DOCUMENTING AND MAINTAINING MODULA-2 PROGRAMS WITH DOgMA

G. Pomberger, T. Prückler, J. Sametinger

Christian Doppler Laboratory for Software Engineering
Institut für Wirtschaftsinformatik
Johannes Kepler University of Linz
A-4040 Linz, Austria

Abstract

DOgMA is a software tool that supports the documentation and maintenance of Modula–2 programs. The tool combines the concepts of hypertext and literate programming to deliver powerful support of the comprehension process. DOgMA can be used to develop new programs, to explore or document existing ones, and to develop source code and documentation in parallel.

The paper presents the concepts used for documentation and maintenance, followed by DOgMA's user interface, and examples of how to use it. Some implementation aspects and experiences conclude the presentation.

1. Introduction

Programmers spend more than half of their time on software maintenance, of which the comprehension process takes more than 50% (see, for example, [Gib89], [Let86], [Par83]). A tool that supports program comprehension on the source code level is of great help (e.g., [Clev88], [Raj90]), especially if documentation is unavailable, incomplete or inconsistent, or if a tool for its automatic production (e.g., [Flet88], [Lan88]) is not available.

The most obvious way to support program comprehension is to produce and maintain adequate documentation, which is needed to understand a software system. But all too often the source code listing is the only accurate and complete representation of the system. By improving the availability of complete and up-to-date documentation, we can reduce software costs considerably.

One way of achieving the goal of significantly better documentation is literate programming (see [Knu84]). The main idea is that when writing programs, we should try to tell humans what we want the computer to do rather than try to instruct the computer what to do. But as humans and computers unfortunately do not speak the same language, there is a need for tool support.

Another important aspect of easing program comprehension is the transformation of the sequential physical structure of both source code and documentation files into a structure that is more adequate for this purpose. In order to understand a software system, it usually does not help very much to sequentially read it. What we need is selective reading and fast and easy access to needed information. Hypertext has proven to be suitable for modeling the logical structure of any document (see [Con87]).

DOgMA supports the literate programming paradigm and is based on hypertext technology with the intention to ease navigation through software systems (both source code and documentation) written in Modula-2. In the following sections we describe the concepts used for supporting the documentation and maintenance process, DOgMA's user interface, and small sample scenarios. Finally, we provide some implementation aspects and share some experiences.

2. Concepts for Documentation and Maintenance Support

Several concepts have been developed to support documentation and/or maintenance. We regard hypertext and literate programming as the most powerful ones and see the combination of these two concepts as resulting in an even more powerful tool. Additionally, we used global text styles to enhance the readability and thus the comprehension of both source code and documentation text.

Literate Programming

The idea of literate programming is to make programs as readable as ordinary literature. The primary goal is not only to get an executable program, but also to get a description of a problem and its solution (including assumptions, alternative solutions and their pros and cons, design decisions, etc.). The idea has been demonstrated successfully on relatively small as well as on large programs (e.g., [Knu84], [Knu86]).

Several tools for literate programming have been developed so far. The original one, named WEB, was developed by Knuth and supported the programming language Pascal (see [Knu84]). Similar systems emerged supporting C (see [Tun89], [Thi86]), Fortran (see [Ave90]), and Smalltalk (see [Ree89]). Even a generator for WEB-systems is available (see [Ram89]) that can be used to construct instances of WEB from a language description. But all these systems, like the original one, are rather clumsy to use and assume that a program and its documentation are written straightforwardly.

In order to incorporate the literate programming concept in DOgMA, we define our documentation structure to consist of a set of documentation chapters. These chapters can be spread over several files, and a hierarchical structure exists among them. A chapter itself consists of a title and documentation text (that is pure ASCII text) intermingled with program text (actually a link to the source code).

