Literate Programming and Documentation Reuse

Bart Childs
Dept. of Computer Science
Texas A&M University
College Station, Texas
77843-3112 USA

Johannes Sametinger
Dept. of Computer Science
Texas A&M University
and
CD Laboratory for Software Engineering
Johannes Kepler University
Linz, Austria

Abstract

Object-oriented programming has brought many advantages to the software engineering community. The reuse of existing software components and application frameworks can improve the productivity in software development considerably. The same object-oriented techniques, i.e., inheritance and information hiding, that ease reusing software, can be applied to documentation and thus, enable its reuse.

One can document each software component–regardless of what a component is–from scratch. This leads to multiple documentation of features that are multiply reused. One can also describe a component's differences to other components. This seems logical for the systems documentation of object-oriented software. However, as will be shown, this kind of reuse can not only be applied to source-code related documentation, but also to documentation, where there is no source code involved at all, e.g., user documentation.

In this paper we describe the concepts for documentation reuse, how these concepts can be realized with a literate programming tool, and the application of documentation reuse.

Key Words: *documentation reuse, reuse of software engineering artifacts, object-oriented programming, literate programming,* `noweb`

1 Introduction

Reusing software components is a typical task in object-oriented programming. Class libraries and application frameworks have increased programmers' productivity. Modification and extension of software components without the need to make changes to the original source code is am advantage of object-oriented technology. This is accomplished by defining classes through describing their differences (i.e., modifications and extensions) to their baseclasses. The standard behavior is inherited from the baseclasses. Reusable software components that are realized in an object-oriented manner can have many elements in common (through inheritance). Adequate documentation is mandatory for software maintenance, as well as for economic reuse of software components. However, overlapping information holds for source code and for documentation. Therefore, the inheritance mechanism should also be applied to the documentation. Object-oriented documentation can be used to define documentation structures, factor common information, and provide different information details for various groups of readers.

Subsequently, we describe the concepts of object-oriented documentation, the `oonoweb` system, which supports the proposed concepts, and examples how these concepts can be successfully applied to various domains.

2 Concepts

Successful reuse of documentation can be achieved by means of

- definition of a common structure for certain documentation parts,
- extraction of common information for several documentation parts,
- reuse and extension/modification of existing documentation (possibly as is),
- definition of views for various readers, e.g., casual to professional users, and,
- object-oriented description of object-oriented software systems.

The key concepts in accomplishing all this are documentation inheritance, documentation abstraction, and documentation views, which are described in

the subsequent sections. Additionally, a combination with literate programming and hypertext can further improve easy access and consistency of the documentation. This has provided the motivation for realizing the concepts with an existing literate programming tool that supports hypertext (see Chapter 3). We start this chapter with a recapitulation of the originating concept, source code inheritance.

2.1 Source Code Inheritance

The source code of an object-oriented software system consists of classes containing variables (structure) and methods (behavior). Objects with the same structure and behavior are described in a class. From a documentor's point of view, classes and methods seem to be equivalent to modules and procedures used in conventional programming. A central difference between modules and classes is the inheritance relationship between classes. A class may inherit the structure and behavior of another class and may extend and modify it. For example, classes *Rectangle* and *Circle* inherit from a class *Shape*, which defines the structure and the behavior that is applicable to all graphical objects. *Rectangle* and *Circle* are called subclasses (or derived classes), whereas *Shape* is called the baseclass. The source code of the classes *Rectangle* and *Circle* contains only the modifications and extensions to the baseclass *Shape* (see Fig. 1).

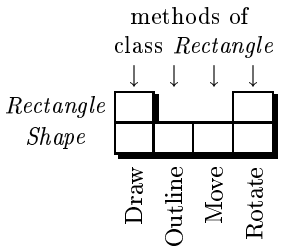

Figure 1: Methods of classes *Shape* and *Rectangle*

The boxes in Fig. 1 indicate the existence of source code for a method. *Rectangle* objects can be drawn, outlined, moved, and rotated, though the class *Rectangle* does not implement the methods *Outline* and *Move*; they are inherited from the baseclass *Shape*. The methods *Draw* and *Rotate* are overridden; i.e., *Rectangle* objects have their own *Draw* and *Rotate* methods, they do not use the methods of the *Shape* class. The arrows (↓) in Fig. 1 indicate the direction of view in order to determine the methods which are provided and used by class *Rectangle*.

