DOgMA:
A TOOL FOR THE DOCUMENTATION & MAINTENANCE OF SOFTWARE SYSTEMS

Johannes Sametinger

TECHNICAL REPORT

JUNE 1991

Institut für Wirtschaftsinformatik

Johannes Kepler University Linz

Austria

Abstract

It is very often necessary to correct faults in software systems, make them usable in changed environments, or improve their quality (e.g., performance). This activity is called software maintenance and is usually done by people other than those who developed a software system. Software maintenance requires detailed documentation and the possibility to attain an overview of the system structure, the interaction of the various components, and the effects of changes.

This report describes the results of a research project in the area of documentation and maintenance whose main focus was to combine state-of-the-art methodologies in a new tool (DOgMA) to improve the support of documentation and maintenance activities. Additional goals were to identify major problems in the field, to determine possible improvements, to investigate how these improvements are supported by tools, to summarize deficiencies of existing tools, and to ameliorate these deficiencies in the design of a new tool.

After a short discussion of the role of documentation and maintenance in the software life cycle, a detailed compilation and definition of terms used is given. Subsequently an outline of the state of the art is described and evaluated. Based on these perceptions, requirements on a documentation and maintenance tool were developed along with concepts for their solution. The resulting tool, DOgMA, serves as a contribution to the improvement of the documentation and maintenance process. A comparison with similar tools and an examination to what extent the goals have been achieved concludes the report.

Table of Contents

1. Introduction

1.1 The Role of Documentation and Maintenance in the Software Life Cycle

In the early days of computing, good programmers strived to produce programs that were concise, tricky and efficient. This goal was justified by sluggishness and limited storage of computers at that time. Needless to say, it was hardly possible for someone else to read and maintain these programs.

Requirements on software increased as computers became faster and were equipped with more storage. Thus software research work concentrated mainly on the areas of software design and development to deal with the increasing requirements. The task of maintaining all the software was considered a minor problem. But after only a few decades we have come to realize that half the software personnel is engaged in software maintenance rather than software development.

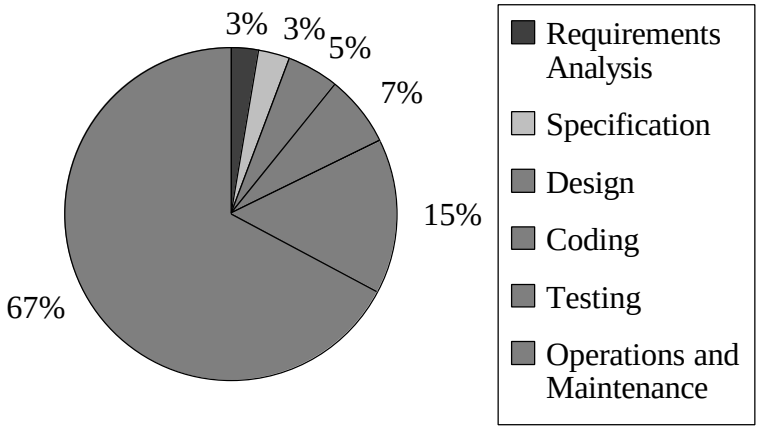

Fig. 1.1 Relative costs in the phases of the software life cycle [Art88]

Software maintenance generally means any work on an existing software system. It cannot be avoided, it is intrinsic to software, and it proves to be the most difficult as well as the most expensive part of the software life cycle.

Many statistics emphasize the important role of maintenance in the software life cycle. Figure 1.1 shows the relative costs in the traditional software life cycle ([Mar83], [Art88]), where the biggest part is taken by operations and maintenance.

Typically, the development phase requires 1 or 2 years, whereas the maintenance phase spans over 5 to 10 years [Fai85]. Additionally, the costs of software maintenance have increased steadily in the past (see Fig. 1.2).

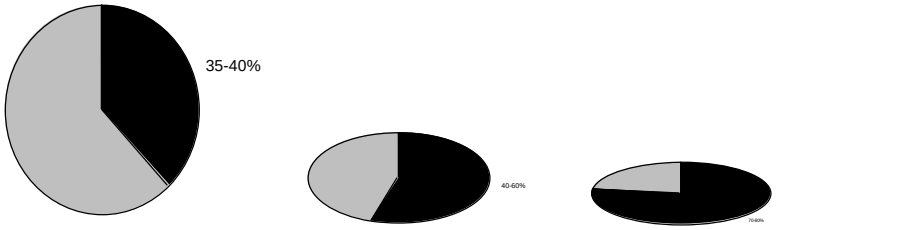

Fig. 1.2 Percentage of software budget spent for maintenance [Pre87]

The message in the statistics is clear, yet maintenance is (negatively) viewed as being [Hal87]:

- difficult
- unfair (due to the lack of needed information)
- a dead-end job (no progress that can be seen)
- a task that is not at the cutting edge of technology

It is no surprise that 90 percent of computer science students would like to be involved in development, while only 10 percent have an interest in maintenance [Per86]. Some of the problems of maintenance are ([Cou85], [Cha86a]):

- poor documentation
- bad system structure
- high complexity
- use of old languages (e.g., Cobol)
- unavailability of the developers (in case questions arise)
- time pressure (imposed by users)

There is no hope that the portion of old code will ever decrease. The following considerations emphasize this fact [Sch87]:

- Functions are added rather than replaced.
- Every new function must be tied into the existing system.
- The organizations' goal is compatibility, not perfection.

Nevertheless, several facts emphasize that software maintenance is attracting more attention from the research community:

- Up to now six Conferences on Software Maintenance have been held (1983 in Monterey, CA, 1985 in Washington, D.C., 1897 in Austin, TX, 1988 in Phoenix, AZ, 1989 in Miami, FL, and 1990 in San Diego, CA, 1991 to be held in Italy).

- The Software Maintenance Association (SMA) was organized as a special interest group in the field of software maintenance. Since 1983 it publishes a monthly maintenance newsletter, the Software Maintenance News, which is usually based on real experience.
- The Annual Meeting and Conference of the Software Maintenance Association is held each year in a different city in North America. It is a gathering of professionals in the field of software maintenance.
- A new Journal of Software Maintenance: Research and Practice has been published since September 1989. The main function of this journal is to publish original work and high quality surveys in the field of software maintenance.
- The Journal of Software Testing, Verification and Reliability began publication in April 1991.
- Special sections on software maintenance are being published in journals, for example, IEEE Transactions on Software Engineering in March 1987, IEEE Software in January 1990.
- A column on Literate Programming was first published in the journal Communications of the ACM in July 1987. The idea of literate programming plays a major role in the documentation and maintenance of software systems. The column has appeared several times since then.
- There are also many conferences which deal with special topics in the field of software maintenance, e.g., International Conference on Software Testing, International Testing Methods Conference, International Conference on Software Maintenance & Reengineering.
- Many workshops and seminars are offered all over the world on software maintenance, configuration management, software testing, software quality management, software project management, software reengineering, reverse engineering, etc.
- An increasing number of papers in the software maintenance area have been appearing in various journals and books.

Development is optional; maintenance is mandatory. Even people whose priority is software development must be interested in reducing maintenance because this implies more resources available for development. Maintenance hardly produces any profit, but it is essential to preserving profits.

1.2 Scope

Some of the major problems with software maintenance are (see [Cha86a]):

- inadequacy of documentation
- poor structure of source code

- high complexity of software systems

This leads to a situation where maintenance programmers spend half of their time on understanding programs (see Fig. 1.3, see also [Gib89]).

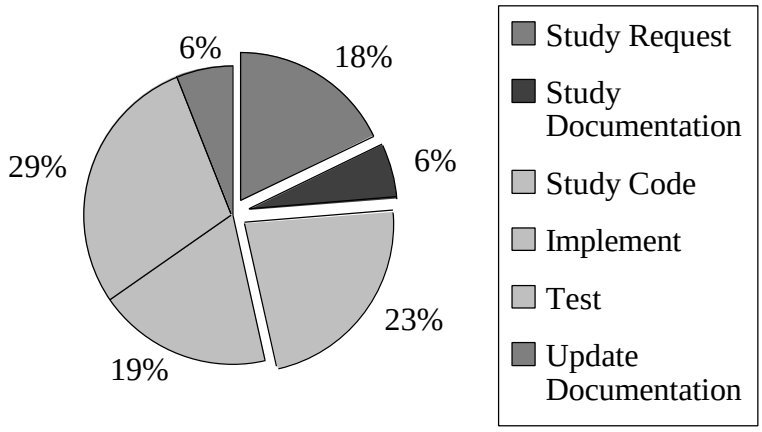

Fig. 1.3 Maintenance personnel activities [Par83]

Insufficient information about a given program can lead the maintenance programmers to erroneous assumptions about the program. In such a case, any modification of the program is very likely to introduce new errors. Therefore tools are needed that provide the proper information about a software system in an appropriate and convenient way.

This report contains the results of a research project of which the main focus was:

- to identify the main problems in the field of documentation and maintenance
- to determine possible improvements
- to investigate how these improvements are supported by tools
- to summarize the deficiencies of existing tools
- to develop a new methodology and a corresponding tool in order to diminish these deficiencies

The presented tool is aimed at providing improvements in the support of documentation and maintenance. It supports understanding both source code and documentation and mastering the complexity of large systems. The name DOgMA is an abbreviation for **DO**cumentation **& MA**intenance, whereby the ‘g’ was chosen because of its similarity with ‘&’.

Several kinds of documentation are differentiated (see Chapter 2). We will mainly concentrate on system documentation, which plays the major role in software maintenance. Creating, studying and updating this documentation are supported.

The software systems being considered are systems written in an algorithmic language with either a conventional or an object-oriented implementation. Poor structures of source code can be found in both conventional and object-oriented software systems. Complexity tends even to be higher in object-oriented systems. Emphasis is laid on

mastering complexity, so the presented tool especially supports the maintenance of object-oriented software systems, too. Understanding of software is one of the keys to making correct modifications.

DOgMA, a software tool for the documentation and maintenance of software systems, has the following characteristics:

- DOgMA was especially designed to support documentation and maintenance. However, its use is in no way limited to that process. It can be used as a software development tool as well.
- Special emphasis was laid on the process of program comprehension as this is the most time-consuming task in software maintenance.
- In DOgMA the concepts of browsing, hypertext and literate programming are amalgamated. This combination offers a new perspective in the field of maintenance.
- Global text styles can be defined for keywords, identifiers, comments, etc. This results in a considerable increase of readability in source code and documentation.
- The possibility to highlight any identifiers makes it easy to locate the occurrences of class, method or variable names both in the source code and the documentation.
- Automatic renaming of identifiers in a certain scope is provided for both the source code and the documentation.
- The possibility to get useful information about identifiers at any location, e.g., declaration, location of the declaration, inheritance path and even verbal descriptions (if available), provides a major improvement in the understanding of software systems.
- DOgMA can be used for the processing of any software systems, even if they were created with another tool.
- DOgMA uses textual external representations of source code and documentation. This enables programmers to process documents created with DOgMA with other tools as well.
- DOgMA was implemented on a SUN workstation under UNIX with a modern and comfortable user interface providing windows, menus, etc. The programming languages C++ and Modula-2 are supported.
- The implementation was made in an object-oriented manner using the programming language C++ and the application framework ET++.
- In the design a strict separation in a language-dependent and a language-independent part was pursued. This allows the adaptation of DOgMA to other programming languages more easily.

The aim in developing DOgMA was to demonstrate possible improvements in the field of software documentation and maintenance. This development did not result in a finished product, but rather emphasis was laid on the creation of a new methodology.

1.3 Survey

The report is subdivided into five parts.

Chapter 2 contains the definitions of terms used in the field of software maintenance and therefore also used in this report.

Chapter 3 gives an overview of the state of the art, describes major problems as well as promising improvements and tools support in the field of software maintenance, and evaluates the current situation.

Chapter 4 is dedicated to the new tool DOgMA, which supports the documentation and maintenance of software systems. The underlying concepts, the user interface, its use (sample scenarios), and some implementation details are presented.

Chapter 5 classifies software documentation and maintenance tools, describes tools similar to DOgMA, and compares them to our tool.

Chapter 6 draws conclusions and presents prospects for the future.

2. Terminology

This chapter explains terms from the field of documentation and maintenance of software systems that are used in this report.

Software maintenance consists of different activities than its counterpart on the hardware platform. The general meaning of maintenance is to keep something functioning according to original specifications. Software maintenance, on the other hand, is generally regarded as any work done on existing software systems (see also [Par86c]).

Strictly speaking, the term software maintenance is even a misnomer. As E.W. Dijkstra noted in a privately published newsletter in 1983: "A program is not subject to wear and tear and requires no maintenance." In [Par86a] we find: "Maintenance means to keep an item in working order after it has already been started and is working correctly, that is, as per the original design." Software is different, because it does not deteriorate and it often does not work correctly from the very start. Besides, maintenance activities like testing, debugging, enhancing a system and changing it due to environmental changes are not real maintenance in the hardware sense.

In [Web83] we find the following definition:

Maintenance

1) a maintaining or being maintained; upkeep, support, defense, etc.
2) means of support or sustenance; livelihood; as, her job provided a mere maintenance.
3) in common law, support or assistance that a person is legally bound to give to another or other.
4) in criminal law, the act of interfering unlawfully in a suit between others by helping either party, as by giving money, etc., to carry it on.

Yet the term software maintenance has been in use over decades. Its proper meaning in this context can be found in a more specific definition in [ANS83]:

Software Maintenance

Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.

Thus by software maintenance we mean the activity of modifying a software product after its delivery. Modifications may include error corrections, enhancements, improvements, etc. A more detailed definition can be found in [Par86a]:

> Software maintenance is the work done on a software system after it becomes operational. It includes: understanding and documenting existing systems; extending existing functions; adding new functions; finding and correcting bugs; answering questions for users and operations staff; training new systems staff; rewriting, restructuring, converting and purging software; managing the software of an operational system; and many other activities that go into running a successful software system.

Based on this knowledge, we can say that software *development* incorporates modifications made on a new system, whereas software *maintenance* means modifications made on an old system. More generally we can even say that maintenance is any work done on old software system. In this sense maintenance is nothing else but continued software development (sometimes also called “dynamic development”). The line drawn between development and maintenance at the time of delivery is somewhat arbitrary. However, there are subtle differences between development and maintenance, or between initial development and continued development.

The fact that the software has been delivered already burdens the act of modifying it because (see [Art88], [Cha86b], [Lef87]):

- Changing the software must not change the way users work.
- Each change requires the consideration of its overall impact.
- Requirements for corrections and enhancements may change (depending on the users).
- Corrections must be made in time to satisfy the users.
- Improvements in the design might become necessary.
- The development team is often replaced with a maintenance team.
- Changes must be compatible with the existing architecture and design.

McClure identifies four main functions that are involved in software maintenance [Clu81]:

- understanding the existing software
- identifying the modification objective and the modification approach
- implementing the modification
- revalidating the modified software

Maintenance is usually divided into three categories:

- corrective maintenance
- adaptive maintenance
- perfective maintenance

This classification was first published in [Swa76] and is defined in [ANS83] as follows:

Corrective Maintenance

maintenance performed specifically to overcome existing faults

Adaptive Maintenance

maintenance performed to make a software product usable in a changed environment

Perfective Maintenance

maintenance performed to improve performance, maintainability, or other software attributes

Swanson's three categories have become classical, but other classifications have been proposed, too. For example, Reutter introduced seven categories [Reu81]:

- emergency repairs (to immediately repair errors on user request)
- corrective coding (to repair errors in order to meet the specification)
- upgrades (to adapt to changes in the processing environment)
- changes in conditions (to adapt to changes in business conditions)
- growth (to adapt to changes in data requirements)
- enhancements (to make additions in response to user requests)
- support (to plan for future features, to measure performance, etc.)

Emergency repairs and corrective coding can be assigned to corrective maintenance, upgrades and changes in conditions correspond to adaptive maintenance, and enhancements belong to perfective maintenance. Growth can be assigned to both adaptive and perfective maintenance, whereas the new category of supportive maintenance is considered by Swanson to be part of all three of his categories (see [Mar83]).

Lin and Gustafson differentiate six categories: corrective, adaptive, retrenchment, retrieving, prettyprinting, and documentation activities [Lin88].

Other categories have been added to Swanson's classification as well, but they never really succeeded like his three classical categories. For example in [Bri87] we find:

Prevented Maintenance

Prevented maintenance is maintenance that you don't have to do because the system was designed not to need it.

According to [Bri87], prevented maintenance can be achieved by a flexible design, quality assurance, extensive testing, and system change forecasts by asking people who have knowledge of future requirements (e.g., the users). It is not possible to eliminate maintenance, but actions can be done during development to reduce it.

The only kind of software the author is aware of that does not need any maintenance is video games. It is not necessary to make any modifications on these games because they are simply replaced with new games. But in this case maintenance is not prevented through careful engineering, but rather ignored by throwing away the old systems after delivery.

In [Pre87] we find the term

Preventive Maintenance

> Rather than waiting until maintenance requests are received, the software is partly or completely redesigned, recoded and retested.

This approach was pioneered by Miller [Mil79], who used the term *structured retrofit*. At a first glance preventive maintenance seems quite costly, but one should consider the following facts [Pre87]:

- Redesigning software using modern concepts will greatly facilitate future maintenance (and thus reduce software life cycle costs).
- Development costs are cut back because a running prototype already exists.
- Due to user experience, new requirements can be taken into consideration.

In [Osb87] preventive maintenance is referred to as *cleaning up code*.

Another category can be found in [Par87a]:

Miscellaneous Maintenance

> The corrective, adaptive and perfective maintenance implies changes to a software system. What if a programmer studied a program, but did not make any changes? Such work can be termed as miscellaneous maintenance activities.

Hall introduces the term *functional maintenance* (also called change maintenance) [Hal87]:

Functional Maintenance

> Functional maintenance changes the function of the software by adding or deleting features.

Another important term in connection with maintenance is maintainability. In [ANS83] it is defined as follows:

Maintainability

1) The ease with which software can be maintained.
2) The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.
3) Ability of an item under stated conditions of use to be retained in, or restored to, within a given period of time, a specified state in which it can perform its required functions when maintenance is performed under stated conditions and while using prescribed procedures and resources.

Maintainability can also be expressed in terms of reliability, modifiability, testability, understandability, portability, etc.

At least as important as a good design is good documentation to ease software maintenance or even to make feasible. Documentation is so important for software maintenance because systems are getting more and more complex, so that it becomes

impossible to understand a system within a reasonable amount of time with only the source code available. In [ANS83] the following definitions are given:

Documentation

1) A collection of documents on a given subject.
2) The management of documents which may include the actions of identifying, acquiring, processing, storing, and disseminating them.
3) The process of generating a document.
4) Any written or pictorial information describing, defining, specifying, reporting or certifying activities, requirements, procedures, or results.

Software Documentation

Technical data or information, including computer listings and printouts, in human-readable form, that describe or specify the design or details, explain the capabilities, or provide operating instructions for using the software to obtain desired results from a software system.

We need to distinguish between user documentation and system documentation. For software maintenance we primarily need system documentation, so whenever we use the term documentation in the following chapters, we actually mean system documentation. In [ANS83] these terms are defined as follows:

System Documentation

Documentation conveying the requirements, design philosophy, design details, capabilities, limitations, and other characteristics of a system. Contrast with user documentation.

User Documentation

Documentation conveying to the end user of a system, instructions for using the system to obtain desired results; for example, a user's manual. Contrast with system documentation.

Unfortunately, system documentation very often falls into at least one of the following categories [Liu78]: no documentation, insufficient documentation, or misleading documentation.

Martin distinguishes four kinds of documentation [Mar83]:

- user documentation (instructions on how to use a program)
- operations documentation (used to direct the execution of programs)
- program documentation (used to understand the program, i.e., system documentation)
- data documentation (description of the data components, i.e., part of the system documentation)

Another category can be found in [Mar83] and [Gla81]:

Historic Documentation

The historic documentation contains information on how a software system has evolved during the development and maintenance phases.

This information can be of great help during maintenance in order to understand design decisions.

System documentation can be further divided into internal and external documentation [Mar83]:

Internal/External Documentation

Internal documentation is embedded in the source code (comments and compiler generated information, e.g., a cross reference listing), whereas external documentation is separate from the source code.

Pomberger differentiates among user, system and project documentation. The latter is a form of historic documentation [Pom86]:

Project Documentation

The project documentation contains information for project management and for project control (e.g., a project plan, an organization plan, the definition of project standards).

Other terms that are frequently used in connection with software maintenance are *reverse engineering*, *software reengineering* and *restructuring*. These terms are not defined in [ANS83], but in [Chi90] we can find the following definitions:

Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction (in contrast to forward engineering, which is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to a physical implementation of the system).

Software Reengineering

Software reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. It encompasses the decomposition of the original source code (reverse engineering) followed by a re-implementation to form a new source code (forward engineering).

Restructuring

Restructuring is the transformation from one representation form to another at the same relative abstraction level. The subject system's external behavior is left unchanged in both functionality and semantics.

In the following chapters we use the term *identifier* very often. Therefore we give its definition derived from [ANS83], too:

Identifier

1) A symbol used to name, indicate, or locate. Identifiers may be associated with such things as data structures, data items, program locations, modules or classes.
2) A character or group of characters used to identify or name an item of data and possibly to indicate certain properties of that data.

The term software tool is defined in [ANS83] as follows:

Software Tool

A computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, automated design tool, compiler, test tool, maintenance tool.

The software tool DOgMA can be assigned to the categories development tool, documentation tool and maintenance tool.

3. State of the Art

This chapter represents an investigation of the state of the art in the field of software documentation and maintenance. Section 3.1 describes major problems and Section 3.2 suggests possible improvements. Finally, in Section 3.3 the current situation is evaluated by comparing the problems with the existing improvements, thus establishing where more research work is needed.

3.1 Existing Problems

Making an attempt to improve documentation and maintenance raises the question "What is wrong with it?" This section describes problem areas in this field. There is no claim to completeness, but rather it is intended to provide the reader with a feeling about the difficulties in the documentation and maintenance area.

3.1.1 Poor Code

Existing code is often very complex, unstructured, inconsistently indented, and difficult to understand. It does not follow any standards and contains cryptic names and misleading comments. (In fact, there is often no corresponding documentation available. This holds especially for old code, e.g., the billions of lines of old Cobol code.) This is one of the main reasons why maintenance programmers spend so much time on understanding existing systems.

Changing these systems usually makes things even worse because changes hardly ever improve the structure of a software system.

3.1.2 High Complexity

Software systems have become increasingly large and complex. Requirements on software systems have steadily increased, even though the level of abstraction increased with the evolution of high-level programming languages. So we now face more and more systems whose very complexity inhibits understanding.

The promising paradigm of object-oriented programming makes software more reliable, reusable and extensible. One can reuse existing code by creating subclasses. Thus the behavior of existing classes can easily be changed by adding and overriding states and methods. One has to understand the structure of the classes, the inheritance mechanism,

the behavior of the classes, the interaction among the objects, etc. in order to understand object-oriented systems. This is increasingly necessary even for developers who reuse existing classes that they did not design, e.g., from application frameworks. Typically, object-oriented applications include the definition of hundreds of classes and methods.

The behavior of object-oriented systems is more dynamic and complex than that of module-oriented systems, which makes the comprehension of these systems more difficult.

3.1.3 Inadequate Documentation

Very often the only information a maintenance programmer can trust is the source code. It is the only accurate, complete and up-to-date representation of the program. However, source code listings are hardly suited to representing design decisions, the global system structure, or the interactions among different system components. System documentation is necessary to enable the maintenance crew to understand a system. It should remain valid as long as the software system is being maintained in order to make the maintenance process feasible.

Nevertheless, system documentation is often inadequate (not meeting the needs of the maintenance personnel) and out of date, and therefore unreliable and misleading. There are several reasons for this situation:

- It is often easier to find a solution than to depict the idea behind it.
- Programmers hate to write documentation, even though they appreciate reading it in order to understand a program.
- The separation of documentation text and source code promotes inconsistency. The corresponding documentation is left unchanged especially when changes (e.g., bug fixes) are made to the source code.
- The production of system documentation has lower priority because users do not need it and therefore it does not have to be delivered.

Good (system) documentation should be concise, complete, current, correct and consistent in style [Gla81].

Another important aspect is the representation of documentation which is usually written as text. But text is often not the proper medium, especially when dynamic aspects have to be described (see also Chapter 3.1.8).

3.1.4 Team Coordination

The following questions may arise when a software product is maintained by a team (see [Bab86], [Tic88]):

- This worked yesterday. What happened?
- Who is responsible for this modification?
- Why did my change to this module disappear?
- Why does the listing not match the code image?
- What happened to the fix I put in yesterday?

Both developing and maintaining programmers struggle to understand what state of progress a program represents. Valuable time is spent trying to find versions of a program, and to detect differences between multiple copies of a program. The following problems are typical [Bab86]:

- *The shared data problem*

  If programmers modify a single copy of a program, then modifications of one programmer can interfere with the modifications of others.

- *The simultaneous update problem*

  If two programmers simultaneously update the same program, then possibly the updates of one of them will be overwritten. This causes the reintroduction of bugs that had already been fixed.

3.1.5 Multiple Maintenance

The multiple maintenance problem (called double maintenance in [Bab86]) arises if multiple copies of the same software exist. Typical questions run as follows:

- Was that bug fixed in this copy, too?
- How many copies exist and where are they?

Bugs that are found in a single copy need to be fixed in all existing copies.

3.1.6 Static Description of Dynamic Behavior

Programs are presented to humans not only linearly but also statically. This is true in spite of the fact that software systems behave highly dynamically. And it is exactly this dynamic behavior that is crucial for program comprehension. But deducing this information from the pure, static source code of a system is too cumbersome for an effective software maintenance process. Documentation can help in this respect, but it, too, is presented in a static form.

3.1.7 Linear Presentation of Complex Structures

Software systems contain complex structures, e.g., data structures and class structures, with lots of interrelations like inheritance in object-oriented systems. In spite of this fact the source code is usually stored in flat text files and processed with ordinary text editors. Programmers are obliged to laboriously get complex structures and interrelations out of the various places in the source code files in order to understand a software system, i.e., to get familiar with it.

The human visual system is optimized for multidimensional data, but software systems are presented in one-dimensional textual form. This leaves much of the human brain unused [Mye86].

3.1.8 Object Code as Black Box

When team coordination and/or multiple maintenance go wrong, it often takes hours of dump reading to find out the reason for unexpected behavior of a software system. The difficulty lies primarily in the fact that it is hardly possible to get any information about an executable object code.

If, for example, something went wrong during the link process (perhaps linking wrong or old versions of some units), it becomes extremely difficult to find out this fact because the resulting object code is something like a black box.

The following information is crucial:

- Which versions went into this object-code?
- When were they compiled and which compiler options were used?
- Which system libraries were linked and which link options were used?

Unfortunately, this information is hardly ever available unless the whole linking and/or even compiling process is repeated. However, even this is not always possible because usually multiple versions are delivered and any client with an arbitrary version might report an error.

3.2 Possible Improvements

The previous sections have shown that software development and especially software maintenance are difficult and large-scale tasks. This section presents an overview of approaches that have delivered or promise improvements in documentation and maintenance of software systems. We differentiate between measures that help a priori in reducing the maintenance effort (Section 3.2.1) and measures that help mastering and improving the maintenance process (Section 3.2.2). We will also try to give an overview of tool support. Again, there is no claim to completeness, but rather the attempt is made to show various directions of improvement.

3.2.1 Preventing Maintenance

The pipe dream of any software engineer is maintenance that does not have to be done because the system was designed not to need it (see Chapter 2). Although it is impossible to totally eliminate the maintenance process, there are measures that can be taken in the earlier phases of the software life cycle in order to reduce the maintenance effort later on. Examples are *structured programming*, *prototyping*, the use of *fourth generation languages*, and *spare parts maintenance*, which are presented in the following subsections. Tools that help in preventing or reducing maintenance are not

regarded as maintenance tools because they are used especially in early life cycle phases. Therefore we refrain from looking at such tools in this subsection.