There are two possibilities to use program text within documentation: 1) single identifiers, 2) any text part, i.e., any number of lines of code in succession within a module or a procedure. Single identifiers can be used right within the documentation text, e.g.:

> …
> The variable **kind** tells the module what to do with the definitions when **EndModule** is called: if the module is a definition module (**kind** = **definitionModule**), then the module is added to the other definition modules; otherwise it is disposed of.
> …

Source code identifiers are shown boldfaced. However, with DOgMA the user can choose any font or style for these identifiers. Program text is clearly separated from the documentation text, for example:

> …
> This procedure has to be called before ANY of the other procedures of this module, because it initializes the necessary data structures (a stack and two variables - see code).
>
> ```
> PROCEDURE BeginModule(VAR module: ModuleType; kind: ModuleKind);
> BEGIN
>    NEW (module);
>    InitStack(module^.stack);
>    module^.kind := kind;
>    module^.importActive := FALSE;
> END BeginModule;
> ```
>
> The first parameter, **module**, is a pointer to an instance of type **ModuleType**, which is to be initialized.
> …

In the WEB system and other similar literate programming tools, the source code and the documentation text are physically intermixed. If the source code is needed in its pure form, e.g., for compilation, a tool has to be used. DOgMA takes a wholly different approach: The source files do not contain any documentation, nor vice versa. Links between the texts and additional information gathered from parsing (see below) the source code allow virtual intermixing of code and documentation for browsing (see Fig. 1).

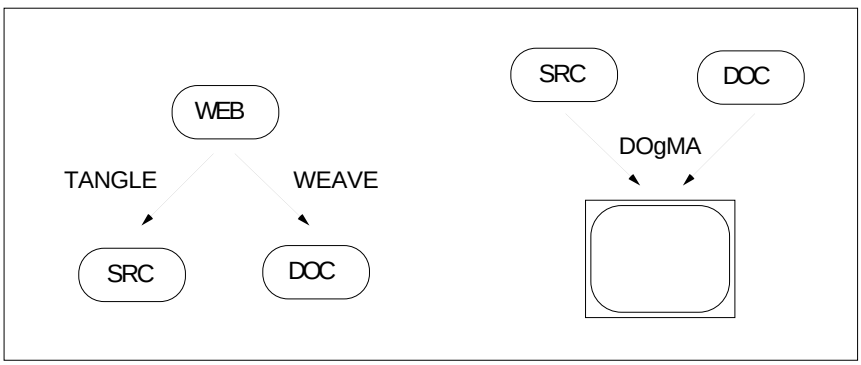

Fig. 1: The document structures of WEB and DOgMA

This strict separation has a crucial advantage: source code files remain unchanged, so they can be compiled without previous conversion and can also be processed by other tools. Besides, DOgMA can be used to directly work on any existing programs. Another advantage is that DOgMA does not force the user to develop program text and documentation text in parallel. Parallel development—which is forced by WEB

systems—might be advantageous. However, in practice this seems to be too restrictive because it requires that programmers be quite familiar with the problem and possible solutions.

Hypertext

Usually text files on a computer are flat; i.e., they are organized in a linear way. This linear organization is not adequate when reading (and writing) the source code and/or the documentation of large software systems. 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 able to move freely through the available information (see [Con87], [Fid88], [Smi88], [Nie90]).

In order to use hypertext concepts in DOgMA we define modules and procedures as hypertext nodes that are managed together with their interrelations, i.e., a module consists of a definition module and an implementation module, a procedure belongs to a certain module and/or to a surrounding procedure, a module is imported by other modules. Additionally, on a second, lower level we even regard single identifiers as hypertext nodes. The following relations exist among identifiers:

- 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 a variable.

Another node of our network is the chapter, which can have superchapters and subchapters and contains links to source code nodes (modules, procedures) and identifiers thereof. A hypertext system can offer the possibility to easily browse through the system by means of the above relations.

Global Text Styles

Increasing the readability of source code and documentation provides a major step in the improvement of program comprehension and thus software maintenance. 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, and identifiers. 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.

This simple global style mechanism can be used to point out various aspects of a software system. Different styles can be used to distinguish among different kinds of identifiers, e.g., local variables, global variables, and parameters. Applying any styles to an identifier of a software system, i.e., the possibility of highlighting a certain 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 definition module (i.e., the exported variables of a module) or the local variables of a module or a procedure. In order to distinguish among different highlighted identifiers (e.g., local and global variables), DOgMA provides the possibility to apply different styles for highlighting. Additionally, the number of occurrences are displayed in the various lists containing the modules, procedures, files, and chapters (for an example see Fig. 2 in the next section).

It is an essential point that the whole functionality provided for the source code also be available within the documentation. This means that the user can obtain information about identifiers and also branch to their definition in the documentation.