2.2 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.

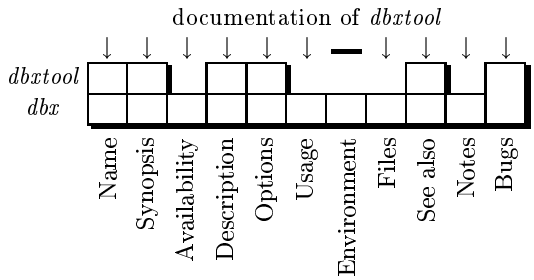

Figure 2: Inherited, overridden, extended, and hidden documentation sections of *dbxtool*

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. The sections *Availability*, *Usage*, *Files* and *Notes* are inherited by *dbxtool*. 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 [9].

2.3 Documentation Abstraction

In object-oriented programming, abstract classes are designed as parents from which subclasses may be derived. Abstract classes are not itself suitable for instantiation. They are used to predefine certain structure and behavior which is then shared by a group of sibling subclasses. The subclasses add different variations of the missing pieces. Documentation has similar structure in many domains, e.g., manual pages and software life-cycle documents. The predefined structure for a certain group of documents guarantees uniform and consistent appearance. It is also possible to factor common information for all the documents, making it easier to make modifications and keep information consistent. The definition of sections of the abstract documentation serves as a guide to consistent documentation and helps identify incomplete parts.

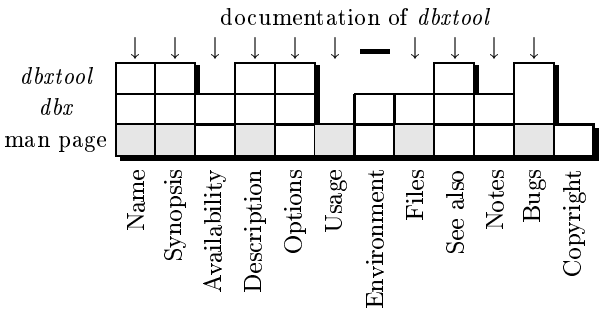

Figure 3: Inherited, overridden, extended, and hidden documentation sections of *dbxtool*

Fig. 3 is another view of the documentation of *dbxtool* in terms of documentation abstraction. The documentation for "man page" defines twelve sections, of which six are designated as having to be overridden (the sections *Name*, *Synopsis*, *Description*, *Usage*, *Files*, and *Bugs*). If such a section is not overridden, as indicated in Fig. 3 for section *Usage*, then the inherited contents of the section should indicate that this information is missing and has to be provided. Tool support is useful in checking completeness and —if incomplete— in spotting the missing sections. The abstract documentation in Fig. 3 contains another section, *Copyright*, which is automatically included for all descriptions inherited thereof. Fig. 4 shows what the abstract documentation for manual pages could look like. Whenever manual pages for a new tool are written, the presence of —information has to be provided— (which is inherited from the abstract manual page) in the documentation indicates that there are still missing parts.

2.4 Two Levels of Documentation Inheritance

A single level of inheritance may not be sufficient for the definition of a convenient documentation structure. Suppose the *Usage* section of *dbx* is further divided into subsections (such as *Filenames, Expressions, Operators*, etc.) and that for the documentation of *dbxtool* we want to override only certain parts and inherit the rest. Of course, we could define sections like *Usage-Filenames, Usage-Expressions*, and *Usage-Operators*. However, the logical structure of the document is better reflected by applying the inheritance mechanism to subsections also. This is shown in Fig. 5, where an additional (abstract) documentation unit has been introduced in order to predefine these subsections.

