Reuse Documentation and Documentation Reuse

Johannes Sametinger
Department of Computer Science,
Texas A&M University, TX, U.S.A.
jsam@cs.tamu.edu
and
CD Laboratory for Software Engineering,
Johannes Kepler University of Linz, Austria
sametinger@swe.uni-linz.ac.at

Abstract

The reuse of application frameworks and class libraries can improve the productivity in software development considerably. Object-oriented techniques, i.e., inheritance and information hiding, that ease reusing software, can be applied to documentation and thus, enable documentation reuse.

One can document a software component from scratch—regardless of what a component is. This leads to multiple documentation of features that are multiply reused and may easily result in inconsistencies. One can also describe a component's differences to other components. This seems logical for systems documentation of object-oriented software. However, this kind of reuse can also be applied to documentation, where there is no source code involved at all.

The information needed for the reuse of software components is not provided by traditional documentation, which is often divided into project documentation, user documentation, and systems documentation. In this paper we define the contents and structure of reuse documentation and apply the concepts of documentation reuse. This results in a reuse documentation hierarchy that defines the structure of various kinds of reusable components and supports consistency and completeness of the information.

Keywords: documentation, reuse, object-oriented programming, reuse documentation, documentation reuse, literate programming

---

This work has been supported by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung*.

1. Introduction

The first step in the reuse process is finding and selecting suitable components. For effective reuse it must be easier to find components than to develop them from scratch. Finding suitable components does not mean finding exactly what is needed. Locating similar components can be sufficient. After components have been found, they must be understood in order to reuse them. Finding and understanding are related, because to select a component for reuse one must know what the component does. Understanding becomes even more important when the component has to be modified. Adequate documentation is significant for this step.

Building a software system out of a bunch of unmodified components is the ideal scenario. Typically, at least some of them will have to be adapted to specific needs of the particular software system to be built. Components can be modified in various ways, e.g., by changing internals or by adding new features. When a component offers the required functionality it has to be incorporated into the software system.

The goal is to maximize reuse and to minimize basic development efforts. But typically, existing components will not suffice to build a new system. At least a few components will have to be built from scratch. Ideally, these components will be inserted into the component library to facilitate their reuse in later projects.

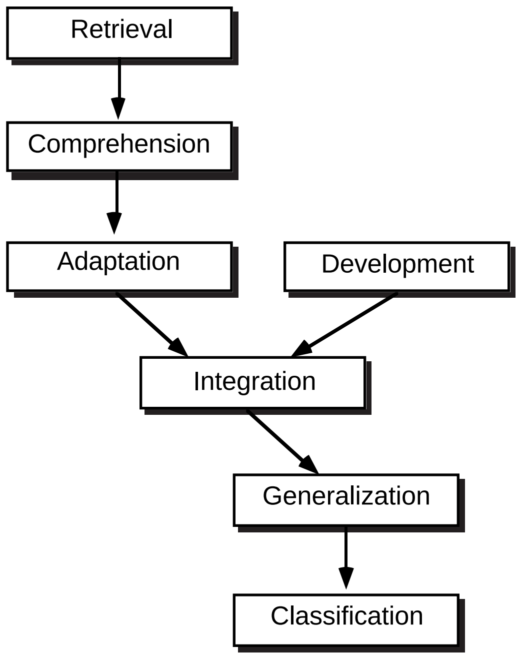

Fig. 1-1: Various Steps of Software Reuse

Fig. 1-1 summarizes the steps encountered in the reuse process. On the upper left side is the process of reusing an existing component (steps Selection, Comprehension, and Adaptation). Development is necessary when no reusable components can be identified. Integration has to be done regardless of whether an old component is reused or a new one has been developed. Finally, a new component or a modified old one is generalized—if necessary, classified, and inserted into a component repository for future reuse. In this paper we will concentrate on the information needed for effective reuse. We refrain from classification techniques and repository aspects, but rather deal with the systematic organization of documentation and the methodical reuse not only of software components but also of their documentation.