3.2.1.1 Structured Programming

The discussion on structured programming started in 1965 with Dijkstra's paper *Programming Considered as a Human Activity*, which contained the notion of top-down design, the emphasis on quality and correctness of programs, and arguments against goto statements [Dij65]. Dijkstra observed that computers were becoming faster and more powerful, whereas the capacity of human brains remained limited.

A matter of course are the characteristic terms of structured programming like top-down design, modular programming, and structured coding. But it took years until these techniques became widely accepted. And it was also a major step forward for software maintenance because structured programs are easier to understand and thus more reliable and maintainable. Maintaining today's complex software systems would be unimaginable if, for example, their program code would modify its own instructions, which was not unusual in the 'unstructured' sixties.

Historical papers on the subject of structured programming by renowned computer scientists like Dijkstra, Knuth and Wirth are collected in [You79].

3.2.1.2 Prototyping

Prototyping allows users to get experience during the development phase, to show the impact of the requirements, and to reduce the development risks (see [Hol84]). Early hands-on experience on a working prototype influences the maintenance phase as well. User enhancements account for almost half of all maintenance work (see [Mar83]). They can be reduced considerably when users have the possibility to explore a software system's functionality and user interface before it is developed. In this sense, prototyping is a means for preventing maintenance (see Chapter 2).

Prototyping is useful not only for specifying the requirements to a system, but also to support the implementation by exploring the structure of the system (e.g., modularization, class hierarchy). This leads to better implementations, which is beneficial in the maintenance phase. Another prototyping approach is repeatedly applying a redesign, reimplementation and reevaluation process, thus developing a prototype to the final product (see also [Bud84], [Pom91]). Again, maintenance benefits from a better implementation.

3.2.1.3 Fourth Generation Languages

Machine languages were the first generation of computer languages. The second generation consisted of assembler languages. High-level algorithmic languages like Fortran and Cobol introduced the third generation. Even modern programming languages like object-oriented ones are still considered to belong to the third generation.

The idea of fourth generation languages was to raise the abstraction level and to give more power to the user. While languages from the first to the third generation are to be used by (professional) programmers, fourth generation languages are intended to be used by nonprogrammers as well. In these languages it usually suffices to specify what is to be done and the system knows how to do it (e.g., by means of report generators or query languages). This offers end users the opportunity to develop and maintain systems of their own.

The positive effects of using fourth generation languages on software maintenance are [Mar83]:

- Programs can be both created and changed faster and more easily.
- It becomes easier to understand another person's applications.
- End users can make their (simple) changes and enhancements themselves.

However, the maintenance process will not disappear; it will even suffer as a result of high complexity and insufficient documentation. (Fourth generation languages are hardly self-documenting as their vendors suggest, [Tin85].) Besides, the fact that end-dusers are not trained in producing maintainable software (ignoring documentation standards, system structures, etc.)—probably they do not even have any motivation to do that—burdens the process of modifying this software subsequently [Par85].

3.2.1.4 Spare Parts Maintenance

In hardware maintenance the removal of defective parts and their replacement with spare parts is routine. This could be applied to software maintenance, too. When requests for corrective maintenance are received, the erroneous parts could be replaced by spare parts [Pre87]. The spare part strategy seems unconventional for software, but, considering all costs during the software life cycle, it might be an advantageous alternative. However, there is little experience in this strategy. Besides, separate programming teams tend to make the same or similar mistakes when using the same specification [Pre87].

3.2.2 Improving Maintenance

In the following subsections we present examples of possible improvements in the process of software maintenance. Each section concludes with a discussion of tool support as this is a crucial point in the acceptance and even in the application of the presented remedial measures.

3.2.2.1 Hypertext

Usually text files on a computer are flat; i.e., they are organized in a linear way. This linear organization is not adequate in many applications. For example, the documentation of a software system should be interleaved with the source code and there are a lot of possible paths to read the available information, depending on the interests of the reader.

Hypertext enables nonsequential writing and reading. It consists of a set of nodes where each node contains some amount of information (some text, a picture, or even a video sequence). These nodes are connected by links and form a directed graph.

Navigating through a hypertext system means following these links. As each node can have several outgoing links, there are many possible sequences in which to inspect the nodes of such a network. This makes the user feel that she/he can move freely through the available information (see [Fid88], [Smi88], [Nie90]).

Vannevar Bush was the first to describe the ideas of hypertext [Bus45]. Although at that time there were no adequate computers available, Bush had a vision of organizing information similar to the human mind, which operates by association. He introduced (although never implemented) memex, a machine for browsing and making links and notes, an on-line text and retrieval system.

About 20 years later Bush's ideas influenced the work of Douglas Engelbart, who then developed NLS (oN Line System). NLS was an experimental tool for storing specifications, plans, designs, programs and documentation and doing planning, designing, debugging, etc. [Eng63]. More and more hypertext systems have emerged with the evolution of cheaper and more powerful computers (see [Con87], [Shn89]).

The fact that special issues and sections about hypertext have been published in the BYTE journal (October 1988) and the Communications of the ACM (July 1988 and March 1990) emphasize the interest of the research community in this field. Books are available on this subject, too (e.g., [Bar88], [Shn89]). Even special conferences are held on the subject of hypertext.

Nonsequential writing is extremely useful for software documentation. Hypertext tools are the new generation of documentation tools (for both user and system documentation). The concepts of hypertext can also be applied to source code and—even more important—it can be used for the integration of source code and documentation.

Some familiar hypertext systems are Neptune, NLS/Augment, NoteCards, Document Examiner, HyperCard, Guide and Xanadu. In [Shn89] an overview of existing hypertext systems is given along with short descriptions. Conklin provides an extensive list of hypertext systems and their features in a clearly arranged table [Con87].

Possible applications of hypertext systems include

- dictionaries
- encyclopedias
- product catalogs
- technical documentation
- help systems
- software engineering tools

Dynamic Design is an example of a software engineering tool developed for the C programming language using the Neptune hypertext system (see [Big87]). A simple hypertext editor for the processing of source code can be found in [Mös90].

Hypertext technology seems to be a suitable approach for organizing both documentation and source code, thus giving users better access to needed information.

3.2.2.2 Comfortable Browsing

A browser presents a hierarchical index to information, where the index is an aid for the user to quickly obtain needed information. Typically browsers are used for the inspection of file systems (the index providing directories and files) or program systems (the index providing the names of modules/classes and procedures/methods).

Comfortable browsing is an important aid in the program comprehension process. For program understanding it is especially important to obtain needed information as fast as possible.

Actually, browsing systems can be regarded as limited hypertext systems because they allow nonsequential reading of information. For example, in a program browser classes and methods (or modules and procedures) represent the nodes. They are linked with certain relations among them, e.g., the methods or subclasses of a class. Usually in browsers the nodes are displayed in one or more lists, where they can be selected for inspection. Depending on a selection, the contents of some other lists may change, which helps to master the complexity. However, a hypertext system does not necessarily use lists to permit following the links.

Examples of comfortable browsing tools are the various Smalltalk browsers (see [Dig89], [Lal90]).

3.2.2.3 Literate Programming

Programs are written to be executed by computers rather than to be read by humans. Ideally, it should be the other way round. When writing programs, we should not try to instruct the computer what to do, but rather we should try to tell humans what we want the computer to do [Knu84].

The idea of literate programming is to make programs as readable as ordinary literature. The primary goal is not to get an executable program but to get a description of a problem and its solution (including assumptions, alternative solutions, design decisions, etc.). The idea has been demonstrated on relatively small as well as on large programs (e.g., [Knu86a], [Knu86b]).

After the idea of literate programming was published by Knuth in 1984 [Knu84], *Programming Pearls* columns about literate programming were presented in the *Communications of the ACM* in 1986 ([Ben86a], [Ben86b] and [Ben87]). The response to these columns was so strong that a column dedicated to literate programming was introduced

([Gil87], [Wal87], [Col88], [Lind89], [Ram89], [Wyk90]). The journal *Structured Programming* has laid emphasis on this theme as well ([Lin89a], [Lin89b], [Tun89], [Bro90a], [Bro90b]). An annotated bibliography of literate programming is given in [Smi91].

With the idea of literate programming Knuth also developed the WEB system to support the new paradigm. The original WEB system supports Pascal as programming language [Knu84]. Other implementations have been made for the programming languages C ([Lev87], [Thi86]), Modula-2 [Sew87], Lisp [Ram88] and Fortran [Ave90]. In order to make WEB available to a much larger audience, the tool SPIDER was developed [Ram89] to construct instances of the WEB tools TANGLE and WEAVE from a language description. WEBs for the languages C, AWK, SSL, and Ada have been generated with SPIDER.

The HSD system developed by Tung is also in some way based on the original WEB system. However, documents are not organized linearly but rather composed of a hierarchically ordered collection of sections [Tun89]. Object-oriented programming languages are supported as well, so a literate programming environment is available for Smalltalk [Ree89].

Programming with documentation rather than with pure source code is a major step towards better program comprehensibility and thus maintainability.

3.2.2.4 Configuration Management

Usually large software products are not a single system but a set of similar configurations. Configuration management is the discipline of controlling these configurations and their evolution. It also helps to coordinate programmers working on the same software product. This is accomplished by [Tic88]:

- identifying components and configurations of a software product
- tracking changes (which component, what reason, what time, by whom)
- automatically putting together configurations
- simultaneously managing updates of components

The use of configuration management techniques is not limited to source code. These techniques can be applied to documentation as well. Similar to source code, multiple versions of specifications, design documents, test reports, etc. exist in large projects. The maintenance of large software systems is hardly possible without configuration management .

For small programming teams, manual configuration control procedures might be sufficient. But tool support is indispensable for large teams. The Unix operating system has become a commercially popular environment for software development. It offers configuration management tools like the Source Code Control System (SCCS, see [Roc75]) and the Revision Control System (RCS, see [Tic82], [Tic85]). Both SCCS and RCS have similar capabilities. They manage the files of a software system and can

produce any version of any module on demand. They also prevent simultaneous updates by different programmers.

One common problem in maintaining consistency is to keep source code and object code equivalent. This is not an easy task when an interface has been changed in a large software system. One possibility to guarantee this equivalence is to rebuild all object codes from scratch. However, in large projects this is not always a suitable solution. Unix offers the tool Make (see [Fel79]), which is used to build an executable image from the source code, ensuring in the process that all recent changes are reflected and minimizing the time to achieve this goal (by recompiling only the necessary modules).

Another example of a software configuration management system is the Domain Software Engineering Environment (DSEE) available on Apollo workstations (see [Leb84]).

3.2.2.5 Change Management

The methodology for controlling changes in the process of maintaining a system is called change management. Change management means documenting, communicating, tracking, maintaining and reporting changes in a software system [Art88]. It is useful for the programmers, the managers and the users to report the project status, to schedule changes, to trace changes, etc.

For every change the following questions should be answered by the change management system [Art88]:

- What is to be changed?
- Why is it to be changed?
- Who wants it to be changed?
- How important is it to make the change?
- How will it be changed?
- When will it be changed?
- Where will it be changed?
- Who will make the change?

A change management system allows inquiries in order to facilitate project management. Changes can be scheduled according to their importance (priority); change requests can be rejected due to budget or personnel constraints or because of technical reasons; change requests cannot get lost because they are documented (some kind of historic and project documentation); changes can be tracked and monitored and thus give an essential support for all maintenance activities.

Change management facilities are often integrated in configuration management tools (see [Zve89]).

3.2.2.6 Software Reengineering

Like many terms in software engineering, the origin of the term reverse engineering lies in the field of hardware. The practice of deducing design decisions from a finished product is not unique to the analysis of hardware [Chi90].

Applied to software systems, reverse engineering helps to get a design-level understanding of a system and its structure in order to aid the process of maintaining this system. This is useful when only the source code of a system is available.

The following objectives are to be accomplished by reverse engineering [Chi90]:

- coping with complexity (e.g., by extracting relevant information)
- generating alternate views (e.g., graphic representations)
- recovering lost information (especially about the design process)
- detecting side effects
- synthesizing higher abstractions (generating alternate views at higher abstraction levels)
- facilitating reuse (existing software becoming a candidate for reuse only when there is more information available than just the source code)

Reverse engineering is the opposite of the well-known (or better known) process of moving from the design to the implementation of a system (also called forward engineering to distinguish it from reverse engineering). Subareas of reverse engineering are redocumentation, restructuring and design recovery. Software reengineering uses both the reverse and forward engineering technologies to achieve a new, better maintainable software product, which may already be modified with respect to new requirements.

The usual meaning of restructuring is transforming a program from an unstructured form (sometimes called spaghetti code) to a structured form (without gotos). In this sense, restructuring enhances the readability and comprehensibility of code by improving its structure. In a broader meaning, restructuring is a recasting technique for reshaping data models, design plans, and requirements structures (see [Chi90], [Hod85]).

Restructuring seems to be especially useful with Cobol code because billions of lines of old, unstructured and undocumented Cobol code exist. But we should never forget that (automatic) restructuring only treats symptoms and does not solve real problems. Bad code will remain bad code, although it may become structured bad code. Restructuring is not a substitute for good design and good documentation. But, nevertheless, it can reduce the maintenance burden significantly [Gam86].

Restructurers exist mainly for Cobol because so many existing software systems were written in Cobol without any structured programming techniques. But other languages are supported as well (e.g., Fortran, C). Examples are Superstructure, Cobol ISF, Decoder and Retrofit (see [Zve89]).

More sophisticated tools support the conversion of source code to design code or a specification language, e.g., Reverse Engineering, CASE Station, Promod [Zve89]. An example of a reformatter is given in [Bla89].

Case studies have shown that reengineering can decrease the complexity and increase the maintainability of software systems [Sne90].

3.2.2.7 Visualization

Graphic aids have long been known to support program comprehension (e.g., flowcharts). A more visual style would improve both the creation and the understanding of programs.

- *Visual programming*

  Visual programming allows users to specify a program in a pictorial fashion. This includes flowcharts as well as graphic programming languages.

- *Program visualization*

  Program visualization, on the other hand, uses graphics to provide information about the state of a program (e.g., the values of the variables). This is especially useful during the debugging process, but can also be used for teaching purposes.

- *Program animation*

  Program animation means the dynamic visualization of program states. Static descriptions are inadequate and insufficient for the process of understanding a complex and highly dynamic software system. Program and algorithm animation seems to be the right step forward to support program explanation and comprehension. The idea is that the user can specify objects she/he is interested in, which are then continuously displayed on the screen. Thus the dynamic behavior of a software system becomes a major contribution to the process of understanding this system. The user needs the possibility to view the execution of a program at any desired level of abstraction in order to fully understand what’s going on in the system.

- *Programming by example*

  Programming by example is another new technology which is intended to make programming easier, especially for non-professionals. It means specifying everything about a program by providing and working on examples. Usually this is also done visually.

In [Mye86] precise definitions of the terms visual programming, program visualization, and programming by example are given, as well as a good overview of existing tools. Examples of tools supporting visual programming are PIGS, Pict and Prograph, which use Nassi-Shneiderman diagrams, flowcharts (with color icons) and a data flow language, respectively.

Tools for program visualization illustrate certain aspects of a software system that was conventionally implemented. Animation tools are program visualization tools with dy-

namic display of the program. Examples of program visualization tools are Pegasys and VIFOR [Raj90]. Balsa and Animation Kit are animation tools. Programming by example is offered by Graphical Thinglab, SmallStar, Rehearsal World, and others.

Research work on visual programming environments is collected in the tutorials [Gli90a] and [Gli90b]. They also contain papers about the tools mentioned above.

3.2.2.8 Self-Identifying Software

The idea of self-identifying software is to identify each change of a software component and to provide this identification for every component in an executable program. This reduces the time needed for determining (the versions of) the parts which a software system being diagnosed consists of (see [Gre88]).

Self-identifying software would require appropriate tool support but unfortunately such tools are hardly available.

3.3 Evaluation of the State of the Art

The previous sections presented exemplary problems and possible improvements in the field of software maintenance. In this section we will investigate how these improvements serve to alleviate the problems.

- *Poor code*

  Poor structure of source code can be avoided when applying *structured programming* techniques. Top-down design, modular programming and structured coding lead to considerably better understandable and thus better maintainable source code. If the source code exists already, then *restructuring* can enhance its readability by improving its structure.

- *High complexity*

  Coping with complex systems is made easier by *literate programming* because this paradigm is aimed at telling humans the essentials about a software system (e.g., design decisions). The idea of *hypertext* also contributes to the mastering of complex systems by supporting fast access to available information (both in the source code and in the documentation). *Fourth generation languages* raise the abstraction level, which is an important step in decreasing the complexity of software systems. *Change management* and *configuration management* help to master the evolution of complex systems by supporting the management of different versions and configurations and by keeping track of the change history. *Reengineering* can help to track relevant information. *Visual programming* as well as *programming by example* are further efforts to raise the abstraction level of programming, whereas *program visualization* is used to graphically present various aspects of a software system. The more complex a system is, the more useful these graphic representations are. The same holds for *prototyping*. Working with a prototype is

better suited for understanding the specification of a (complex) system than reading numerous pages of written text. *Self-identifying software* becomes important when the number of components (modules, classes) and their versions increases.

- *Inadequate documentation*

  The quality of the documentation can become considerably better when applying the idea of *literate programming*. *Reengineering* tools can generate different views of a software system (e.g., diagrams). With *fourth generation languages* the need for documentation may decrease due to the higher abstraction level. Working *prototypes* can replace large amounts of written documentation in the specification phase. The behavior of a prototype is considerably better documentation than any written text. The same holds for any *visualization* of any aspects of a software system.

- *Team coordination, multiple maintenance*

  Both *configuration management* and *change management* emerged because of the problems encountered in team coordination and multiple maintenance. They were proposed for handling the questions mentioned in Sections 3.1.4 and 3.1.5.

- *Static description of dynamic behavior*

  The importance of dynamic aspects is increasing (e.g., in object-oriented software systems). *Program visualization* and especially *program animation* offer the opportunity to present this crucial aspect in an adequate manner. In this sense *prototypes* also play a major role because they consider the dynamic aspects of specifications.

- *Linear Presentation of dynamic behavior*

  With *hypertext* the complex structure of a software system can be presented to the user in an adequate manner. The power of nonsequential reading offers the flexibility to meet individual needs in studying complex information webs.

- *Object code as black box*

  Self-identifying software can shed more light on object code by providing important information about all its constituent parts. Reengineering techniques (e.g., disassembling) can also help to get information about object code.

Figure 3.1 summarizes our evaluation of the state of the art. It is interesting to note that spare parts maintenance does not address any of the enumerated problems. The reason for this is that the idea behind spare parts maintenance is simply to develop multiple parts of a software system in order to speed up corrective maintenance in these parts.

	Poor Code	High Complexity	Inadequate Documentation	Team Coordination	Multiple Maintenance	Static Description	Linear Presentation	Object Code as Black Box
Structured Programming	8	8						
Prototyping		8	8			8		
4th Generation Languages		8	8					
Spare Parts Maintenance								
Comfortable Browsing		8					8	
Hypertext		8					8	
Literate Programming		8	8					
Configuration Management		8		8	8			
Change Management		8		8	8			
Software Reengineering	8	8	8					8
Visualization		8	8			8		
Self-Identifying Software		8						8

Fig. 3.1 Evaluation of the state of the art

The number of crosses in a column in Fig. 3.1 does not indicate how well a specific problem is under control. High complexity, for example, is still a problem in software projects. One of the reasons for this is that literate programming, hypertext, fourth generation languages, change management, configuration management, software reengineering, visualization, prototyping and self-identifying techniques are hardly ever used together.

It seems that all mentioned problems are faced by useful improvements. Yet maintenance remains a problem. The main reason for maintenance still being a major problem is the lack of supporting tools. We do not, for example, benefit from hypertext and/or literate programming if we do not have the appropriate tools. Below we give a short evaluation of the tool support.

- Comfortable *browsing tools* are available for modern programming languages (especially object-oriented ones) like Smalltalk, but almost none are available for old languages like Cobol or Fortran.
- Hypertext tools have been conceived for an amazingly broad range of applications, but mostly not for software development and/or maintenance. The use of hypertext

tools designed for this application area is hindered by the difficulty in integrating them with other software tools.

- Many *literate programming tools* have been developed, but even though tools are available and the paradigm is widely accepted as being useful, these tools are hardly used except by their creators (see also [Wyk90]). Besides the fact that still nobody likes to write documentation, the main reason for this situation is that nearly all of these tools lack a modern user interface. They are simply to cumbersome too work with.
- *Configuration management tools* often only focus on various aspects. One of the reasons that there is hardly any universal tool is the fact that research work is far from being completed in these area.
- The same holds for *change management*.
- Tool support for *software reengineering* is mainly focused on subareas thereof, e.g., redocumentation.
- *Visual programming tools* and program visualization tools are areas of active research. Interesting systems have been developed already. Both static and dynamic code visualization are not unusual (e.g., flowcharts, highlighting the part of the source code being currently executed), whereas data visualization is less supported, even though it is at least equally important. Visual programming tools exist mainly for programming with flowcharts or Nassi-Shneiderman diagrams, or for specifying the user interface of a software system.
- Inconceivably, the idea of *self-identifying software* seems not to be worth being supported by tools.

Summarizing, we can say that the tools lack completeness, compatibility and uniformity (see [Mar83]):

- No system has a complete set of tools.
- Tools are language-dependent and machine-dependent.
- Tools on the same machine and for the same language are often incompatible.

Unix, with its extensive tool support, can be regarded as a programming environment (see [Ker76]). However, the toolset provided is not complete either, and, of course, many of the tools are language-dependent. Even compatibility, one of Unix's strengths, is not always ensured.

The main problem with the tool support is that even if various tools are compatible and can be used together, their combined use is in most cases less powerful than a single tool that combines the concepts of these tools. For example, if a hypertext tool, a literate programming tool and a debugging tool can be used together, then the user does not benefit from any hypertext or literate programming features when working with the debugger. However, it would be advantageous to have comfortable access to

the documentation or any other important information during the debugging process also.

Another drawback of most of the tools is their clumsy user interface which does not match modern and widely accepted techniques like menus, windows and wysiwyg (what you see is what you get).

[Tah90] gives an annotated CASE bibliography containing over 120 items. The fact that only two items were classified under the keyword maintenance, one under documentation and two under reverse engineering, further emphasizes that these terms play a minor role in software engineering.

Trying to build a software tool that incorporates all conceivable features for the maintenance process would certainly not be satisfactory. This only leads to colossal software systems that are hardly maintainable themselves, and even difficult to develop with justifiable effort.

The study of existing tools and the above considerations led the author to pick up new and promising ideas, amalgamating them to build a new tool that is a step forward in the improvement of the all too neglected and yet immensely important task of documenting and maintaining software systems.

4. DOgMA: A New Tool for Documentation & Maintenance

This chapter describes DOgMA, the author's contribution to the improvement of documentation & maintenance. Although DOgMA is basically a software development environment, it was designed with the intent to particularly support documentation and maintenance. Section 4.1 describes the goals behind the design of DOgMA. Section 4.2 contains the underlying concepts. The user interface is introduced in Section 4.3. Sample scenarios are given in Section 4.4. Section 4.5 introduces parameterization possibilities. Section 4.6 gives some implementation aspects. Finally, Section 4.7 presents some statistics and measurements.

4.1 Goals

In the previous chapter we saw that too often existing tools deal only with certain subareas, do not combine various concepts, lack compatibility, and suffer from clumsy user interfaces. These observations led to the definition of the following goals for the development of a new maintenance tool.

- *Concentration on the essentials*

  Rather than trying to support all activities of the software maintenance process, the main goal should be to reach a considerable reduction in software maintenance costs with justifiable effort.

- *Synthesis of new concepts*

  The synthesis of useful new concepts like literate programming and hypertext is expected to emerge as a much better aid than the combined use of tools for each concept (e.g., the separate use of hypertext tools and literate programming tools).

- *Compatibility*

  An isolated tool does not help very much unless it supports all activities of the software life cycle process. Therefore an easy integration in an existing environment is essential.

- *Modern user interface*

  Even though modern graphic user interfaces are widely accepted and used, the majority of existing tools does not support such simple things as menus or (multiple) windows. Ease of use is a crucial point for the acceptance of tools; their use must not require the reading of extensive user manuals. This holds especially for unpopular activities like documentation and maintenance.

- *Modern implementation techniques*

  Providing our new tool with a modern user interface additionally offers a good opportunity to use modern design and implementation techniques like object-oriented programming and application frameworks. These techniques are said to reduce design and implementation effort considerably, especially for applications with graphic user interfaces.

One of the major problems with software maintenance is program comprehension. It is the most time-consuming task (see Section 1.1). And one of the most important things to improve program comprehension is system documentation. Thus the improvement of both program comprehension and documentation promises a major reduction in software maintenance costs. We define the following subgoals for the achievement of such an improvement :

- *Motivation to write documentation*

  One of the problems with documentation is that programmers do not want to document. Better tool support for its creation and especially the integration with the source code should provide better motivation.

- *Documentation access*

  Documentation does not help much if it is not accessible. The user has to be told whether documentation exists for a given part of the source code and where it is.

- *Documentation consistency*

  Documentation is seldom up-to-date and therefore gives incorrect or misleading information. This is certainly worse than no documentation at all. A tool cannot really check consistency of a documentation text, but it should help the programmer to check it.

- *Documentation completeness*

  The amount of documentation will vary from project to project. However, a maintenance tool should be able to tell the user which parts of the systems are documented and where (see documentation access), and which are not.

Good documentation is the best way to improve program comprehension. But there are other possibilities for improvement as well.

- *Information access*

  As important as the access to documentation is the (fast) access to the source code itself. Understanding a piece of code hinges on the answers to many questions, e.g.: Where is this variable defined/set/used? An effective maintenance tool must help the user to answer questions like these in an easy and fast manner.

- *Preventing side effects*

  Side effects of a maintenance change are very often triggered by just ‘making a little change’. The reason for this is incomplete understanding of the system to be maintained and can again be prevented by an easy and fast access to relevant information (see above).

- *Dealing with high complexity*

  Being better able to deal with complex software systems is another important goal because new paradigms like object-oriented programming tend to result in even more complex system structures.

Maintenance is work done on existing source code and the corresponding documentation. But as maintenance is considered as continued development (see Chapter 2), the development of new software has to be supported as well because all these development activities are carried out during maintenance also. Therefore a maintenance tool must support the maintenance of already existing systems as well as the creation of new ones.

- *Maintenance support*

  The tool must be able to support the user in the understanding of existing source code, even if there is no documentation available and the code was created with another programming system.

- *Development support*

  The tool must be usable in the maintenance as well as in the development phase; i.e., it must also assist in the development of software systems.

In the conclusion in Chapter 7 we will examine to what extent these goals have been achieved.

4.2 Concepts

We already mentioned that program comprehension is one of the major problems with software maintenance and that system documentation is a crucial point for that task. Hypertext technology shows the greatest promise to improve the understanding of software systems and the idea of literate programming is expected to advance the quality of both source code and documentation. Combining hypertext and literate programming is supposed to strengthen the integration of source code and documentation. The resulting permanent availability of the documentation has many

advantages. The consistency and the completeness of documentation can be checked more easily. The motivation to put down ideas and design decisions and to keep them up-to-date will increase, which results in better information for the maintenance staff.

In the following we will demonstrate in more detail how the used concepts are applied to our tool. This is also partially described in [Sam90] and [Sam92].

4.2.1 The Structure of Source Code

The overall structure of a software system plays an important role for maintenance people getting well acquainted with it. Questions like the following arise:

- Which components does the system consist of?
- What are the interrelations among these components?
- How are the interfaces defined for the various components?
- What is the control flow and the data flow within and among these components?

Generally speaking, a software system can be regarded as an information web which consists of nodes and links. Modern design and implementation paradigms are module-oriented and object-oriented programming. The architecture of module-oriented software systems differs essentially from object-oriented systems. We therefore have to investigate how they correspond with the general information web model and what the differences are between object-oriented and module-oriented systems.

