OBJECT-ORIENTED DOCUMENTATION

Johannes Sametinger

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

Abstract

Object-oriented programming improves the reusability of software components. Extensive reuse of existing software enhances the importance of documentation. In order to increase the productivity in documenting and to make the structure of documentation better suitable for object-oriented software systems, we suggest to apply object-oriented technology 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, interests of various groups of readers (e.g., reusers, maintenance staff) can be taken into account, and easy access to relevant information can be given.

In this paper we briefly describe a documentation scheme for object-oriented software systems. This scheme distinguishes among overview, external view and internal view of both static and dynamic aspects of software components. Then we apply inheritance by simply reusing and extending existing documentation where appropriate, and enforce information hiding by providing an access control mechanism. This improves the reusability and accessibility of documentation. Additionally, we present an exemplary tool and relate our experience with object-oriented documentation.

Introduction

The object-oriented programming paradigm achieves a major improvement in the reusability of existing software components. However, increasing reuse intensifies the need for precise documentation to express the capabilities of reusable components and encourages the reuse of various components of existing documentation, too. Software documentation is usually divided into user documentation, system documentation and project documentation (see [ANS83, Pom86]). In this paper we concentrate on system documentation, which describes interfaces and implementation aspects to facilitate reuse and to enable maintenance. Documentation schemes for conventional software systems do not address the special needs of software reusers that arise with the widespread use of object-oriented programming. Reusing existing software will become a very important technique that will significantly

improve the productivity of programmers as well as the overall quality of software systems. To achieve this goal, we have to succeed in providing the right portion of information about which components come into question for reuse and how to reuse these components.

For those readers unfamiliar with the object-oriented paradigm, we introduce the terms that are used throughout this paper in the next section. This section can be skipped by readers that are used to object-oriented terminology.

Object-Oriented Programming Terminology

Objects: A running object-oriented software system consists of objects. Each object has structure and behavior. For example, a rectangle object's structure might consist of an *origin* and an *extent* (called instance variables) and its behavior might include *Draw* and *Rotate* (called methods). Objects become active by executing one of their methods, in which they can change their state and send messages to other objects, which in turn invokes the execution of the corresponding methods of those objects.

Classes, **methods**: The source code of an object-oriented software system consists of classes containing the variables (structure) and the methods (behavior). Objects with the same structure and behavior are described in one class. So far, from a documentor's point of view, classes and methods seem to be equivalent to modules and procedures used in conventional programming.

Inheritance: 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 superclass (or base class). The source code of the classes *Rectangle* and *Circle* contains only the modifications and extensions to the superclass *Shape* (see Fig. 1).

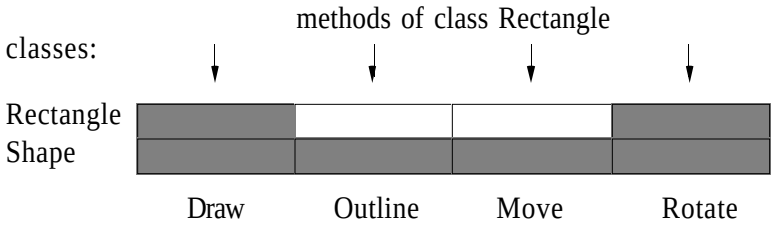

Fig. 1: Methods of classes *Shape* and *Circle*

The gray 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 superclass *Shape*. The methods *Draw* and *Rotate* are overridden; i.e., rectangle objects have their own draw and rotate methods, they do not use the (hidden) methods of the *Shape* class.

Extension, **modification, reuse**: Supported by the inheritance mechanism, extension and modification of existing classes is achieved by adding variables, and both adding and overriding methods. Usually object-oriented programmers do not write programs from scratch because it is often possible to reuse existing classes by extending and modifying them. In contrast to conventional programming, this can be done without changing the existing source code.

Class libraries, **application frameworks**: The reuse of classes becomes an inherent part of the object-oriented software development process. This results in collections of highly reusable classes called class libraries. Class libraries that define an application skeleton are called application frameworks. The skeleton consists of classes that have to be reused in order to easily implement an application with a modern graphic user interface.

