The Role of Documentation in Programmer Training

From Conventional Documentation to Literate Programming, Hypertext and Object-oriented Documentation

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

Abstract

High-quality software documentation reduces the maintenance burden and improves productivity by enhancing reusability. Well documented software systems are also needed for students to learn from designs and implementations of experienced engineers.

Documentation is neglected in software education to a great extent. Neither documentation skills are taught, nor well documented systems are used for learning purposes. The availability of programming tools already plays a major role for choosing a programming language for software education. Usually and unfortunately, neither the availability of documentation tools nor the availability of documentation support in programming tools have an impact on this choice.

Furthermore, conventional documentation does not seem to be very suitable and attractive for learning and teaching. This paper outlines how concepts like literate programming, hypertext and object-oriented documentation can be combined to improve software documentation quality and accessibility. This can result in a more effective education of our future software engineers.

1. Introduction

Software engineering is the practical application of scientific knowledge for the economical production and the useful and economical employment of reliable and efficient software [Pom93]. The need for documentation is obvious for software maintenance and software reuse. But we also have to direct our attention to the role of documentation in programmer training.

Maintenance plays a key role in software engineering. It consumes the greatest proportion of expenses in the software life cycle [Art88]. The most difficult problem in modifying software systems is understanding the intent of the original programmers. Although tools that support program comprehension on source code level are of great help, adequate documentation is the most obvious and effective way to support this comprehension process. Software documentation is a necessity to enable maintenance, and increasingly attention is being paid to it in practice.

Object-oriented programming is the programming paradigm of the nineties. Typically, object-oriented systems are not built from scratch, but rather class libraries and application frameworks are extended. However, efficient reuse of existing components is possible only when documentation is available that makes it possible to identify reusable components in a certain context and to actually reuse them by providing all necessary information.

One of the key skills for a good software engineer is the ability to devise excellent designs for software systems. Besides practicing, the only way to considerably improve one's ability to design software systems is to study existing designs. This requires the availability of well designed software systems and their corresponding, well written documentation. Without documentation it is difficult to duplicate the chain of reasoning of the original author of a complex system. Thus documentation is crucial not only for maintenance and reuse but also for software education. Excellently designed and well documented software systems also prove extremely useful in the education of software engineers.

The study of both good quality design and documentation is inexcusably neglected in programmer training. This is partly due to the fact that most of the source code is proprietary and, if available, lacks adequate documentation. Nevertheless, there is enough well designed and publicly available source code around that can be used for the purpose of knowledge transfer from experienced software engineers to fledgling ones.

We need more attention to software documentation education to change the role of documentation from an awkward appendage to a matter of course. Additionally, concepts and tools are needed that help in analyzing and learning from existing, good software architectures and documentation for educational purposes.

So far, writing documentation has not been practical when a software system is frequently changing. Yet particularly when a software system is frequently changing, documentation is required to facilitate these changes. This apparent contradiction can be resolved by using technologies like literate programming, hypertext and object-oriented documentation, as will be shown in this paper.

2. Conventional Documentation

Conventional documentation is simply a linear text that possibly contains some graphics and is physically separated from the program text. Typically it is written after finishing the source code of a software system. Nowadays this kind of documentation is typical, and it almost always is incorrect, inconcise, incomplete and inconsistent.

When writing software documentation is taught at all, then typically students are first instructed in the possible structure and contents. Then they have to write parts or all of the documentation for an example application that they had to program. In many cases, practical programmer training stops at the point when development of a software system is finished. As a result, documentation is hardly ever written and also not really needed, except for getting good marks when it is required by the teacher.

Public domain software could be used for studying existing designs, but it hardly ever comes with adequate documentation. So even if the software contains good design examples, they are not really useful for educational purposes. And even if suitable documentation is available, it is hard for students to use it, because conventional documentation is organized linearly. This handicaps selective reading and writing. Besides, there is no connection to the program text. This separation makes it difficult not only to check the completeness and the consistency of the documentation. Additionally, it is difficult to find the corresponding location of some source code, e.g., the description of a certain procedure or method.