2. Component Comprehension

Information about various aspects has to be provided in order to effectively and correctly reuse a software component. This includes information that enables the evaluation of components in a set of possible candidates, the understanding of a component's functionality, the use of a component in a certain environment, and the adaptation of a component for specific needs. Good documentation of components is essential to software reusability. Traditional documentation usually does not meet these needs. In the literature a lot of advice has been given about what should be provided for reusable software components. The following entries have been distilled from [Braun 94, Karlsson 95, Krueger 92, Meyer 94, NATO]:

- *Component Name*: name, possibly giving a hint about a component's functionality
- *Identification*: Is the component a candidate for potential reuse in a certain scenario? (i.e., a

clear, concise initial statement about the component's functionality for initial selection)?

- *Specification*: What is the component's functionality in full detail?
- *Status*: What is the quality, test, maintenance, financial, etc. status of the component?
- *Technical Restrictions*: What are the technical restrictions on the use of the component (e.g., capacities, programming language, operating system dependencies)?
- *Commercial or Legal Restrictions*: What are the commercial or legal restrictions on the use of the component (e.g., purchase, special license, or permission required)?
- *Problems*: Are there any outstanding problem reports (e.g., known bugs, desired enhancements)?
- *Recommended Enhancements*: Are there any known possible enhancements (e.g., to improve performance/maintainability, make the component more robust, extend the scope of reuse)?
- *Resource requirements*: What (amount of) system resources is required for using the component (e.g., memory, processor, communication channels)?
- *Installation*: How is the component adapted to a new application?
- *Adaptation*: How and to what specific needs can the component be adapted?
- *Usage*: How can the component and/or its functions be used (correctly)?
- *Support*: What is the point of contact to get help (e.g., in adapting the component)?
- *Relations to other components*: Can this component be used stand-alone or must other components be used with this one together?
- *Alternatives*: Are there similar components, that could be used instead of this one?
- *History*: What is the history and current version of the component (including all prior versions, their developers, and dates of release)?
- *Test support*: Are a test environment and/or test cases available for the component?
- *Classification*: How is the component classified to enable future retrieval (depending on the used classification scheme)?
- *Implementation*: How is the component implemented?

The amount and kind of information needed strongly depends on the form of the reusable component. Reusing assembler routines requires different information than reusing an object-oriented class or a self-contained application. It is important that the documentation is considered an essential part of the software component. Without proper documentation a component is useless. Neither can it be retrieved when needed, nor can it be reused and adapted with reasonable effort. Documentation standards have to be established in order to guarantee the availability of important information and the completeness and consistency of this information. A consistent structure makes the documentation more readable and better understandable. It helps the reuser in finding relevant parts and decreases the time needed for evaluation, actual reuse, and adaptation of components.

Another important aspect is that each component has its self-contained documentation. Letting the reuser filter out a component's documentation from a big document describing a set of components is bad practice. A component's requirements, design, test, reuse information must stand alone and should be minimal in dependencies and references to other documents. Additional requirements to the documentation are its availability in machine-readable form. This allows us to reuse it for derivative components and modify it according to the modifications made to the component. Clarity and understandability should be a matter of course for any documentation. It is of special importance in fostering reuse.

In Section 3 we will describe the concepts for documentation reuse that help in defining documentation structures, and keeping documentation complete and consistent. In Section 0 we will apply these concepts to information needed for reuse. Finally, in Section 5 we will deal with oft encountered inconsistencies between source code and documentation.

3. Documentation Reuse

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 without need of modification),
- definition of various views for different kinds of readers, e.g., casual users and 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 5). We start this chapter with a recapitulation of the originating concept, source code inheritance.

3.1 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 one class. From a documentor's point of view, classes and methods seem to be equivalent to modules and procedures used in conventional

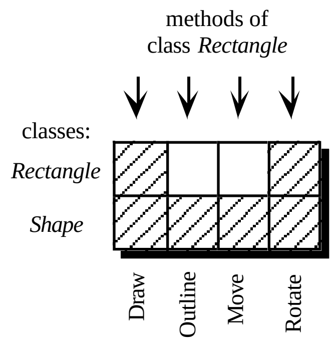

Fig. 3-1: Methods of classes *Shape* and *Rectangle*

programming. One of the main differences between modules and classes is the inheritance relationship between classes. A class may inherit the structure and behavior of another class and additionally 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. 3-1).

The hatched boxes in Fig. 3-1 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.

3.2 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.

