Documentation Inheritance in Literate Programs

Bart Childs
Department of Computer Science
Texas A&M University, USA
bart@cs.tamu.edu

Johannes Sametinger
Department of Computer Science
Brown University, USA
js@cs.brown.edu
and
CD Lab for Software Engineering
Johannes Kepler University Linz, Austria

Abstract

Object-oriented programming has resulted in the reuse of class libraries and application frameworks. This can considerably improve the productivity in software development. Black-box reuse boosts productivity more than white-box reuse. However, white-box reuse is the usual means of dealing with common parts in different applications, assuming that the same developers are building these applications or they somehow know about these commonalties.

With a case study we will demonstrate that both code and documentation can and have to be reused systematically and that there is a need for methods and tools for doing so. Literate programming and software reuse are not in contradiction. However, current literate programming systems do not explicitly support software reuse, even though, as the study will show, there is a high demand for doing so.

We will also present a way for systematic reuse of documentation. Inheritance and information hiding, that ease reusing object-oriented software, will be applied to literate programs and thus, enable their reuse.

Key Words: literate programming, software reuse, software documentation, object-oriented programming, documentation inheritance

1 Introduction

Software reuse is the process of creating software systems from existing software rather than building them from scratch [13]. Reusable software has many benefits, see [1, 13, 15].

Donald Knuth proposes to create each software system as a piece of literature [12]. Can this literature be cut into components and be reused in various contexts? We believe that the idea of literate programming is important in getting well documented and structured software systems. The question that arises is: "How do literate programming and software reuse fit together?"

We will demonstrate the need for explicit software reuse in some literate programs. We will accomplish this by comparing existing software systems both in reuse of code and documentation. We were interested in finding out, how much reuse had been done, how it had been done, and whether there are possible improvements in the way it had been done. We want to demonstrate that it is possible to determine the amount of white-box reuse, that both source code and

documentation can be reused, that more institutionalized software reuse can be done by simply making existing software more reusable, and that a system done in a literate manner can be done with significant reuse in a systematic and formalized manner. In order to achieve systematic and planned reuse for literate programs we adopt object-oriented techniques.

The structure of the paper is as follows: Chapter 2 gives an overview of reuse intentions, i.e., of black-box, white-box, and glass-box reuse. Chapter 3 gives a short introduction to literate programming. In Chapter 4 we present an overview of the systems we considered for investigation. In Chapter 5 we explain how the results had been determined. The results are presented in Chapter 6. Chapter 7 describes our approach of applying object-oriented techniques to documentation and literate programs. Finally, conclusions appear in Chapter 8.

2 Reuse Intentions

Depending on whether the internals of a software component are visible to reusers we speak of black-box or white-box reuse. If a component is a black box we cannot modify its internals, we use it as-is. White-box components are usually modified, even though this is not necessarily the case. They offer both as-is reuse and by-adaptation reuse. The term glass-box reuse has been coined recently and means white-box visibility but black-box, i.e., as-is reuse [4].

- *Black-Box Reuse*
  Reusing a component as a black box means using it without seeing and knowing any of its internals. Usually, a black box is reused as-is. Object-oriented techniques allow modifications of black boxes by making modifications and extensions to a component without knowing its internals. However, components have to be designed so that such modifications become viable.

- *White-Box Reuse*
  White-box reuse is the typical case of today's still widespread unplanned ad-hoc reuse. It means reuse of components of which internals are changed for the purpose of reuse.

- *Glass-Box Reuse*
  The term glass-box reuse is used when components are used as-is like black boxes but its internals can be seen from outside. This gives the reuser information about how the component works without letting him to change it.

Black box reuse is more difficult to achieve than white-box reuse but promises higher quality and reliability of the resulting software system. The potential of customizing black-box components can increase their reuse potential but has to be considered and designed.