The separation of source code and documentation is one of the key determinants for the neglect of documentation in education. Programming and documenting seem to be two totally different activities. Of course, the more important one—programming—is done first and, unfortunately for the latter—documenting—there is not sufficient time available. This separation is also one of the main reasons for the absence of well documented sample programs. Students could benefit from programs that are well documented, but such programs hardly exist. Literate programming promises to provide a solution to this dilemma.

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 [Knu92]. 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.).

With the idea of literate programming Knuth also developed the WEB system to support his paradigm. WEB programs consist of a series of documentation sections which contain documentation text and source code. Each section describes a certain aspect of the software system and has references to related sections. For compilation, the source code is extracted from the sections.

The original WEB system supports the programming language Pascal. Other implementations have been made for C, Modula-2, Lisp and Fortran. In order to make WEB available to a much larger audience, a tool was developed to construct instances of the WEB tool from a language description [Ram89]. Object-oriented programming languages are supported as well; for example, literate programming environments have been developed for C++ [Sam92] and Smalltalk [Ree89].

Figure 1 contains two sections of a literate program taken from [Knu84]. For every section the WEB system generates hints about where the code in this section is used and which sections refer to its code.

The integration of source code and documentation makes a positive influence on the correctness, consistency and completeness of documentation. Besides, programming with documentation rather than with pure source code is a major step towards better program comprehensibility and thus maintainability. Well documented examples, like [Knu86], are of high value to students as they provide all necessary information to study both the architecture of existing software systems and their corresponding documentation.

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

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

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  {max. 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. 1: Sample literate program [Knu84]

Unfortunately, too few examples are available because literate programming has not been a success in practice. So what are the disadvantages? Despite the glorious idea of integrating source code and documentation, the WEB system and its successors are rather useless to practitioners because they impose parallel and linear development of both source code and documentation. This is realistic only when one has a good idea of the program to be developed. But usually, experiments and redesigns are needed because the structure of a system is not clear from the very beginning. This scenario of evolutionary, exploratory, and incremental software development characterizes many projects (especially student projects) and must be supported by methods and tools for documentation as well.

In order to benefit from the advantages of literate programming we have to overcome the linear and parallel development but adhere to the integration of source code and documentation. This is where hypertext comes on to the scene.

4. Hypertext Documentation

Usually documentation is stored in text files that are flat; i.e., they are organized in a linear way. This linear organization is not adequate. The documentation of a software system should be interleaved with the source code, and there need to be many possible paths to read the available information, depending on the interests of the reader. A 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 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 gives the user the feeling of free movement through the available information (see [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, an on-line text and retrieval system for browsing and making links and notes. 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 for 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]). Possible applications of hypertext systems include dictionaries, encyclopedias, product catalogs, technical documentation, help systems, and software engineering tools.

Hypertext tools are the new generation of documentation tools. Nonsequential reading and writing is extremely useful for software documentation, but 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. The possibility of selectively reading parts of the source code and the documentation is crucial for the comprehension process. With documentation organized as hypertext and tools providing the above features, it is much easier for both students and practitioners to explore software systems. Sample software systems and good quality documentation—if available—can more easily be studied and checked for consistency and completeness.

The motivation to write documentation increases with hypertext, because it becomes possible to write down ideas and design decisions and connect them with the corresponding source code locations without the need to create separate documents. Completing a web of ideas and design decisions that were written down during the development phase is far easier and less tedious than writing documentation from scratch after development. The fact that software systems change during development becomes less serious because the corresponding documentation parts are easily available and can thus be kept consistent.

The nodes and links in a hypertext documentation have to be well structured. This is a major factor that determines how easy it is to use and update the documentation. Additional features like indexes (for looking up information alphabetically), searches (e.g., keyword search), filters (for limiting the displayed information), bookmarks (for marking specific locations), and path histories (for keeping track and going back) further enhance the opportunities of hypertext.

Arbitrary nodes and links can be defined when writing a hypertext, and there exist numerous possible ways through a hypertext when reading it. Unfortunately, too much freedom can cause difficulty with both reading and writing. We need predefined structures and guidelines that help in producing documentation and lessen the danger of getting lost in a complex information web. Object-oriented documentation described in the next section defines a suitable structure for object-oriented software systems. Additionally, it takes the enhanced reusability and extensibility of the object-oriented programming paradigm into account.