Clients, **heirs**, **friends**: Classes, like modules, support information hiding. Variables and methods can be private (not accessible from outside), protected (accessible to subclasses), and public (accessible to the clients of a class) (see Fig. 2). This follows the C++ programming language (see [Str91]). Classes like *Rectangle* and *Circle* that inherit from class *Shape* are called its heirs; they have access to the public and protected parts of *Shape*. Classes that use *Shape*, e.g., by sending messages to shape objects, are called clients and only have access rights for the public part of *Shape*. Obviously objects of class *Shape* have access to the private part also. This access might be made available to a specific client class also, which we then call a friend of *Shape*. A friend relationship is

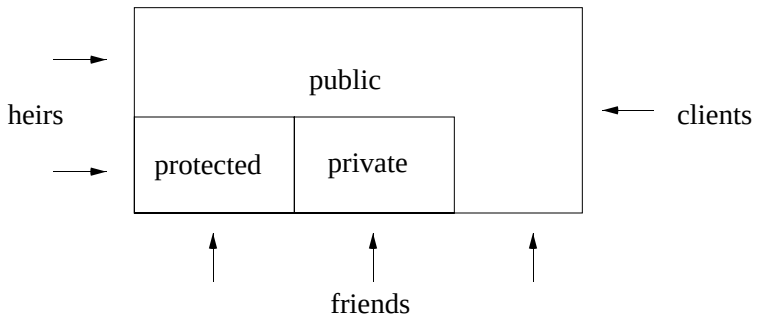

Fig. 2: Access rights of clients, heirs and friends

useful when classes are strongly coupled.

Reuser: Both programmers developing client classes and heir classes (subclasses) of a class are reusers of this class. For example, class *Shape* has been reused for developing class *Rectangle*, and class *Rectangle* could be reused to implement a graphics editor. Reusers are not interested in the implementation of the reused class and thus have different information and documentation demands than programmers who maintain the class or write friend classes.

Additionally, *polymorphism* and *dynamic binding* are terms that play a major role in object-oriented programming. However, they are of minor relevance for the documentation we describe in this paper. Therefore, we refrain from describing them here. For more detailed and more extensive descriptions of the concepts and terms described in this section, we refer the reader to [Mey87] and [Str91].

Class libraries and application frameworks must provide extensive documentation in order to facilitate reuse and achieve widespread acceptance. This documentation has to be integrated and reused in an application's documentation, just as the prefabricated software components are integrated into an application's source code. In the next section we briefly describe a scheme for system documentation of object-oriented software systems. (By software system we mean a class library, e.g., an application framework, or an application program.) Based on this scheme, we subsequently demonstrate how to improve the structure, the reusability, and the accessibility of various parts by applying object-oriented techniques.

Documentation Scheme for Object-Oriented Software

Typically, object-oriented software systems are extensions to class libraries or application frameworks. This characterization should become true for the documentation as well. Hence, such documentation does not describe the entire system from scratch; instead, it contains a description of all extensions and modifications of the reused components and describes all system-specific parts as well. It is assumed that separate library documentation is available which—similar to the code—provides the base for the entire documentation.

Different documentation is needed by people who maintain software components and people who reuse them. Hence, we distinguish between reuser information (needed for reuse) and implementation descriptions (needed for maintenance). The dynamic behavior of object-oriented software systems is usually more complex than that of conventionally implemented systems. Thus, we also differentiate static aspects of a system (its architecture) and its dy-

	overview	external view	internal view
static view	static overview	class interface description	class implementation description
dynamic view	dynamic overview	task interface description	task implementation description

Fig. 3: Documentation scheme for object-oriented software systems

namic behavior (e.g., control flow). Finally, an overview of a system is needed to make a decision on whether to reuse existing software components and to ease the familiarization process for programmers (reusers and maintainers). These different information needs of various groups of readers result in six different documentation parts (see Fig. 3), of which the two internal view parts are intended primarily to support software maintenance. The other four parts are also necessary for the maintenance personnel, but their primary goal is to facilitate the reuse of the software described.

