumentation muß ebenfalls gewährleistet bleiben. Dies ist meist nur mit großem Aufwand möglich, was dazu führt, daß Umbenennungen, die meist die Lesbarkeit eines Softwaresystems erhöhen würden, einfach unterlassen werden.

In DOgMA können Bezeichner sehr einfach umbenannt werden. Man tippt den neuen Namen in eine Dialog-Box und automatisch wird sichergestellt, daß alle Vorkommnisse in Quelltext und Dokumentation geändert werden.

Informationen über Bezeichner

DOgMA liefert für jeden beliebigen Bezeichner eine Reihe nützlicher Informationen (z.B. Ort der Deklaration). Diese Information enthält auch eine Kurzbeschreibung, falls eine solche nach der Deklaration in einem Kommentar steht. Das ist für das Verstehen von fremden Code eine große Hilfe.

Die Annahme, daß in einem Kommentar nach einer Deklaration eine Kurzbeschreibung dieses Bezeichners enthalten ist, trifft in den meisten Fällen zu. Außerdem führt diese Annahme im schlimmsten Fall zur Anzeige eines Kommentars, der nicht für die Beschreibung des Bezeichners gedacht war. Andererseits motiviert diese Tatsache dazu, bei allen Deklarationen konsequent eine kurze Beschreibung anzufügen. So erhält man mit minimalen Aufwand eine gute Unterstützung beim Einarbeiten.

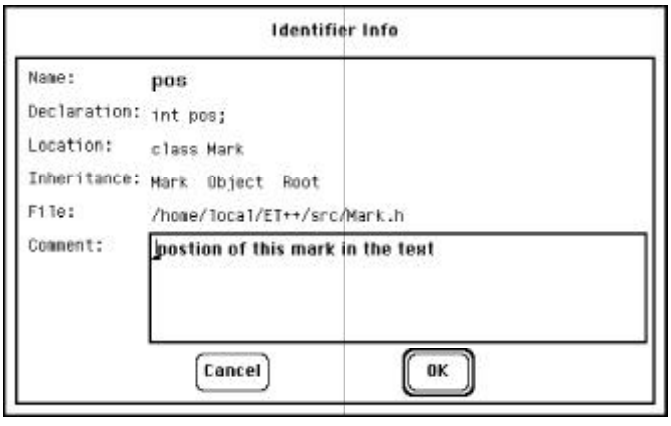

Abb. 2.3-14 Informationen über einen Bezeichner

In Abb. 2.3-14 sehen wir Information über einen Bezeichner *pos*. Wir erfahren, daß *pos* eine Integer-Variable ist, die in der Klasse *Mark* definiert wurde. Weiters ist der Vererbungspfad von *Mark* angegeben, die Datei, in der die Klasse von *Mark* und somit die Deklaration von *pos* enthalten sind, und eine kurze verbale Beschreibung. DOgMA "weiß" natürlich nicht, daß *pos* eine Integer-Variable ist, sondern zeigt lediglich jene Zeile an, in der *pos* definiert wird. In den meisten Fällen steht darin die Information, in die der Benutzer interessiert ist. Selbstverständlich sind diese Informationen über Bezeichner auch in der Dokumentation zugänglich. Diese Vorgehensweise ist in vielen Fällen von Vorteil, da das Schreiben von kurzen Kommentaren bei der Deklaration von Bezeichnern nicht unüblich ist.

Ein-/Ausblenden von Kommentaren

Wenn ein Softwaresystem mit DOgMA entwickelt bzw. dokumentiert wird, dann besteht keine Notwendigkeit, Informationen in umfangreichen Kommentaren in den Quelltext zu plazieren. Vielfach wird dies jedoch getan, da keine Möglichkeit besteht, Verbindungen zischen Quelltext und Dokumentation herzustellen. In solchen Fällen ist

es vorteilhaft, Kommentare ausblenden zu können, damit sie einen bereits mit dem System vertrauten Leser nicht ständig irritieren und behindern. Man kann dabei noch unterscheiden, ob man alle Kommentare, oder nur mehrzeilige ausblenden will. Abbildung 2.3-15 zeigt eine Klasse mit versteckten Kommentaren, deren Vorhandensein allerdings nicht verschwiegen wird.

```
class HyperText: public StyledText {
/**/
protected: //
    class MarkList          *marks;
    class HyperTextView     *tv;
    class HyperProject      *project;
    class HyperMark         *currentMark;      /**/
    class NodeItem          *nodeItem;
    class TextCopyMark      *textCopyMark;     //
    class HyperMark         *firstIdent;       /**/
    bool   showEscape;    //
    bool   textDirty;     //
    int    nrLines;       //
private:
    int             curPos;  //
    char            *title;  //
    bool            alreadyProcessed; /**/
public:
    /**/
    …
};
```

Abb. 2.3-15 Versteckte Kommentare

2.3.4 Erfahrungen

Auf universitärem Boden spielt die Wartungsphase in der Regel eine untergeordnete Rolle. Erste Erfahrungen mit DOgMA resultieren aus dem Einsatz in Forschungs- und Studentenprojekten. Da bei der objektorientierten Programmierung jedoch der Wiederverwendung eine zentrale Rolle zukommt, verschieben sich die Schwerpunkte der Wartung immer mehr in die Entwicklungsphase hinein. Bei der Entwicklung eines Softwaresystems kommt man heute nicht mehr ohne komplexe Klassenbibliotheken und sog. *Application Frameworks* aus, wodurch die Einarbeitung und das Verstehen fremden Quelltextes auch hier eine zentrale Tätigkeit dar stellt.