Two levels of inheritance are important for the documentation of object-oriented software systems also. In this case we need a second level of inheritance for the description of methods. Methods are inherited from baseclasses; but documentation for a single method must be further dividable in order to allow convenient adaptation. In Fig. 6 we have the documentation of a class *Shape*, which consists of three sections (*Description, Layout, Event Dispatching*) plus the documentation of the methods *Draw*, *Outline*, *Move* and *Rotate*. The documentation of each method consists of the sections *Description, Interface,* and *Categories*. The documentation of class *Rectangle* overrides the *Description* and adds an *Implementation* subsection for the methods *Draw* and *Rotate*.

```
abstract manual page
Name
---information has to be provided---
Synopsis
---information has to be provided---
Availability
Refer to ``Installing OS 4.1'' on how to install
  optional software.
Description
---information has to be provided---
Options
no options available
Usage
---information has to be provided---
Environment
no environment variables used
Files
---information has to be provided---
See also
OS 4.1 Programmer's Guide
Notes
no notes
Bugs
---information has to be provided---
Copyright
by Company XYZ, 1996
```

Figure 4: Possible documentation abstraction for manual pages

Considering more than two levels of inheritance is possible. However, we have not encountered a practical example needing more than two.

2.5 Documentation Inclusions and References

For documentation to be readable, information about a unit should not be spread over several files and/or directories. We need either the full documentation of a unit with all inherited documentation included, or cross-references to the inherited information (with page numbers for printed documentation or links for online documentation).

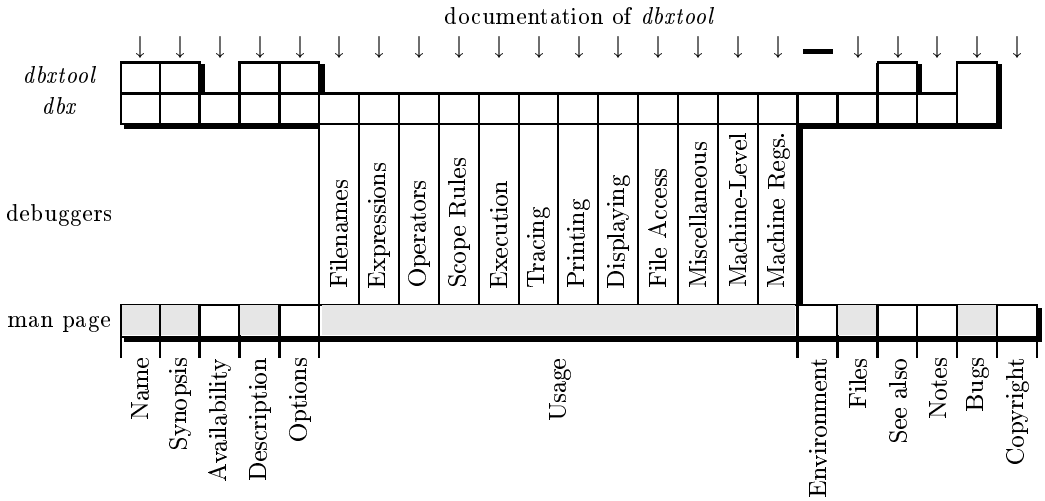

Figure 5: Sections of *dbxtool* using two abstractions and two levels of inheritance