Our goal is to have reusable source-code, documentation, and literate program (source code plus documentation) boxes. The purpose of source code is to provide functionality. Thus, we want to reuse source code as black-box whenever possible. The internals are not relevant to the reuser of the source code. The purpose of documentation is to provide information, i.e., the documentation itself. Hence, we have to reveal the internals, but do not want to have them modified. This is glass-box reuse. The documentation part of literate programs describes internals (implementation details) and does not have to be revealed to reusers. Literate programs should be reused as black boxes like pure

source code, but should also offer extension and modfication mechanisms as provided by object-oriented source code.

3 Literate Programming

Human readers are necessary for maintenance activities and that is an area of prime importance in the study of software engineering. We agree with Knuth's claim that literate programming is a process which should lead to more carefully constructed programs with better, relevant 'systems' documentation [12]. The author of codes in a literate programming style has to keep in mind that the human reader is as important as the machine reader. We take Knuth's style of literate programming as prototypical. It was used in writing the second version of the TEX typesetting system [8, 9] and its related components. This WEB system, as he used it, leads to:

- top-down and bottom-up programming through a structured pseudo-code,
- programming in small sections where most sections of code and documentation (section in this use is similar to a paragraph in prose) are approximately a screen or less of source (documentation and code),
- typeset documentation (after all, it was for and in TEX),
- pretty-printed code where the keywords are in bold, user supplied names in italics, etc., and
- extensive reading aids which are automatically generated including table of contents and index.

The value of each of these items is dependent upon the programmer, as always. For example, the index mentioned in the last item can be supplemented by user supplied entries in addition to those automatically generated (which are similar to compiler cross reference lists.) If the author does not furnish these, it is our opinion that the modifier *literate* cannot be justified.

4 The Subject Systems

We have studied four WEBs from the TEX system: TEX, a book quality formatting system [8, 9]; METAFONT, a system that enables a programmer/artist to create a family of fonts for TEX's use [10, 11]; *DVItype*, a prototypical reader of device independent (*DVI*) files which are the output of TEX [7]; and METAPOST, a close relative of METAFONT that enables the creation of high quality graphics as encapsulated PostScript files [5, 6]. The most outstanding feature of the TEX system is the complete and careful documentation that is included. Several of the WEBs were written by Knuth himself and some of the others were obviously carefully reviewed by him.

- TEX
  The TEX processor converts a plain text file containing document markup into a device independent graphics metafile. It also inputs a number of other files in this process to get font charcteristics, document styles, etc.
- METAFONT
  The METAFONT processor operates in a manner that is similar to the TEX processor and at the same time is quite different.

METAFONT does significant graphics interpretations, solving of equations, and other items associated with the creation of a consistent family of fonts.

- METAPOST
  METAPOST is a close relative to METAFONT. Instead of creating a font which has a family of related glyphs constructed using common strokes, serifs, etc. The output of METAPOST is book quality figures.

- *DVItype*
  The *DVItype* processor was created to serve two purposes. First, in the early days when porting TEX was a common activity, it served as a great debugging aid. Second, since it properly reads all possible *DVI* files, it gave creators of programs to input *DVI* files and output printer files a big help.

We have also studied the *tangle* and *weave* processors associated with the CWEB and FWEB systems but due to space limitations will not present the results in this paper. More details on this case study can be found in [2].

5 Reuse Evaluation

Reuse can be identified at a number of different levels, i.e., words, phrases, sentences, lines, paragraphs, sections, and chapters.

We decided to base our general comparison on chapters. This means that we first decided which chapters of two systems were being compared in more detail. The more detailed comparison were done automatically and were based on lines and words. Taking any semantic information into account seemed impossible to be done automatically. Yet, comparing lines and words has given a good indication about the reuse. If line reuse was high, then obviously much reuse had been done. If line reuse was low, but word reuse was high, then much reuse had been done, but the reused text had been modified on a more local basis. Finally, if both line and word reuse were low, then apparently there was not much reuse at all.

$$R = (100 - \frac{C}{T}) \times 100$$

R	reuse percentage from comparing lines. 0 means no reuse, 100 means everything (all lines) of file `a` had been reused in file `b`.
C	number of lines to be changed in file `a` in order to get contents of file `b`
T	total number of lines of file `a`

Figure 1: Reuse Evaluation

The number of lines to be changed in a file `a` in order to convert it to the contents of file `b` opposed to the total number of lines (of file `a`) gives an indication of how much of file `a` had been reused in file `b`. Determination of the reuse factor is shown in Fig. 1.