Die Einarbeitung des von uns häufig verwendeten *Application Frameworks* ET++ (Weinand 1989) konnte mit DOgMA wesentlich erleichtert werden. Das leichte Auffinden jeglicher Information beschleunigt den Prozeß der Einarbeitung enorm. Selbst ohne vorhandene Dokumentation war durch die Vernetzung auf Struktur- und Bezeichner-Ebene eine sichtliche Produktivitätssteigerung zu verzeichnen.

Die Möglichkeit zur Integration der Dokumentation wirkte sich nicht unmittelbar aus, da sich niemand die Mühe machte, vorhandene Beschreibungen mit dem Quelltext zu verweben. Aber wir konnten von seiten der Studenten eine Qualitätssteigerung und bessere Brauchbarkeit der erstellten Dokumentationen verzeichnen.

Solche Kleinigkeiten wie beispielsweise das einfache Umbenennen von Bezeichnern darf man ebenfalls nicht außer Acht lassen. Dies führt zu aussagekräftigeren Namen, da man beim Erstellen von Programmen nicht immer auf Anhieb den richtigen Namen für einen Bezeichner findet und sich außerdem die Bedeutung eines Bezeichners im Laufe der Entwicklung ändern kann. Ohne entsprechende Werkzeugunterstützung unterbleiben solche Namensänderungen leider allzuoft.

2.3.5 Zusammenfassung

DOgMA ist eine für die Programmiersprache C++ entwickelte Programmierumgebung, die insbesondere Tätigkeiten der Wartungsphase unterstützt. Durch die Integration von Hypertext- und Literate Programming-Konzepten konnte eine wesentliche Produktivitätssteigerung beim Einarbeiten und Verstehen von fremden Programmen erzielt werden. Durch den Einsatz DOgMAs schon in der Entwicklungsphase läßt sich weiters eine positive Auswirkung auf Qualität, Konsistenz und Vollständigkeit der Dokumentation erwarten. Eine wesentliche Kostenreduktion in der Wartungsphase, das anfänglich gestecktes Ziel bei der Entwicklung von DOgMA, kann unter Betracht der Bedeutung von Dokumentation und der Tätigkeit des Einarbeitens und Verstehen-Müssens in der Wartungsphase als erreicht betrachtet werden.

2.3.6 Literatur

Bush 1945 — Bush V: As We May Think, Atlantic Monthly, pp. 101-108, July 1945.

Conklin 1987 — Conklin J.: Hypertext: An Introduction and Survey, Computer, Vol. 20, No. 9, pp. 17-41, September 1987.

Digitalk 1989 — Smalltalk/V PM: Tutorial and Programming Handbook, Digitalk Inc., 1989.

Fiderio 1988 — Fiderio J.: Hypertext: A Grand Vision, BYTE, Vol. 13, No. 10, pp. 237-244, October 1988.

Gibson 1989 — Gibson V.R., Senn J.A.: System Structure and Software Maintenance Performance, Communications of the ACM, Vol. 32, No. 3, pp. 347-358, 1989.

Knuth 1984 — Knuth D.E.: Literate Programming, The Computer Journal, Vol. 27, No. 2, pp. 97-111, 1984.

Knuth 1986a — Knuth D.E.: Computers and Typesetting, Volume B, TEX: The Program, Addison-Wesley, Reading, MA, 1986.

Knuth 1986b — Knuth D.E.: Computers and Typesetting, Volume D, METAFONT: The Program, Addison-Wesley, Reading, MA, 1986.

Knuth 1992 — Knuth D.E.: Literate Programming, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

LaLonde 1990 — LaLonde W.R., Pugh J.R.: Inside Smalltalk, Volume 1, Prentice Hall, Inc., 1990.

Nielsen 1990 — Nielsen J.: The Art of Navigating through Hypertext, Communications of the ACM, Vol. 33, No. 3, pp. 296-310, March 1990.

Sametinger 1990 — Sametinger J.: A Tool for the Maintenance of C++ Programs, Proceedings of the Conference on Software Maintenance, San Diego, CA, pp. 54-59, 1990.

Sametinger 1992a Sametinger J., Pomberger G.: A Hypertext System for Literate C++ Programming, Journal of Object-Oriented Programming, Vol. 4, No. 8, pp. 24-29, January 1992.

Sametinger 1992b Sametinger J.: DOgMA: A Tool for the Documentation and Maintenance of Software Systems, VWGÖ, Vienna 1992.

Shneiderman 1989 Shneiderman B., Kearsley G.: Hypertext Hands-on: An Introduction to a New Way of Organizing and Accessing Information, Addison-Wesley, Reading, MA, 1989.

Smith 1988 Smith J.B., Weiss S.F.: Hypertext, Communications of the ACM, Vol. 31, No. 7, pp. 816-819, July 1988.

Stroustrup 1991 Stroustrup B.: The C++ Programming Language (Second Edition), Addison-Wesley, 1991.

Weinand 1989 Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming, Vol. 10, No. 2, 1989.