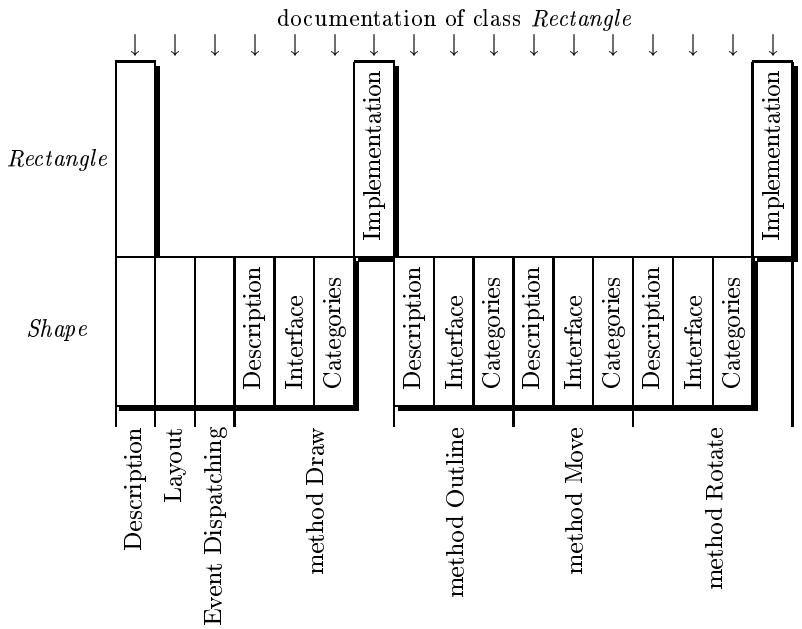

Figure 6: Sections of class *Rectangle* using two levels of inheritance

```
class Collection
base class for collections of objects
...
Collection Types
The subclasses of Collection implement different
ways of storing and accessing the objects...
Dynamic Creation and Object Copying (class Object)
see page 34.
...
```

Figure 7: Sample output with reference to an inherited section

Fig. 7 shows part of the documentation of a class *Collection*. The section *Dynamic Creation and Object Copying* is inherited from class *Object* and can be read on page 34 of the documentation. This avoids waste of paper in printed documentation. For online documentation the inclusion of inherited sections will enhance readability and avoid the excessive use of links. Then too, it may make the document overly redundant.

It is also useful to have a table of contents for a unit, where for each section (including the inherited ones) the corresponding unit and the page number (printed documentation) or a link (online documentation) are specified.

Fig. 8 shows the online output, i.e., instead of page numbers links are provided for direct access to the various sections. This mechanism can also be used in order to list all methods of a certain class and provide references to their documentation.

2.6 Documentation Views

Information filtering is important for efficient access to huge amounts of information. Defining categories for documentation sections is a simple, yet powerful mechanism to provide various views on a document and meet different documentation needs of various readers. Fig. 9 shows what information might be provided to a casual user of *dbxtool*. The presence of an arrow in the figure indicates that the corresponding section is part of the view. Please note that sections that are not used for that view do not have a horizontal line (—), i.e., they are not hidden. Hidden sections are not available for any view, whereas sections without an arrow may be used for other views. For example, a professional user would get the other sections as well.

When documenting source code, a useful control mechanism is the distinction among private, protected and public sections, as is done in the programming language C++. This distinction determines access rights for clients, heirs and friends of classes. Public sections

```
[*] class Collection
base class for collections of objects

[*] List of Sections
 1. List of Sections (Collection), see page [<-]
 2. List of Methods (Collection), see page [->]
 3. Description (Collection), see page [->]
 4. Memory Management (Collection), see page [->]
 5. Collection Types (Collection), see page [->]
 6. Retrieval of Elements (Collection),
    see page [->]
 7. Iterators (Collection), see page [->]
 8. Enumerating Objects (Collection),
    see page [->]
 9. History (Collection), see page [->]
10. Class Descriptors and Dynamic Type-Checks
    (Object), see page [->]
11. Dynamic Creation and Object Copying
    (Object), see page [->]
12. Object Input/Output (Object), see page [->]
13. Object Comparison (Object), see page [->]
14. Change Propagation (Object), see page [->]
15. Flag Handling (Object), see page [->]
...
```

Figure 8: Sample (online) output with a table of sections

can be read by everyone and are devoted to describing how to use a class. Protected sections contain more detailed information that is needed to build subclasses. Finally, private sections contain additional implementation details that are exclusively intended for development and maintenance personnel (see Fig. 10).