The Information Web of Object-Oriented Software Systems

Object-oriented software systems consist of a set of classes. A class describes the implementation of a set of objects. A class (subclass) can inherit the properties of another class (superclass). If a class is the subclass of only one superclass, we speak of single inheritance. If a class has more than one superclass, we speak of multiple inheritance (for details, see [Mey87]).

Objects (i.e., instances of classes) communicate via sending and receiving messages which trigger the performance of a specific operation. The operation performed is described in a method. The behavior of subclasses can be changed by overriding methods of the superclass.

The information web of an object-oriented system consists of classes and methods with the following interrelations:

- A class can be the subclass or superclass of another class.
- A class has a set of methods.
- A method belongs to a class.
- A method can be overridden in a subclass.

Figure 4.1 shows the relations of a class to two superclasses, three subclasses and three methods.

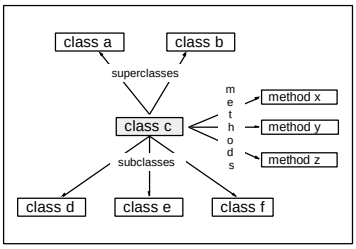

Fig. 4.1 Relations of classes in an object-oriented system

Still other relations exist based on identifiers used in a software system:

- An identifier is defined in a class or method.
- The use of an identifier is related to a specific definition of this identifier and to other uses of the same identifier.
- A comment possibly contains a short description of an identifier, e.g., the description of a class, a method, or an instance variable. (Instance variables are variables defined in a class.) Such a comment is related to all occurrences of an identifier.

The Information Web of Module-Oriented Software Systems

Module-oriented systems consist of modules and procedures rather than of classes and methods. Usually a module (or package) consists of a definition module and an implementation module (or package specification and package implementation). The definition module specifies what a module does and the implementation specifies how this is done.

A module can use another module by importing its definition. Hence in a module-oriented software system we can define the following relations:

- A module consists of a definition and an implementation.
- A module imports other modules.
- A procedure belongs to a module.

We can define relations based on identifiers, as we did for object-oriented systems:

- An identifier is defined in a module or procedure.
- The use of an identifier is related to a specific definition of this identifier and to other uses of the same identifier.
- A comment possibly contains a short description of an identifier, e.g., the description of a module, a procedure, or any other data.

The Information Web of Systems Written in C++

Any software system is implemented in a certain programming language. The question arises whether the language has an essential influence on the structure and representation of the system's information web. We use C++ as an example of an object-oriented programming language and will take a closer look at software systems written in that language. C++ was chosen because on the one hand it is expected to gain widespread use in industry and on the other hand tool support did not keep up

with the evolution of this language and cannot be compared with, for example, tool support for Smalltalk.

The programming language C++ is not a pure object-oriented language but rather an object-oriented extension of C [Str86]. A C++ program system consists of a set of files that contain class definitions, method implementations and global declarations. The global declarations can be used in more than one class and their corresponding methods.

There is no restriction on what has to be written in a single file. A file can contain more than one class definition and a class definition together with its method implementations can be spread over several files. We use the common extensions ‘.h’ for files containing class definitions and ‘.c’ for files containing the implementation of a class, i.e., the implementation of their methods; they are called the *h-files* and *c-files*, respectively.

In order to use a specific class, its h-file has to be *included*. (This is done with a special preprocessor statement.) Besides the inheritance relation shown above, there exists another relation among the files of a program system written in C++, the include relation. This means that usually one has to inspect different files in order to find out the whole story about a C++ class.

Thus the following relations can be defined in a C++ software system:

- A class definition is contained in a file.
- A class inherits from another class.
- A method is contained in a file.
- A method belongs to a specific class.
- A method is overridden in a subclass.
- A file is included by other files.

An identifier may not only be defined in a class or method, but also global in a file, which allows its use in more than one class. The parallel structure of files and classes complicates the information web of C++ systems, which even further impedes the process of getting familiar with such a system.

Programming languages have an important impact on the structure of a software system. Therefore a tool for software maintenance should be capable of presenting various system structures to the user. It must be conceived in a language-independent manner. In order to demonstrate a tool’s qualification for language-independency, at least two different languages have to be supported. It is reasonable to choose both an object-oriented and a module-oriented language for this purpose, as this greatly influences the overall structure of a software system. We decided on C++ as a representative of object-oriented programming languages and Modula-2 as an example of a module-oriented language. Modula-2 [Wir85] was chosen as representative of a module-oriented programming language because it contains all essential concepts of this paradigm without being needlessly complex.

In the following sections examples will be shown in C++ notation. However, the concepts presented can be applied to both module-oriented and the object-oriented software systems.

Collapsing Text

The information web described so far depends on the syntax of the programming language that is used, i.e., on the concepts of the language. However, this is not sufficient for program comprehension because entities like methods or procedures are not subdivided further. We need a possibility to structure larger text pieces beyond the programming language's syntax in order to represent logical contexts. This can be achieved by grouping text parts and giving them a name. A software tool can replace these text parts by their name, thus hiding the details by either collapsing or expanding these texts. This improves the lucidity of larger text pieces significantly. For example, a method that creates a menu bar can logically be divided into groups for each menu entry.

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

Fig. 4.2 Collapsed text parts

Figure 4.2 shows a C++ method that creates seven menus for a menu bar. The source code for creating the menus is hidden.

During development by stepwise refinement, the different steps of the design process can be realized with such collapsed text parts. Information about design decisions is usually lost, but is of crucial interest for maintenance programmers. Text parts can be collapsed in both object-oriented and module-oriented software systems.

Information about Identifiers

One of the most annoying problems in trying to understand a software system is the fact that needed information is spread over the whole system. It is a tedious task to find it when needed. For example, a method or procedure consists of many identifiers of which the meaning is of crucial interest for comprehension. A maintenance tool must provide this information in an easy and comfortable way. This information should contain:

- the declaration of the identifier
- the location of the declaration (which file, method, module, etc.)

- additional information, depending on the programming language (e.g., the inheritance path in object-oriented systems)
- a verbal description of the identifier

Actually, a verbal description of an identifier would be the most useful information. The question arises how such a verbal description can be provided. The simplest and most obvious way to accomplish this is to place such descriptions into comments at the definition (point of declaration) of an identifier. Figure 4.3 shows declarations in C++ with subsequent comments containing short verbal descriptions of the declared identifiers.

```
int curPos;             //index to next character to read
int readLen;        /*current length (to be used for reading only)*/
char   eot;                 //symbol to be returned on end of text
char   *title;          //title of text
bool   alreadyProcessed;        /*this text has been read (and processed) already*/
```

Fig. 4.3 Verbal descriptions of identifiers in comments

This procedure is beneficial in many cases as writing such comments at the point of declaration is not an unusual programming practice. Besides, working with a tool that provides such a feature might be a motivation to consistently describe each declared identifier verbally.

The Structure of Files

In many programming languages a compilation unit (a file) contains several nodes of the software system's information web. A module in Modula-2 contains several procedures and a C++ file usually contains several classes and/or methods. With common text editors it is a tedious task to find out the contents of a file, e.g., which classes and methods are defined in a certain file. The file structure is an essential part of the whole system structure. It is needed, for example, to determine the scope of global identifiers.

The compilation unit (file) is a node of the information web, too (see above). We can easily provide the file structure by showing an outline of it. In the outline any nodes of the system's information web (e.g., class definitions, methods) are simply replaced with their names. Thus, whenever a node of the information web appears in the text of any other node, it is substituted by the name of this node.

Figure 4.4 shows the outline of a C++ source code file that includes three other files and contains several class definitions (but no global declarations).

```
#ifndef HyperMark_First
#define HyperMark_First

#include "ET++.h"
#include "Mark.h"
#include "StyleDialog.h"

class HyperMark
class IdentDefMark
class IdentUseMark
class KeywordMark
class CommentMark

#endif HyperMark_First
```

Fig. 4.4 Outline of a file

4.2.2 The Structure of Documentation

A software system consists not only of its source code; the corresponding documentation is an integral part of it, too. Unlike the source code, the documentation does not obey formal structural rules. However, a well defined structure is equally important for the documentation in order to ease the familiarization process. We therefore define such a structure in this section.

Documentation consists of a set of documentation chapters (parallel to the code structure). Chapters are organized very similar to classes; i.e., there exists a hierarchical structure among them. A chapter itself consists of a title, documentation text (plain ASCII text) and program text (actually a link to the source code; see Section 4.2.3).

The documentation chapters form a tree:

- A chapter has several subchapters and at most one superchapter.
- There exists an order among sibling chapters (i.e., the subchapters of the superchapter).

Figure 4.5 shows the relations of chapter c, which has a superchapter and three subchapters and is related to a previous and a next sibling chapter.

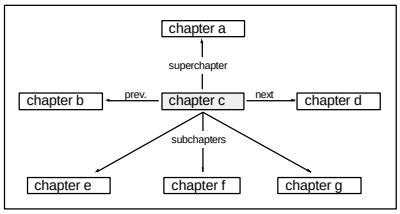

Fig. 4.5 Relations among chapters

Like the source code, the documentation has to be stored in files which can contain one or more chapters. Storing each chapter in a separate file would lead to an unnecessarily large number of files.

4.2.3 Integration of Source Code and Documentation

For software maintenance it is crucial that any information about a software system be easily accessible. This holds not only for information inherent to the source code but also for the documentation which contains information that cannot be represented within the source code (e.g., design decisions, description of the overall system structure). We therefore define in this section how the integration of source code and documentation can be accomplished.

The text of a chapter contains pure documentation text and program text (a link to the source code). Existing tools for the integration of source code and documentation (i.e., literate programming tools) allow an arbitrary mixture thereof. However, they do not explicitly consider the use of single source code identifiers within the documentation text. But this is crucial for many reasons because these identifiers have a special meaning, and a maintenance tool should provide the same information about them as it does within the source code (see above).

We therefore have two different possibilities to use program text in the documentation:

1) single identifiers
2) any text part, i.e., any number of lines of code in succession within the source code

Single identifiers can be used anywhere within the documentation text (see Fig. 4.6).

With these constants we add two entries in the menu **identMenu** in the method **CreateMenuBar** of the class **HyperTextDocument**.

Fig. 4.6 Code identifiers in the documentation text

For better readability, source code identifiers in the figures are shown boldfaced. One or more code lines are clearly separated from the documentation text as shown in Fig. 4.7.

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**.

Fig. 4.7 Source code within documentation text

There should not be any restriction concerning the combination of source code and documentation. A documentation chapter might contain the description of a single class, method, module, or procedure. It might also contain the description of facts that are spread over several source code units.

As it does make sense to include only source code parts that logically belong together, we define that the whole text of a source code unit (e.g., classes or methods) or collapsed text parts can be integrated in the documentation. This is only a minor restriction because the user is free to collapse any parts of the source code which do not overlap (see Section 4.2.1).

Figure 4.8 schematically shows that a chapter contains a title, documentation text with source code identifiers in between, and plain source code.

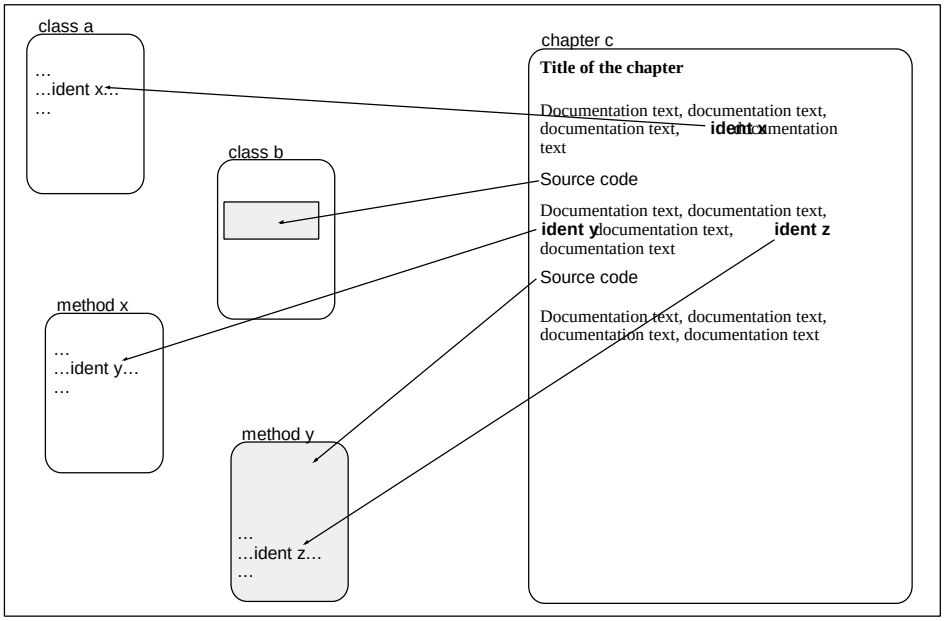

Fig. 4.8 Links from documentation text to the source code

4.2.4 Automatic Creation of the Information Web

The information web must be created automatically by the system with as few actions as possible on the part of the user. The maintenance tool has to establish:

- the links within the source code (i.e., links among classes, methods, modules, procedures, and identifiers)
- the links within the documentation text (i.e., links among chapters)
- the links between source code and documentation text
- collapsed text parts
- remarks about identifiers
- file outlines

Actually, the links within the source code can be derived automatically, whereas links between source code and documentation text have to be defined by the user. But once these links are defined, they have to be kept beyond the session.

4.2.5 Browsing Features

Effective browsing through the information web of a software system is extremely important in order to facilitate the familiarization process for program comprehension and to allow efficient working on this software system. Thus comfortable and powerful browsing features are an essential part of a maintenance tool. Browsing must be possible:

- within the source code
- within the documentation
- between source code and documentation
- based on identifiers (within the source code and within the documentation)

This can be accomplished by basing the browsing features on the relations among the nodes in the information web. Thus browsing becomes possible:

- from a class to any of its superclasses, subclasses or methods
- from a method to its corresponding class or the same method in a superclass
- from any source code part to the corresponding documentation chapter
- from an identifier (either in the source code or in the documentation) to the definition of this identifier

In order to inspect several parts of a software system simultaneously, users must also be able to create several copies of the browser.

The complex information web of large software systems can cause users to get lost in it. This can be prevented by providing useful information about this web. The information should contain:

- the name of the file currently inspected
- the name of the class, method, module or procedure currently inspected
- additional information like the inheritance path (in object-oriented systems)

Another useful way to prevent the user from getting lost is to let her/him go back the way she/he came. Thus a powerful history function must tell the user at any time which parts of the system have been inspected previously and enable her/him to go back to any of these places.

4.2.6 Increasing the Readability

Increasing the readability of source code and documentation provides a major step in the improvement of program comprehension. Better readability can be achieved by simply using different styles both in the source code and in the documentation.

In the source code the definition of a global font, size and style (plain, bold, italics, outline, shadow) is useful for

- keywords
- comments
- identifier definitions (this is especially useful for languages that allow the declaration of identifiers anywhere within the source code, e.g., C++)

In the C++ class definition of Fig. 4.9 we can see the names of the instance variables and the method names at a glance. Furthermore, we can immediately see that four methods are commented out.

```
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();
     …
};
```

Fig. 4.9 Global styles in C++ source code

These global styles are helpful in the source code parts of the documentation, too. Additionally, the definition of global styles for chapter titles and source code identifiers within the documentation text further enhances the readability (see Fig. 4.10).

The global styles need to be applicable not only on the screen but also on paper for printing both pure source code and documentation, and they have to be freely chosen by the user.

This simple global style mechanism can even be used to point out various aspects of a software system. Different styles can also be used to distinguish among different kinds of identifiers, e.g., local variables, global variables, and parameters. Applying arbitrary styles to an identifier of a software system, i.e., the possibility of highlighting an identifier, is of great help in making it detectable throughout the whole system.

Based on the possibility to highlight single identifiers, identifiers can be highlighted that are defined in a certain part of the source code. This feature enables the user to recognize, for example, the identifiers defined in a class definition (i.e., the instance variables of a class) or the local variables of a method.

In order to distinguish among different highlighted identifiers (e.g., local and global variables) different styles must be applicable for highlighting. Figure 4.11 shows a C++ method where instance variables of its class are boldfaced and local variables are shown

in italics and a different font. Using different colors instead of or in addition to the styles will even further increase the readability.

4.2 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:
    …
```

If the identifier is to be pasted, we send the text the message **PasteMark** and provide the mark of the identifier as its 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 updated correctly in PasteMark already
  SetChanged();
  return newMark;
}
```

We just call the method **PasteMark** of the **HyperMark** we get as the first parameter.
…

Fig. 4.10 Global styles in the documentation

It is rather cumbersome to inspect a whole software system in order to find all occurrences of certain identifiers. Therefore the user has to be provided with the information where highlighted identifiers can be found (e.g., in which classes, methods and files).

4.2.7 Hardcopy Documentation

Although the importance of printed documentation vanishes due to the powerful interactive browsing features, it is still necessary to get a printed copy of both documentation text and source code. Thus we need the facility to directly send a document to the printer or to create a file where documentation text and source code are intermixed as seen on the screen (see Fig. 4.10).

To get a complete documentation, the chapters have to be printed in preorder and automatically numbered. Also, a table of contents has to be created automatically. It is not necessary to provide printed hints about where an identifier used in the documentation text is defined in the source code or where a piece of source code in the documentation text is copied from. Usually this will be clear from the context. Besides, this information

is easily provided interactively. Printing documentation is only considered a possibility of saving it. The power of nonsequential reading makes printed documentation less important and less useful.

```
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
   }
}
```

Fig. 4.11 Highlighted identifiers in a C++ method

4.2.8 Processing Subsystems

Very often software systems are not developed from scratch, but rather already existing code is reused. In fact, the ease of reusing existing source code is one of the major advantages of object-oriented programming (e.g., application frameworks). When working on such systems, programmers are not interested in the implementation of the reused source code part, but in its definition only (in order to facilitate the reuse process). Additionally, large software systems are often divided into (possibly overlapping) subsystems with different people working on them. In order to work on a subsystem, many definitions of the rest of the software system are necessary for program comprehension. The implementation thereof is not important.

A maintenance tool has to support the processing of subsystems and to distinguish between source code of that subsystem under consideration and the other parts from which only the definitions should be available. It has to be guaranteed that changes are made to the affected subsystem only. For example, the definitions of an application framework must not be changed when reusing it.

We use a project description file both to enable users to work on subsystems and to establish write protection for the rest of a software system. The project description file contains the file names of the subsystem under consideration (in the notation of the programming language used). Files that are not listed in the project description file cannot be changed by the user.

```
//sample project description file
#include "HyperCmdNo.h"
#include "HyperMark.h"
#include "HyperMark.c"
#include "HyperProject.h"
#include "HyperProject.c"
#include "HyperText.h"
#include "HyperText.c"
#include "HyperTextDocument.h"
#include "HyperTextDocument.c"
#include "HyperTextView.h"
#include "HyperTextView.c"

#include "intro.d"
#include "hypertext.d"
#include "literate.d"
#include "implement.d"
#include "copyPaste.d"
#include "conclusion.d"

#include "makefile"
```

Fig. 4.12 Example of a project description file

Figure 4.12 shows an example of a project description file. Several source code files (h-files and c-files) as well as documentation files (d-files) and a makefile are included. Our maintenance tool has to load a software system based on this project description file, i.e., all files that are included directly or indirectly are loaded but the user is only allowed to change the contents of files that are listed explicitly.

4.3 User Interface

The user interface concept is based on modern application frameworks and the supported concepts thereof (see [Shn86], [Wei89]). Ease of use was one of the major goals in its design. There exist three main parts that will be described in detail in the following subsections: the application window, the hypertext window and the file window. The hypertext window is the essential part of DOgMA, whereas the application window is used as a control panel and the file window provides only an additional feature for simple text file editing.

The supported programming language has only minor effects on the user interface. In the following sections the user interface of the C++ version of DOgMA, which has

been implemented already, is presented. A Modula-2 version is currently under development.

4.3.1 The Application Window

The application window is part of every application that is implemented with the application framework ET++ (see Section 4.6). It is a kind of a control panel for creating and opening documents and for quitting the application.

Fig. 4.13 Application window

DOgMA's application window provides the following six buttons (see Fig. 4.13):

- *about*
  Selecting the about button pops up a little dialog box with some information about DOgMA.
- *new project*
  Selecting the new project button provides an empty project description file in a hypertext window.
- *open project*
  Selecting the open project button enables the user to open an existing project description file. The project is read in automatically and shown in a hypertext window.
- *new file*
  With the new file button a new file can be created in a file window.
- *open file*
  The open file button is used for editing arbitrary text files in a file window.
- *quit*
  Selecting the quit button ends a session with DOgMA. If there are still single files or projects open that have been changed, then the user is asked whether they should be saved on disk.

4.3.2 The Hypertext Window

The hypertext window is used for working on a project and represents the proper maintenance tool. It provides a menu bar, two selection lists, an information box, and an editor window (see Fig. 4.14).

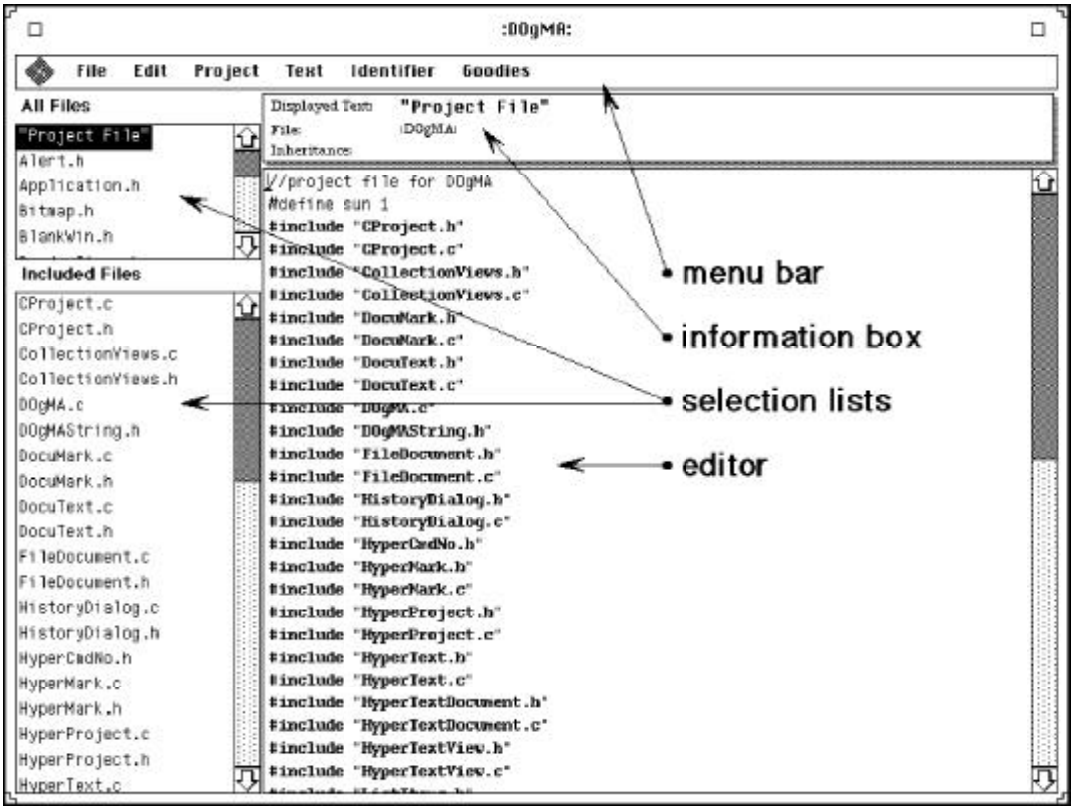

Fig. 4.14 Hypertext window

The Menu Bar

The menu bar is used to perform various commands. It is clearly separated into seven groups:

- The ***file menu*** offers commands for loading and storing files.
- The ***edit menu*** offers commands for simple text editing, e.g., cut, copy, paste, and enhanced editing commands like inserting and deleting of nodes in the information web.
- The ***project menu*** offers commands applicable to a whole project, like saving and/or closing a project.
- The ***text menu*** offers common hypertext commands.
- The ***identifier menu*** offers hypertext commands based on identifiers.
- The ***goodies menu*** offers some special commands.

The menu entries are described in more detail below.

The Information Box

The information box is used to display relevant information about the node currently under inspection. This prevents users from getting lost in complex software systems. In the C++ version of DOgMA this information contains:

- the name of the code or documentation currently shown
- its file name (containing the directory path)

- the inheritance path (i.e., all the superclasses) of the class or the method currently shown

The Selection Lists

A classification of nodes is of great help in mastering the complexity of large systems. Powerful classification mechanisms are provided by Smalltalk systems where users can introduce arbitrary categories. We adopt a simple but still very useful version of this mechanism by providing predefined categories. The nodes of the various categories are shown in the upper selection list. The text bar over the list indicates the category of nodes that are displayed in this list.

In order to understand a certain piece of code (the content of a node), many related pieces (nodes) have to be inspected, too. This is due to the fact that global variables are used, instance variables and methods are inherited from superclasses, etc. Therefore, the lower selection list shows nodes that bear a certain relation to the one selected in the upper list. The text bar over the lower list indicates the relation among the nodes of this list and the node selected in the upper list.

To choose another category or another relation, these text bars can be used as popup menus (see Fig. 4.15). The following categories can be chosen in the upper list:

- ***all classes*** of the system
- ***implemented classes***, i.e., classes of which the implementation part is loaded, too
- ***all chapters*** of the documentation
- ***top chapters*** of the documentation hierarchy
- ***all files*** of the system
- ***implementation files***, i.e., files that contain method implementations
- ***documentation files***, i.e., files that contain documentation chapters

The following relations can be selected for the lower list (see Fig. 4.15):

for classes:

- their ***superclasses***
- their ***subclasses***
- their ***method implementations***

for chapters:

- their ***superchapters***
- their ***subchapters***
- their ***sibling chapters***

for files:

- their ***included files***
- their ***substitutions***, i.e., the classes and method implementations they contain

Selecting an entry in any of the two selection lists causes DOgMA to browse to the corresponding node in the information web.

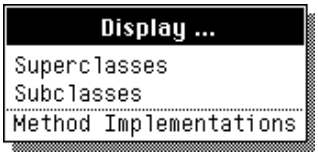

Fig. 4.15 Popup menu for lower selection list

The Editor

The editor window displays the code part according to the selections made in the *selection lists* on the left side or in the *information box* (see below). It offers the usual text editing capabilities, e.g., cut, copy, paste, automatic matching of parentheses (by simple double clicking in the source code).

4.3.3 Browsing Features

Effective browsing is essential in the maintenance process. DOgMA supports this activity in many ways:

- simple browsing features
  by selecting an item in one of the selection lists
- enhanced browsing features
  by selecting an item in the information box
- identifier-based browsing features
  by using the interrelations among the identifiers in the software system
- history-based browsing features
  by selecting one of the previously inspected items
- documentation-based browsing features
  by using the interrelations between source code and documentation text

Both simple and enhanced browsing features are presented in this section. The other features will be described in succeeding sections where the various menus are presented (identifier-based browsing in Section 4.3.9, history-based browsing in Section 4.3.7, and documentation-based browsing in Section 4.3.8).

Simple Browsing Features

One possibility to browse from one piece of code to another is to select an item from the upper selection list. In this case the appropriate information is displayed. Choosing among the available categories and selecting items shown in this list makes it possible to reach all locations of a software system.

However, another and more useful way to reach other parts of the system is to follow one of the relations associated with the displayed node. Thus selecting items shown in the lower list enables the user to browse (in C++)

- from a class to its superclass or any of its subclasses or method implementations
- from a file to any of its included files or its substitutes

- from a documentation chapter to any of its subchapters, sibling chapters, or to its superchapter

Enhanced Browsing Features

When trying to understand a certain piece of code, information in related code pieces is of crucial importance. In object-oriented software systems these related code pieces are the superclasses and the overridden methods because they contain most of the information that is needed to understand a subclass or an overridden method.