Subsequently we briefly outline the six documentation parts. (For more details see [Sam93].)

Static Overview

The static overview contains the description of the overall implementation (e.g., supported platforms, hardware requirements, the programming language), the structure of the software system (e.g., components of the system comprising basic classes, application classes, container classes), the organization of the classes (e.g., class hierarchy, client relations), and brief descriptions of all classes.

Figure 4 shows the class hierarchy of the classes *Object*, *Shape*, *Rectangle*, and *Circle* used throughout this paper.

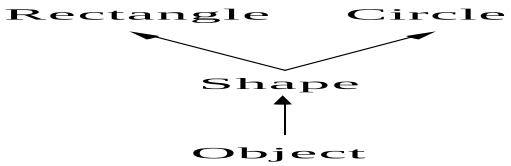

Fig. 4: Class hierarchy

Dynamic Overview

The dynamic overview describes the various concepts that are necessary to understand the dynamic behavior of the software system under consideration. Typical examples of these

concepts in an application framework for graphical user interfaces are event handling and general control flow, but also process and interapplication communication models, the handling of undoable commands, change propagation, and window and/or view updating policy.

A description of how rectangle and circle objects are informed about events (mouse click, key input) would be part of the dynamic overview of our example classes.

Class Interface Description

Class interface descriptions depict all classes from a static and external point of view. They should help to answer questions concerning the capabilities of classes and about how to use the various operations of a particular class. These descriptions are important references for programmers who either reuse or maintain a software system.

A short example description of class *Rectangle* could take the following form:

class Rectangle:	
short description:	The class Rectangle represents visual rectangle objects that are drawn on the screen. …
superclass:	Shape
methods:	
Draw (…):	The rectangle object is drawn on the screen.
Outline (…):	The outline of the rectangle object is drawn on the screen.
Move (…):	The coordinates of the rectangle object are changed according to the parameters.
Rotate (…):	The rectangle object is rotated according to the parameters specified.

Task Interface Description

Class libraries and application frameworks usually consist of hundreds of classes and thousands of methods. It is extremely difficult for programmers to use existing classes when only their interface descriptions are available. To facilitate reuse of complex libraries, programmers have to be provided with detailed information on what has to be done to fulfill a certain task, e.g.: Which classes have to be used? Can they be used directly, or is it necessary to derive subclasses? Which methods have to be overridden? Which messages have to be sent (and in which order)?

An example task interface description would be the description of how class *Shape* and its subclasses can be used with other classes to implement a simple graphics editor.

Class Implementation Description

The class implementation description characterizes all classes from a static and internal point of view. It should clarify the concept and the internal structure of a class so that even

software engineers not involved in the class's development can understand and maintain it. Class implementation descriptions are intended to be read by the maintenance (and development) personnel only, not by reusers. Important components of the class implementation description are the description of the internal structure of the class, the use of other classes (components), and the name, purpose and type of all methods and variables. Furthermore, methods that implement non-trivial algorithms should be described here. The class implementation description can be seen as an extension to the interface description. Hence, things described in the interface description do not have to repeated here.

A short example description of class *Shape* could look like this:

class Shape:

variables:

origin:	The variable *origin* identifies the left top point of the graphical object.
extent:	The variable *extent* specifies the right bottom point relative to the origin.

methods:

Outline (…):	The rectangle object is rotated according to the parameters specified.

Whether the variables *origin* and *extent* are described in the interface or the implementation description depends on whether they are accessible by clients or not.

Task Implementation Description

The task implementation description characterizes the implementation of important tasks from a dynamic and internal point of view. It contains descriptions about how the system (not an individual class thereof) fulfills a certain task, e.g.: Which classes/methods are involved in a task? Answers to these questions are important for anyone trying to comprehend the functioning of a software system. Consequently, task implementation descriptions—like class implementation descriptions—are primarily dedicated to maintenance programmers.

An example would be the description of how a simple graphics editor would determine which of the shape objects (existing at run-time) a user clicked at with the mouse.

Any documentation that can be directly related to a specific component of the source code, i.e., both the interface and the implementation descriptions of classes, are very well suited for reuse. Thus, in the following we will concentrate on documentation of classes and apply object-oriented techniques to both their interface and their implementation descriptions and thus improve reusability and extensibility of documentation as with the source code.