The whole documentation of a class (or a method) is visible only for friends. Reusers who build subclasses (heirs) see only a subset of this documentation; they do not have access to private sections, which typically describe implementation details (*Implementation* sections in Fig. 10). Clients' access is further restricted to public sections, which contain general interface descriptions (*Description, Layout, Method Descriptions* and *Interfaces* in Fig. 10). Please note that, similarly to the source code, private sections of the documentation are not inherited; i.e., private documentation of the classes *Rectangle* would not become part of the documentation of any subclass thereof.

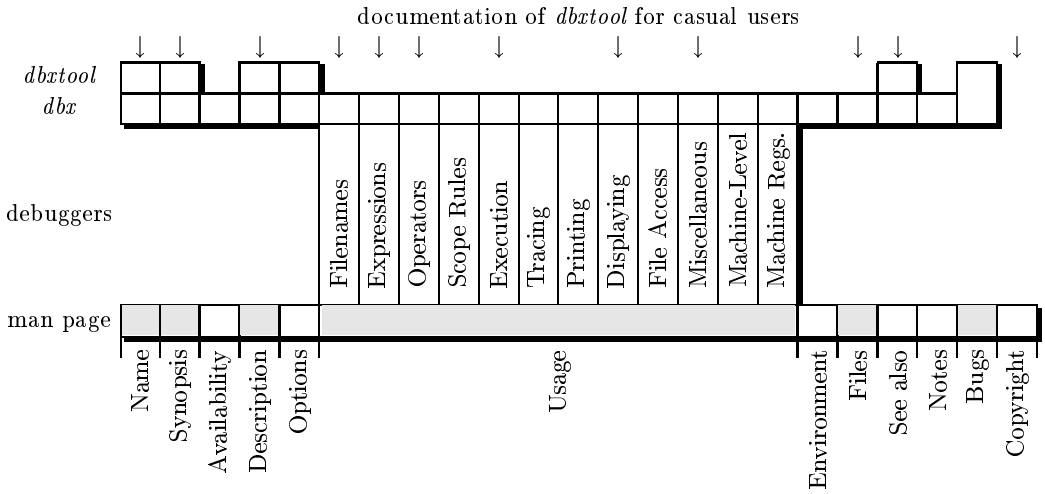

Figure 9: A casual user's view of *dbxtool*'s documentation

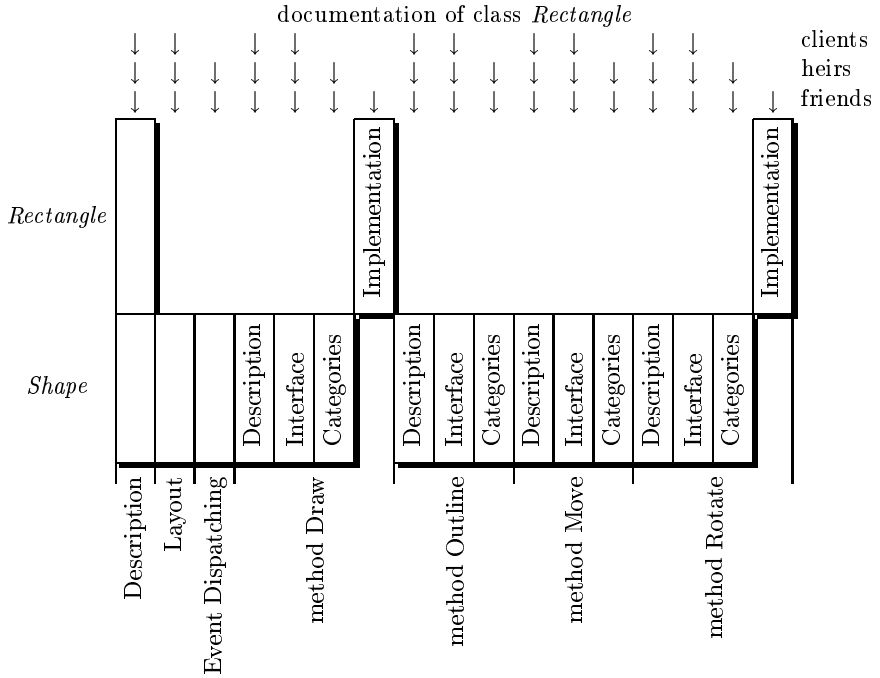

Figure 10: Documentation sections for clients, heirs, and friends

3 Oonoweb

In order to realize the ideas presented in the previous section we have taken an existing literate programming tool as a starting point and augmented it with the presented features. The following sections describe the literate programming system noweb, why we selected it, and how these features can be incorporated into it.

3.1 Literate Programming

Programs are written to be executed by computers rather than to be read by humans. However, when writing programs, the goal of telling humans what we want the computer to do should be more important than instructing the computer what to do [3]. The idea of literate programming is to make programs as readable as ordinary literature. The primary goal is not just to get an executable program but to get a description of a problem and its solution (including assumptions, alternative solutions, design decisions, etc.).

We agree that literate programming is a process leading to more carefully constructed software systems with better documentation. Most literate programming tools automatically provide extensive reading aids like tables of contents and indexes. We believe that these tools can and should be used for the entire documentation of software systems. Of the entire documentation only a small part will have source code included. The advantage is that the whole system is documented in a consistent way, and as will be shown in the subsequent sections, documentation reuse can be applied very easily. Naturally, developing large software systems requires tools like browsers in addition to a simple documentation system. In this paper we will concentrate on the reuse aspect.

3.2 The noweb System