As empty lines are considered to be equal, the reuse factor, naturally, is greater than zero, if empty lines appear in both files. Thus, it is crucial that empty lines be eliminated before the reuse factor is determined. Of course, for two equal files the result of R is 100. When empty lines are eliminated then, R is usually zero for nominally different files.

Lines can be very similar and, for example, differ only in a single word. Therefore, we considered the reuse on a word basis as well. We will denote R_l and R_w as reuse factors considering lines and words, respectively.

The interesting thing is to compare R_l and R_w. Usually, they do not differ very much, with R_w slightly higher than R_l. Sometimes, however, R_w is significantly higher than R_l. This is the case, when reused text had been modified extensively, leading to many different lines (and a lower R_l), but still remaining many equal words (leading to a higher R_w).

6 Results

TEX, METAFONT, METAPOST, and *DVItype* operate on various common files, e.g., device independent files, font metric files, log files. Some kind of similarity is certainly to be expected. Browsing the sources of TEX and METAFONT in book form [9, 11] reveals many chapters with the same title. Similarities in these chapters are obvious even from just turning the pages. METAPOST is a direct derivative from METAFONT. This should show the highest degree of reuse among them all. In the following description of the results `a` → `b` means the reuse of `a` in `b`, i.e., how much of `a` had been reused in `b`.

- TEX → METAFONT
  TEX contains about 21,500 lines and 122,000 words. METAFONT consists of about 20,500 lines and 110,000 words. TEX and METAFONT are divided into 55 and 52 chapters, respectively. 26 of these chapters have the same title in both systems. These chapters contain 33.4 percent of the lines of the TEX system. 14.3% of the lines and 21.5% of the words of TEX are reused in METAFONT. Of the 26 chapters with the same title 42.8% of the lines and 60.7% of the words are reused in the corresponding chapters in METAFONT.

- *DVItype* → TEX
  Out of 15 chapters in *DVItype* six appear with the same title in TEX. The descriptions of the character set and the device-independent file format have a reuse factor of about 70 percent. These two chapters comprise a quarter of *DVItype*.

- METAFONT → METAPOST
  The highest reuse factor had been determined in comparing METAFONT and METAPOST. Even though the size of about 20,000 lines and more than 100,000 words, more than 60 percent of METAFONT are reused in METAPOST. METAFONT has 52 chapters, METAPOST has 49. 44 chapters appear in both systems with the same title. A total of 24 chapters has a reuse factor higher than 90 percent. Except for three chapters all the other chapters have a reuse factor higher than 70 percent.

Tables 1, 2, and 3 summarize the results. In Table 1 the number of lines and words are given for the subject systems. Table 2 contains the reuse factors yielded by comparing the entire systems. In Table 3 only similar chapters are considered. Column *Portion* specifies how much these chapters contribute to the whole system. For example, the chapters common to TEX and METAFONT contribute 33.4% to TEX. Considering only these chapters for the comparison yields a line reuse of 42.8% and a word reuse of 60.7% (as opposed to 14.3% and 21.5% shown in Table 2).

System	*Lines*	*Words*
TEX	21,541	122,137
METAFONT	20,481	109,307
METAPOST	20,460	104,375
DVItype	2,136	13,606

Table 1: Line and word lengths

Systems	R_l	R_w
TEX → MF	14.3 %	21.5 %
dvi → TEX	18.8 %	32.1 %
MF → MP	63.4 %	67.0 %

Table 2: Reuse Factors

The results clearly demonstrate that software reuse had been applied to both code and documentation. This was done primarily by code and documentation scavenging. There was a certain degree of reuse that we had expected in the results. We did not expect the differences between line and word reuse. These differences are due to the fact that extensive word-smithing on many segments of code and documentation had been done to present information in the best possible manner.