DOgMA provides easy access to this information by simply selecting one of the superclasses in the inheritance path (shown in the information box). This enables the user to inspect any of the superclasses of a class and get back again by simply selecting the first item of the inheritance path again.

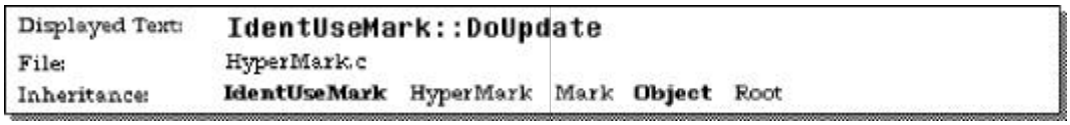

Fig. 4.16 Enhanced browsing

Additionally, if a method of a class is selected, then all the classes in the inheritance path that implement the same method are highlighted. Thus the user can see at a glance which classes implement a certain method. In Fig. 4.16 the class names IdentUseMark and Object are highlighted (shown in boldface) because they implement the method DoObserve.

Selecting a highlighted superclass immediately shows the appropriate method of the superclass. If the method implementations of a class are not loaded—their files were not specified in the project description file—, then the definition of the appropriate method is selected in the class definition. For example, when an application framework is used, then we might be interested only in the definition of the framework's class definitions. In this case the corresponding method implementations would not be loaded.

Figure 4.17 shows the definition of a method in one of the superclasses. This was accomplished by simply clicking at the superclass in the information box.

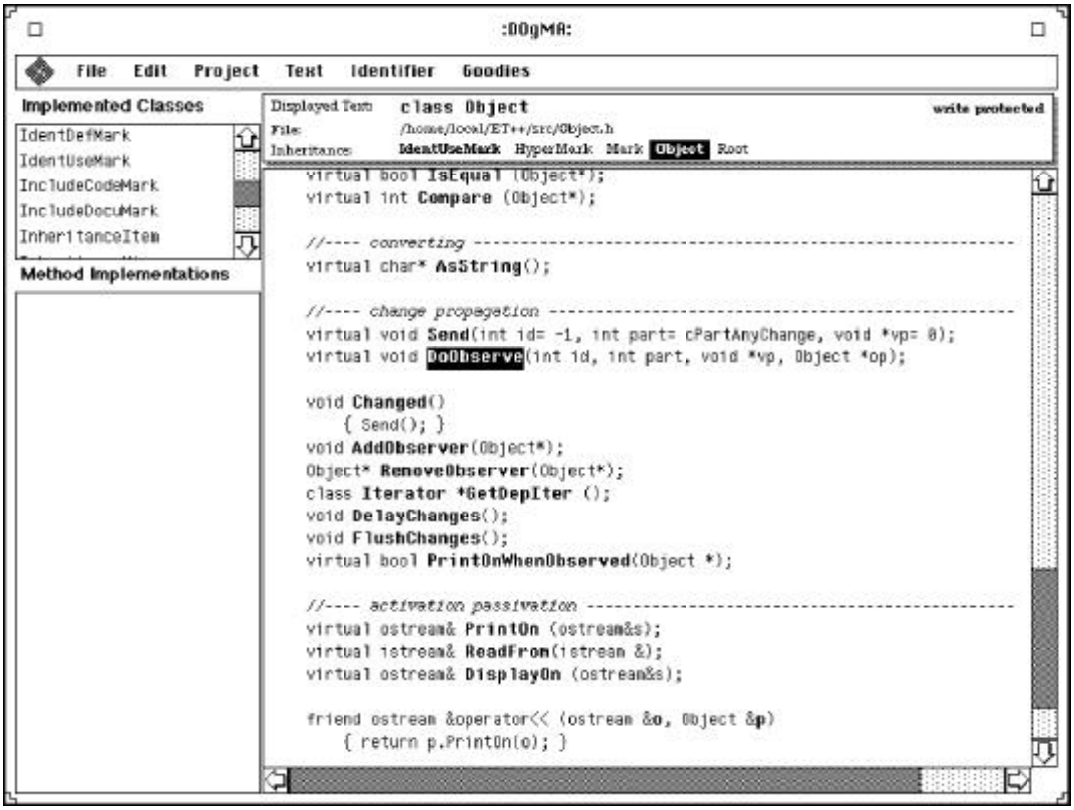

Fig. 4.17 Method definition in a superclass

In C++ the files that contain the classes and methods under consideration are also of interest because they can contain global declarations that are used in these classes and methods. Therefore, DOgMA supports also to select the file path shown in the information box. This facility offers an easy way to browse to the file where a specific code part is contained. Using this possibility together with the features of the inheritance path allows us to inspect all the files coherent to the superclasses and their method implementations.

Please note the ‘write protected’ in the upper right corner of the information box in Fig. 4.17. This means that the displayed class Object does not belong to the part of the system that is being worked on (i.e., it has not been specified in the project description file). Therefore, the user cannot make any changes to it (see Section 4.2.10).

4.3.4 The File Menu

The file menu provides commands for loading and saving files, for closing the hypertext window and for showing the application window (see Fig. 4.18).

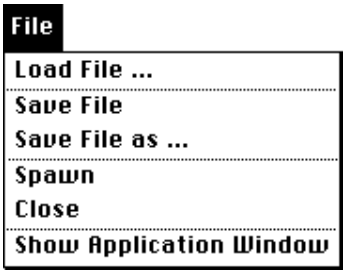

Fig. 4.18 The file menu

Please note that whenever a menu entry ends with three dots (…) then the command is not executed immediately, but rather the user is asked for more information. For example, if a file is to be loaded, then the user has to specify which file to load.

- *Load File…*

  Loading a new file is useful when the user processes a subsystem only and then wants to inspect a file which is not part of this subsystem (e.g., to inspect the implementation of a class defined in another subsystem). With the *Load File…* command a new file can be added to the already loaded software system. The specified file as well as all its directly and indirectly included files that have not yet been loaded are added. Furthermore, the hypertext links are automatically established and the list entries are updated accordingly.

- *Save File/Save File As…*

  During long sessions with any tool the ability to occasionally save any changes that were made is very important. The currently displayed file can be saved to disk, either overwriting the existing file (*Save File*) or creating a new file (*Save File As…*). If a class or a method is displayed in the editor window, then the file containing the class or method is saved.

- *Spawn*

  The *Spawn* command allows inspection of various parts of a software system simultaneously by creating a new copy of the hypertext window. An arbitrary number of browsers can be created with this command.

- *Close*

  Closing a hypertext window is accomplished by selecting the *Close* command. If modifications have been made since the last save and there are no other hypertext windows open for the same project, then the user is asked whether these changes should be saved.

- *Show Application Window*

  The application window is needed for opening files and projects and for quitting DOgMA. If it is hidden on the screen by other windows, then it can be brought to the front by choosing this command.

4.3.5 The Edit Menu

The edit menu provides functionality for simple text editing, line positioning, searching, replacing, inserting and deleting classes, methods, chapters and files (see Fig. 4.19).

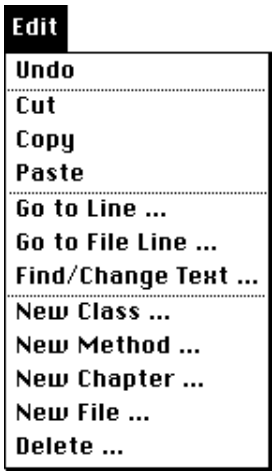

Fig. 4.19 The edit menu

- *Undo*

  This command can be used for undoing any previously executed command. The ability to undo previous activities is one of the most useful and important commands for any software tool because its absence often inhibits users from effectively working with it.

- *Cut/Copy/Paste*

  These menu commands provide the usual cut/copy/paste functionality.

- *Goto line…/Goto file line…*

  If the source code has been changed and stored back onto the corresponding files, then a recompilation can be started. Compilers usually report errors with the line number and an error message. As DOgMA cuts the content of source code files into little pieces, it becomes difficult to position at certain line numbers. Therefore, the command *Goto file line…* is available, which displays the proper class, method or file and positions at the specified line number (within the currently displayed file). The command *Goto Line…* is used to position at a certain line number within the displayed text. Figure 4.20 shows the dialog for positioning at a certain line.

Fig. 4.20 Dialog for line positioning

- *Find/Change Text…*

  Finding and changing text is one of the most useful commands for software engineers that have only a simple text editor available. In a modern environment for development and maintenance, its use should be replaced by more comfortable commands (e.g., changing the name of an identifier in a certain scope). Nevertheless, there are cases where finding and changing arbitrary text strings is necessary (e.g., to find string constants). Figure 4.21 shows the dialog box for this command (see also [Wei89]).

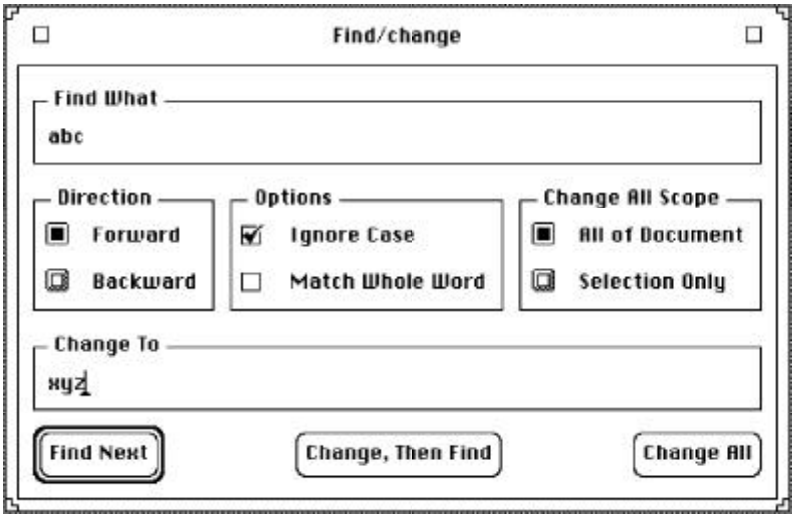

Fig. 4.21 Find/change text

- *New Class…*

  DOgMA asks the user for the name of the new class, an optional superclass, the name of the file where the class definition is to be stored, and the name of the file where the constructor and the destructor methods are to be stored. A constructor and a destructor method are automatically created because they are implemented in almost every class.

- *New Method…*

  DOgMA asks the user for the name of the new method, the corresponding class, and the name of the file where the method is to be stored.

- *New Chapter…*

  The user is asked for the name of the new chapter, an optional superchapter, and the name of the file where this chapter is to be stored.

- *New File…*

  DOgMA asks the user for the name of the new file and whether it is to be a definition, an implementation, or a documentation file. This information is needed because different text templates are provided for these files.

- *Delete…*

  The delete command is used to remove files, classes, methods and/or chapters.

Whenever a new node (class, method, chapter, file) is inserted, DOgMA provides a text template which can be completed by the user. The text templates can arbitrarily be defined by the user (see Section 4.5). The user enters the name of the new node and selects the classes and the files from a list provided by DOgMA.

4.3.6 The Project Menu

Any commands that are applied to the whole project are included in the project menu (see Fig. 4.22).

Fig. 4.22 The project menu

- *Show Project File*

  The project description file is displayed when this command is chosen.

- *Read Project*

  When a project is opened, then it is read in automatically. However, if a new project description file is created or new entries are added to an existing one, then reading the project has to be started explicitly.

- *Parse Project*

  Reading a project causes the hypertext web to be generated based solely on classes, methods, chapters and files. With the command *Parse Project* a full syntax analysis is accomplished, extending the hypertext web based on identifiers. Parsing a project is separated from reading a project because a full syntax analysis is needed only when the user wants to get as much information about a software system as possible. If a user is familiar with a system already and only wants to make changes, then she/he does not need this information and a simple reading of the project will suffice.

- *Save Project*

  DOgMA provides comfortable working with classes and methods rather than with files only. Whenever changes have been made in a software system, DOgMA remembers which files were affected. Applying the command *Save Project* causes DOgMA to ask the user whether each changed file is to be saved to disk.

- *Close Project*

  As with *Save Project* the user is asked whether changed files should be saved. Additionally, the project is closed; i.e., all the hypertext windows of this project disappear. Unless there are any other projects or files open, only the application window will remain on the screen, allowing the user to quit DOgMA or to open other projects and/or files.

- *Define Styles*

  In order to enhance the readability of the source code (see Section 4.2.6), the user can define global styles for various syntactic constructs, e.g., comments, keywords (see Fig. 4.23).

  Global styles can be defined:

  - for keywords, comments, substitutions, identifier definitions, and identifier uses in the program text
  - for chapter titles, unresolved code links (see below) and code identifiers in the documentation text
  - in three different styles for the highlighting of identifiers in both the program and the documentation text

  For source code within the documentation, the same global styles are used as in the plain source code. Code identifiers are those identifiers that are used directly within the documentation text. The definition of a global style for these identifiers allows an immediate recognition of them. Unresolved code links are described in Section 4.4.5.

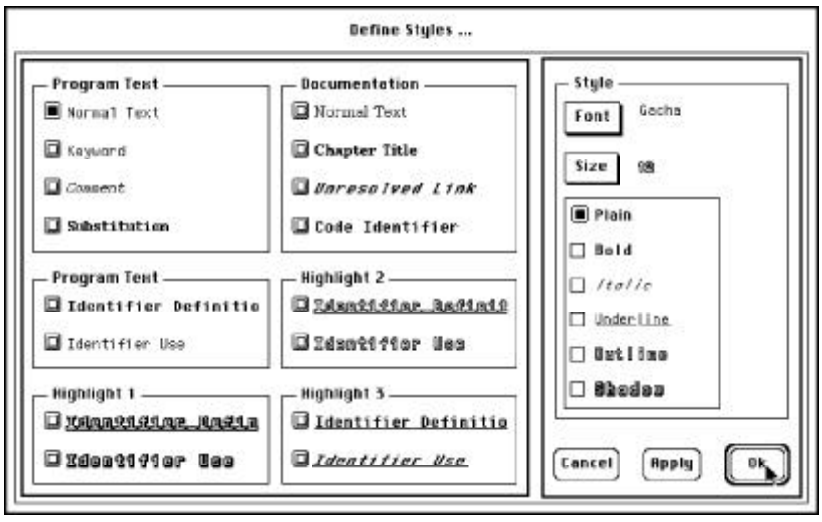

Fig. 4.23 Global text styles

Each global style is defined by its font (e.g., Gacha, Bookman), its size (e.g., 10 point, 12 point), and its style (e.g., plain, bold, underline).

- *History…*

  A powerful history function is necessary to prevent users from getting lost in a complex software system. DOgMA remembers the browsing path taken by the user and thus enables her/him to undo any browsing activities and get back to where

she/he came from. A history dialog appears on the screen containing the names of the previously inspected items. Selecting any one of these items causes DOgMA to branch to the appropriate part of the system.

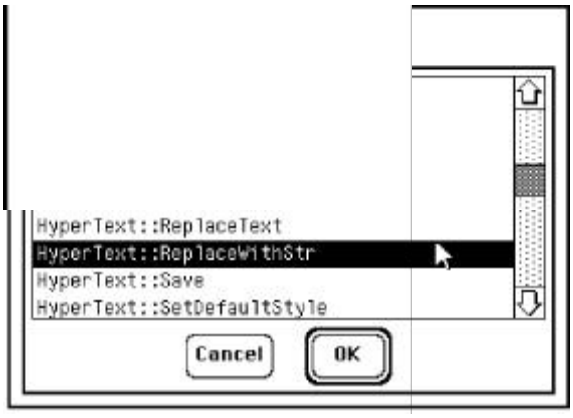

Fig. 4.24 History dialog

Figure 4.24 shows the history dialog with various methods (in C++ notation) which allows the user to select an arbitrary item of her/his browsing history.

4.3.7 The Text Menu

The text menu provides commands that are applied to text parts (see Fig. 4.25).

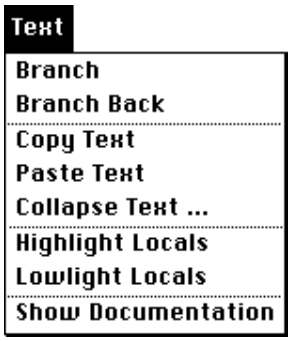

Fig. 4.25 The text menu

- *Branch*

  The outline of a file contains substitutions of classes and methods (see Section 4.2.1). With the *Branch* command we can branch to the corresponding class or method. Besides, with this command we can branch from an include statement to the included file. This branching can also be done by displaying the included files or the substitutions in the lower selection list and selecting the appropriate item.

- *Branch Back*

  With the *Branch Back* command the user can branch to the previously inspected text part (see also the *History* command). It can be applied repeatedly, thus traversing the browsing history in reverse order.

- *Copy Text/Paste Text*

  With the *Copy Text* and the *Paste Text* commands source code parts can be integrated in the documentation. Either a whole text (class, method, etc.) or a collapsed text can be used. The text in the source code is simply copied and pasted into the documentation text. All hypertext links are established automatically.

- *Collapse Text*

  In order to define collapsed text parts, it suffices to select the corresponding text and select the menu entry *Collapse Text*. DOgMA automatically establishes the hypertext information and guarantees that the collapsed text part is saved to disk correctly so that it is possible to use it in another session again.

- *Highlight Locals/Lowlight Locals*

  In order to highlight the identifiers defined in the currently shown text, the command *Highlight Locals* is used (see Section 4.2.6). This enables the user to highlight the instance variables and method names of a class definition, the parameters and local variables of a method, or the global variables of a file. Figure 4.26 shows both global and local identifiers highlighted. In the two lists at the left side and also in the inheritance path, we see the number of occurrences in the classes and the methods. This information is provided for files also (i.e., if files are displayed in the selection lists). For classes two numbers are listed. The first one specifies the number of occurrences in the class definition and the second one specifies the number of occurrences in the corresponding method implementations. The highlighting can be undone with the *Lowlight Locals* command.

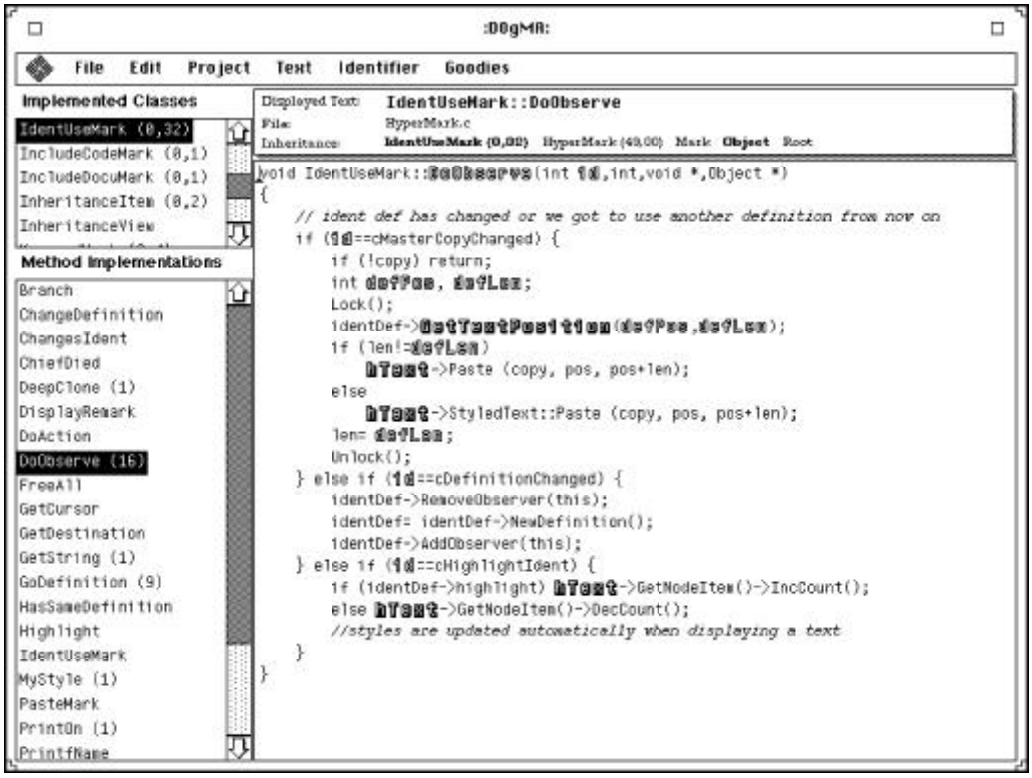

Fig. 4.26 Highlighted identifiers

- *Show Documentation*

  It is important to know whether documentation exists for a certain part of the source code and where it is. With DOgMA the documentation text of any source code part can easily be found by applying the *Show Documentation* command. If a collapsed text is selected, then the documentation of this text is shown; otherwise the documentation of the displayed class, method or file is shown (see Section 4.2.3). If there is no documentation available, then the menu entry *Show Documentation* is dimmed and cannot be selected.

- *Show Code*

  If documentation text is displayed, then the menu entry *Show Documentation* is replaced with *Show Code*. With this command it is possible to branch from any source code part in the documentation to the corresponding place in the source code itself. To branch back, we either apply the command *Branch Back* or the *Show Documentation* command that is now available.

4.3.8 The Identifier Menu

The identifier menu consists of menu entries that are applied to single (source code) identifiers (see Fig. 4.27). These commands are available only when the software system has been parsed (see menu entry *Parse Project*). If the software system has not been parsed, then the whole menu is dimmed.

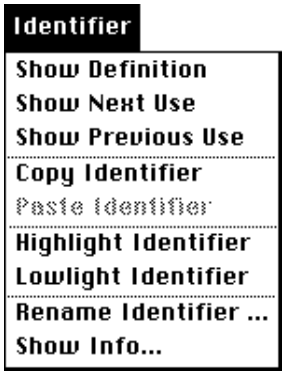

Fig. 4.27 The identifier menu

- *Show Definition*

  Finding the definition of an identifier is frequently required when trying to understand a piece of code. Therefore DOgMA offers the possibility to select an identifier and jump to its definition immediately, no matter where this definition is in the system.

- *Show Next Use/Show Previous Use*

  In order to inspect all occurrences of an identifier, the commands *Show Next Use* and *Show Previous Use* can be used to browse to the next or previous use of an identifier. The definition of the identifier can be used as starting point.

- *Copy Identifier/Paste Identifier*

  Similar to the *Copy Text* and *Paste Text* commands, these commands are used to integrate source code identifiers within the documentation. Identifiers can be copied both within the source code and in the documentation (if an identifier had been copied already from the source code). Pasting an identifier in the documentation establishes the hypertext links and guarantees that the identifier is highlighted and updated correctly (see menu entry *Change Identifier*).

- *Highlight Identifier/Lowlight Identifier*

  These commands have the same effect as their counterparts in the text menu (*Highlight Locals* and *Lowlight Locals*). However, highlighting can be applied to a single identifier with these commands (see Section 4.2.6).

- *Rename Identifier…*

  When working with a simple text editor, changing the name of an identifier is usually done with the *Find/Change Text…* command (see above). However, this has many disadvantages. The user can never be sure that all occurrences have been considered because the use of an identifier can be spread over several files. On the other hand, there may exist different identifiers with the same name but in a different scope. It is very difficult to consider this fact when using a command that operates on text strings only. With the menu entry *Rename Identifier…* a selected identifier can be renamed in the whole software system (both in the source code and the documentation). This is simply done by entering the new name and pressing the OK button (see Fig. 4.28). With this command it is guaranteed that all occurrences of the identifier are renamed and that identifiers with the same name but a different scope are left unchanged.

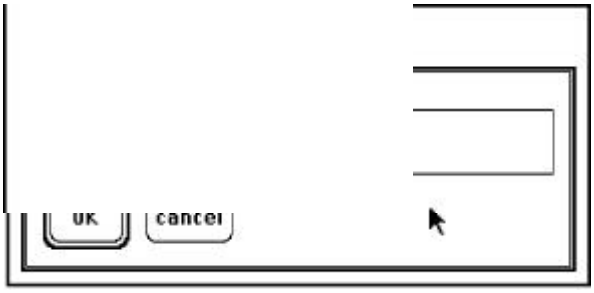

Fig. 4.28 Renaming an identifier

  This command can be activated only if the identifier to be changed is defined in the inspected subsystem. This prevents the change of identifiers in reused source code. Besides, changing an identifier which is not part of the subsystem under consideration would lead to an incomplete change when only the implementation part of this subsystem is loaded.

- *Show Info…*

  If a short description exists for an identifier (a short comment after its declaration), then this description together with some other information (e.g., point of

declaration) can be shown wherever this identifier is used (see Section 4.2.1). This is an essential aid in supporting program comprehension.

We assume that a comment after the definition of an identifier contains a description of this identifier. Assuming this style for existing programs might—at worst—lead to the display of a comment that was not intended to describe an identifier. But on the other hand, the system can provide the user with information that is very useful for program comprehension and can do so with little effort on the part of the user (i.e., by writing a short comment for all or many identifier definitions).

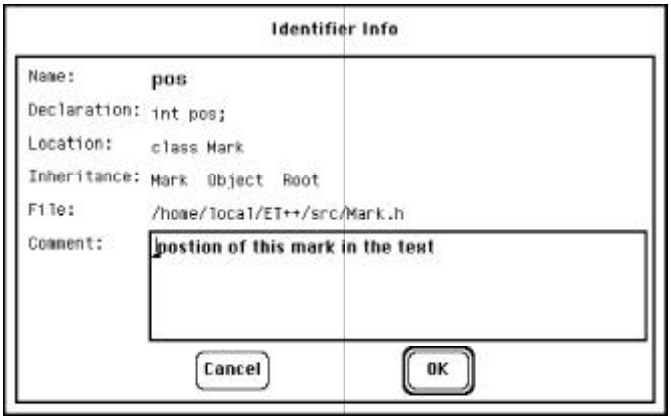

Fig. 4.29 Information about an identifier

In Fig. 4.29 information was requested about the identifier *pos*. DOgMA tells us that pos is an integer variable which is defined in the class Mark. Furthermore, the inheritance of this class, the file where it is stored and a verbal description are given. Note that DOgMA does not ‘know’ that pos is an integer variable, but rather displays the contents of the line where pos is defined. In most cases the source code line of an identifier’s definition provides the information the user is interested in.

The browsing capabilities provided for the source code are supplied for the documentation, too. So it is also possible to jump from an identifier in the documentation (either a single identifier within the documentation text or an identifier within a source code part) to its definition in the source code.

4.3.9 The Goodies Menu

The goodies menu contains some additional useful commands (see Fig. 4.30).

Fig. 4.30 The goodies menu

- *Use Highlight 1/Use Highlight 2/Use Highlight 3*

  In order to distinguish among various identifiers, we can use different styles (see *Define Styles…*). The highlight commands use either the first, second or third highlight style, depending on which of the three *Use Highlight* commands is active.

- *Show Comments/Hide Long Comments/Hide All Comments*

  It is sometimes useful to collapse comments when they are used very extensively. DOgMA provides the possibility to either hide all comments or long comments only (see Fig. 4.31). A comment is considered long if it spreads over more than one line, i.e., if it contains at least one newline character. Only the content of the comments is hidden; their delimiters (// and /**/ in C++) remain visible.

```
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:
    /**/
    …
};
```

Fig. 4.31 Hidden comments

The commands in the goodies menu are used like switches. When one of the highlight or comment entries is chosen, it is dimmed (to permit easy recognition of which one is active). They remain active until another one is selected.

4.3.10 The File Window

Inspecting the contents of an arbitrary file is often necessary when maintaining a software system. Therefore DOgMA also supports the editing of simple text files to prevent the need for an additional text editor. This is accomplished with the file window, which is a subset of the hypertext window containing only a menu bar (with fewer menus) and an editor window (see Fig. 4.32).