5. Object-oriented Documentation

Class libraries and application frameworks must provide extensive documentation in order to facilitate their use in education. This documentation has to be integrated and reused in the student's documentation just as the prefabricated software components are integrated in their source code. Object-oriented programming improves the reusability of software components. In order to increase the productivity in documenting and to make the structure of documentation better suited for object-oriented software systems, we suggest that object-oriented technology be applied to the documentation, too. This makes it possible to reuse documentation by extending and modifying it without making copies and without making any changes to the original documentation. Additionally, easy access to relevant information can be given by using the inheritance mechanism. Object-oriented documentation is another step toward integrating source code and documentation, and a way of providing non-linear access to relevant information about a software system.

The reuse of object-oriented software components is facilitated by the inheritance mechanism. Inheritance can be viewed as both extension and specialization (see [Mey87]). A class D directly or indirectly inherits from one or more superclasses A, B, and C. The features of the superclasses are a subset of the features of class D; i.e., D inherits whatever A, B, and C provide and includes its own extensions. On the other hand, inheritance is used to realize an is-a relation. For example, a rectangle (D) is a special visual object (A) with the features of a visual object but specialized behavior (specialization). For this purpose, methods can also be disabled in subclasses.

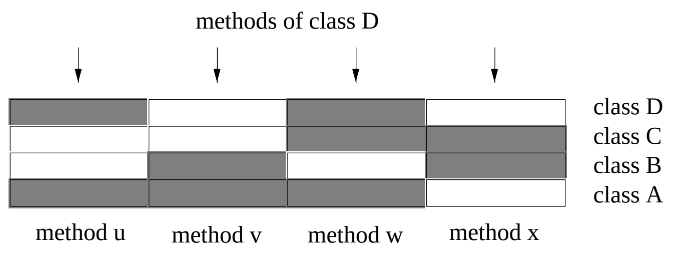

Fig. 2: Inherited and overridden methods of a class

Figure 2 graphically represents the inheritance mechanism. In this example class A provides three methods, u, v, and w. Class B (a subclass of A) overrides v and adds a method x. Class C is a subclass of B and overrides the methods w and x. Finally, class D overrides the methods u and w. The gray and the white boxes in Fig. 2 indicate the availability of methods. Class D provides the methods u and w of its own, method v inherited from class B, and method x inherited from class C. Overriding a method means either replacing the overridden method or extending it, i.e., invoking the overridden method in the overriding one. However, from the class's users' point of view there is no difference between an overriding and an extending method.

As with the source code, a class should inherit the documentation of its superclasses. However, the benefits of inheritance would not be worth the effort when applied only to a class's documentation as a whole. Therefore, we suggest dividing it into (arbitrary) sections. A section is a portion of documentation text with a title. The sections can be defined by the user (programmer) and used for inheritance in the same way as methods. Similarly to methods, sections are either left unchanged, removed, replaced, or extended in subclasses (see Fig. 3). Examples of such sections are: short description, conditions for use, instance variables, instance methods. We further suggest defining a basic set of sections that has to be provided for each class (e.g., those listed above). Depending on the classes, other sections have to be added, e.g., event handling, change propagation.

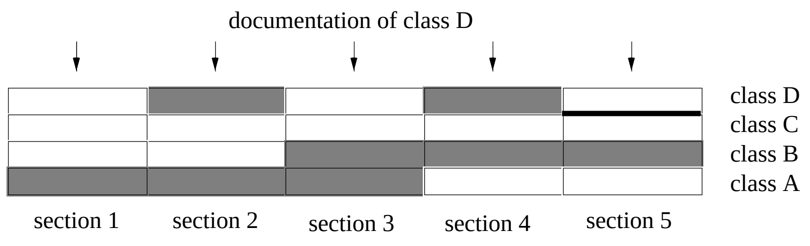

Fig. 3: Inherited and overridden documentation sections of a class

Figure 3 contains the structure of the documentation of the classes A, B, C, and D. The documentation of class A consists of 3 sections, and classes B and C have five documentation sections, whereas class D has only four. Class D inherits sections 1 and 3 from the classes A and B, respectively, has section 2 and four of its own, and removes section 5. Removing documentation sections is important when a specialized behavior is realized in subclasses, e.g., object list and sorted object list. Please note that the documentation of class C consists of five parts, though not an extra line of documentation has been written for this class. For more details about object-oriented documentation see [Sam94].