Each system was created as a self-contained, homogeneous work. To achieve this, reused parts from other systems sources were reworked and adapted carefully. Such adaptations included changing the system name (e.g., TEX to METAFONT), changing the word order or modifying single words for better layout results. Often, these adaptations were real improvements like the addition of index entries (which may also have been useful in the original.) This is white-box reuse at its best.

Systems	*Portion*	R_l	R_w
TEX → MF	33.4 %	42.8 %	60.7 %
dvi → TEX	34.9 %	53.8 %	75.2 %
MF → MP	80.8 %	78.5 %	85.1 %

Table 3: Reuse Factors of similar Chapters

The following question arises, though: "How can we achieve the demonstrated degree of reuse and adaptation without scavenging code and documentation?" Our answer to that question is simple: "*Not at all.*" We argue that writing and documenting a software system from scratch will lead to different program and documentation structure than building it by reusing existing components. The intent to deliver reusable components as a by-product of a "build it from scratch" model also leads to a different structure (at least to more self-contained parts of the system.)

Object-oriented development systems were not readily and widely available at the time TEX was built. If the TEX systems had been implemented in an object-oriented manner, many classes would likely have been reused not by direct modifications but by building subclasses. Documentation needs a similar way to be adapted and reused without direct modification, e.g., by means of object-oriented documentation [17].

7 Documentation Inheritance

Systematic reuse of documentation can be achieved by means of [18]

- definition of a common structure for certain documentation parts,
- extraction of common information for several documentation parts,

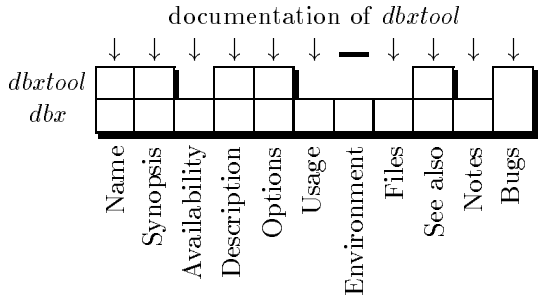

Figure 2: Documentation Inheritance

- reuse and extension/modification of existing documentation (possibly without the need of direct modification), and
- definition of various views for different kinds of readers, e.g., casual users and professional users.

The key concept in accomplishing this is documentation inheritance. As with object-oriented source code, a documentation unit should inherit the documentation of its base unit. A section is a portion of documentation text with a title. The sections can be defined by the programmer/technical writer and used for inheritance in the same way as methods. Similar to methods, sections are either left unchanged, removed, replaced, or extended.

Fig. 2 contains the structure of the documentation of the UNIX tools *dbx* and *dbxtool*. The documentation of *dbx* consists of eleven sections; *dbxtool* has six documentation sections. *dbxtool* inherits the sections *Availability*, *Usage*, *Files* and *Notes*. It has its own sections on *Name*, *Synopsis*, *Description*, *Options*, and *See also*. The section *Environment* is not applicable to *dbxtool* and thus is hidden. This is indicated by a horizontal line rather than an arrow in the figure. The bugs of *dbx* are also available in *dbxtool*, therefore the *Bugs* section had been extended. For more details on this kind of documentation inheritance see [17, 18].

Inheritance can be applied to pure documentation, i.e., documentation without any source code, to systems documentation of conventional software systems, and to systems documentation of object-oriented software systems. The example in Fig. 2 gives a glimpse of how documentation can be reused in the manual page domain, i.e., in user documentation of tools and applications.

TEX, METAFONT, METAPOST, and *DVItype* which have been implemented/documented as literate programs give an excellent example of where documentation could be reused to a big extent. We have found out that a lot of implicit, ad-hoc reuse had been done. This ad-hoc reuse can easily be made explicit by using object-oriented concepts for documentation. The results of this investigation strongly motivated us in providing the prerequisites of explicit documentation reuse.

In order to realize these ideas we have taken an existing literate programming tool and augmented it with the presented features. Noweb is a literate programming tool like WEB. It has been designed to be as simple as possible but meet the needs of literate programmers. Noweb's primary advantages are simplicity, extensibility, and language independence. The primary sacrifice relative to WEB is that code is not prettyprinted and that indexing is not done automatically [16]. More details on this prototype implementation are given in [3].