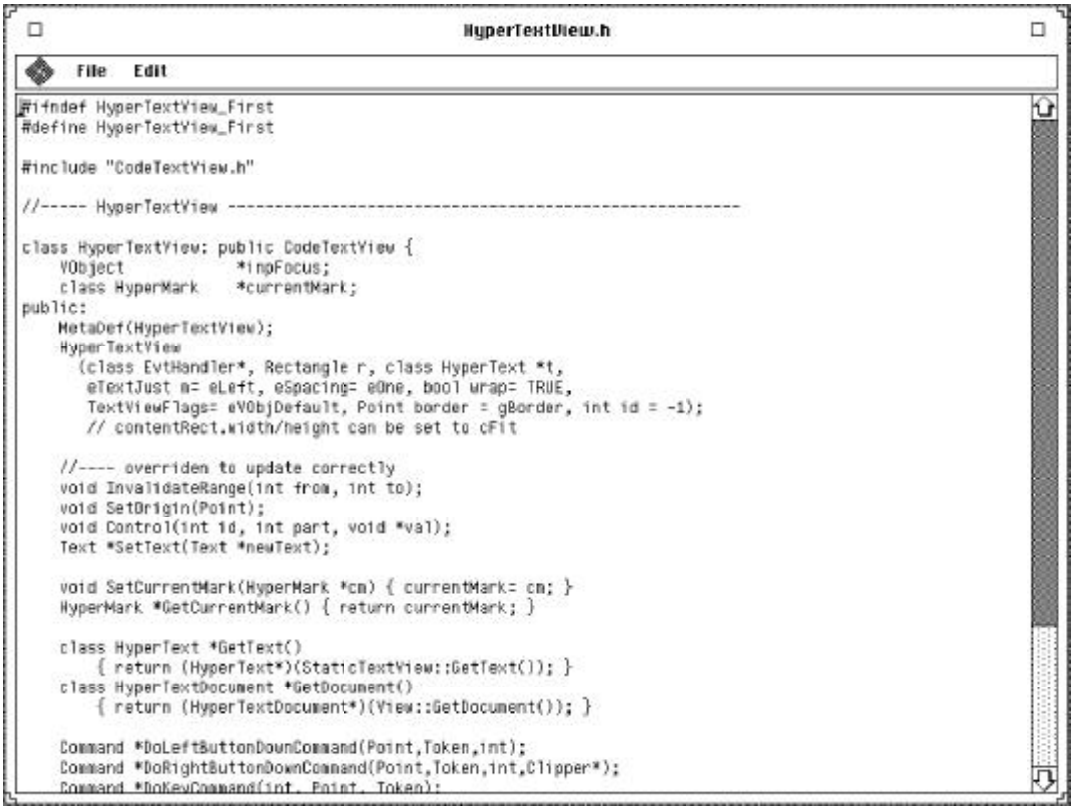

Fig. 4.32 File window

The Menu Bar

The menu bar of the file window contains only the file and edit menus (see Fig. 4.33), whereby some entries are omitted. The functionality of the remaining entries is the same as in the hypertext window and is described in previous sections.

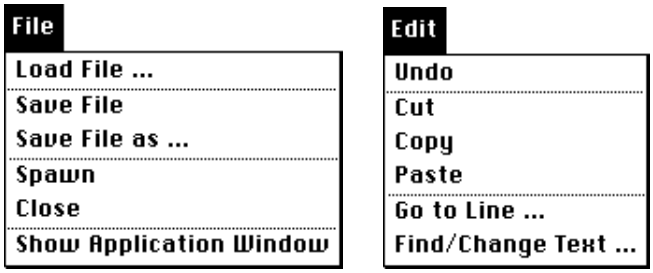

Fig. 4.33 Menus of the file window

The Editor Window

The editor window displays the open file and offers the usual text editing capabilities like in the hypertext window, e.g., cut, copy, paste.

DOgMA meets the requirements for a comfortable and easy-to-use interface. It is comprised of simple elements like menus, lists and dialog boxes. Users were able to use it with only a short introduction and demonstration of the various features without the need for any user documentation.

4.4 Sample Scenarios

This section shows how various questions about a software system can be answered with DOgMA, provides sample documentation, and demonstrates how documentation can be read, written and modified.

4.4.1 Answering Questions

We already mentioned that various questions arise when a maintenance programmer is trying to understand a software system. Providing answers to these questions in an efficient way significantly speeds up the comprehension process and thus helps reducing maintenance costs.

The following subsections demonstrate with a few typical examples how DOgMA can be used to find answers to various questions related to program understanding.

Where is this variable used?

Particularly when there is no documentation available and there are hardly any comments in the source code, it is often difficult to guess the meaning of a variable. Possibly even its name is not very expressive. Finding out where in the system this variable is used is a first and important step in casting light on the subject.

With DOgMA this is an easy task because it suffices to select the identifier and highlight it (see menu entry *Highlight Identifier* above). Then in the selection lists we can see where and how often this identifier is used. Additionally, in the editor window we easily recognize all occurrences of the identifier by a different font, style and/or size.

Where is this message sent? Which methods send a particular message?
Where is this procedure called? Which other procedures call a particular procedure?

The same holds for methods and procedures as for variables. Sometimes it is useful and necessary to find all callers of a particular method or procedure. Again, with DOgMA this is easily done because highlighting the method/procedure name tells us where else in the system this method/procedure is used (called).

Which classes implement a particular message?

Knowing the implementors of a particular message is crucial to the understanding of an object-oriented software system. With DOgMA this question cannot be answered by simply highlighting an identifier, but with the information box it is easy to find out which of the superclasses implement a certain method of its subclasses (these classes are shown in boldface, see Section 4.3.3). Besides, with the ability to display and browse to subclasses, it is relatively easy to find out the implementors of a message.

Admittedly, a graphic representation would ease to answer this question. However, this is deferred to future versions.

Where are the parameters set in this method/procedure?

In inspecting a method or procedure, it is important to find out where input parameters are used and where the output parameters are set. Again, by simply highlighting the parameters, we find all their occurrences. DOgMA does not distinguish between the use and modification of a variable. However, seeing all occurrences will certainly provide sufficient information.

4.4.2 Sample Documentation

Unfortunately, many software systems are either undocumented at all or their documentation is incomplete and/or inconsistent. In this case we recommend maintenance programmers to document at least their changes made on this system. DOgMA is both suited for subsequent documentation an existing system and for documentation of changes made in a system.

Let’s assume that we introduce a new feature in the software system we are working on and that we want to describe the implementation of this feature in a new chapter. Introducing the new feature usually necessitates the insertion of code at several places in the existing code.

We might start writing the documentation text and the source code in parallel. But this is possible only when we know exactly what to do. Alternatively, we can first insert and test the source code (supposing it is not too extensive) and write down the documentation text afterwards, integrating it with the source code. This procedure—though despised by theorists—has proven very useful in practice. Figure 4.34 shows the content of a chapter that actually describes part of the addition of the menu entries Copy and Paste Identifiers of DOgMA itself.

Any global text styles that the user has defined for the source code are used in the documentation text, too. This is the reason why we can see identifier definitions in boldface. Similarly, the style for identifiers used within documentation text can be defined. In the example above these identifiers are also boldfaced.

4.2 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:
   …
case cIdentPaste:
   …
   break;
```

…
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 the first parameter. **PasteMark** is a virtual method of the class **HyperMark** which is overridden in the marks for identifiers (i.e., in **IdentDefMark** and **IdentUseMark**). In **HyperMark** we have to do nothing:

```
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 (…)
{
  return identDef?identDef->PasteMark(ht,from,to):0;
}
```

The definition finally does the pasting:

```
HyperMark *IdentDefMark::PasteMark (…)
{
  ...
}
```

Two cases have to be distinguished: In a documentation text we generate a **DocuIdentMark** and in a source text we generate a regular 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.

Fig. 4.34 Sample documentation

4.4.3 Reading Documentation

There are two possible ways to browse through a system for reading the documentation. One can either inspect the source code and look at the documentation, where available. Or one can read the documentation and look at the source code when the documentation does not provide enough information. (DOgMA facilitates comfortable work with documentation, but it cannot guarantee complete and high quality documentation. This remains the responsibility of the user.)

Similarly to navigating through classes, methods and files, the DOgMA allows easy access of chapters based on their interrelations. When reading a documentation chapter, one can easily inspect related chapters, i.e., subchapters, the superchapter, and sibling chapters.

There are several possibilities to branch to the source code because all source code parts used in documentation text offer the same hypertext features as their counterparts in the source code:

- One can select an identifier and jump to its definition. (The definition is always in the source code.)
- One can jump to the location from which a block of source code is taken/copied.
- Highlighting of identifiers is done in the documentation, too, so it is easy not only to find out where an identifier is used in the source code, but also to find its uses in the documentation text.

Whenever documentation text is available to the source code, the menu entry *Show Documentation* is enabled. This is the case when a piece of source (i.e., a class, method, file, or part thereof) has been used somewhere in the documentation. Issuing the menu command is sufficient to branch to the documentation.

4.4.4 Writing Documentation

Documentation should be written at the time of coding. Unfortunately, this is seldom done, either because of time constraints or because of the fact that code changes, which necessitates a time consuming change of the documentation, too. Therefore our tool supports both cases: writing documentation for already existing source code and writing documentation together with writing the source code.

The source code for fulfilling a task is usually spread over several locations—even several files. This is especially true for object-oriented systems, where many methods are overridden in subclasses.

To write documentation for already existing source code, we create a new chapter, write down text and simply include identifiers and text parts from the source code. This is done by simply activating the (menu) commands for copying and pasting identifiers and texts.

Below we will briefly demonstrate the task of creating the sample documentation presented in Section 4.4.2. Let's assume that we create source code and documentation in parallel. It is convenient to open two hypertext windows and to use one for the source code and the second for the documentation text.

As the first step we create a new chapter (with the menu command *Insert Chapter…*) and start typing the text of the first paragraph (see Fig. 4.34). To integrate the source code identifier *SetClipMark*, we simply select this identifier somewhere in the source code, select the menu entry *Copy Identifier* (in the hypertext window with the source code), and paste the identifier by choosing *Paste Identifier* (in the hypertext window with the documentation text).

In the next step we want to write source code. We simply insert the code at the proper place in the source code window and define a collapsed text part (menu entry *Collapse Text…*). We select this text part and choose the menu entry *Copy Text* and paste this text into the documentation by choosing *Paste Text*.

If additions are made to the source code at multiple locations in order to perform a single task (as is often necessary in object-oriented systems), then we recommend replacing all these additions with the same title. For example, we used the title 'Copy/Paste of Identifiers' for all source code pieces that were needed to accomplish this task. The same title can be used in different classes, methods and files.

These steps of writing documentation and source code text and copying and pasting identifiers and texts is repeated until the chapter is finished.

4.4.5 Modifying Documentation

The most obvious way to make changes in the documentation is by simply typing in documentation text and using the cut, copy and paste commands for both simple text and source code.

However, there are several possibilities to indirectly modify the documentation:

- An identifier has been changed in the source code:
  The system automatically changes this identifier at all occurrences both in the documentation text and in the source code.
- Any other changes have been made in the source code:
  As there exist links between source code and documentation text, these changes are made automatically in the documentation, too.
- *An identifier or a text part of the source code that is used in the documentation has been removed, let's say, with another text editor:*
  In the documentation we can see the name and the location of the identifier or the text part instead of the actual source code (see Fig. 4.35).

By using a different font for unresolved links to the source code (bold and italic in Fig. 4.44), the user immediately recognizes them and can see how the link was originally defined. In Fig. 4.44 we see that the text part "Copy/Paste Identifiers" in the method DoMenuCommand of the class HypertextDocument and the identifier SetText in the definition of the class Hypertext are not defined any longer.

4.2 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** .

@"HypertextDocument::DoMenuCommand;Copy/Paste Identifiers"

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

...

Fig. 4.35 Unresolved links to the source code

4.5 Parameterization

The usefulness of a software tool and its acceptance by users is highly influenced by the tool's ability to meet individual needs. Therefore, DOgMA offers several parameterization possibilities which are described in the following sections.

In the UNIX operating system, environment variables can be used for parameterization. With the *setenv* command, values can be assigned to these variables. For example, *setenv SIZE 20* assigns the value 20 to the environment variable SIZE.

4.5.1 Directory Paths

Large software systems consisting of subsystems are usually spread over several file directories. In order to enable an automatic reading of large systems the various directory paths are communicated to DOgMA by setting the variables INCLUDE1, INCLUDE2, etc. (see Fig. 4.36). With this mechanism different versions of a subsystem or even of the whole system can simply be loaded by setting the INCLUDE variables accordingly.

```
setenv INCLUDE1 $ET_DIR/src
setenv INCLUDE2 /usr/include
setenv INCLUDE3 Documentation
setenv INCLUDE4 $ET_DIR/src/SUNWINDOW
```

Fig. 4.36 Directory paths

If a file is included in a software system, then DOgMA looks for this file in the current directory. If it cannot be found there, then it looks in the path specified for INCLUDE1, etc.

4.5.2 Size of History

The browsing history is more important for novice programmers than for professionals. Therefore the number of browsing steps DOgMA remembers can be set by the user so as not to needlessly waste storage. This is done by setting the variable HISTORY_SIZE (see Fig. 4.37).

```
setenv HISTORY_SIZE 50
```

Fig. 4.37 Size of the history

4.5.3 Width of the Selection Lists

Depending on project conventions, the length of file, class and method names may vary. To avoid long names being truncated in the selection lists or space being wasted when the names are much shorter, the width of the selection lists can also be specified by the user.

```
setenv LIST_WIDTH 180
```

Fig. 4.38 Width of the selection lists

The horizontal size is specified in pixels by setting the variable LIST_WIDTH (see Fig. 4.38). The standard size that we have used for our screen dumps (see Fig. 4.14) is 180 pixels.

4.5.4 Text Templates

Almost every programmer prefers a different indentation philosophy, and style conventions vary from project to project. When the user adds a new class or method, DOgMA provides a textual template which can be completed. In order to meet individual needs, these templates can be customized. This is done by setting the following variables:

- DEF_FILE: the text of a definition file (file containing class definitions only)
- IMPL_FILE: the text of an implementation file (file containing method implementations)
- DOCU_FILE: the text of a documentation file (file containing documentation chapters)
- CLASS: the text of a class definition without a superclass (root class)
- SUBCLASS: the text of a subclass definition
- CONSTRUCTOR: the text of a constructor method
- DESTRUCTOR: the text of a destructor method
- METHOD: the text of any other method
- CHAPTER: the text of a documentation chapter

DOgMA replaces any %s in these texts with the actual name of the file, chapter, class or method. Figure 4.39 contains an example of the definition of the templates. For better readability the variable names are shown in boldface.

```
setenv DEF_FILE '#ifndef %s_First\
#define %s_First\
#include "xxx.h"\
\
#endif %s_First'
setenv IMPL_FILE '//$%s$\
#include "%s.h"\
\
MetaImpl (%s,(I_O(xxx)));\
'
setenv DOCU_FILE 'empty documentation file'
setenv CLASS 'class %s: {\
protected:\
public:\
   MetaDef(%s);\
   %s();\
   ~%s();\
};'
setenv SUBCLASS 'class %s: public %s {\
protected:\
public:\
   MetaDef(%s);\
   %s();\
   ~%s();\
};'
setenv CONSTRUCTOR 'void %s::%s()\
{\
}'
setenv DESCRUCTOR 'void %s::~%s()\
{\
}'
setenv METHOD 'void %s::%s()\
{\
}'
setenv CHAPTER '%s\
{\
}'
```

Fig. 4.39 Text templates

Note that the backslash is used to continue a text in the next line. Thus parameterization templates for other programming languages can easily be defined as well.

4.6 Implementation Aspects

DOgMA was implemented in an object-oriented manner with C++ under UNIX on a SUN workstation using the application framework ET++ ([Wei88], [Wei89]). Space limitations prohibit a detailed description of the implementation; instead we concentrate on the most important aspects.

Section 4.6.1 provides an overview of the overall structure of DOgMA's implementation. In Section 4.6.2 the interface between the language-dependent and the language-independent part is given. This interface is essential for supporting any other programming languages with DOgMA. Section 4.6.3 describes the language-dependent part of the C++ version of DOgMA by outlining the most important aspects in the static analysis of C++ programs. This section is also intended to give the reader an impression of the effort needed for adapting DOgMA to a new programming language. The use of an application framework greatly influences the implementation of a software system. Thus the impact of using ET++ in developing DOgMA is described in Section 4.6.4. Finally, current restrictions and possible improvements are sketched in Section 4.6.5.

4.6.1 Overall Structure of the System

The tool is clearly separated into two parts: a language-independent hypertext browser (see [Con87]) and a language-dependent static analyzer (C++ and Modula-2) that collects needed information about the inspected program.

Language-Independent Hypertext Browser

The language-independent hypertext browser controls the user interface and manages the following information about a software system:

- text pieces (e.g., class descriptions, method implementations)
- any relations among these text pieces for browsing (e.g., inheritance, include relations, methods of a class)
- classification of text parts (e.g., keywords, comments, identifiers)
- relations among identifiers (the definition of an identifier and its uses)
- additional information (e.g., inheritance path, file location)

Based on this (language-independent) information, the tool manages easy browsing through a software system.

Language-Dependent Source Code Parser

The language-dependent parser analyzes the source code, cuts it into small pieces of text (classes and methods), and passes information to the hypertext browser, e.g.:

- the definition of an identifier
- the use of an identifier

- any keyword
- any comment
- any inheritance relation
- the location of files (directory path)

Documentation Text Parser

Documentation text, like source code, is stored as text. Links to the source code are marked with a special character and consist of the name of the source code part (e.g., ‘class HyperText’, ‘HyperText::Cut’) followed by the name of the identifier or the text part (e.g., ‘class HyperText;marks’, ‘HyperTextDocument::DoMenuCommand;Copy/Paste of Identifiers‘).

The documentation text parser has to establish the links among the chapters and between documentation text and source code parts and identifiers.

4.6.2 Interface between the Hypertext Browser and the Parser

In order to be able to use DOgMA for other programming languages, the interface between the language-independent hypertext system and the language-dependent source code parser is crucial. An important part of DOgMA’s class hierarchy is shown in Fig. 4.40, whereby classes of the application framework ET++ are drawn in grey boxes. The only language-dependent class, CProject, is shown with a thick border.

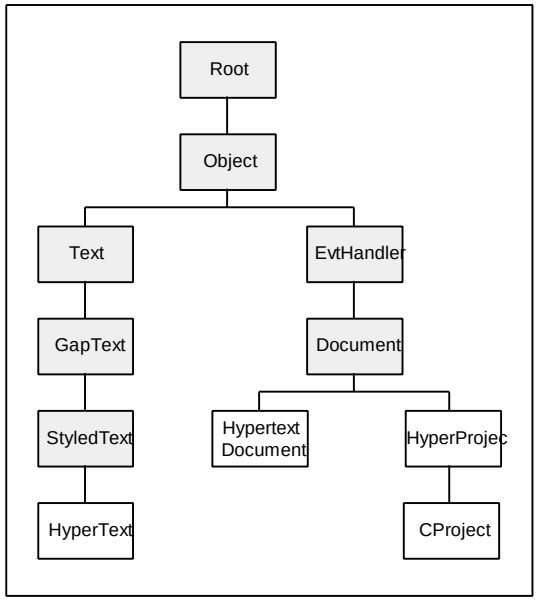

Fig. 4.40 Class hierarchy

The most important classes of DOgMA’s implementation are HyperText, HyperTextDocument and HyperProject. The class HyperText provides text editing capabilities (inherited from the ET++ class GapText), various fonts and styles (inherited from the ET++ class StyledText) and hypertext features (implemented in the class HyperText itself).

The class HyperTextDocument provides the usual document processing capabilities like loading and storing of files. The class HyperProject provides general information about a loaded software system. Additionally, it provides two methods, ReadProject and ParseProject. ReadProject is responsible for reading a software system and establishing global links (classes, methods, etc.). ParseProject has to do static analysis of the loaded system in order to get information about identifiers, comments, etc. This information is passed to the hypertext browser by sending messages to objects of class HyperText.

The methods ReadProject and ParseProject are responsible for both reading and parsing the source code and the documentation text. They are language-dependent; therefore they are overridden in a separate subclass (CProject in Fig. 4.40) which contains all language-dependent parts of the system. If DOgMA is to be adapted to another programming language, then the main task is writing a new subclass of HyperProject with the methods ReadProject and ParseProject.

4.6.3 Static Analysis of C++ Programs

C++ is an object-oriented superset of the programming language C [Str86]. We will present some details about the static analysis of C++ programs because the structure and the history of the language burden the development of tools for it.

The compilation of C++ usually consists of three parts: the C preprocessing (*cpp*), the transformation to C (*cfront*), and finally the compilation of the C program (*cc*) (see Fig. 4.41).

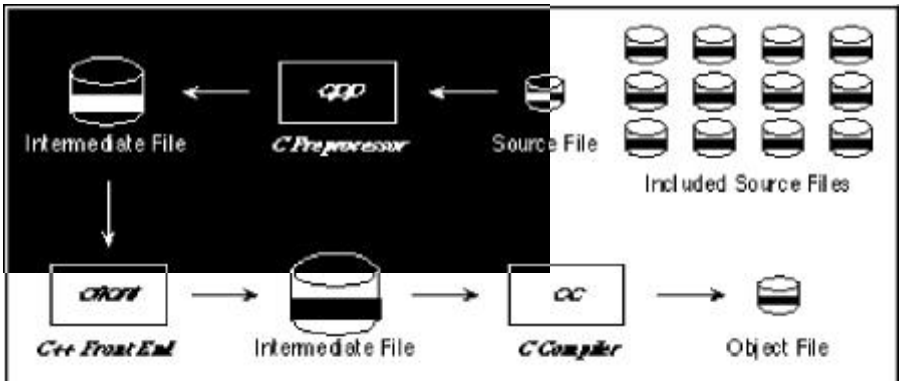

Fig. 4.41 C++ compilation

The *C Preprocessor (cpp)* first reads the source code and processes the preprocessor statements (lines beginning with a '#'). It includes other files, handles the definition of identifiers and replaces these identifiers in the subsequent text with their defined strings (with parameters), and skips parts of the text according to if-then-else-statements, which requires the evaluation of constant expressions.

The *C++ Front End (cfront)* parses the output of the preprocessor and generates a C program. To do this, a full syntactic and semantic analysis is necessary.

Finally, the *C Compiler (cc)* reads the output of the C++ front end and generates object code. In some implementations object code is generated directly; i.e., the transformation to C code is omitted.

In order to get the information needed for our hypertext browser, the syntactic and semantic analysis of *cfront* has to be carried out. This analysis cannot be done with the output of the preprocessor because the preprocessor generates a new intermediate source file and it would be impossible to determine exactly the definition and use of identifiers of the original source file. Therefore these two steps have to be integrated; i.e., the functions of *cpp* and *cfront* have to be carried out simultaneously.

Our language-dependent parser has to recognize and perform any preprocessor statements (include files, manage a symbol table of preprocessor-defined symbols, evaluate constant expressions in if-then-else statements and possibly skip lines). Whenever an identifier is read from the regular C++ code, we must check whether it is a preprocessor-defined identifier. In this case this identifier (and possibly an argument list) have to be replaced with the appropriate string and passed on for further analysis. Otherwise the hypertext browser has to be informed about an identifier definition or use. The same holds for keywords and comments.

4.6.4 Impact of Using an Application Framework

Reusability of existing code is a major advantage of the object-oriented programming paradigm. Thus predefined standard applications can easily be reused and extended flexibly. These standard applications are called application frameworks.

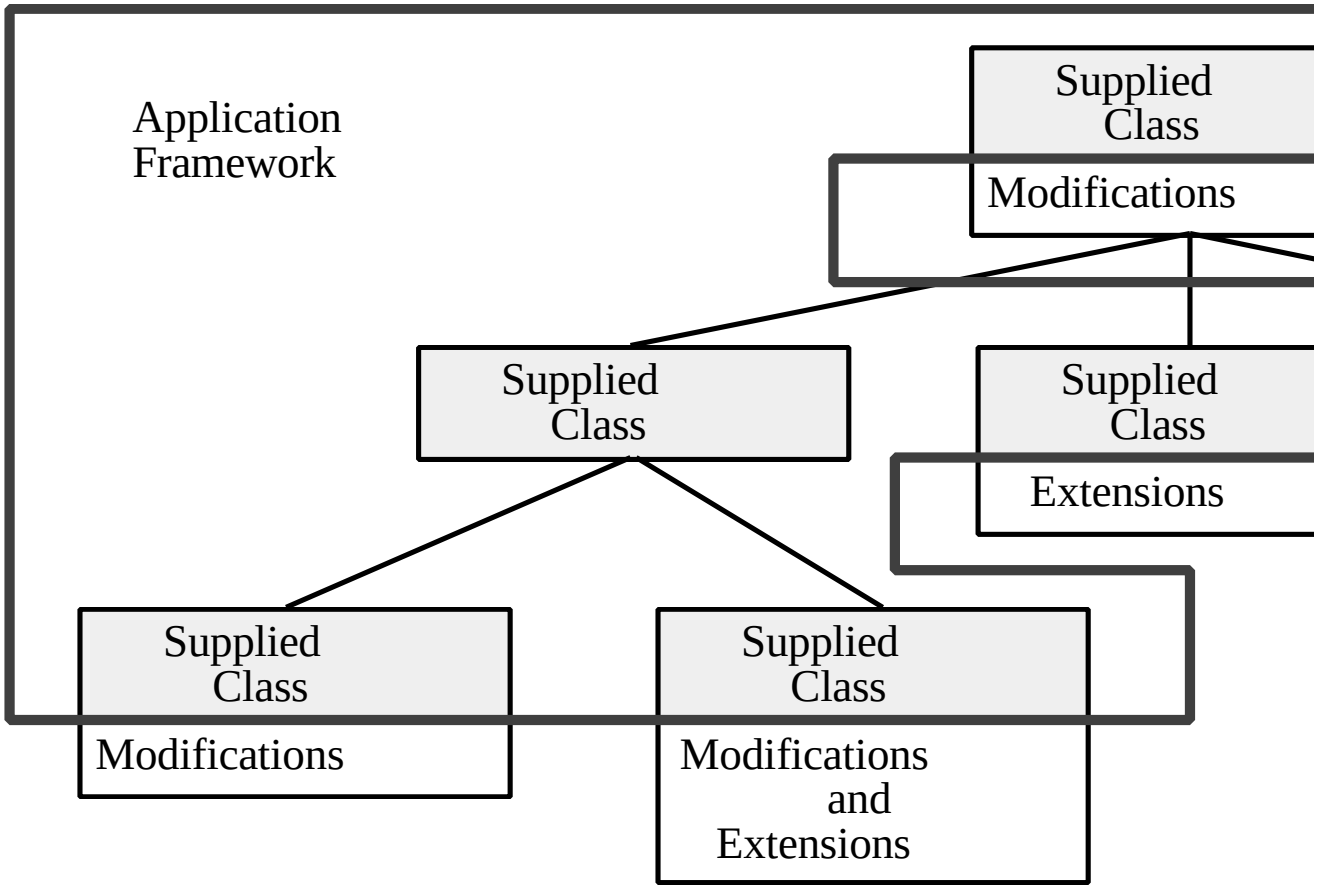

Fig. 4.42 An expanded application framework

The basic organization of an application designed with an application framework is shown in Fig. 4.42 (see also [Sch86]).

The activity of developing a new application is reduced to modifying existing classes by appending code to them (building subclasses) and to adding new classes.

DOgMA was implemented using the application framework ET++ ([Wei88], [Wei89]). ET++ integrates user interface building blocks, basic data structures, and support for object input/output. The main goals were to ease the development of highly interactive applications and to provide them with consistent user interfaces. DOgMA is a highly interactive application which benefits from ET++ in several ways. The following functionality is provided by ET++ and did not have to be taken into consideration explicitly when developing DOgMA:

- window resizing, window movements
- menu handling
- event handling
- basic text editing (cut, copy, paste)
- finding/changing text
- styled texts
- undoable commands

- scrolling of window contents
- display of lists
- file handling

This resulted in a considerable reduction of source code to be written and hence also development time. However, there were also some problems with using ET++:

- Due to the high complexity, it takes a significant amount of learning effort to use ET++. (This holds for other application frameworks as well.)
- There was no documentation available, which made it necessary to study the class implementations. (This is by no means desirable when using an application framework.)
- Sometimes a certain behavior of a class was undesirable, but it was very difficult to circumvent it because this was not intended by the designers of the framework. For example, the class *Document* incorporates a mechanism for determining whether a document has been changed. When the user closes a document, she/he is automatically asked whether the changes should be saved. Usually this built-in behavior is welcome because it relieves programmers of that task. However, DOgMA's document structure is more complex because it consists of several documents in the notion of ET++. Therefore, the built-in behavior was undesired and had to be turned off, which was rather cumbersome, as the class *Document* was not designed to support that.

However, the advantages outweighed the drawbacks by far. The benefits of using an application framework even increase when developing more applications with it because the learning effort vanishes.

4.6.5 The Implementation of Hypertext

In DOgMA hypertext nodes exist for classes, methods, files and chapters. For this purpose an abstract class *NodeItem* was introduced which was specialized for the various node kinds (see Fig. 4.43).

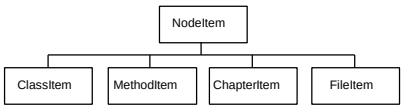

Fig. 4.43 Class hierarchy for hypertext nodes

The class *NodeItem* contains information that is peculiar for every hypertext node (e.g., a reference to the text of this node). The specialized classes contain references representing the various links defined for them (e.g., references to subclasses, superclasses).

The hypertext information for texts is stored in mark lists. Mark lists are lists of objects that contain additional information about a text, e.g., style information or hypertext information (see Fig. 4.44). Each object in a mark list refers to a portion of text that

starts at some position and extends for a certain length and contains additional information about the text portion it refers to. For example, an object marking the use of an identifier contains a reference to the object referring to the definition of this identifier. This references are not restricted to a single text; e.g., the use of an identifier can be in a different text than its definition.

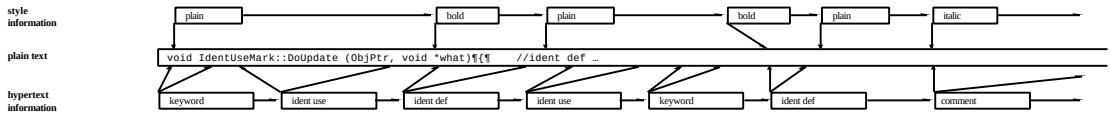

Fig. 4.44 Hypertext and style information as mark lists

When the user wants to highlight an identifier, the definition object of this identifier sends a message to all uses which enforce the style information to be updated correspondingly. Additionally, a counter in the corresponding node object is incremented. This counter is displayed in the selection lists and represents the occurrences of an identifier.

The positions stored in each mark object have to be updated whenever the user makes any changes in a text (deleting or inserting characters). However, this can be done fast enough so that the editing process is not distinctly slowed down.

4.6.6 Problems with the Text Structure of ET++

Source code must not exist in multiple editions when the user copies it into the documentation, but rather references have to be established to use the same text in several contexts (see Fig. 4.45). This avoids consistency problems and guarantees that any changes are immediately reflected anywhere a text is used.

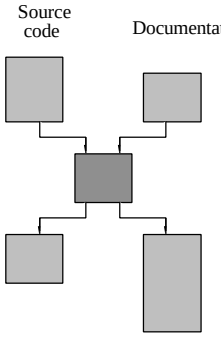

Fig. 4.45 Integration of source code and documentation

However, this is not possible with the text structure provided by ET++ classes, which model a text as an array of characters.

We decided to use the existing text classes for the first version of DOgMA instead of reimplementing them. In the sense of experimental prototyping, this approach was thought to provide useful information about the drawbacks of the existing classes. This

information can serve as a base for the design of a new implementation. The following drawbacks were encountered:

- Collapsing or expanding text parts causes the following steps to be executed: Cutting the text to be removed, pasting the textual counterpart and updating the style and hypertext information (removing the marks of the old text, inserting the marks of the new text and updating the positions of the other marks).

  With a new text implementation this procedure should be replaced by simply updating some references.

- Integrating source code and documentation requires duplicating text parts of the source code together with its mark information. To prevent consistency problems, we do not allow the user to edit source code within the documentation in the current version of DOgMA.

  This is the major drawback of the existing text classes because it even imposes restrictions on the user. In a new implementation it must be possible to make multiple references to a text without copying neither the plain text nor any marks.

The current version of DOgMA has proven very useful for both its users and its author in spite of the restriction that currently source code cannot be edited within documentation text. On the one hand, the author had the possibility to gather information and experience for the reimplementation of the text classes (which also requires a good knowledge of the existing classes). On the other hand, DOgMA was earlier available for users than it would have been without reusing the existing classes.

4.6.7 External Storage of Source Code

For compatibility reasons the source code has to be stored in its original form; i.e., no hypertext information is filed with it. This is necessary to use other tools like compilers and debuggers of which the functionality is not provided by our maintenance tool. Besides, a maintenance tool must be capable of processing existing source code which has been created with another programming tool. This becomes possible when using textual representation of the source code.