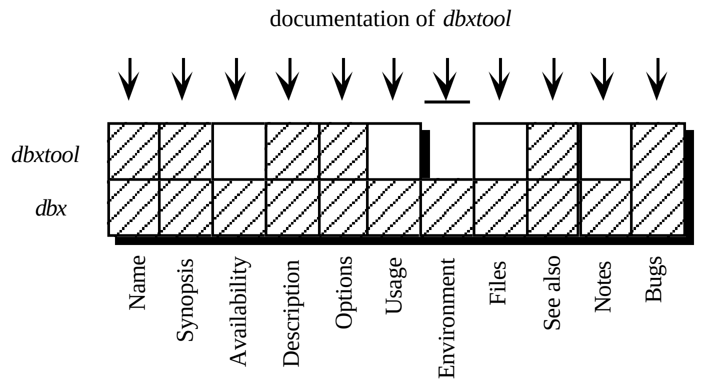

Fig. 3-2: Inherited, overridden, extended, and hidden documentation sections of *dbxtool*

Fig. 3-2 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. 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 [Sametinger 94].

3.3 Documentation Abstraction

In object-oriented programming, abstract classes are designed as parents from which subclasses may be derived. Abstract classes are not themselves suitable for instantiation. They are used to predefine certain structures and behaviors which are then shared by groups 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.

Fig. 3-3 is another view of the documentation of *dbxtool* in terms of documentation abstraction. The documentation for "abstract manual 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. 3-3 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 the completeness of documentation and—if incomplete—in spotting the missing sections. The abstract documentation in Fig. 3-3 contains an additional section *Copyright*, which is automatically included for all descriptions inherited thereof.

Fig. 3-4 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

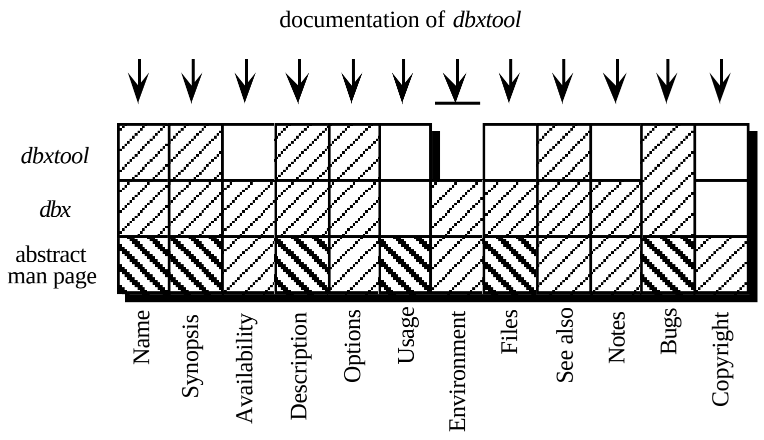

Fig. 3-3: Sections of *dbxtool* using documentation abstraction

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.
…
See also
OS 4.1 Programmer's Guide
Notes
no notes
Bugs
---information has to be provided---
Copyright
© by Company XYZ, 1996

Fig. 3-4: Possible documentation abstraction for *manual pages*

page) in the documentation indicates that there are still missing parts, i.e., sections to be written.

3.4 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. 3-5, 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. 3-6 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*. Considering more than two levels of inheritance is possible. However, we