Reusing Documentation

The reuse of software components is facilitated by the inheritance mechanism. Inheritance can be viewed as both extension and specialization (see [Mey87]). Class *Rectangle* inherits from superclasses *Shape* and *Object*. The features of the superclasses are a subset of the features of class *Rectangle*; i.e., *Rectangle* inherits whatever *Shape* and *Object* provide and includes its own extensions. On the other hand, inheritance is used to realize an is-a relation. For example, *Rectangle* is a special visual object (*Shape*) with the features of a visual object but specialized behavior (specialization).

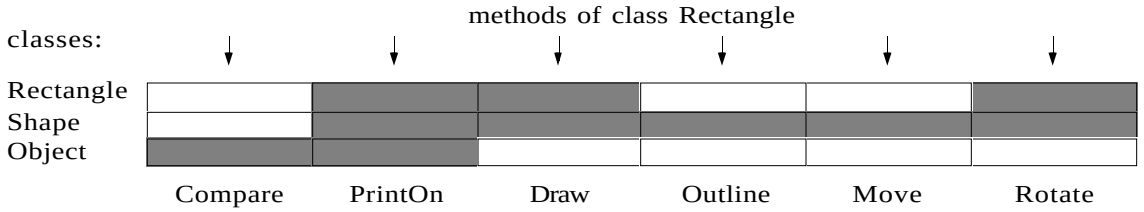

Fig. 5: Inherited and overridden methods of class *Rectangle*

Figure 5 graphically represents the inheritance mechanism. In this example class *Object* provides two methods, *Compare* and *PrintOn*. Class *Shape* (a subclass of *Object*) overrides *PrintOn* and adds the methods *Draw, Outline, Move*, and *Rotate*. Class *Rectangle* is a subclass of *Shape* and overrides the methods *PrintOn*, *Draw*, and *Rotate*. Again, the shaded boxes in Fig. 5 indicate the existence of methods. Class *Rectangle* provides the methods *PrintOn*, *Draw*, and *Rotate* of its own. The methods *Outline* and *Move* are inherited form *Shape*, *Compare* is inherited from *Object*.

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 viewpoint of a class' reuser 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 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. Examples of such sections are: short description, conditions for use, documentation of instance variables, and description of 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 class, other sections have to be added, e.g., event handling, change propagation.

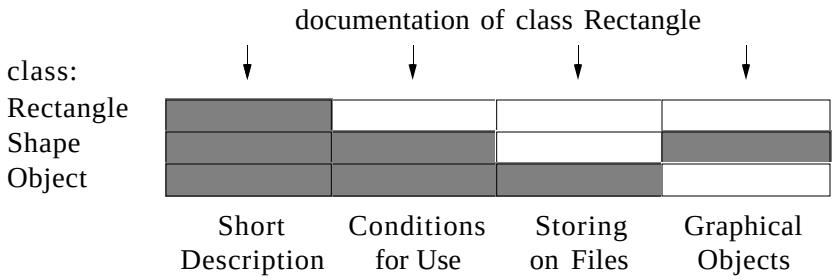

Fig. 6: Inherited and overridden documentation sections of class *Rectangle*

Figure 6 contains the structure of the documentation of the classes *Object*, *Shape*, and *Rectangle*. The documentation of class *Object* consists of 3 sections; classes *Shape* and *Rectangle* have four documentation sections. Class *Rectangle* inherits the section *Storing on Files* from the class *Object* and the sections *Conditions for Use* and *Graphical Objects* from class *Shape*; it has an own *Short Description*. Please note that the documentation of class *Rectangle* consists of four parts, though only a short description has been written for it.

The documentation of methods is organized the same way as that of classes (see Fig. 7). It is worth mentioning that there might be classes that do not implement a certain method. Naturally, they do not contain any documentation for this method. However, both the method and its documentation are available in these classes through inheritance.