```
PullDownBar *HyperTextDocument::CreateMenuBar()
{
    ObjList *list= new ObjList;
    //@(:"Sun Menu"
    sunMenu= new PullDownMenu(new ImageItem(SunImage, Point(23)));
    sunMenu->AppendItems("About DOgMA", cABOUT, 0);
    list->Add(new PullDownItem(sunMenu));
    //@)
    //@(:"File Menu"
    fileMenu= new PullDownMenu("File");
    fileMenu->AppendItems
       ("Load File ...",    cLoadFile,
        "-",
        "Save File",        cSAVE,
        "Save File as ...", cSAVEAS,
        "-",
        "Spawn",            cSPAWN,
        "Close",            cCLOSE,
        "-",
        "Show Application Window", cSHOWAPPLWIN,
        0);
    list->Add(new PullDownItem(fileMenu));
    //@)
    …
    return new PullDownBar(this, list);
}
```

Fig. 4.46 External storage of collapsed text parts in C++

Storing the source code in its original textual form is possible because all the hypertext information is inherent to it. With a simple static analysis of the source code, the nodes (classes, methods, modules, procedures) and their links can be reestablished.

There is only one exception: collapsed text parts are not supported in programming languages. Therefore, we simply use comments to identify collapsed text parts and to name them. Figure 4.46 gives an example of a C++ source code with comments for the identification of collapsed text parts (see also Fig. 4.2).

The special character @ identifies comments for collapsed text parts. If the @ is followed by an opening parenthesis, a colon, and a string, then the succeeding text should be replaceable by this string. The end of the replaceable text is identified by a comment containing the special character @ and a closing parenthesis.

4.6.8 External Storage of Documentation

Saving the documentation in textual form also has many advantages due to compatibility reasons. First, the documentation can be processed with other tools as well, and second, already existing documentation text can easily be used with our maintenance tool.

```
chapter “Copy/Paste of Identifiers”:”Implementation”
{
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 @"class HyperProject;SetClipMark".

@"HyperTextDocument::DoMenuCommand&Copy/Paste Identifiers"

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

@"HyperText::PasteMark"

We just call the method @"HyperText::PasteMark;PasteMark" of the
@"class HyperMark;HyperMark" we get as the first parameter.
...
}
```

Fig. 4.47 External storage of documentation

In order to reestablish the information web, links to the source code are marked with a special character and consist of the name of the source code part (e.g., ‘class HyperText’, ‘HyperText::Cut’) followed by the name of the identifier or the text part (e.g., ‘class HyperText;marks’, ‘HyperTextDocument::DoMenuCommand;Copy/ Paste of Identifiers‘).

Figure 4.47 shows how the documentation chapter of Fig. 4.10 is stored in a text file. The keyword “chapter” is used to easily identify chapters when reading the documentation. It is followed by its name (in quotes) and an (optional) name of its superchapter. The special character @ is used to identify links to the source code. To simplify the process of recognizing documentation chapters, their content is externally enclosed by braces.

4.6.9 Current Restrictions and Possible Improvements

DOgMA has been implemented, but certain details have not yet been completed and are scheduled for inclusion in future improvements, e.g.:

- Incremental static analysis is not supported, i.e., when changes are made to a software system hypertext information cannot be updated correspondingly.
- All information about a software system is kept in main memory. The use of a database is being considered.
- Multiple inheritance is not supported.

- Graphic information representation (graph and tree browsers) is still missing, but should be easy to implement.
- Text processing features have to be improved.
- Documentation support is not yet complete (e.g., automatic numbering of chapters, generating a table of contents, printing).
- Search-and-replace operations function on the currently displayed text only. They must be applied to the whole software system.
- An initial setting of the global text styles should also be parameterizable. So far the initial values are set by DOgMA.
- DOgMA should explicitly call the user's attention to inconsistencies in the documentation (unresolved code links).
- Inclusion of graphics in the documentation is still missing.
- The source code cannot be edited within the documentation.
- It should be possible to interactively specify the width of the selection lists.

The main goal in developing DOgMA was to demonstrate possible improvements in the field of documentation and maintenance rather than to create a perfect tool. Nevertheless, DOgMA has still proved a useful support, especially in the comprehension process, in spite of the shortcomings mentioned above.

4.7 Measurements and Statistics

The part of the application framework ET++ used for DOgMA consists of about 150 classes. To implement DOgMA about 50 classes have been added with about 500 method implementations for the language-independent part. The definition of the parts of ET++ that were used (the definition files only) encompasses about 10.000 lines of code, DOgMA accounts for about additional 9.000 lines (class definitions and method implementations). The language-dependent part for the C++ parser is implemented in C and accounts for about 28,000 lines of code. An existing parser and preprocessor were adapted for that purpose.

Figure 4.48 presents some measurements that were made processing a subsystem of DOgMA itself. This subsystem consists of the whole source code of DOgMA without the language-dependent parser, which is implemented in pure C, and the definition files of the application framework ET++ which were used in implementing DOgMA. This was the typical configuration used by the author when working on DOgMA with DOgMA. The measurements were made on a Sun SPARC station 1+ with 8 MB RAM.

It has to be mentioned that it takes a while to load and parse the software system (40 seconds for loading and an additional 110 seconds for parsing). This is not that bad when considering the fact that compiling takes about 3 minutes. However, loading and parsing has to be done only once and the benefits of these 2.5 minutes of waiting can be seen in the other measurements. It hardly ever takes longer than a second to get any information about the loaded software system. And this is what counts, because the user

needs information fast and in an easy manner in order to promote continuity in her/his trains of thought.

Total number of files	140
Total number of classes	200
Total number of method implementations	500
Total number of lines of code	19,000
Size of object code	1.8 MB
Time needed for loading the system	40 sec
Time needed for parsing the system	110 sec
Main memory needed (virtual)	11.5 MB
Real main memory used	2.5 MB
Getting information about an identifier	0.5 sec
Highlighting an identifier	0.5 sec
Highlighting instance variables of a class	1 sec
Changing the name of an identifier	2 sec
Branching to the definition of an identifier	1 sec
Branching to the documentation of a class	1 sec

Fig. 4.48 Measurements and statistics

5. Comparison with Similar Tools

In this chapter we introduce tools similar to DOgMA. In order to determine tools that are similar to DOgMA, a classification of tools is presented in Section 5.1. The categories considered for comparison are browsers, hypertext systems, and literate programming systems, which are presented in the Sections 5.2, 5.3, and 5.4, respectively. The browsing, hypertext and literate programming features of DOgMA are also summarized and compared with the other tools in the corresponding subsections. Finally, Section 5.5 provides a summarizing comparison of all these features of the presented tools.

5.1 Classification of Tools

A software tool is a computer program used for developing, testing, analyzing or maintaining another computer program or its documentation (see Chapter 2). Typical categories of software tools are [Mar83]:

- system design tools
- analysis tools
- programming tools
- testing and debugging tools
- documentation tools
- maintenance tools

As software maintenance is considered as continued development (see Chapter 2), all tools used for development are also useful and necessary for maintenance. However, the term maintenance tools is used in this context for tools that are primarily useful in the process of software maintenance.

Documentation tools include both tools for preparing documentation (used mainly during development) and tools for automatic generation of documentation (applied to existing systems). The situation is similar with testing and debugging tools. They are needed during initial development as well as (even more) during the maintenance phase.

There are numerous ways to classify the types of tools. Usually they are classified by function because functional areas are easy to identify. The delineation of the various categories of software tools is sometimes as arbitrary as is the strict separation of development and maintenance. Nevertheless, the assignment of tools to certain categories is helpful for obtaining a general view of them.

The following (functional) classification divides software maintenance tools into 11 categories and was made by the Federal Software Management Support Center at the U.S. General Services Administration ([Rom86], [Par87b]):

1) test coverage monitors
2) translators
3) reformatters
4) data standardization tools
5) cross reference analyzers
6) documentation tools
7) source comparators
8) file comparators
9) data manipulation tools
10) restructurers
11) code analyzers

In order to make maintenance tools accessible for a larger audience, the Federal Software Management Support Center has developed a programmer's workbench which consists of 10 tools to assist maintenance programmers. They fall into the categories presented above. One tool covers two categories; therefore 11 categories are covered by only 10 tools (see also [Abi88]).

A more exhaustive classification with 23 categories and assignment of commercially available tools to these categories can be found in [Zve89]. Unfortunately, the description of the tools is very poor and somewhat unsystematic.

As the above classifications do not cover all maintenance activities and do not consider new technologies, we will introduce one of our own. Our classification will be based on the activities presented in Section 3.2.2.

1) *Browsing Tools*
   A browser presents a hierarchical index to information, where the index is an aid for the user to quickly obtain needed information (see Chapter 3.2.2). Browsing tools help maintenance programmers primarily to get familiar with a software system.

2) *Hypertext Tools*
   Hypertext tools support nonsequential reading and writing. This approach is extremely useful for software documentation. Hypertext tools are the new generation of documentation tools (for both user and system documentation). The concepts of hypertext can also be applied to source code and—even more important—it can be used for the integration of source code and documentation.

3) *Literate Programming Tools*
   Literate programming tools can be regarded as documentation tools, but a separate category is justified because of the importance of the concept of literate program-

ming and the fact that these tools support not only documentation but also the process of designing and coding.

4) *Visual Programming Tools*
Visual programming tools use graphics in order to support the programming, debugging and understanding of software systems. These are tools for visual programming, program visualization, program animation, and programming by example.

5) *Documentation Tools*
Documentation tools support the creation and maintenance of documentation. Redocumentation is a subarea of reengineering [Chi90]. So documentation tools that automatically produce documentation from the source code are reengineering tools as well. But we count them among documentation tools in spite of this fact.

6) *Debugging Tools*
Debugging tools support the monitoring of program execution. They not only help to find errors but are also very useful for understanding programs.

7) *Testing Tools*
Testing tools are considered a superset of test coverage monitors. In this category we also include any other test tools like test driver tools, test planning tools, test data generators, etc.

8) *Reengineering Tools*
Reengineering has emerged as an independent discipline aimed at the decomposition and comprehension of existing source code. Restructuring is part of reengineering, therefore we replace the category restructurers by reengineering tools. We also count reformatters in this category because reformatting can be regarded as a simple kind of restructuring which enhances the readability of a program.

9) *Configuration Management Tools*
Configuration management is a major problem for managing and maintaining large software products which do not consist of a single version of a system but encompass a set of similar configurations.

10) *Software Engineering Environments*
The pipe dream of any software engineer is an integration of the tools mentioned above to a compatible toolset and also the integration of the concepts found in various tools to become part of a powerful software engineering environment.

11) *Other Tools*
All other tools (e.g., the ones mentioned by the Federal Software Management Support Center but not taken into consideration by our classification so far) will be collected in this category, i.e., translators, source and file comparators, data standardization and manipulation tools, and code analyzers.

DOgMA can be assigned to the first three categories: browsing tools, hypertext tools and literate programming tools. Therefore, in the following sections we will systematically contrast browsing, hypertext and literate programming features of various tools.

A true comparison of software tools could be accomplished by using the tools in parallel for some time, thus finding out the differences (see [Smn85]). However, such a comparison using multiple maintenance teams would cost too much. Besides, one test would not be enough, and so far nobody knows which variables to control. Some minor comparisons have been made that way to test structuring engines (i.e., restructuring tools) for Cobol [Smn86], but the tests were not considered very conclusive: they did not last long enough and the changes performed were too trivial.

5.2 Browsers

For comparison with DOgMA we chose the Smalltalk-80 and the Smalltalk/V browsers, the Omega browser, and the browsers available with the application framework ET++ (which was used to implement DOgMA). The Smalltalk browsers are considered for comparison because their early development was a major contribution in the evolution of browsing systems. Besides their functionality and comfortable user interfaces can still be compared with newer browsers. Omega is a new system which provides interesting and promising browsing capabilities. Finally, ET++ offers the inspection of various dynamic aspects which are very important for program comprehension but not considered by other browsing systems.

5.2.1 The Smalltalk-80 Browser

The Smalltalk-80 browser (see [Gol84], [Lal90]) presents categories to organize classes within the system and to organize methods within a class (see Fig. 5.1).

The browser contains five panes and a switch between instance and class. The panes depend on each other; i.e., making a selection in a pane specifies the information to be displayed in another pane. The switch is used to distinguish between methods that are sent to the instances of a class and methods that are sent to the class itself. (For details concerning classes and their instances see [Gol85].)

The classes are organized according to categories which can be seen in the first pane (class categories). Choosing an item in this pane causes the classes of the chosen category to be displayed in the next pane. Selecting a class causes the display of its message categories in the message categories pane. Choosing a message category causes the appropriate messages to be displayed in the message pane.

In the text pane the source code of the appropriate classes and messages are displayed depending on the selections made above. Selecting a category provides a template for creating a new class or message.

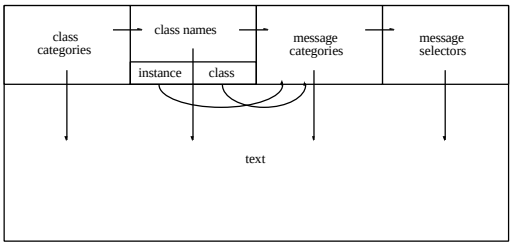

Fig. 5.1 Structure of the Smalltalk-80 system browser

There is much useful information about a class that can be retrieved with a Smalltalk-80 browser:

- a description of classes and methods (comments)
- a classification of classes and methods
- access to classes that implement a particular message
- access to methods that send or implement a particular message
- access to methods that reference a particular variable or literal

It is also possible to browse a subset of the system simply by spawning a new browser which gives access only to a certain class category, a certain class, a certain method category or even a certain method. With a class hierarchy browser the superclass and the subclasses of a class can be inspected comfortably.

Rating

The Smalltalk-80 browser is a powerful tool in coping with the complex structure of (object-oriented) Smalltalk systems. The classification of classes and methods and the possibility to easily get the answers to various questions concerning the structure of a software system are useful aids in program comprehension. Unfortunately, browsing is limited by being based only on the relations among classes, methods and their categories. The documentation is also neglected, except the rudimentary facility of using comments to describe classes and methods and to present them to the user on demand.

5.2.2 The Smalltalk/V Browser

The Smalltalk/V environment (see [Dig89]) provides a class hierarchy browser that shows the interrelationships of the classes within the system (see Fig. 5.2).

The class hierarchy list appears in the upper left pane, where the classes of the system are presented in a hierarchical order. The instance/class radio buttons are used to select between the display of instance or class methods and variables. In the center pane the variables of the selected class and its superclasses are displayed and the message pane shows the methods of a class.

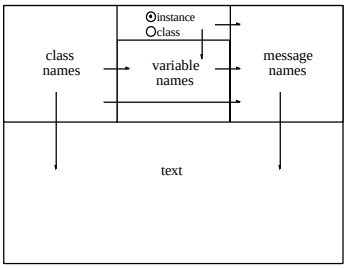

Fig. 5.2 Structure of the Smalltalk/V class hierarchy browser

If a variable is selected, then the list of methods is reduced to the methods which reference the selected variable. Like the Smalltalk-80 system, the Smalltalk/V system helps the user by answering a lot of questions about the structure of the system, for example:

- Senders: Which methods send a specific message?
- Local senders: Which methods of the current class and its subclasses send a specific message?
- Implementors: Which classes implement a specific message?
- Local implementors: Which classes of the current class and its subclasses implement a specific message?
- Which methods change the value of a specific variable?
- Which methods use the value of a specific variable?
- Which methods change or use the value of a specific variable?

Rating

The Smalltalk/V browser is very similar to the Smalltalk-80 browser, i.e., powerful browsing capabilities are degraded by neglected documentation. Again the Smalltalk/V browser does not provide any hypertext capabilities but, nevertheless, supports comfortable browsing features enabling nonsequential reading of a software system. Smalltalk/V does not offer class and method categories, but instead also bases its browsing on variables. This helps in finding the occurrences of variables, which is very useful in the maintenance process.

5.2.3 ET++ Browsers

The application framework ET++ is provided with browsers for programmers using the framework. As C++ is the implementation language of ET++, the browsers support C++, too. However, there is a major difference between ET++ browsers and other browsers, because ET++ browsers are not tools that can be started to inspect any software system, but rather they are available for a system that has been implemented with ET++ and are available only when the application is running. Thus it is not possible to inspect any software system written in C++. But the browsers have the advantage that they partially represent the object structure of the running ET++ application, which is a major advantage for understanding a system.

Four kinds of browsers are supported:

- *Source browser*
  for browsing through the source code and getting important information about it
- *Hierarchy browser*
  for graphically displaying the inheritance hierarchy and other important information
- *Structure browser*
  for graphically displaying the object structure of the running application together with useful information about the relationships of these objects
- *Inspector*
  for inspecting the objects of the running application by displaying the values of their instance variables

The source browser (see Fig. 5.3) provides a list of all classes (upper left pane). Selecting one of the classes causes this class to be displayed with all its superclasses and methods in the center pane. The methods list can be reduced depending on whether the methods are public, protected or private (see [Str86]). In the third pane methods can be displayed that implement, override or inherit a specific message. The text pane does not display classes and methods, but rather shows the contents of complete files, but it is automatically positioned to the corresponding class or method chosen.

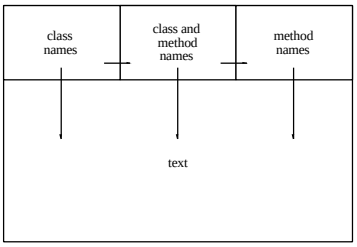

Fig. 5.3 Structure of the ET++ source browser

The hierarchy browser is used to graphically display the inheritance, client and member relationships of a system. Thus it is easy to find out where in the system a specific class is used (client) and which other classes a specific class uses for its instance variables (members). This information is displayed graphically using different colors.

The structure browser graphically represents all objects of a running application (i.e., the dynamic incarnations of classes) and shows different relationships among these objects, e.g., to which other objects a specific object points, by which objects a specific object is referenced.

The inspector allows a detailed inspection of any object of the running application (see Fig. 5.4). In the upper left pane a list of all classes with the corresponding number of incarnated objects is displayed. If a class is selected, then all objects of this class are displayed in the center pane. Selecting any object causes this object to be displayed in the lower pane; i.e., all the instance variables of the corresponding class and its superclass are shown together with their values. The upper right pane can be used to show objects that reference a specific object.

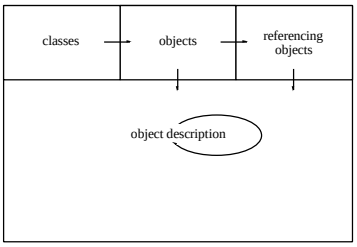

Fig. 5.4 Structure of the ET++ inspector

Rating

ET++ browsers are especially useful for maintenance programmers because they graphically display information and they also present dynamic aspects of a software system. This is an enormous help in program understanding. However, ET++ browsers lack the possibility to inspect an arbitrary software system (written in C++). They are available only for running applications that were implemented using the application framework ET++. Similar to the Smalltalk browsers, there is no documentation or hypertext support.

5.2.4 Omega Browsers

Omega is an object-oriented programming language which uses the concept of prototypes instead of classes (see [Bla91]). The Omega system is highly interactive and provides a type browser and an object editor. Types (as used in Omega) and classes (as used in Smalltalk and C++) are somewhat different, but for our purpose this difference is not significant. In the following context a reader not familiar with prototypes can regard types as classes.

The type browser shows the hierarchy of all types of an Omega software system. Selecting one of them opens an object editor (see Fig. 5.5).

Depending on the selections made in the buttons of the upper row of the object editor, a list of either variables or methods is displayed below (including all the inherited ones). Both the methods and variables are shown with additional iconic information about location, visibility and override status. Thus the user can see at a glance whether a method is implemented only once (in the current inheritance path), overrides a method or is itself overridden.

The text of methods is specified and displayed in a separate text window. Types and variables are fully specified with the object editor (without typing any text).

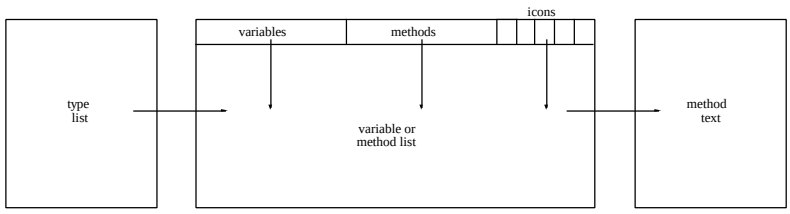

Fig. 5.5 Structure of Omega's browsing facilities

Rating

Omega is a new programming language and system still under development. However, its browsing capabilities are very promising. Much information that is important during maintenance is provided by Omega in a most convenient way, e.g., the visibility of variables and methods and the fact whether and where methods are overridden in the inheritance hierarchy. Documentation support as well as any hypertext features are not considered so far.

5.2.5 Browsing Features of DOgMA

DOgMA's general browsing facility is based on classes, methods and files. Its structure can be seen in Fig. 5.6.

Similar to the other browsers presented above, DOgMA uses the interrelationships among classes, methods and files for browsing. One of DOgMA's strengths is its simple but powerful browsing facility of the inheritance information. In the inheritance information the user can immediately see how methods are overridden in the superclasses and can very easily inspect inherited methods. Additionally, browsing is possible based on the interrelations among identifiers and between documentation and source code (for details see Section 5.3).

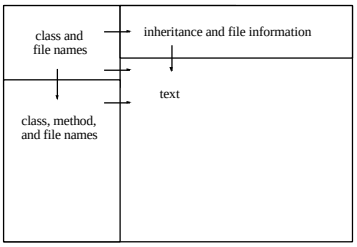

Fig. 5.6 Structure of DOgMA's browsing facility

Rating

DOgMA's browsing facilities take documentation fully into consideration and provide hypertext features which are integrated both in the source code and the documentation. This is considered a major improvement in the browsing power. Besides, comments are supported not only for classes and messages as, for example, in the Smalltalk-80 system, but for any identifier of a software system.

5.2.6 Comparison of Browsing Features

The comparison is based on the following features: browsing based on classes, methods, variables, files and/or chapters; multiple windows; categories; hypertext and graphic support.

- *Browsing based on classes, methods, variables, files and/or chapters*

  Browsers are based on logical units like classes and methods. This means that they provide lists with these items and thus offer easy and comfortable access to them.

Typical browsers for object-oriented software systems are based on classes and methods. Smalltalk/V displays variable names also. When the user selects one of these variables, the message (method) list is reduced to those using the selected variable. Omega offers a variable list as a comfortable way of specifying them, but without any browsing functionality. In ET++ the inspector displays existing objects of a running application where the variables of the object and thus the class can be seen. File-based browsing is supported only by DOgMA. However, in Smalltalk and Omega files do not have any significance. Besides, DOgMA is the only system that integrates documentation with the source code and therefore also provides browsing capabilities based on documentation chapters. The benefits of the variables found in Smalltalk/V are offered in DOgMA by means of the powerful hypertext concept (e.g., to find out which methods use a specific variable).

- *Multiple windows*

  Multiple windows are essential for any browsing system because it is crucial to inspect several parts of a software system simultaneously. Needless to say, all the presented browsers support this feature.

- *Categories*

  Categories are important for coping with complexity because they offer a possibility to reduce the information being displayed. A full category concept can be found only in the Smalltalk-80 system. ET++ and Omega provide a simplified classification mechanism in the possibility to reduce the displayed information depending on the visibility (public, private) and the location (shared, local: in Omega only). DOgMA's categories are restricted to implemented classes, implementation files and documentation files (see Section 5.3).

  The reason why DOgMA does not fully provide categories lies in the fact that the programming language C++, which does not have any categories, is supported. They would have to be introduced artificially (possibly similar to collapsed text parts by using special comments) but would never be available for existing software systems.

- *Hypertext*

  The hypertext concept offers very powerful browsing capabilities in addition to the mere display and selection possibility in ordinary lists. This concept is integrated only in DOgMA.

- *Graphic support*

  Graphics ease the understanding of the logical structure of a software system. ET++ graphically displays the inheritance hierarchy and the object structure. Omega provides a graphic representation of the inheritance. Graphic support in DOgMA is not implemented so far, but planned for future versions.

Figure 5.7 summarizes our comparison of the browsing features. All presented browsing systems are rather powerful. DOgMA is distinguished from these systems by its

support of hypertext features (which enhances browsing essentially) and by its incorporating the documentation.

	Smalltalk-80	**Small-talk/V**	**ET++**	**Omega**	**DOgMA**
classes	yes	yes	yes	yes	yes
methods	yes	yes	yes	yes	yes
variables	no	yes	no	no	no
files	—	—	no	—	yes
chapters	—	—	—	—	yes
multiple windows	yes	yes	yes	yes	yes
categories	yes	no	yes[1)]	yes[1)]	yes[1)]
hypertext	no	no	no	no	yes
graphics	no	no	yes	yes	no

Fig. 5.7 Comparison of browsing features

1) to some extent only

5.3 Hypertext Systems

The following hypertext systems will be compared with DOgMA: Dynamic Design, DIF, Guide and She. Dynamic Design and DIF were chosen as representatives of software engineering environments with inherent hypertext capabilities. Guide and She serve as examples of simple hypertext editors. Guide is a commercially available general purpose system, whereas She is not marketed and was especially designed for the manipulation of source code.

5.3.1 Dynamic Design

Dynamic Design, a CASE environment developed at Tektronix, is C-based and administers project components in what is called a Hypertext Abstract Machine (HAM, see [Big87], [Big88]). The project components include all documents generated during the software life cycle process.

Hypertext nodes hold project components (text, graphics, object code, etc.). Links can either point to an entire node or to any part thereof, e.g., to an identifier in the source code. Both links and nodes have attributes. A node identifies the project component it contains and a link indicates the type of relation that is provided by the link. Links may, for example, exist between specification and source code or between a module with a variable and the module with the definition of this variable.

Using attributes and values for nodes and links extends the usefulness of hypertext technology. Attributes identify and categorize nodes and links, which can be used to locate or filter information in query operations.

The concept of contexts entails the collection and partitioning of nodes and links into sets. Contexts offer the grouping of common nodes and links into subgraphs, which makes it possible to support version trees, configuration management, and local workspaces for different programmers (which may or may not overlap).

Rating

Dynamic Design fully supports both hypertext technology with the concepts of contexts and attributes, and the integration of source code and documentation. But despite this integration, Dynamic Design cannot be regarded as a literate programming environment because the interconnection of source code and documentation is too loose. The fact that the linking process is not automated has also to be considered a drawback.

5.3.2 DIF—Documents Integration Facility

DIF is the acronym for Documents Integration Facility, a hypertext system to integrate and manage documents of the software life cycle. The system was developed at the University of Southern California (and presented in a multitude of papers, e.g., [Gar87], [Gar88a], [Gar88b]). Segments of the documents are considered to be the nodes and are stored in files, whereas the links between the nodes, i.e., the relationships between the document segments, are stored in a relational database.

Through the integration with several software tools, an integrated software engineering environment is made available. For example, DIF provides revision management facilities by means of using RCS [Tic85] und presents a software system graphically by using a system visualizer.

Additionally, DIF supports the definition of forms and basic templates, which ensure that all projects have the same document structure. Thus the software engineer only has to fill in the forms rather than also defining the structure. Operational links can be defined between source code and object code; activating these links results in the execution of the code.

Rating

DIF also fully supports hypertext technology and integrates source code and documentation. But again it cannot be regarded as a literate programming environment due to the same reasons as mentioned for Dynamic Design, and it does not automate the linking process.

5.3.3 Guide

Guide is a text editor from OWL International that allows the creation of hypertext-type documents called guidelines on Apple Macintosh and IBM PC (see [Bro86],

[Her87]). A guideline is a mixture of text and graphics which can contain buttons. Buttons provide links to hidden text and graphics. The following buttons are supported:

- *Replacement buttons*

  Replacement buttons afford the ability to expand or collapse text and graphic pieces, thus hiding a certain amount of information and revealing it on demand.

- *Note buttons*

  Note buttons provide additional information about an item. Activating a note button displays this information in a pop-up window.

- *Reference buttons*

  Reference buttons point to another part of the same or another guideline document. They allow nonlinear branching through a guideline.

A powerful history function offers backtracking and prevents the user from getting lost in a complex information web. The various buttons in a guideline document can be recognized by means of different styles, which can arbitrarily be selected by the user.

Rating

Guide is a nice hypertext system, but unfortunately it can neither be applied to source code nor can it be used for the integration of source code and documentation. Its use is limited to pure documentation because hypertext information is stored with text, which makes this text unusable for compilers. However, it has to be mentioned that the primary goal for developing Guide was to support nonsequential documentation texts, e.g., help systems.

5.3.4 She—A Simple Hypertext Editor

She is the abbreviation for simple hypertext editor, which was developed by H. Mössenböck at the Eidgenössische Technische Hochschule (ETH) in Zürich [Mös90]. The editor is intended for manipulating source code files, but it is language-independent and can be used for any text files. She provides the following hypertext features:

- *Folding*

  It is possible to replace certain pieces of text with other text pieces and exchange them with a mouse click. This enables users to hide details of a program by replacing any text (e.g., statement sequences, complicated expressions, extensive comments) with a meaningful name. The user can expand or collapse the text pieces and therefore inspect the program at various levels of detail. A simple but powerful application of this feature is to keep the different steps of the design process when developing by stepwise refinement.

- *Annotating*

  Annotating is similar to folding. The difference is that the associated text is not replaced by another text, but rather a small pop-up window is displayed showing this

text. This feature can be used to explain complicated parts of a program without inflating the source code with comments.

- *Linking*

  In the sense of hypertext, different locations in a program text can be linked together even if they are in different files. This makes it easy to browse through a system, for example, from the use of an identifier to its definition or from the source code to the corresponding documentation.

These features prove especially powerful because the She editor provides the possibility to use global annotations and links. Instead of annotating and linking all identifiers by hand, the system remembers the information in a global dictionary which makes it available for all occurrences of identifiers (even if they are entered afterwards). Unfortunately, with this technique we cannot distinguish between identifiers with identical names but different scopes.

Additionally, She provides different fonts and—although She is language-independent—an extension has been made to automatically establish folding, annotating and linking features in an Oberon program text, which is especially useful for the creator of She, who primarily uses this language.

Rating

She can be regarded as a very useful improvement over an ordinary text editor. The provided features—though simple—are a valuable addition which meets the needs of maintenance programmers to a high extent. However, different scopes of identifiers are not supported—which is especially important for large software systems—and documentation aspects are not considered.

5.3.5 Hypertext Features of DOgMA

Essentially DOgMA supports nearly all the hypertext features available in the tools presented above. However, there are some minor differences:

- *Links*

  We have to distinguish between links that are generated automatically and links that can be inserted manually. All links within the source code are generated automatically (e.g., from any identifier to its definition, from a class to its superclass). Links between documentation and source code have to be inserted manually (from identifiers in the documentation text to their definition in the source code, and from code pieces in the documentation to the corresponding parts in the source code).

- *Replacements*

  Similar to She and Guide, DOgMA allows pairwise replacement of text pieces, thus making it possible to hide details.

- *Notes*

  Notes cannot be inserted manually. However, for every identifier (e.g., class names, method names, variable names) in the source code there exists an automatic annotation providing information about declaration, inheritance and, if a comment is available, a verbal description of this identifier.

DOgMA also provides a powerful history mechanism. What is missing so far is the support of attributes (as found in Dynamic Design) and the concept of contexts because these features are incoherent with existing software systems.

Rating

In comparison with the other tools DOgMA's hypertext features together with the fully automated generation of the hypertext information provides much more support of the maintenance process. This is mainly due the the fact that as much information as possible is made available, and also due to the high integration of source code and documentation by taking up the idea of literate programming.

5.3.6 Comparison of Hypertext Features

The following hypertext features are considered for the comparison: links, replacements, notes, attributes, contexts, automatic hypertext generation and history functionality.

- *Links*

  Links are the essential concept of hypertext and are present in all systems.

- *Replacements*

  Replacements provide the possibility to replace text pieces with other text pieces and to exchange them on demand. This is also an essential hypertext concept for nonsequential reading because details can be hidden. Replacements, like links, are provided by all systems.

- *Notes*

  Assigning annotations to text pieces like replacements highly supports nonsequential reading. Again, annoying details are hidden and can be shown on demand. Notes are also provided by all systems. DOgMA automatically offers notes for any identifier in the source code with additional information (e.g., point of declaration).

- *Attributes*

  Attributes are very useful for information filtering. They are missing in Guide and She. DOgMA does not explicitly support attributes, but provides some categories in the selection lists, thus restricting the nodes being displayed in these lists. Only Dynamic Design provides lots of attributes applied to nodes and links. The reason for this (current) restriction in DOgMA is the fact that a full implementation of

attributes is not practicable without the use of a database. Using a database is being considered, but has not been implemented so far. The same holds for contexts.

- *Contexts*

  The collection and partitioning of nodes and links into sets is especially useful for multiperson design and documentation of large-scale software systems. Again, Guide, She and DOgMA do not support this concept.

- *Automatic hypertext generation*

  Usually hypertext information (links, replacements and notes) has to be input by hand. However, She provides a global dictionary and an extension for Oberon to generate this information automatically. DOgMA also establishes this information automatically.

- *History*

  The browsing history is an important aid in preventing the user from getting lost in a complex information web. Hypertext systems usually provide this information. Of the systems chosen for our comparison, only the She editor does not remember the browsing path.

Figure 5.8 summarizes our comparison of the hypertext features. Maintenance programmers have to work with already existing systems which (usually) were implemented without any hypertext system. DOgMA is the only system that gives high support for dealing with such systems because all links based on source code are generated automatically and thus help the maintenance crew to easily get a lot of important information about the existing code. She also provides some help for already existing source code, but, in this sense, is less powerful than DOgMA.

	Dynamic Design	**DIF**	**Guide**	**She**	**DOgMA**
links	yes	yes	yes	yes	yes
replacements	yes	yes	yes	yes	yes
notes	yes	yes	yes	yes	yes
attributes	yes	yes	no	no	no
contexts	yes	—	no	no	no
automatic generation	no	no	no	yes	yes
history	yes	yes	yes	no	yes

Fig. 5.8 Comparison of hypertext features

5.4 Literate Programming Systems

Literate programming systems support the integration of documentation text and source code. The following systems were chosen for comparison: WEB, the original literate programming system by Knuth; HSD, a structured method for literate programming; an environment for literate Smalltalk programming; and a proposal for an interactive environment for literate programming.

The WEB system was chosen because it is the original system and most of the other literate programming systems available are based on it. This holds for HSD as well, but HSD provides a major improvement in its hierarchical structuring of documents. The environment for literate Smalltalk programming was chosen to serve as an example of a system that supports an object-oriented programming language. Besides, it offers a matured user interface, which is missing in the other systems. Finally, a proposal for an interactive literate programming environment is presented which is intended to offer a major improvement to existing WEB-based systems.

5.4.1 The WEB System

The WEB system is the original literate programming system developed by the founder of literate programming, D.E. Knuth [Knu84]. The name WEB expresses the fact that a program consists of a web of ideas. WEB can be used to write literate Pascal programs. The system is a combination of both the programming language Pascal and the document formatting language TEX. When using WEB, one writes a literate program that serves as input for two simple tools (see Fig. 5.9). With the tool WEAVE and the TEX processor, a hardcopy documentation of the program can be generated. The TANGLE processor produces a Pascal source file that can be compiled, linked, and executed.

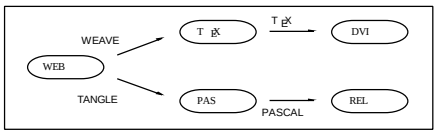

Fig. 5.9 The WEB system [Knu84]

Small examples of WEB programs can be found in [Knu84], [Ben86a], and [Ben86b]. Large programs are given in [Knu86a] and [Knu86b].

WEB programs are divided into sections that are numbered sequentially. Every section contains an optional commentary, optional macro definitions, and optional program text.

- *Commentary*

  The commentary contains explanations of the macro definitions and/or the program text that follows (e.g., a description of what the source code is supposed to do, preconditions, postconditions).

- *Macro Definitions*

  Macros are used to substitute some source code for an identifier (possibly with a parameter). This allows, for example, the declaration of constants or the definition of output-oriented commands like writing a string followed by a number. (This requires two statements in Pascal.)

- *Program Text*

  This part of a section contains the specification of the source code. The source code can be specified directly (in the form of Pascal statements) or indirectly as a reference to code that appears in other sections. Furthermore, the possibility to concatenate the program text of different sections provides the ability to present the code and text in any desired order.

Figure 5.10 shows a sample section containing commentary text, a macro definition, and program text.

4. The Global variables *M* and *N* have already been mentioned; we had better declared them. Other
global variables will be declared later.