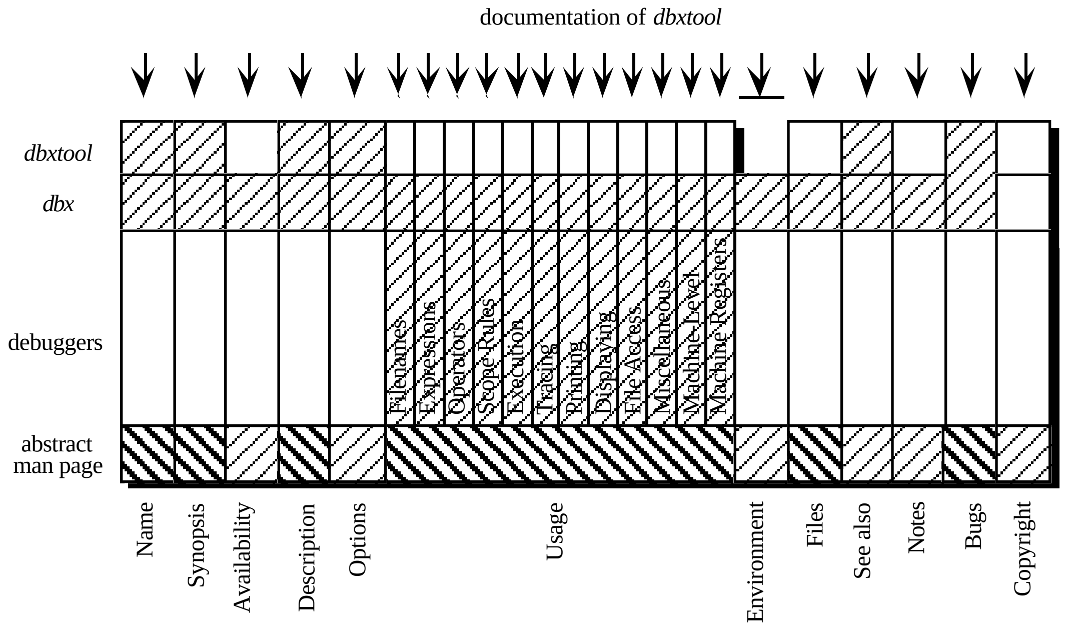

Fig. 3-5: Sections of *dbxtool* using two abstractions and two levels of inheritance

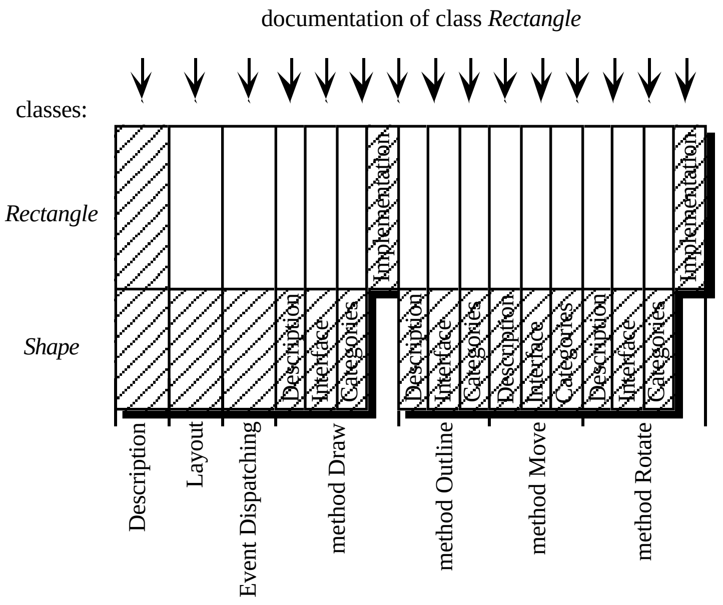

Fig. 3-6: Sections of class *Rectangle* using two abstractions and two levels of inheritance

contemplate two levels only, because so far we have not encountered a practical example where two levels were not sufficient.

3.5 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). Fig. 3-7 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. In printed documentation references to page numbers is preferred in order to avoid waste of paper. For online documentation the inclusion of inherited sections may 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

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.
…

Fig. 3-7: Sample output with reference to an inherited section

[*] 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 [->].