Noweb is a literate programming tool like WEB (see [3]). A noweb document consists of a series of chunks that can appear in arbitrary order. Each chunk contains either code or documentation. For additional structuring LaTeX commands like \section, and \subsection can be used. Indexing and cross-referencing information can be provided for chunks and for programming language identifiers. Fig. 11 is taken from the noweb distribution. It consists of a documentation chunk followed by a code chunk. Double brackets ([[]]) in the documentation text enclose source text. Double angles (<<>>) in the code determine other chunks.

The design of noweb was purposefully 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. Noweb has been in use for many years at Princeton and elsewhere for tens of thousands of lines of code in languages as awk, C, C++, Icon, Modula-3, PAL, perl, Promela, and Standard ML [7].

```
@ This program has no input, because we want
to keep it simple.  The result of the program
will be to produce a list of the first thousand
prime numbers, and this list will appear on the
[[output]] file.

Since there is no input, we declare the value
[[m = 1000]] as a compile-time constant.  The
program itself is capable of generating the
first [[m]] prime numbers for any positive
[[m]], as long as the computer's finit
limitations are not exceeded.

<<program to print the first thousand prime
  numbers>>=
program print_primes(output);
  const m = 1000;
    <<other constants of the program>>
  var <<variables of the program>>
  begin <<print the first [[m]] prime numbers>>
  end.
```

Figure 11: A Simple noweb Example

Noweb works with any programming language and supports TeX, LaTeX, and HTML back ends. Thus, you can either produce printed or online documentation. Cross-references in printed documentation are provided by means of page numbers and links in online documentation. For more information on literate programming and the noweb system we refer the reader to [3, 6] and [7, 8], respectively.

Noweb is implemented with pipes, i.e., noweb source is transformed into an intermediate code, to which various filters are applied in order to accomplish certain tasks, such as indexing and cross-referencing (see [2] on software architectures). At the end of the pipe are filters that make a transformation to either TeX, LaTeX, or HTML. Due to this architecture extending the functionality of noweb can easily be done by implementing new filters. This is what we do in order to implement the concepts of documentation reuse, which is described in the next section.