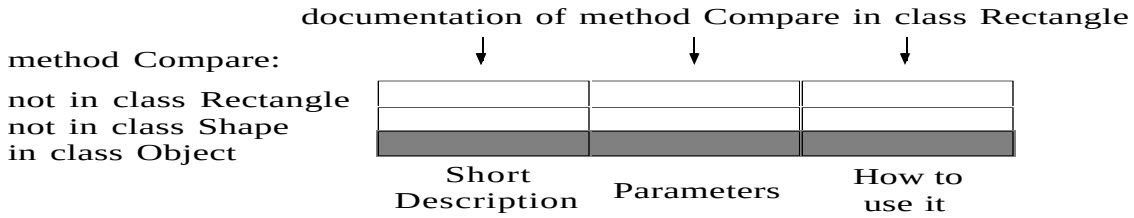

Fig. 7: Inherited and overridden documentation of a method

In Fig. 7 the original documentation of method *Compare* in class *Object* consists of the sections *Short Description*, *Parameters*, and *How to use it*. Classes *Shape* and *Rectangle* do not override the method *Compare* (see Fig. 5) and thus inherit both the method and its corresponding documentation from class *Object*.

The object-orientedness of documentation presented so far is useful for both interface descriptions (needed for reuse) and for implementation descriptions (primarily needed for maintenance), but no explicit distinction is made between them.

Control of Access to Documentation

Information hiding plays a major role in dealing with very large software systems. Besides hiding implementation details of classes, this means controlling access to variables and methods of classes. A useful control mechanism is the distinction among private, protected and public variables and methods, as is done in the programming language C++. This distinction determines access rights for clients, heirs and friends of classes.

This simple, yet useful mechanism can also be used for documentation sections in order to meet different documentation needs of various readers. 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 (additionally) needed to build subclasses. Finally, private sections contain additional implementation details that are exclusively intended for development and maintenance personnel (see Fig. 8).

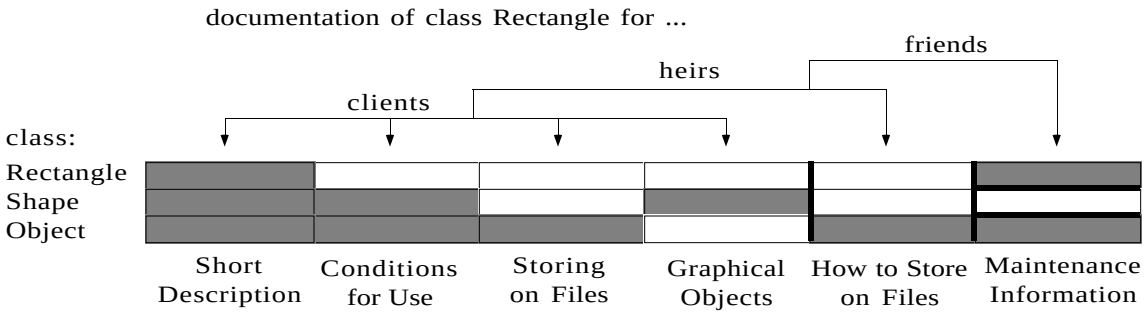

Fig. 8: Documentation sections for clients, heirs, and friends

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 (*maintenance information* in Fig. 8). Clients' access is further restricted to public sections, which contain general interface descriptions (*Short Description*, *Conditions for Use*, *Storing on Files*, *Graphical Objects* in Fig. 8). Please note that, similarly to the source code, private sections of the documentation are not inherited; i.e., private documentation of the classes *Object* and *Shape* is not part of the documentation of class *Rectangle*. In Fig. 8 this is indicated by the thick horizontal lines. The implementor (or friends) of class *Rectangle* cannot read the private implementation details of *Rectangle*'s superclasses unless the latter are declared as readable by heirs, i.e., unless they are described in protected sections.

Tool Support

The mechanisms of inheritance and information hiding are rather useless, unless they are supported by tools to give fast access to relevant parts of the documentation. A possible way to achieve this goal is to print the documentation sections for all classes and methods. This can be done in various ways:

- Only those documentation sections are printed that are specifically written for a certain class or method. This has the disadvantage that the documentation of all superclasses also has to be inspected in order to find out the whole story about a class.