…

Fig. 3-8: Sample (online) output with a table of sections

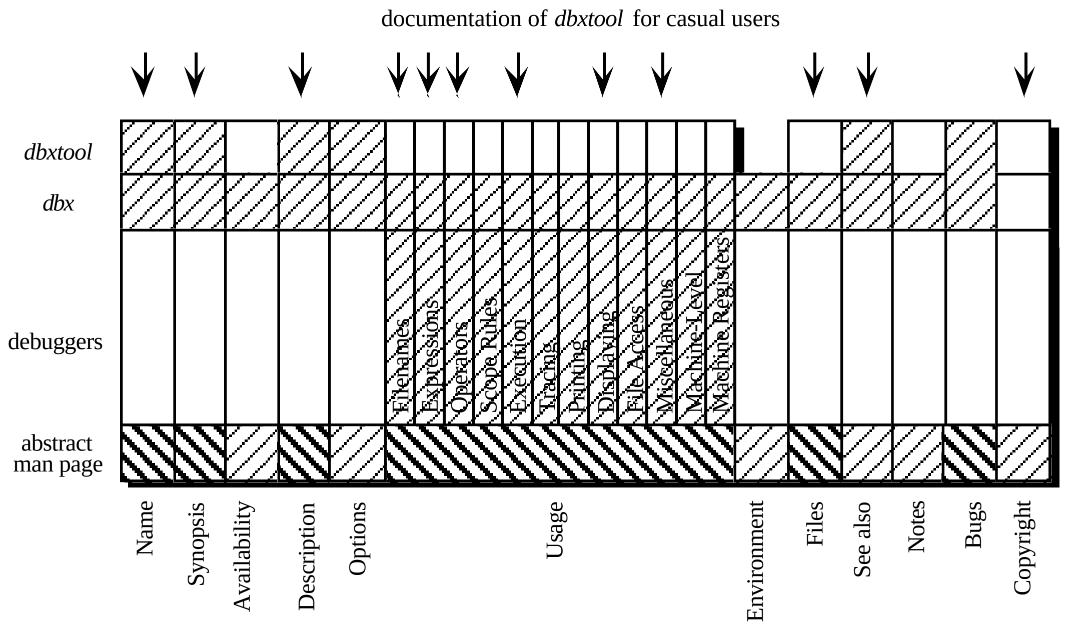

Fig. 3-9: A casual user's view of *dbxtool*'s documentation

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. 3-8 shows the online output, where links rather than page numbers are provided for direct access to the various sections.

3.6 Documentation Views

Nowadays huge amounts of information is readily available. Information filtering is important for efficient access. 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. 3-9 shows what information might be provided to a casual user of *dbxtool*. 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 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. 4-10). 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. 4-10). Clients' access is further restricted to public sections, which contain general interface descriptions (*Description*, *Layout*, *Method Descriptions* and *Interfaces* in Fig. 4-10).

4. Reuse Documentation

According to documentation abstraction demonstrated in the previous sections, reuse documentation should be defined and created for each component. The size, layout and contents of such a manual may vary according to the type of components. The information mentioned in Section 2 should be combined in a single document to be created for reuse purposes. We suggest an outline consisting of four different parts. Part I contains

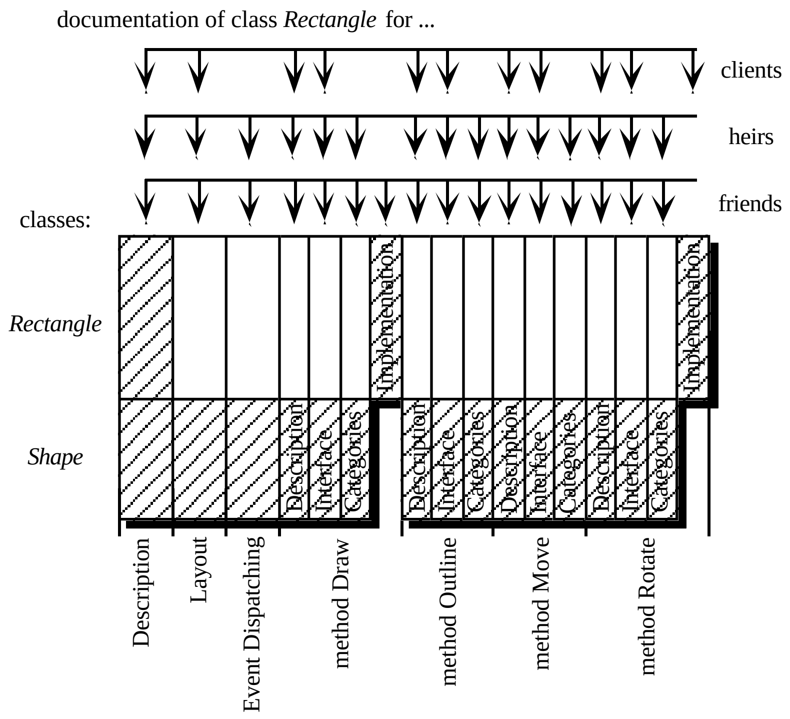

Fig. 4-10: Documentation sections for clients, heirs, and friends

general information about a component for evaluation purposes. It should provide enough information to decide whether a component is a possible candidate in a certain reuse scenario, but refrain from being too detailed. If the information in this part contains too many details the evaluation process will be slowed down. However, a final decision on which component to choose out of a set of possible candidates may require the inspection of information of the other parts also. Part II contains the essential information for actual reuse. It should carry all the details necessary for installing, using, and adapting the component. Part III contains administrative information like legal constraints and available support. Part IV contains more detailed information for the evaluation of a component like known bugs, limitations, and quality statements. Any other information not covered by the first four parts is subsumed in Part V.