3. DOgMA's User Interface

DOgMA's user interface concept is based on modern application frameworks and the concepts supported thereby (see [Shn86], [Wei89]). It provides a menu bar, an information box, two selection lists, and an editor window (see Fig. 2, which also contains the various menus at the bottom). The upper selection list displays either the modules or the files of a software system. The lower one displays the procedures and functions of a module, or those modules imported by a module selected in the upper list. The editor window displays the code part according to the selections made in the two lists on the left side. The menu bar is used for the activation of commands, and the information box displays the file name of the code currently shown.

The Menu Bar

The menu bar is used to perform various commands. It is clearly separated into seven groups and offers commands for loading and storing files, for saving and/or closing a project, for simple text editing (e.g., cut, copy, paste, and undo) and enhanced editing commands (like inserting and deleting of procedures).

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. The information contains the name of the code or documentation currently shown, its file name (containing the directory path), and its module (for procedures and functions).

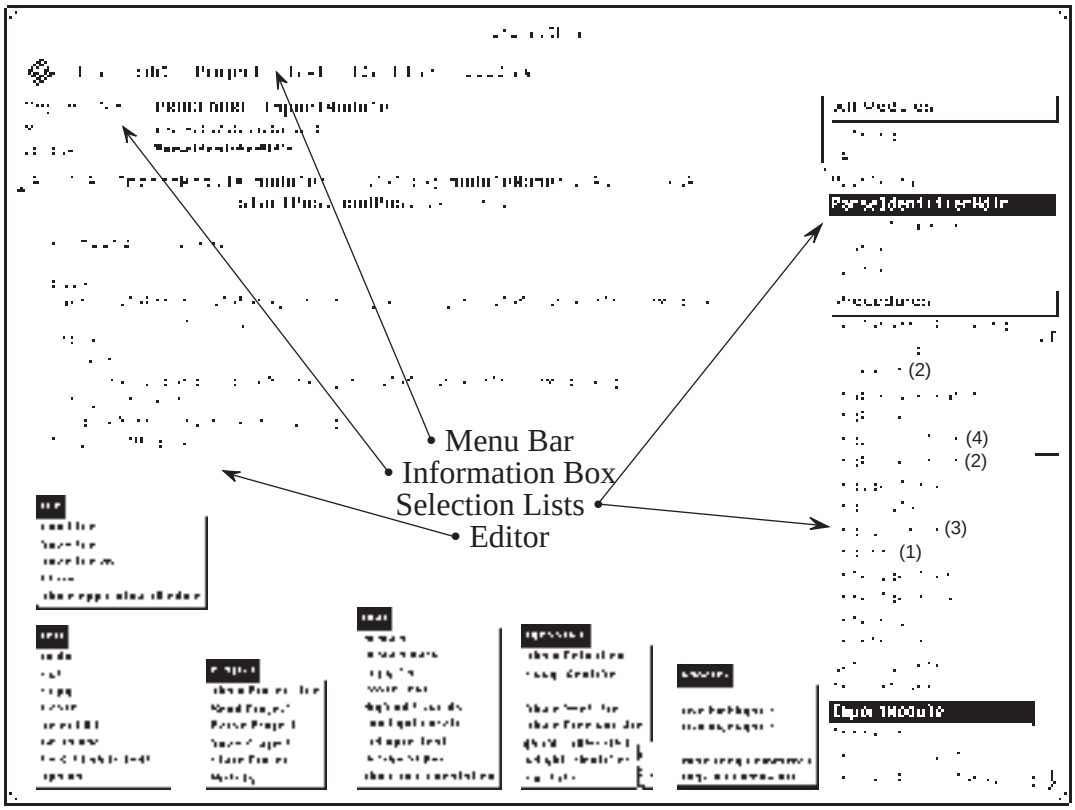

Fig. 2: User Interface

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 the 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. 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 pop-up menus. The following categories can be chosen in the upper list: *all modules, editable modules, all files, editable files, documentation files, documentation chapters*. (The distinction between editable and read-only files and modules is a simple way to avoid unwanted changes in parts of a project.) The lower list enables the user to select *imported modules, procedures/functions,* and *subchapters*.

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

The Editor

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

Using Global Styles for Program Comprehension

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 reduce maintenance costs. The following subsections demonstrate with a few typical examples how DOgMA's global styles can be used to find answers to various questions related to program comprehension.

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 defined and used is a first and important step in casting light on the subject.

Where is this procedure/function called? Which modules call a particular procedure/function?

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

Where are the parameters set in this procedure?