3.3 Units

A documentation unit typically describes a logical unit such as a software component, i.e., a module, a class, an asset. The documentation is usually divided into sections, subsections, and paragraphs in order to describe various aspects of that unit. The example in Fig. 12 shows how the input for the manual pages of the source-level-debugger *dbx* would look like. A unit can be the subunit of another unit. This is specified with the `\baseunit` command. In Fig. 12 unit *dbx* is subunit of `man_page` and inherits documentation from it, i.e., the sections defined therein.

```
\unit{dbx}
\baseunit{man_page}

\section{Name}
dbx - source-level debugger

\section{Synopsis}
dbx [-f fcount] [-i] [-Idir] [-k]
  [-kbd] [-P fd] [-r] [-s startup]
  [-sr tstartup] [objfile [corefile|process-id]]

\section{Availability}
This command is available with the Debugging
software installation option.  Refer to
Installing SunOS 4.1 for information on how
to install optional software.

\section{Description}
dbx is a utility for source-level debugging and
execution of programs written in C, or other
supported languages such as Pascal and
\FORTRAN~77.  dbx accepts the same commands
as dbxtool(1), but uses a standard terminal
(tty) interface.
...
```

Figure 12: Sample `oonoweb` documentation unit

3.4 Sections and Subsections

Sections and subsections are used for the distinction of two levels of inheritance. Whenever a section is contained in a baseunit but not in the subunit, then the whole section is inherited. If a section is available in both the baseunit and the subunit, then the section is overridden. If, however, sections contain subsections, then inheritance is applied to these subsections as well. Subsections can also be inherited or overridden, compare Fig. 12 and Fig. 13.

```
\unit{dbx}
...
\section{Usage}
Refer to dbx in the Debugging Tools manual.
The most useful basic commands about are run;
run the program being debugged; where, obtain
a stack trace with line numbers; print,
display variables; and stop, set breakpoints.
\subsection{Filenames}
Filenames in dbx may include shell meta-
characters.  The shell used for pattern matching
is determined by the SHELL environment variable.
\subsection{Expressions}
dbx expressions are combinations of variables,
constants, procedure calls, and operators.
Variables are either variables in the program
being debugged or special dbx variables whose
names begin with $.
...
```

Figure 13: Sample `oonoweb` sections and subsections

3.5 Extensions and Concealments

In the example in Fig. 2 we have seen that it is sometimes necessary to either conceal (section *Environment*) or extend inherited sections (section *Bugs*). Fig. 14 shows how this can be achieved with the commands `\conceal` and `\extend`. The documentation of *dbxtool* will not contain the *Environment* section and will have the *Bugs* section extended by the text given.

```
\unit{dbxtool}
...
\section{Environment}
\conceal
\section{Files}
core default core file
.dbxinit local dbx initialization file
\section{Bugs}
\extend
The interaction between scrolling in the source
subwindow and dbx's regular expression search
commands is wrong.  Scrolling should affect
where the next search begins, but it does not.
```

Figure 14: An Extended and a Concealed `oonoweb` Section

3.6 Inclusions and Cross References

If `\xrefInherit` is specified for a unit, then cross-references are included, if `\includeInherit` is specified, then the inherited documentation is physically included. If none of these commands is specified, then– similar to source code–inherited documentation is not mentioned at all, except when a section is extended (see section 3.5) then the inherited text is included by default.

Commands `\xrefInherit` and `\includeInherit` can be specified for individual sections. This gives the user the possibility to reference big sections and include small ones. Fig. 15 contains an example of included, referenced, and extended sections.

A table of contents (i.e., a list of sections) can simply be included with the `\sectionlist` command (see Fig. 16). For each section it specifies the corresponding unit and the page number (printed documentation) or a link (online documentation). The same holds for the second level of inheritance, for which `\subsectionlist` is used.

3.7 Views

In order to provide various views to the documentation we offer the possibility to apply attributes to sections. To each section (or subsection) various attributes can be specified by using the `\attribute` command. Additionally, with `\attributeOn` and `\attributeOff` it is possible to specify attributes for more than one section. In Fig. 17 the attribute `professionalUser` is specified globally, i.e., for all sections, whereas the attribute `casualUser` is specified only for certain parts of the documentation.