PART I: General Information

1. *Introduction*
   name, identification, overview of component
2. *Classification*
   information used for the classification of the component like a list of keywords, type of component (e.g., C++ class, C function, OpenDoc application)
3. *Function*
   overview of all externally visible operations

PART II: Reuse Information

4. *Installation*
   steps (if any) to be done to incorporate the component into a system, e.g., installation of an application
5. *Integration*
   detailed information for effective reuse of the component, including interfaces, sample scenarios, diagnostic procedures (what to do if a problem occurs)
6. *Adaptation*
   means of adaptation to specific needs with detailed information about how to accomplish this, e.g., available options, subclassing

PART III: Administrative Information

7. *Procurement and Support*
   source (if component is not directly available in repository), ownership (any legal or contractual restrictions), maintenance (available support, points of contact)
8. *History*
   version history, dates of releases, main differences to old versions

PART IV: Evaluation Information

9. *Quality*
   information about verification, applied tests, test results, available test data, retesting procedures
10. *Performance*
    e.g., assumptions, resource requirements (disk, CPU, main memory)
11. *Alternative components*
    any known components that might serve as an alternative to this one
12. *Known Bugs*
    any outstanding problem reports (e.g., known bugs, desired enhancements)
13. *Limitations*
    e.g., capacities, programming language, operating system dependencies
14. *Possible Enhancements*
    any possible enhancements, e.g., to improve performance/maintainability, make the component more robust, extend the scope of reuse
15. *Interdependencies*
    any dependencies to other components, requirements to the environment

PART V: Other Information

16. *Index*
    Providing an index should be considered for complex components that require extensive documentation.
17. *References*
    references to literature or other documentation (e.g., systems documentation)

Such an outline should be defined and consistently used for all components in a repository. Naturally, adaptations may be appropriate depending on the nature of the components being stored. Documentation inheritance (see Section 3.2) allows the definition of a hierarchy of documentation outlines for different kinds of components, as is depicted in Fig. 5-1. This guarantees consistent documentation structure for all components with adaptations according to the type of a component. If a component is reused for the development of a software system, the component's documentation becomes part of the documentation of the entire system. Any adaptations made to the component have to be clearly documented as well. Ideally, this is also done by documentation inheritance without any direct modifications to the original documentation.

Functions and classes are the kind of software components that are most often reused today. They will, for example, not need any information on installation (entry 4). However, this may be essential information for the reuse of filters, applications, or megaprograms. These are examples of higher levels of abstraction for reusable software components. For example, filters can be reused by combining them in pipes [Garlan 93], standalone applications can be reused by means of command languages like Tcl [Ousterhout 94], and megaprograms are a conglomerate of huge, self-contained, stand-alone applications [Wiederhold 92]. Documentation inheritance can be further utilized by extracting information that is common to a set of reusable components, e.g., to all classes of a class library (see XYClass in Fig. 5-1).

5. Integration of Source Code

Source code components should be reusable without knowing their internals. However, sometimes it is necessary to modify a component by making direct changes to its implementation. This