- The inherited sections are also printed for each class and method. This eases reading the documentation considerably but requires multiple printing of sections. The sections either can be divided into groups for clients, heirs and friends, or the documentation can be separately printed for each of these user groups, which requires additional multiple printing of sections.

A better solution is to provide a tool that manages the sections of a software system and provides suitable documentation of the classes and methods for either clients, heirs, or friends. DOgMA (see [Sam92b]) is a tool that supports browsing mechanisms for classes and methods; i.e., it displays the methods of a class for clients, heirs and friends. The same mechanism is provided for documentation sections (see Fig. 9). The user interface consists of a menu bar, an information box (containing information like the currently displayed text or the inheritance), a text editor, and selection lists for classes, methods and documentation sections. The selection lists of DOgMA can be used to display various aspects of a software system. For example, it is possible to display a class's methods for clients, for heirs and for friends. Accordingly, the documentation sections relevant for clients, heirs or friends can be displayed in the lower right list.

In Fig. 9 the documentation section Windows and Icon of the class DProject is displayed; actually this section is inherited from the superclass Document. Whenever the user selects a class or method in the two selection lists at the upper right corner, the lower list displays all documentation sections that belong to this class or method. Each section title is preceded by the name of the class in which the section is defined. In Fig. 9 we can see that the various sections are inherited from the superclasses Document, EvtHandler, and Object.