When inspecting a 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 at a glance. DOgMA does not distinguish between the use and modification of a variable. However, seeing all occurrences will certainly provide sufficient information.

The Integration of the Documentation

There are two possible ways to browse through a system in order to read 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 (see Fig.3). Similarly to navigating

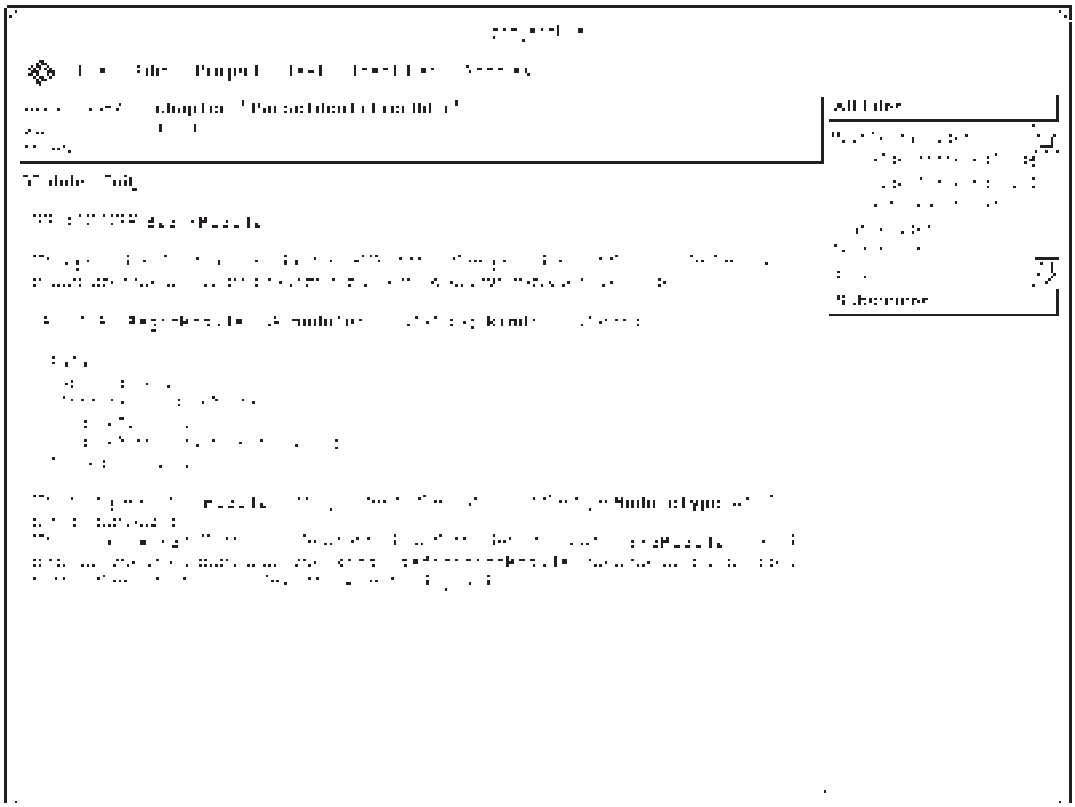

Fig. 3: Documentation text in DOgMA

through modules and procedures, 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.
- One can highlight identifiers 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.

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 DOgMA supports both cases: writing documentation for already existing source code and writing documentation together with writing the source code.

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, any changes in the source code are automatically made in the documentation, too:

- An identifier has been changed in the source code:
  DOgMA 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.

4. Implementation Aspects

DOgMA was implemented under UNIX on a SUN workstation. The implementation is clearly separated into three parts: a language-independent hypertext browser (see [Big88], [Con87]), a language-dependent static analyzer used to get needed information about the inspected program text, and a simple parser for the documentation text. Due to space limitations only the basic system structure of DOgMA is described.

Language-Independent Hypertext Browser

The language-independent hypertext browser (adapted from a version of DOgMA that supports object-oriented programming, see [Sam90], [Sam92]) controls the user interface and manages the following information about a software system:

- text pieces (e.g., chapters, modules, procedures)
- any relations among these text pieces for browsing (e.g., subchapters, import relations)
- classification of text parts (e.g., keywords, comments, identifiers)
- relations among identifiers (the definition of an identifier and its uses)
- additional information (e.g., file location)

Based on this (language-independent) information, DOgMA facilitates easy browsing through a software system.