The output of the documentation can be controlled with a command line option. Various views can be specified for the documentation and cause the output of sections that have the attribute specified. Where sections or subsections are inherited, they are also considered for output if they have the attribute specified. Thus, multiple views of documents can be generated.

4 Examples

The concepts discussed in the previous sections 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 examples of the previous sections have given a glimpse of how documentation can be reused in the manual page domain, i.e., in user documentation of tools and applications.

```
dbxtool
Name
dbxtool - SunView interface for the dbx
source-level debugger
Synopsis
dbxtool [-d] [-i] [-k] [-kbd]
  [-I directory] [objectfile [corefile]]
Availability
see Availability in dbx.                    xref
Description dbxtool, a source-level
debugger for C, Pascal and FORTRAN 77
programs, is a standard tool that
runs within the SunView environment.
It accepts the same commands as dbx,
but provides a more convenient user
interface.
Usage
see Usage in dbx.                           xref
Files                                       incl
core default core file
.dbxinit local dbx initialization file
...
Notes                                       xref
see Notes in dbx.
Bugs
see Bugs in dbx.                            xref
The interaction between scrolling in
the source subwindow and dbx's regular
expression search commands is wrong.
Scrolling should affect where the
next search begins, but it does not.
```

Figure 15: Included, referenced, and extended (*Bugs*) sections

Knuth's TEX and METAFONT are implemented/documented as literate programs [4, 5] are excellent examples where documentation could be reused to a big extent. We have studied these [1] and determined extensive implicit, ad-hoc reuse was done. This ad-hoc reuse can easily be made explicit by using the concepts for documentation reuse presented in this paper. In fact, the results of this investigation strongly motivated us in providing the prerequisites of explicit documentation reuse.

```
\unit{dbxtool}

\section{Section List}
\sectionlist

\section{Name}
...
```

Figure 16: Section list

```
\unit{dbxtool}
\baseunit{dbx}
\attributeOn{professionalUser}
...
\section{Name}
\attribute{casualUser}
...
\section{Synopsis}
\attribute{casualUser}
...
\section{Description}
\attribute{casualUser}
...
\section{Options}
...
\section{See also}
\attribute{casualUser}
...
\section{Bugs}
...
```

Figure 17: Attribute specification for the *dbxtool* documentation

Documenting an object-oriented software system provides the most obvious applicability of documentation reuse. If systems documentation and source code are similarly structured, documentation reuse is a matter of fact. This had been partly shown in [9] already.

5 Conclusion

We have presented a comfortable and natural means of reusing any kind of documentation. This can be done by defining common structures, extracting common information, extending and modifying sections, and defining various views on documentation. The introduced concepts are being implemented as an extension to the `noweb` literate programming system. The future goal is to have software systems built from reusable components and to have their documentation built upon these components' documentation. Even though we are still a long way from that scenario, literate programming and explicit documentation reuse can help in improving the quality of our software systems and in increasing the productivity of software engineers.

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 via `oonoweb` 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.

References

[1] Childs B., Sametinger J.: "Literate Programming from the Viewpoint of Reuse," to appear.

[2] Garlan, D., Shaw, M.: "An Introduction to Software Architecture," in "Advances in Software Engineering and Knowledge Engineering," Vol. 1, World Scientific Publishing Company, 1993, and Carnegie Mellon University, Technical Report, CMU/SEI-94-TR-21.

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

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

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

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

[7] Ramsey N.: "Literate programming simplified." *IEEE Software*, Vol. 11, No. 5, pp. 97-105, September 1994.

[8] Ramsey N.: "The noweb Hacker's Guide," August 1994.

[9] Sametinger J.: "Object-oriented Documentation," *ACM Journal of Computer Documentation*, Vol. 18, No. 1, pp. 3-14, January 1994.

The current address of Johannes Sametinger is
Department of Computer Science
Box 1910
Brown University
Providence, RI 02912

Current email addresses:
`js@cs.brown.edu`
`bart@cs.tamu.edu`