Object-oriented documentation is built according to the inheritance structure of a software system and offers both easy extensibility and easy reusability. The simple structure helps students to find their way through the information more easily. The well-defined structure of object-oriented documentation can be combined with hypertext and literate programming to achieve high flexibility and higher integration with the source wherever wanted.

6. Experiences

Many of our students who write their thesis use the public domain application framework ET++ for their programming work (see [Wei89]). ET++ consists of hundreds of classes and thousands of methods. Documentation text for the most important classes and methods is available, but is stored in separate files and thus not easily accessible. For beginners it is even rather cumbersome to get relevant information about certain classes and methods out of the source code because the data is usually spread over the descriptions of several classes due to inheritance.

The use of application frameworks and class libraries is inherent to object-oriented programming. But so far such frameworks and libraries represent a hurdle to be surmounted by beginners only with great effort. This is due to the usually enormous complexity, and also to the lack of good documentation. Thus, despite the fact that these frameworks and libraries are developed by very experienced engineers, this experience hardly ever finds its way to software engineering beginners.

We have built a programming environment prototype, which combines the concepts of literate programming and hypertext [Sam91]. The tool automatically creates hypertext links for the source code, e.g., from classes to superclasses and from identifier uses to identifier definitions. This enables useful browsing capabilities and supports the comprehension process enormously. Additionally, documentation text can be integrated with the source code via hypertext links in the sense of literate programming [Sam92]. We also divided the existing documentation into sections and applied the inheritance mechanism as described in the previous section [Sam94].

The students rate the tool as being very useful for getting familiar with an unknown class library. They like the object-oriented access to the library's documentation and the hypertext and literate programming features they can use for writing their own documentation. Unfortunately, no hypertext links are established for the library documentation and no source code is integrated in the sense of literate programming. But once such tools are widely available appropriate documentation will hopefully come with the libraries. Then we will be able not only to reuse the source code but also to fully exploit the knowledge that went into building those libraries.

7. Conclusion

Documentation is important not only for software maintenance and reuse, but also for the education of future software engineers. On the one hand, we have to train our students to write good quality documentation. This does not mean that they should be able to write novels, but rather everyone must be able to sketch design decisions and implementation details and to organize them in a well structured way, so that it is readily understandable and useful for others. On the other hand, well documented, high-quality systems can and should be used as examples in education in order to transfer design knowledge. These goals cannot be achieved with conventional documentation. We need concepts like literate programming, hypertext, and object-oriented documentation plus the corresponding tools in order to improve the quality and also the availability of documentation, and to profit from it in education again.

8. References

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

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

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

[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.

[Knu84] Knuth D.E.: Literate Programming, The Computer Journal, Vol. 27 No. 2, pp. 97-111, 1984, also contained in [Knu92].

[Knu86] Knuth D.E.: Computers and Typesetting, Addison-Wesley, Reading, MA, 1986.

[Knu92] Knuth D.E.: Literate Programming, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

[Mey87] Meyer B.: Object-Oriented Software Construction, Prentice Hall, 1988.

[Nie90] Nielsen J.: The Art of Navigating through Hypertext, Communications of the ACM, Vol. 33, No. 3, pp. 296-310, March 1990.

[Pom93] Pomberger G.: Software engineering education—adjusting our sails. Education and Computing, Vol. 8, pp. 287-294, Elsevier, 1993.

[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.

[Sam91] Sametinger J.: DOgMA: A Tool for the Documentation and Maintenance of Software Systems, Technical Report, University of Linz, Department of Software Engineering, June 1991, available via anonymous ftp from ftp.swe.uni-linz.ac.at.

[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.

[Sam94] Sametinger, J.: Object-Oriented Documentation. Journal of Computer Documentation, Vol. 18, No. 1, pp. 3-14, January 1994.

[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.

[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.