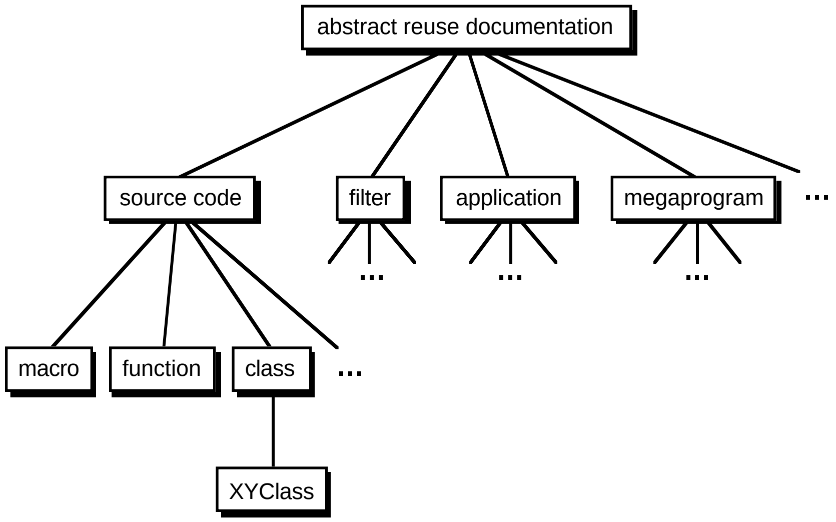

Fig. 5-1: Reuse Documentation Hierarchy

can be caused by the need to modify or enhance a component's behavior or to eliminate flaws or existing restrictions. It is important that the documentation describing the implementation of the component is available for that purpose and kept consistent with the changes made. The concept of literate programming supports this consistency. Naturally, legal restrictions may prohibit the availability of a component's implementation and allow its reuse as a black box only.

Donald Knuth stated that 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 [Knuth 92]. 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 documentation reuse can easily be applied.

Noweb is an example for a flexible literate programming tool [Ramsey 94]. 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. A noweb document consists of a series of chunks that can appear in arbitrary order. Each chunk contains either code or documentation. Indexing and cross-referencing information can be provided for chunks and for programming language identifiers. Noweb works with any programming language and supports TeX, LaTeX, and HTML back ends. Thus, it can either produce printed or online documentation. Cross-references in printed documentation are provided by means of page numbers and links in online documentation.

The implementation of any source code components should be documented in a literate way. In addition to that, reuse of documentation should be exploited. To experiment with these ideas we are currently working on augmenting the noweb system with features to support object-oriented documentation [Sametinger 95]. This will give us the opportunity to define documentation structures, reuse documentation, integrate source code into the documentation, produce high quality printed output, and produce online documentation with hypertext links.

6. 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 proposed structure for reuse documentation will not meet everyone's needs. However, it can serve as a start for the definition of such structures. The use of documentation hierarchies helps in presenting the commonalties and differences of various structures and in keeping documentation consistent, complete, and up-to-date. Furthermore, literate programming should be used in order to guarantee consistency with source code. The definition of various views on documentation enables the combination of information of different readers in single documents, like reuse documentation and systems documentation. This will also help in reducing redundancy and keeping information consistent.

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, explicit reuse documentation and documentation reuse will improve the quality of our software systems and increase the productivity of software engineers.

7. References

[Braun 94] Braun Christine, *Reuse*, in *Encyclopedia of Software Engineering*, by Marciniak John J. (Editor-in-Chief), Vol. 1, pp. 1055-1069, John Wiley & Sons, 1994.

[Garlan 93] Garlan, D., Shaw, M.: *An Introduction to Software Architecture*, in *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing Company, 1993.

[Karlsson 95] Karlsson Even-André: *Software Reuse: A Holistic Approach*, John Wiley & Sons, 1995.

[Knuth 92] Knuth Donald E.: *Literate Programming*, Leland Stanford Junior University, 1992.

[Krueger 92] Krueger Charles W.: *Software reuse*, Computing Surveys, Vol. 24, pp. 131-183, June 1992.

[Meyer 94] Meyer Bertrand: *Reusable Software: The Base object-oriented component libraries*, Prentice Hall, 1994.

[NATO] *NATO Standard for the Development of Reusable Software Components*, Vol. 1 (of 3 Documents), NATO Communications and Information Systems Agency.

[Ousterhout 94] Ousterhout John K.: *Tcl and the Tk Toolkit*, Addison Wesley, 1994.

[Ramsey 94] Ramsey N.: *Literate programming simplified*. IEEE Software, Vol. 11, No. 5, pp. 97-105, September 1994.

[Sametinger 94] Sametinger Johannes: *Object-oriented Documentation*, ACM Journal of Computer Documentation, Vol. 18, No. 1, pp. 3-14, January 1994.

[Sametinger 95] Sametinger Johannes: *Literate Programming and Documentation Reuse*, to be published.

[Wiederhold 92] Wiederhold Gio, Wegner Peter, Ceri Stefano: *Toward Megaprogramming*, Communications of the ACM, Vol. 35, No. 11, pp. 89-99, November 1992.