Language-Dependent Source Code Parser

The language-dependent parser (implemented in Modula-2) analyzes the source code, cuts it into small pieces of text (modules and procedures), and passes information to the

hypertext browser, e.g.: the definition of an identifier, the use of an identifier, any keyword, any comment, the location of files (directory path).

There are two levels of information available: When the system is read in, modules, procedures and imports are recognized and dealt with. On request the whole system is parsed to gather information about each identifier (definitions and uses), the keywords and comments. Both steps have been designed to work with syntactically incorrect programs, which is useful in development and debugging periods. We have chosen to carry out the parsing of the system only upon request because it is rather time-consuming and only necessary if the user is not yet familiar with the system.

Documentation Text Parser

Documentation text, like source code, is stored as pure ASCII text. Links to the source code are marked with a special character and consist of the name of the source code part (e.g., `‘MODULE Scanner’`, `‘PROCEDURE BeginModule’`) followed by the name of the identifier or the text part (e.g., `‘PROCEDURE BeginModule;kind’`). DOgMA creates this external representation automatically. The user does not need to work with these cryptic files at all. The documentation text parser analyzes the documentation files, cuts them into small pieces of text (chapters), and reestablishes the links to the source code.

5. Application and Experience

DOgMA has proven to be of great help in mastering the complexity of both our own research developments, including DOgMA itself, and software written by others. A big benefit of DOgMA is its usefulness to students. Its simple mechanism for browsing allows them to obtain useful information about a software system and thus to understand faster what is happening. Our industrial partners also rate DOgMA as a great help, especially in understanding existing systems.

A tool cannot replace good documentation. At most it can produce more documentation automatically, but this will never be a real substitute for documentation written by hand (e.g., the description of concepts). DOgMA supports the literate programming paradigm. Unfortunately, there are no literate Modula-2 programs available, but the possibility to comfortably (re-)document software systems is an important step towards better (i.e., consistent and complete) software documentation.

Activities of software maintenance and software development are very similar. Thus a real maintenance tool also supports software development. Usually program comprehension alone plays a somewhat minor role in the development process. But browsing facilities and the comfortable possibility of parallel documentation are crucial points in software development, too.

6. References

[Big88] Bigelow J.: Hypertext and CASE, IEEE Software, pp. 23-27, March 1988.

[Clev88] Cleveland L.: A User Interface for an Environment to Support Program Understanding, Proceedings of the Conference on Software Maintenance, pp. 86-91, 1988.

[Con87] Conklin J.: Hypertext: An Introduction and Survey, Computer Vol. 20, No. 9, pp 17-41, Sept.87.

[Flet88] Fletton N. T., Munro M.: Redocumenting Software Systems Using Hypertext Technology, Proceedings of the Conference on Software Maintenance, pp. 54-59, 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.

[Knu84] Knuth D. E.: Literate Programming, The Computer Journal, Vol. 27 No. 2, pp 97-111, 1984.

[Knu86] Knuth D.E.: Computers and Typesetting, Volume B, TEX: The Program, Addison-Wesley, Reading, MA, 1986.

[Lan88] Landis L. D., et al.: Documentation in a Software Maintenance Environment, Proceedings of the Conference on Software Maintenance, pp. 66-73, 1988.

[Let86] Letovsky S., Soloway E.: Delocalized Plans and Program Comprehension, IEEE Software, pp. 41-49, May 1986.

[Par83] Parikh G., Zvegintzov N.: Tutorial on Software Maintenance, IEEE Computer Society, pp. 61-62, 1983.

[Raj90] Rajlich V., et al.: VIFOR: A Tool for Software Maintenance, Software—Practice and Experience, Vol. 20, No. 1, pp. 67-77, January 1990.

[Sam90] Sametinger J.: A Tool for the Maintenance of C++ Programs, Proceedings of the Conference on Software Maintenance, San Diego, USA, 1990.

[Sam92] Sametinger J., Pomberger G.: A Hypertext System for Literate C++ Programming, Journal of Object-Oriented Programming, 1992.

[Shn86] Shneiderman B., et al.: Display Strategies for Program Browsing: Concepts and Experiment, IEEE Software, pp. 7-15, May 1986.

[Wei89] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No.2, Springer International 1989.

[Wir85] Wirth N.: Programming in Modula-2, 3rd corrected edition, Springer-Verlag, New York, NY, 1985.