```
    define M_max= 5000   {maximum value of M}

<Global variables 4>+
M: integer;    {size of the sample}
N: integer;        {size of the population}
```

See also Sections 7, 9, and 13.
This code is used in Section 3.

Fig. 5.10 Example of a WEB section [Ben86a]

The ‘<‘- and the ‘>‘-sign followed by a ‘+‘ embrace the name of the code of the section. The code can be used in other sections by specifying its name (see Fig. 5.11).

3. Here is an outline of the entire Pascal program:

```
program sample;
    var <Global variables 4>
    <The random number generation procedure 5>
    begin <The main program 6>
    end.
```

Fig. 5.11 Use of program text in another section [Ben86a]

The number following the name is generated automatically by the WEB system (the number of the section where a code is defined). Please note that the code is used (in Section 3) before its definition (in Section 4).

The program text of a section can be extended in later sections. This allows the introduction of further global variables in other sections (at places where they are used) (see Fig. 5.12).

```
<Global variables 4>++
size: integer;  {the number of elements in set S}
```

Fig. 5.12 Extension of program text [Ben86a]

For every section the system generates hints that tell the user where the code of this section is used and which sections refer to its code (see Fig. 5.10). Additionally, a table of contents and two indexes for identifiers and section names of the WEB program are generated automatically.

Although the generated documentation of the WEB system is appealing, it is cumbersome to input a WEB program. Figure 5.13 shows the text that is necessary to generate a WEB section.

Rating

In developing WEB, its author, D.E. Knuth, concentrated on the essentials of his idea of literate programming. Unfortunately, this resulted in a poor user interface. WEB cannot be used interactively, but rather works in batchmode. Users benefit from the printed output only. Additionally, users have to know the programming language, the document formatting language, and the syntax of WEB programs itself in order to write WEB programs. Another disadvantage of WEB is that it supports one level of abstraction only. Programmers not familiar with a software system might want to read the system's description section by section. But this is unsuitable for an experienced maintenance programmer who sometimes might want to inspect the source code alone.

```
ØThis program has no input, ...

Since there is no input, we declare ...

\[The program text below specifies the ‘‘expanded meaning’’ of
‘X2:Program to print $\ldots$ numbers\X; notice that it involves the top-
level descriptions of three other sections. When those top-level
descriptions are replaced by their expanded meanings, a syntactically
correct \PASCAL\ program will be obtained.\]