Object-oriented literate programming is the careful design of source code and documentation of self-contained components (functions, mod-

ules, classes, etc.) that allow their systematic reuse with object-oriented concepts like inheritance. Still, many problems remain open for future research. For example, so far we consider object-oriented techniques only when weaving the documentation, but not when tangling source code, because object-oriented programming languages provide these techniques anyway. It would be interesting to determine the usefulness of applying object-oriented techniques to modular programming languages. Another issue not addressed so far is multiple inheritance. Experience will show whether there is a need for multiple inheritance for documentation similar to source code.

8 Conclusions

We have investigated some TEX systems for reuse. TEX and its related systems are implemented as literate programs. Thus, reuse not only means reuse of source code but also of documentation. Comparing two software systems by first determining chapters or files where similarities can be expected and then by doing automatic comparison based on lines and words proofed to be quite useful in determining the amount of white-box reuse.

The systems under consideration have been implemented in the conventional way, i.e., from scratch by scavenging existing documentation and code. The amount of ad-hoc reuse, that had been done, turned out to be surprisingly high. Also, the different results achieved by comparing lines and words indicate, that any reused text had been carefully edited and adapted. Thus, a lot of (successful) ad-hoc reuse had been done by Knuth (and others), but no attention had been directed to providing, and reusing, reusable components.

We suggest the application of object-oriented techniques to documentation and especially to literate programs in order to benefit from the ease of extension and modification. Documentation inheritance provides the technical prerequisite for reuse. However, like source code, documentation must be designed (and written) for reuse.

References

[1] Braun Christine: "Reuse," in [14], pp. 1055–1069, 1994.

[2] Childs Bart and Sametinger Johannes: "Literate Programming from the Viewpoint of Reuse," to be published, 1996.

[3] Childs Bart and Sametinger Johannes: "Literate Programming and Documentation Reuse," *4th International Conference on Software Reuse*, Orlando, Florida, pp. 205–214, 1996.

[4] Goldberg Adele and Rubin Kenneth S.: *Succeeding with Objects: Decision Frameworks for Project Management,* Addison-Wesley, 1995.

[5] Hobby John: "A user's manual for MetaPost," Computing Science Technical Report No. 162, AT&T Bell Laboratories, April 1992.

[6] Hobby John: "Introduction to MetaPost," *EuroTEX '92 Proceedings*, pp. 21–26, September 1992. TEX Users Group.

[7] Knuth Donald E. and Fuchs David R.: "TEXware," Stanford Computer Science Report 1097, April 1986.

[8] Knuth Donald E.: *The TEX Book*, Volume A of Computer & Typesetting, Addison-Wesley, 1986.

[9] Knuth Donald E.: *TEX: The Program*, Volume B of Computer & Typesetting, Addison-Wesley, 1986.

[10] Knuth Donald E.: *The METAFONT Book*, Volume C of Computer & Typesetting, Addison-Wesley, 1986.

[11] Knuth Donald E.: *METAFONT: The Program*, Volume D of Computer & Typesetting, Addison-Wesley, 1986.

[12] Knuth Donald E.: *Literate Programming*, Stanford University Center for the Study of Languages and Information, Leland Stanford Junior University, 1992.

[13] Krueger Charles W.: "Software Reuse," *Computing Surveys*, Vol. 24, pp. 131–183, June 1992.

[14] Marciniak J. (Editor-in-Chief): *Encyclopedia of Software Engineering*, Vol. 1, John Wiley & Sons, 1994.

[15] Mili Hafedh, Mili Fatma, and Mili Ali: "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, pp. 528–562, June 1995.

[16] Ramsey Norman: "Literate programming simplified," *IEEE Software*, Vol. 11, No. 5, pp. 97–105, September 1994.

[17] Sametinger Johannes: "Object-Oriented Documentation," *ACM Journal of Computer Documentation*, Vol. 18, No. 1, pp. 3–14, January 1994.

[18] Sametinger Johannes: "Reuse Documentation and Documentation Reuse," *TOOLS Europe '96*, Paris, France, pp. 17–28, Feb. 1996.