Ø<Program to print...Ø>=
program print_primes(output);
const Ø!m=1000;
Ø<Other constants of the programØ>Ø;
var Ø<Variables of the programØ>Ø;
begin Ø<Print the first |m| prime numbersØ>;
end.
```

Fig. 5.13 WEB code for generating a section [Knu84]

5.4.2 HSD—Hierarchical Structured Document

HSD is a programming tool for the automatic generation of source code and documentation from a document description which is hierarchically structured [Tun89]. Unlike

in the WEB system, documents in HSD are not organized linearly, but rather are composed of a hierarchically ordered collection of sections. To express the structure, HSD provides a graphic user interface and uses structured representations like trees. Thus a literate program is a combination of graphic and textual objects.

An HSD section consists of comments, program code, names of subsections, and commands to define and add program code to boxes. Boxes are used to group closely related declarations and/or statements. (In WEB each section defines one box in this sense.)

HSD programs are specified in a graphic document description language (GDDL). In GDDL each node consists of its name and the corresponding text. The HSD system shows the structure of a software system by displaying the names of the nodes on the screen. The associated text blocks are shown on demand. The documentation and the source code are generated by traversing the specification. Like in WEB, the specification can be decorated with typesetting commands.

Rating

HSD's advantage compared to the WEB system lies in its possibility to hierarchically structure documents and to graphically display and edit this structure. But HSD still uses typesetting commands instead of interactively providing features for processing text in wysiwyg manner.

5.4.3 An Environment for Literate Smalltalk Programming

A programming environment based on a variant of WEB was developed for Smalltalk [Ree89]. The main goals of this development were the analysis, design, implementation, and maintenance of object-oriented systems in the Smalltalk-80 language.

Fig. 5.14 Structure of the Galley editor

The data model consists of a directed graph (nodes and attributed relations) which can be processed with the Galley editor (see Fig. 5.14). The margin part at the left contains an iconic presentation of the document structure (e.g., text, figure, section, title page). It can be used to edit the structure. The right part shows the contents, where various editors (e.g., for text and graphics) are integrated.

Special nodes are used for class and method definitions. They provide a browsing facility for easy access to the Smalltalk program library. Besides, the Smalltalk compiler can be activated, which makes this code ready for execution. Even several versions of the same code may exist. In this case the version recompiled last is executed. Program fragments are always compilable Smalltalk code, i.e., classes or methods.

An alphabetically ordered index of classes and methods with page numbers (rather than section numbers like in the WEB system) is also provided automatically.

Rating

The Smalltalk environment provides the best user interface yet. Both the document structure and its contents can be edited easily and interactively without the need for any WEB or text formatting commands. The fact that only compilable code fragments can be used in the documentation without the possibility to split them further must be considered as a drawback. With this restriction it is not possible, for example, to describe a global variable (instance variable) together with the code parts where it is used. Furthermore, in object-oriented systems it is often the case that the implementation of a certain feature is spread over several source code units (see also the example in Section 5.4.2).

5.4.4 An Interactive Environment for Literate Programming

The WEB system and all the systems based on it work in some kind of a batch-mode. Brown [Bro90a] proposes an interactive environment for literate programming. However, only a prototype is available so far. The main idea is to provide the advantages of a printed WEB listing on the screen also (e.g., complete index, cross reference listing, print preview). Additionally, the idea of hypertext is incorporated by regarding modules as hypertext nodes and defining the following links:

- Module m uses module n.
- Both modules m and n contain the same index entry.
- Module m is extended by module n, e.g., by adding a global variable.
- Both modules m and n use the same variable.

Buttons are used to provide this hypertext functionality. For example, a button is offered for every module that uses the module that is currently displayed on the screen.

Other planned features are the integration of a debugger, graphic representations of various aspects of the system, and a personal preference database to parameterize the user interface.

Rating

The proposed interactive environment eliminates the shortcomings of existing WEB systems and additionally introduces hypertext features, which are an essential improvement for maintenance programmers. However, DOgMA’s powerful hypertext capabilities exceed them by far. Again the proposal is strongly based on WEB trying to bring the advantages of printed WEB documents to the screen.

5.4.5 Literate Programming Features of DOgMA

The literate programming features of DOgMA are described in detail in Chapter 5. As all available literate programming environments are based on the WEB system, we will

describe DOgMA's literate programming features—which are not based on WEB—by comparing them with the WEB system. The differences are as follows:

- *Document structure*

  The document structure is one of the major differences between WEB and DOgMA. In WEB the documentation and the source code are intermixed in a single document, whereas in DOgMA they are strictly separated and presented simultaneously to the user (see Fig. 5.15).

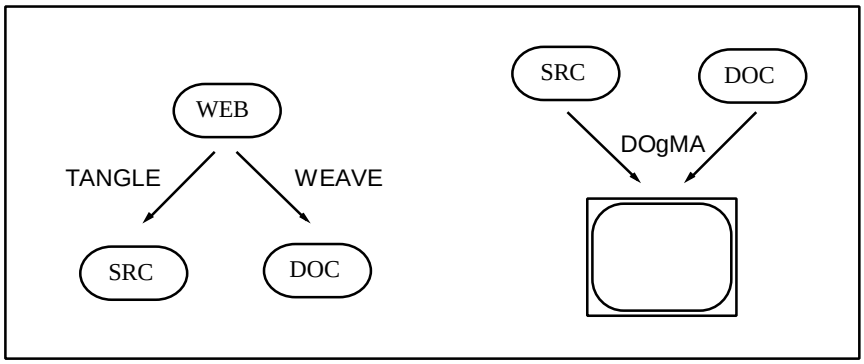

Fig. 5.15 The document structures of WEB and DOgMA

  The separation of source code and documentation text has some major advantages. First, it is possible to inspect the source code alone. This is useful when the documentation is already known and the user is familiar with the structure of the system. The documentation text of any source part can easily be inspected any time, though. Second, it enables us to process existing software systems which have not been developed with a literate programming style. Using DOgMA in this case does not provide the advantages of literate programming immediately, but it is possible to gradually create documentation. For example, the changes made during the maintenance phase can be documented in the sense of literate programming. Using the WEB system for existing source code would require making (bulky) changes in it.

- *Batch-orientation*

  WEB users write a mixture of different languages (programming language, text formatting language, WEB specification language). DOgMA users create documentation interactively; they have to know only the programming language.

- *Compile/Edit/Run*

  WEB-based literate programming systems are batch-oriented. They generate a source listing which is processed by the compiler. Compilation errors are reported with references to the source code listing. It remains the user's task to find the corresponding location in the WEB document. To speed up the turnaround cycle, WEB users tend to make modifications in the generated source code and transfer these changes to the WEB document afterwards (even though the generated source code is hardly readable). DOgMA users make this modification in the source code, and these modifications are automatically reflected in the documentation.

- *Documentation update*

  If any changes are made in the source code, then the corresponding changes have to be made in the documentation, too. This holds for both DOgMA and WEB. However, if source code identifiers are changed in DOgMA, then the documentation is updated automatically, even if the user is inspecting the source code alone.

- *System development*

  WEB forces its users to develop a software system together with its documentation. DOgMA does not force, but only supports this procedure. Additionally, parts of the source code can be developed in advance; the documentation can be added afterwards.

In DOgMA one has the possibility to read the documentation, i.e., documentation text intermixed with program code. The same holds for WEB documents. Additionally, in DOgMA one can view the source code alone and at any location one can easily branch to the corresponding documentation. In DOgMA one can write documentation (chapters) without source code and also source code without any documentation. This seems to be incidental, but is very important in practice. Writing source code without any documentation happens quite often due to pragmatical reasons. It is crucial for the acceptance of a tool that this is possible and that the documentation can be added subsequently in an easy and comfortable manner.

In DOgMA it is easy to (partially or completely) document existing program systems because one can write documentation text and include any part of the source code. In WEB this is hardly possible because one would have to make changes in the original source code, move code blocks around and insert documentation text within the source code.

Rating

Recapitulating, we can say that DOgMA provides a major improvement over WEB-based systems by physically separating source code and documentation. This eases subsequent documentation and the processing of existing source code. However, subsequent documentation is not promoted by DOgMA, but rather it is supported because of its importance in practice. DOgMA is interactive, does not prolong the turnaround cycle, and automatically makes consistent changes of source code identifiers in the documentation.

5.4.6 Comparison of Literate Programming Features

The literate programming comparison is based on the following features: hierarchical structure, delayed code, source view, automatic documentation update, subsequent documentation, what you see is what you get, hypertext, integrated compiler, and graphic support.

- *Hierarchical structure*

  WEB documents are organized linearly; i.e., they are comprised of a sequence of numbered sections. The hierarchical structures found in the other systems are better for capturing the logical structure of a software system.

- *Delayed code*

  One of the major advantages of literate programming systems is the ability to group closely related declarations and/or statements together, e.g., to declare a global variable where it is actually needed. Thus the code sequence presented to the user is different from the code sequence supplied to the compiler. This feature can be found in all systems except the Smalltalk system, where a program fragment must always be a compilable Smalltalk code (a class definition or a method definition).

- *Source view*

  WEB documents contain both documentation and source code. It is not possible to inspect the source code alone (except the generated output of the TANGLE tool, which is not very helpful). However, the pure source code is very important for the maintenance process if the programming staff is familiar with the software system already and does not want to be annoyed by the documentation all the time. DOgMA is the only tool that keeps source code and documentation text separate and therefore allows inspection of the source code alone. Please remember that consistency problems are solved in spite of this fact (see also the next entry).

- *Automatic documentation update*

  The reason why all WEB systems physically integrate source code and documentation is motivated in the consistency problem. So if any changes are made in the source code, the user immediately sees the corresponding documentation and is encouraged to change it accordingly. DOgMA does not support this physical integration, but guarantees that source code in the documentation is up-to-date and that any inconsistencies (unresolved code links) can easily be found by the user.

- *Subsequent documentation*

  Adapting existing source code to a literate program is rather cumbersome with WEB-based systems because the source code has to be modified extensively in order to integrate it with the documentation text. With DOgMA this task is handled easily. The source code remains unchanged (perhaps some text parts are collapsed, which increases source code readability, too), whereas the documentation emerges separately. It is also possible to describe the system only partially, e.g., the changes that are made during the maintenance process.

- *What you see is what you get (wysiwyg)*

  One of the drawbacks of WEB systems is the use of a text formatting language and the WEB specification language. Nowadays users want to make modifications directly on the screen, rather than inserting batch commands and waiting for the results. Another disadvantage of these languages is that users have to learn them.

- *Hypertext*

  As the structure of source code is inherently nonsequential, the concept of hypertext is essential for literate programming systems, too. Brown proposes a hypertext interface, but uses only the relations found in the WEB system. DOgMA was designed as a hypertext tool and therefore heavily supports this concept.

- *Integrated compiler*

  WEB-based systems extend the edit/compile/run cycle to a edit/tangle/compile/run cycle. Only the Smalltalk system offers an integrated compiler, which shortens development time considerably. In DOgMA a compiler is not available directly, but the edit/compile/run cycle remains unchanged because source files are stored separately from the documentation, which makes the tangle-step superfluous.

- *Graphic support*

  Graphic presentations are very helpful for documentation. Unfortunately, they are hardly supported by literate programming systems. Obviously the only right step will be to use existing, powerful word processing systems that allow the integration of graphics, but so far only simple text editors have been used. This holds for DOgMA as well, but graphic support is planned for future versions.

Figure 5.16 summarizes our comparison of the literate programming features. DOgMA's advantages are the possibility to view and process the plain source code and—what is especially important in that case—to automatically update the documentation when changes were made in the source code. Additionally, the hypertext features greatly enrich the usefulness of the literate programming aspect.

	WEB	**HSD**	**Smalltalk**	**Interactive**	**DOgMA**
hierarchical structure	no	yes	yes	yes	yes
delayed code	yes	yes	no	yes	yes
source view	no	no	no	no	yes
automatic doc. update	no	no	no	no	yes
subsequent documentation	no	no	yes	no	yes
wysiwyg	no	no	yes	yes	yes
hypertext	no	no	no	yes	yes
integrated compiler	no	no	yes	no	no
graphics support	no	no	yes	no	no

Fig. 5.16 Comparison of literate programming features

5.5 Summary of the Comparison

The previous sections provided a detailed comparison of DOgMA with browsing, hypertext and literate programming tools. One might claim that DOgMA exceeds the functionalities of the other tools only to a small extent: The provided browsing features are powerful but not unique. The same holds for the hypertext capabilities, which even lack the support of contexts, and attributes are provided only to a minor extent. In literate programming the advantages of DOgMA are somewhat clearer.

However, it was not intended to outdo any existing tools in their very domain, but rather to combine various concepts in order to provide new possibilities in the support of documentation and maintenance. In fact, DOgMA's strength emerges from the amalgamation of the ideas of nonsequential reading and writing and literate programming.

Another characteristic and advantage of DOgMA is its special focus on the process of software maintenance, which resulted in additional features like enhancing the readability through global text styles, finding the occurrences of identifiers through highlighting, automatic renaming of identifiers both in the source code and in the documentation, and providing information about identifiers. These features are not encountered in the tools presented above.

6. Conclusion and Prospects

Chapter 6 draws its conclusion by answering the question whether the goals defined in Section 5.1 have been achieved. Furthermore, experience gained with DOgMA is presented and a discussion of its possible integration with software development environments adds a future dimension.

We repeat the definition of the goals presented in Section 5.1 and comment on the results.

- *Concentration on the essentials*

  The intention in developing DOgMA was to achieve a reduction in maintenance costs. This was accomplished by concentrating on the support of program comprehension which requires most of the time in the process of software maintenance.

- *Synthesis of new concepts*

  In fact the synthesis of the concepts of hypertext and literate programming turned out to decisively contribute to the usefulness of DOgMA.

- *Compatibility*

  DOgMA is not fully integrated with other tools that support other aspects of the software life cycle. However, the plain text was chosen for the external representation of both the source code and the documentation. This ensures unrestricted use of any other tools available as long as their external representation of data is textual, too.

- *Modern user interface*

  DOgMA is highly interactive and its user interface is comprised of simple and easy to use elements like menus and lists. It turned out that users were able to use it with only a short introduction and demonstration without the need for any user documentation.

- *Modern implementation techniques*

  The development of DOgMA would not have been possible without object-oriented programming and the reuse of a powerful application framework. Using the object-oriented paradigm with an application framework saves much time in creating a modern user interface and hence helps in concentrating on application-specific problems.

- *Motivation to write documentation*

  If a tool can ever motivate someone to write documentation, then it must do so by providing an easy possibility to write down thoughts at the time these thoughts occur or by offering a comfortable way of subsequently doing it. We regard the integration of source code with the documentation and the opportunity to simply work on both of them simultaneously and with the same tool as a major step towards improving the willingness of programmers to write (system) documentation.

  Walkthroughs, even though sometimes regarded as too costly, have the advantage of reducing the number of compilations and tests and help in maintaining quality and adherence to standards (see [Par86b]). If walkthroughs are not applied to the pure source code but rather to the documentation which contains the source code, then the quality of the documentation would benefit from these advantages as well. The educational aspect of such documentation walkthroughs would increase the quality of the documentation considerably.

- *Documentation access*

  DOgMA integrates source code and documentation and tells the user whether any documentation exists for a specific part of the source code (see menu entry *Show Documentation*, Section 5.3).

- *Documentation consistency*

  The consistency of the documentation is partially ensured automatically because source code parts in the documentation text are kept up-to-date. Even when parts of the source code have been removed, this can easily be seen in the documentation by means of unresolved links (see Section 5.4.5). As the source code is always up-to-date, checking the text of the documentation can easily be done by simply reading it. This is at least by far simpler than finding inconsistencies in the documentation where no or old source code is integrated.

- *Documentation completeness*

  It is hard to determine when documentation is really complete. However, with DOgMA it is easy to find out whether or not a part of the source code is documented. If we define completeness as the fact that there exists documentation for every part of the source code, then DOgMA can help to determine whether this is fulfilled and for which parts documentation is still missing.

- *Existing code support*

  DOgMA operates on pure source code files, which makes it better suited for processing existing source code than other tools. We believe that DOgMA provides a comfortable way to (re)document an existing system, or at least to document changes on it. If documentation also exists, then with DOgMA it is easy to establish links between documentation and source code and hence take advantage of this integration.

- *Development support*

  DOgMA can be used as development tool as well. By the way, initial versions of it were used to develop DOgMA itself. Many features that were primarily thought to support the maintenance of software systems have been proved themselves very useful during the development process. As it becomes increasingly important for software development to reuse existing code (e.g., application frameworks, tool boxes), the ‘maintenance aspect’ increases for development.

- *Information access*

  DOgMA offers various powerful browsing features, provides information about identifiers, supports global text styles for better readability, and highlights identifiers to help to answer a lot of important questions. Any information available through static analysis is provided to the user as clearly as possible.

- *Preventing side effects*

  DOgMA promotes the understanding of a software system and answers many questions about the source code. This helps to prevent side effects in the source code. Additionally, documentation side effects (like deleting and renaming identifiers in the source code) either can easily be checked (unresolved code links) or are even prevented (renaming identifiers) (see also [Fre82]).

- *Dealing with high complexity*

  In order to deal with complex systems, it is important to allow nonsequential reading, provide information about the logical structure of the software system, and to enable the simultaneous inspection of various parts of the system. DOgMA has been conceived to work on complex software systems.

Through DOgMA’s powerful browsing, hypertext and literate programming capabilities it helps in dealing with the complexity of software systems, in providing better documentation (documentation access, documentation consistency, documentation completeness, motivation to write documentation), and offers nonsequential reading and writing for source code and documentation. Visual programming aspects are regarded to further improve program comprehension to a large extent. Thus visual programming features are to be considered as one of the next steps in DOgMA’s development.

Experience with DOgMA

Early versions of DOgMA were used primarily to develop DOgMA itself. This early experience was most helpful for improvements. DOgMA’s use proved to be very beneficial in the following points:

- *Using existing code*

  Mastering the complexity of an existing application framework is essential for effective software development. DOgMA provides easy access to the existing class hier-

archy, provides inheritance information, easy access to overridden methods, etc. Besides, the highlighting mechanism is of great help in studying the use of methods and in finding out the proper meaning of instance variables. This was especially necessary due to the lack of any documentation of the application framework ET++.

Before early versions of DOgMA were ready for use, the familiarization process with ET++ had to be accomplished with simple file browsers. For example, finding the overridden counterparts of a method in the superclasses required several cumbersome steps. First, the class definition had to be inspected to find out the name of the superclass. Usually, the file name could be deducted from the class name. When several classes were combined in one file, the use of the UNIX *grep* command was required which searches for a string in various files. If the appropriate file was found and opened in a new file browser, then the method could be found with a string search unless it did not exist for this class. If the method could not be found, then the previous steps had to be repeated for the next superclass.

Remember, with DOgMA the existence of overridden methods can easily be seen in the information box and they are at hand with a single mouse click.

- *Finding dead code or variables*

  Continued development and maintenance of large software systems and even the evolutionary approach in software development occasionally produces dead code. Usually such code is left untouched because it is hard to determine whether the code is really ‘dead’. With DOgMA it is no problem to answer this question. For example, the question *Is this instance variable or method really needed any longer?* can easily be answered by simply highlighting the variable or method.

  Finding a dead variable without adequate tool support again requires a cumbersome string search in various file. In the development of DOgMA finding dead code and variables was sometimes necessary due to the evolutionary process taken. Especially when classes were split or merged, a cleanup was needed to remove superfluous variables and methods.

Actually, the experience gained with DOgMA by the author was mainly made during a software development process, but an essential part of this process was the reuse of an already existing class hierarchy. Besides, we would like to recall at this point that the line between software development and software maintenance is drawn rather arbitrarily (see Chapter 2). Without using (early versions) of DOgMA the (continued) development of DOgMA itself would have been a mess! It is impossible to measure the profit in using DOgMA, but comparing various maintenance activities performed with and without DOgMA (see above) gives an idea of its usefulness.

DOgMA is currently being used by other members of our research group and by students as well. They also have to reuse the application framework ET++ and some of them even have to get acquainted with the source code of DOgMA itself in order to make some enhancements and additions. They rate the main advantage of the tool as its support of understanding foreign source code, of getting an overview of the existing

class hierarchy and of easily getting all the information that is needed to understand a piece of code. The time needed by our students to take advantage of ET++ was reduced from months to weeks.

We have little experience so far in the literate programming part of DOgMA. Only minor parts of DOgMA itself were documented subsequently. However, students are now using DOgMA as a literate programming tool by developing source code and the corresponding documentation in parallel. The benefits of literate programming are undisputed and we are optimistic about DOgMA's ability to effectively support this paradigm.

Integration with Software Engineering Environments

DOgMA has proved to be a powerful tool, but it covers only parts of the software engineering range of activities. DOgMA was designed to allow its use together with other software tools on the Unix platform. In order to increase programmer productivity and to make it better suitable for larger projects with multiple programmers, its extension and tighter integration with other tools would be desirable.

Other tools could profit from the symbiosis of hypertext and literate programming resulting from integration with DOgMA. It should be possible to start the compiler within DOgMA and compiler errors should be displayed along with the corresponding source code within the documentation. The debugger should also work with the documentation so that the user hardly ever sees the pure source code and gets used to working with the documentation. Naturally, DOgMA's hypertext capabilities are also most appreciated during the debugging process.

These are the most immediate steps for attaining a powerful software engineering environment. The support of change management and configuration management becomes important for large projects. Again, integration with the concepts of hypertext and literate programming can improve productivity of programmers and the quality of the product considerably. For example, every change made (e.g., bug fix) could be described in a documentation chapter. Hypertext links could easily be established among error reports and corresponding changes in documentation and source code. Connections among different versions of a software system would help to solve the multiple maintenance problem.

7. References

[Abi88] Abi R.: Software Maintenance: Tools and Techniques—How to reduce the maintenance blues, System Development, pp. 3-6, August 1988.

[ANS83] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 729-1983, The Institute of Electrical and Electronics Engineers, Inc., 1983.

[Art88] Arthur L.J.: Software Evolution: The Software Maintenance Challenge, John Wiley & Sons, 1988.

[Ave90] Avenarius A., Oppermann S.: FWEB: A Literate Programming System for Fortran8x, ACM Sigplan Notices, Vol. 25, No. 1, pp. 52-58, Jan. 1990.

[Bab86] Babich W.A.: Software Configuration Management: Coordination for Team Productivity, Addison-Wesley, Reading, MA, 1986.

[Bar88] Barret E. (Ed.): Text, ConText, and HyperText: Writing with and for the Computer, MIT Press Series in Information Series, 1988.

[Ben86a] Bentley J.: Programming Pearls: Literate Programming, Communications of the ACM, Vol. 29, No. 5, pp. 364-369, May 1986.

[Ben86b] Bentley J.: Programming Pearls: A Literate Program, Communications of the ACM, Vol. 29, No. 6, pp. 471-489, June 1986.

[Ben87] Bentley J.: Programming Pearls: Abstract Data Types, Communications of the ACM, Vol. 30, No. 5, pp. 284-290, April 1987.

[Big87] Bigelow J., Riley V.: Manipulating Source Code in Dynamic Design, Conference on Hypertext, pp. 397-408, November 1987.

[Big88] Bigelow J.: Hypertext and CASE, IEEE Software, Vol. 5, No. 3, pp. 23-27, March 1988.

[Bla89] Blaschek G., Sametinger J.: User-adaptable Prettyprinting, Software—Practice and Experience, Vol. 19, No. 7, pp. 687-702, July 1989.

[Bla91] Blaschek G.: Type-Safe OOP with Prototypes: The Concepts of Omega, to be published.

[Bri87] Brill A.E.: Prevented Maintenance, Computerworld, Vol. 20/21, pp. 83-84, Dec. 29, 1986/Jan. 5, 1987, contained in [Par88], pp. 339-340.

[Bro86] Brown P.J.: Interactive Documentation, Software—Practice and Experience, Vol. 16, No. 3, pp. 291-299, March 1986.

[Bro90a] Brown M., Childs B.: An Interactive Environment for Literate Programming, Structured Programming, Vol. 11, No. 1, pp. 11-25, 1990.

[Bro90b] Brown M., Cordes D.: Literate Programming Applied to Conventional Software Design, Structured Programming, Vol. 11, No. 2, pp. 85-98, 1990.

[Bud84] Budde R., et al.: Approaches to Prototyping, Springer; 1984

[Bus45] Bush V: As We May Think, Atlantic Monthly, pp. 101-108, July 1945.

[Cha86a] Chapin N.: Software Maintenance: A Different View, Data Management, Vol. 24, pp. 30-35, February 1986, contained in [Par88], pp. 302-304.

[Cha86b] Chapin N.: Veil of Obscurity Masks Need for Maintenance Training, Computerworld, Vol. 20, p. 59, April 28, 1986, contained in [Par88], pp. 317-318.

[Chi90] Chikofsky E.J., Cross II J.H.: Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, Vol. 7, No. 1, pp. 13-17, January 1990.

[Clu81] McClure C.L.: Managing Software Development and Maintenance, Van Nostrand Reinhold Publishing, 1981.

[Col88] Colner D.: Literate Programming: Expanding generalized Regular Expressions, Communications of the ACM, Vol. 31, No. 12, pp. 1376-1385, December 1988.

[Con87] Conklin J.: Hypertext: An Introduction and Survey, Computer, Vol. 20, No. 9, pp. 17-41, September 1987.

[Cou85] Couger J.D.: Motivating Maintenance Personnel, Computerworld: in depth, Vol. 19, pp. ID/5-ID/14, August 2, 1985, contained in [Par88], pp. 278-280.

[Dij65] Dijkstra E.: Programming Considered as a Human Activity, Proceedings of the 1965 IFIP Congress, Amsterdam, North-Holland Publishing Co., 1965, contained in [You79], pp. 3-9.

[Dig89] Smalltalk/V PM: Tutorial and Programming Handbook, Digitalk Inc., 1989.

[Eng63] Engelbart D.C.: A Conceptual Framework for the Augmentation of Man’s Intellect, in Vistas in Information Handling, Vol. 1, Spartan Books, London, 1963.

[Fai85] Fairley R.E.: Software Engineering Concepts, McGraw-Hill, 1985.

[Fel79] Feldman S.I.: Make—A Program for Maintaining Computer Programs, Software—Practice and Experience, Vol. 9, No.4, pp. 255-266, April 1979.

[Fid88] Fiderio J.: Hypertext: A Grand Vision, BYTE, Vol. 13, No. 10, pp. 237-244, October 1988.

[Fre82] Freedman D.P., Weinberg G.M.: A Checklist for Potential Side Effects of a Maintenance Change, Handbook of Walkthroughs, Inspections, and Technical Reviews; Little, Brown and Company, 1982, contained in [Par88], pp. 93-100.

[Gam86] Gamble S.L.: What Tangled Webs We Weave: The Threat of Unstructured Cobol, Business Software Review, Vol. 5, pp. 28-32, November 1986, contained in [Par88], pp. 335-336.

[Gar87] Garg P.K., Scacchi W.: Maintaining Software Life Cycle Documents as Hypertext: Issues, Analysis, and Directions, Technical Report, University of Southern California, Los Angeles, Computer Science Dept., 1987.

[Gar88a] Garg P.K., Scacchi W.: A Software Hypertext Environment for Configured Software Descriptions, Proceedings of the International Workshop on Software Version and Configuration Control, in [Win88], 1988.

[Gar88b] Garg P.K., Scacchi W.: A Hypertext System to Manage Software Life Cycle Documents, Proceedings of the 21st Annual Hawaii International Conference on System Sciences, Vol. 2: Software, 1988.

[Gib89] Gibson V.R., Senn J.A.: System Structure and Software Maintenance Performance, Communications of the ACM, Vol. 32, No. 3, pp. 347-358, 1989.

[Gil87] Gilbert J.: Literate Programming: Printing Common Words, Communications of the ACM, Vol. 30, No. 7, pp. 594-599, July 1987.

[Gla81] Glass R.L., Noiseux R.A.: Software Maintenance Guidebook, Prentice-Hall, 1981.

[Gli90a] Glinert E.P. (Ed.): Visual Programming Environments: Applications and Issues, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Gli90b] Glinert E.P. (Ed.): Visual Programming Environments: Paradigms and Systems, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Gol84] Goldberg A.: Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA, 1984.

[Gol85] Goldberg A.: Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1984.

[Gre88] Greene L.H.: Self-Identifying Software, Proceedings of the Conference on Software Maintenance, Phoenix, AZ, pp. 126-131, 1988.

[Hal87] Hall R.P.: Seven Ways to Cut Software Maintenance Costs, Datamation, Vol. 33, July 15, pp. 81-84, 1987, contained in [Par88], pp. 358-360.

[Her87] Hershey W.: GUIDE—Review, BYTE, Vol. 12., No. 11, pp. 244-246, October 1987.

[Hod85] Hodil E.D., Richardson G.L.: New Faces for Old Systems, Computer Decisions, Vol. 17, July 15, pp. 52-61, 1985, contained in [Par88], pp. 281-282.

[Hol84] Hollinde I., Wagner K.H.: Experience of Prototyping in Command and Control Information Systems, in Approaches to Prototyping, Springer 1984.

[Ker76] Kernighan B.W., Plauger P.J.: Software Tools, Addison-Wesley, Reading, MA, 1976.

[Knu84] Knuth D.E.: Literate Programming, The Computer Journal, Vol. 27 No. 2, pp. 97-111, 1984.

[Knu86a] Knuth D.E.: Computers and Typesetting, Volume B, TEX: The Program, Addison-Wesley, Reading, MA, 1986.

[Knu86b] Knuth D.E.: Computers and Typesetting, Volume D, METAFONT: The Program, Addison-Wesley, Reading, MA, 1986.

[LaL90] LaLonde W.R., Pugh J.R.: Inside Smalltalk, Volume 1, Prentice Hall, Inc., 1990.

[Leb84] Leblang D.B., Chase R.P.: Computer-Aided Software Engineering in a Distributed Workstation Environment, ACM Sigplan/Sigsoft Symposium on Practical Software Development Environments, April 1984.

[Lef87] Lefkon R.: Maintenance Manager: How To Be a Drill Sergeant and a Good Guy, Too; Computerworld: In Depth, Vol. 21, February 9, pp. 61-75, 1987, contained in [Par88], pp. 340-345.

[Lev87] Levy S.: WEB adapted to C, another approach, TUGboat, Vol. 8, No. 1, pp. 12-14, April 1987.

[Lin88] Lin I., Gustafson D.A.: Classifying Software Maintenance, Proceedings of the Conference on Software Maintenance, Phoenix, AZ, pp. 241-247, 1988.

[Lind89] Lindsay D.: Literate Programming: A File Difference Program, Communications of the ACM, Vol. 32, No. 5, pp. 740-755, June 1989.

[Lin89a] Lins C.: A First Look at Literate Programming, Structured Programming, Vol. 10, No. 1, pp. 60-62, 1989.

[Lin89b] Lins C.: An Introduction to Literate Programming, Structured Programming, Vol. 10, No. 1, pp. 107-112, 1989.

[Liu78] Liu C.C.: A Look at Software Maintenance, contained in [Par88], pp. 61-71.

[Mar83] Martin J., McClure C.: Software Maintenance: The Problem and its Solution, Prentice Hall, 1983.

[Mey87] Meyer B.: Object-Oriented Software Construction, Prentice Hall, 1988.

[Mil79] Miller J.C.: Structured Retrofit, in [Par88], pp. 179-180.

[Mös90] Mössenböck H.: She: A Simple Hypertext Editor for Programs, Technical Report, Nr. 145, Eidgenössische Technische Hochschule Zürich, December 1990.

[Mye86] Myers B.A.: Visual Programming, Programming by Example, and Program Visualization: A Taxonomy, ACM Conference Proceedings, CHI ‘86: Human Factors in Computing Systems, pp. 59-66, 1986, contained in [Gli90b], pp. 33-40.

[Nie90] Nielsen J.: The Art of Navigating through Hypertext, Communications of the ACM, Vol. 33, No. 3, pp. 296-310, March 1990.

[Osb87] Osborne W.: Software Maintenance: Thriving On Respect, Computerworld: In Depth, Vol. 21, July 13, pp. 77-82, 1987, contained in [Par88], pp. 363-365.

[Par83] Parikh G., Zvegintzov N.: Tutorial on Software Maintenance, IEEE Computer Society, 1983.

[Par85] Parikh G.: The Fourth Generation Maintenance Gap, Information Center, Vol. 9, pp. 44-47, 1985, contained in [Par88], pp. 259-263.

[Par86a] Parikh G.: What is Software Maintenance, ACM Software Engineering Notes, Vol. 11, No. 4, pp. 49-52, 1986, contained in [Par88], pp. 29-32.

[Par86b] Parikh G.: Maintenance Walkthroughs Boost Morale, Ensure Quality; Computer World, Vol. 20, April 28, p. 57, 1986, contained in [Par88], p. 320.

[Par86c] Parikh G.: The Third Head of the “Information Age Trinity”, Data Processing Digest, 1/1986, contained in [Par88], pp. 11-12.

[Par87a] Parikh G.: The Several Worlds of Software Maintenance—A Proposed Software Maintenance Taxonomy, ACM Software Engineering Notes, Vol. 12, No. 4, pp. 51-53, 1987, contained in [Par88], pp. 45-50.

[Par87b] Parikh G.: Making the Immortal Language Work (Right), Business Software Review, Vol. 6, April 1987, contained in [Par88], pp. 350-352.

[Par88] Parikh G.: Techniques of Program and System Maintenance, Second Edition, QED Information Sciences, Inc., 1988.

[Par88a] Parikh G.: Improved Maintenance Techniques: The Application of Improved Programming Technologies to Existing Systems, contained in [Par88], pp. 181-186.

[Per86] Perry W.E.: Are Maintenance Careers Dead Ends?, Government Computer News, Vol. 5, pp. 25, 28, January 17, 1986, contained in [Par88] pp. 300-302.

[Pom86] Pomberger G.: Software Engineering and Modula-2, Prentice Hall, 1986.

[Pom91] Pomberger G., et al.: Prototyping-Oriented Software Development—Concepts and Tools; Structured Programming, Vol. 12, No. 1, 1991.

[Pre87] Pressman R.S.: Software Engineering: A Practitioner’s Approach, 2nd edition, McGraw-Hill, 1987.

[Raj90] Rajlich V., et al.: VIFOR: A Tool for Software Maintenance, Software—Practice and Experience, Vol. 20, No. 1, pp. 67-77, January 1990.

[Ram88] Ramsdell J.D.: SchemeTEX—Simple Support for Literate Programming in Lisp. TEX hax Digest, Vol. 88, No. 39, April 23, 1988.

[Ram89] Ramsey N.: Literate Programming: Weaving a Language-Independent WEB, Communications of the ACM, Vol. 32, No. 9, pp. 1051-1055, September 1989.

[Ree89] Reenskaug T., Skaar A.L.: An Environment for Literate Smalltalk Programming, OOPSLA '89 Proceedings, pp. 337-345, October 1-6, 1989.

[Reu81] Reutter J.: Maintenance Is a Management Problem and Programmer’s Opportunity, AFIPS Conference Proceedings on 1981 National Computer Conference (Chicago), Vol. 50, May 4-7, pp. 3443-347, 1981.

[Roc75] Rochkind M.J.: The Source Code Control System, IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, pp. 364-370, December 1975.

[Rom86] Roman D.: Classifying Maintenance Tools, Computer Decisions, Vol. 18, June 30, pp. 36, 40-41, 68-71, 1986; digest in [Par88], pp. 331-332.

[Sam90] Sametinger J.: A Tool for the Maintenance of C++ Programs, Proceedings of the Conference on Software Maintenance, San Diego, CA, pp. 54-59, 1990.

[Sam92] 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.

[Sch86] Schmucker K.J.: Object-Oriented Programming for the Macintosh, Hayden Book Company, 1986.

[Sch87] Schneidewind N.F.: The State of Software Maintenance, IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, pp. 303-310, March 1987.

[Sew87] Sewel E.W.: How to MANGLE your Software: The WEB-System for Modula-2, TUGboat, Vol. 8, No. 2, pp. 118-122, July 1987.

[Shn86] Shneiderman B., et al.: Display Strategies for Program Browsing: Concepts and Experiment, IEEE Software, Vol. 2, No. 5, pp. 7-15, May 1986.

[Shn89] Shneiderman B., Kearsley G.: Hypertext Hands-on: An Introduction to a New Way of Organizing and Accessing Information, Addison-Wesley, Reading, MA, 1989.

[Smi88] Smith J.B., Weiss S.F.: Hypertext, Communications of the ACM, Vol. 31, No. 7, pp. 816-819, July 1988.

[Smi91] Smith L.M.C, Samadzadeh M.H.: An Annotated Bibliography of Literate Programming, ACM Sigplan Notices, Vol. 26, No. 1, pp. 14-20, January 1991.

[Smn85] High Noon: True Test of a Software Maintenance Tool, Software Maintenance News, Vol. 3, pp. 1-2, August 1985, contained in [Par88], pp. 280-281.

[Smn86] High Noon Part II: The Quest for a True Test of a Software Maintenance Tools, Software Maintenance News, Vol. 4, pp. 1-2, August 1986, contained in [Par88], pp. 333-335.

[Sne90] Sneed H.M., Kaposi A.: A Study on the Effect of Reengineering upon Software Maintainability, Proceedings of the Conference on Software Maintenance, San Diego, CA, pp. 91-99, 1990.

[Str86] Stroustrup B.: The C++ Programming Language, Addison-Wesley, Reading, MA, 1886.

[Swa76] Swanson E.B.: The Dimensions of Maintenance, Proc. 2nd Int. Conference on Software Engineering, San Francisco, pp. 492-497**,** October 1976.

[Tah90] Tahvanainen V., Smolander K.: An Annotated CASE Bibliography, ACM Software Engineering Notes, Vol. 15, No. 1, pp. 79-92, January 1990.

[Thi86] Thimbleby H.: Experience of ‘Literate Programming’ Using CWEB (a variant of Knuth’s WEB), The Computer Journal, Vol. 29, No. 3, pp. 201-211, 1986.

[Tic82] Tichy W.F.: Design, Implementation, and Evaluation of a Revision Control System, Proceedings of the 6th International Conference on Software Engineering, September 1982.

[Tic85] Tichy W.F.: RCS - A System for Version Control, Software—Practice and Experience, Vol. 15, No. 7, pp. 637-654, July 1985.

[Tic88] Tichy W.F.: Tools for Software Configuration Management, Proceedings of the International Workshop on Software Version and Configuration Control, in [Win88], 1988.

[Tin85] Tinnirello P.C.: Software Maintenance In Fourth Generation Language Environments, digest in [Par88], pp. 275-278.

[Tun89] Tung Sho-Huan: A Structured Method for Literate Programming, Structured Programming, Vol. 10, No. 2, pp. 113-120, 1989.

[Wal87] Wall D.W.: Literate Programming: Processing Transactions, Communications of the ACM, Vol. 30, No. 12, pp. 1000-1010, December 1987.

[Web83] Webster's new universal unabridged Dictionary, New World Dictionaries/ Simon and Schuster, 2nd edition, 1983.

[Wei88] Weinand A., Gamma E., Marty R.: ET++—An Object Oriented Application Framework in C++, OOPSLA '88, ACM Sigplan Notices, Vol. 23, No. 11, pp. 46-57, 1988.

[Wei89] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No.2, 1989.

[Win88] Winkler J.F. (Ed.): Proceedings of the International Workshop on Software Version and Configuration Control (Grassau), B.G. Teubner, Stuttgart, January 27-29, 1988.

[Wir85] Wirth N.: Programming in Modula-2, 3rd corrected edition, Springer-Verlag, New York, NY, 1985.

[Wyk90] Van Wyk C.J.: Literate Programming: An Assessment, Communications of the ACM, Vol. 33, No. 2, pp. 361-365, March 1990.

[You79] Yourdon E.N. (Ed.): Classics in Software Engineering, Yourdon Press Computing Series, 1979.

[Zve89] Zvegintzov N. (Technical Ed.): Software Maintenance Tools, Release 2.0, Software Maintenance News, March 1, 1989.

8. Figures