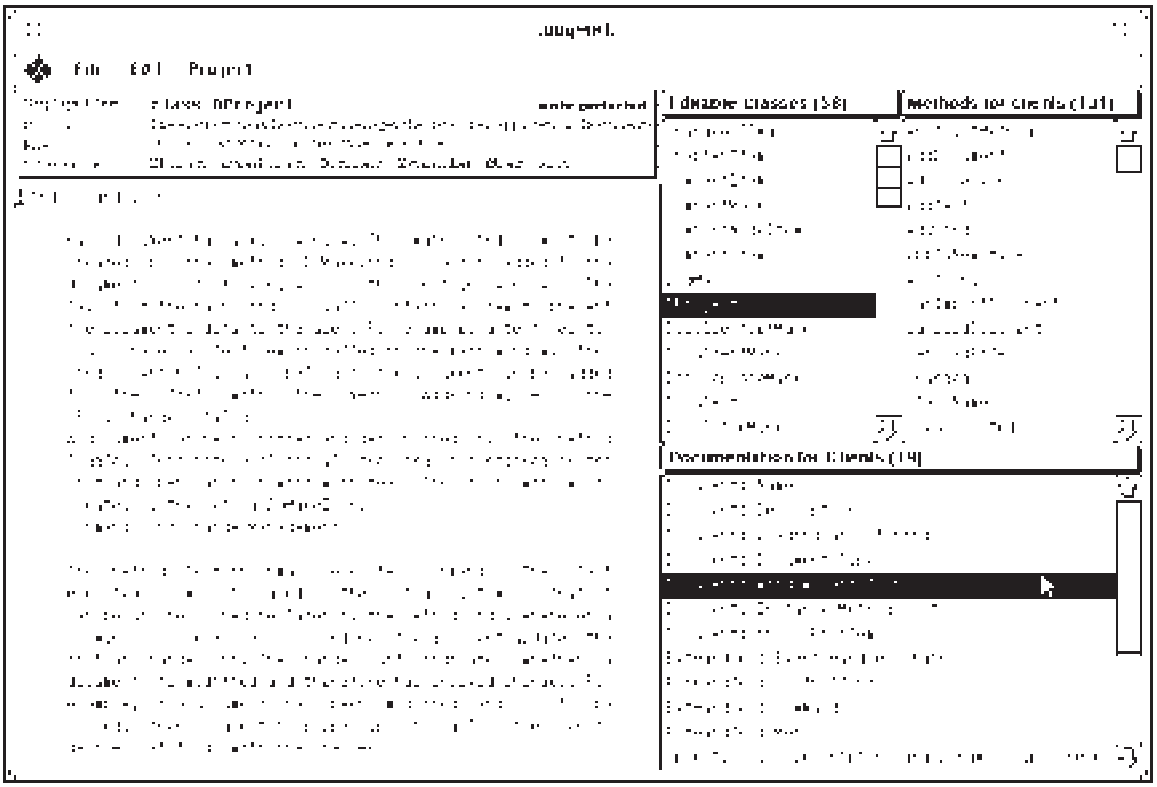

Fig. 9: User Interface of DOgMA

Experience

In our research projects we use the public domain application framework ET++ (see [Wei89]), for which detailed documentation for the most important classes and methods is available. Unfortunately, it is rather cumbersome to get relevant information about certain classes and methods because the data is usually spread over the descriptions of several classes (superclasses). Therefore, we divided the documentation into sections (e.g., description, instance variables, methods, example) to be used with our tool DOgMA.

Although the documentation of ET++ had not been written with inheritance and information hiding in mind, the benefits of applying these mechanisms were enormous. DOgMA previously already provided comfortable browsing mechanisms that were highly esteemed by programmers (especially beginners) using the complex application framework ET++. The possibility to get the part of the documentation that is relevant for using a special class or method, even when it is spread over many superclasses, made reusing a complex class library much easier.

Table 1 gives an impression of the reuse factor of existing documentation. The application framework ET++ consists of 229 classes and about 3400 methods, of which only 35 classes

	documented framework classes	all framework classes	all application classes	documented framework methods	all framework methods	documented application methods	all application methods
number	35	229	58	518	3 403	178	1 159
written sections	9.1	1.4	0.0	8.7	1.3	0.0	0.0
inherited sections	12.4	1.9	19.9	0.0	0.0	8.5	1.3
client sections	15.6	2.4	15.9	6.7	1.0	7.5	1.2
heir sections	19.5	3.0	19.9	7.7	1.2	8.5	1.3
friend sections	21.5	3.3	19.9	8.7	1.3	8.5	1.3

Table 1: Documentation Statistics of ET++ and an Example Application

and about 500 methods are documented. On the average about 9 documentation sections had been written for both classes and methods. Classes additionally inherit an average of about 12 sections from their superclasses. This results in about 16, 20, and 21 sections for clients, heirs, and friends of a class, respectively. In this example methods never inherit documentation. This stems from the fact that the same scheme had been used to document the methods.

Obviously, the 35 documented ET++ classes are the ones that are most often reused. An application program (DOgMA itself) that is based on ET++ consists of 58 classes and about 1100 methods. Before we started to write application-specific documentation, the application's classes had an average of about 15 client and about 19 heir and friend sections already. Please note that the number of heir and friend sections is equal for the application's classes and methods because at that time there was no private documentation for the application. The application's methods that inherit documentation have an average of 7 to 9 documentation sections. The average number of sections of all application methods is relatively low because there are many application-specific methods that do not exist in the application framework and thus cannot inherit any documentation.

The experiment with the documentation of ET++ demonstrates the usefulness of object-oriented documentation. We believe that reusing documentation can even be improved when possible reuse is in the mind of documentation writers. Besides, the distinction of private,

protected and public sections has to be considered more carefully. This, naturally, was not done by the creators of the ET++ documentation.

Conclusion

We presented a possible documentation structure for object-oriented software systems. The suggested structure organizes the documentation as set of sections for classes and methods. By applying object-oriented techniques, i.e., inheritance and information hiding, to the documentation also, reusability, modifiability, and organizability can greatly be enriched for the documentation. The presented mechanisms have been integrated into an existing browsing tool that now enables users to access relevant documentation fast and easily. Experience has shown that the reuse process can be drastically improved by providing comfortable access to relevant information in both the source code and in the corresponding documentation.

Further important steps in improving the quality and the accessibility of system documentation will be achieved by applying the concepts of hypertext (see [Con87]) and literate programming (see [Knu84]). We made experiences with the combination of these concepts already (see [Sam92a]), and believe that together with object-oriented technology we will be able to drastically reduce the documentation problem.

References

[ANS83] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 729-1983, The Institute of Electrical and Electronics Engineers, Inc., 1983.

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

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

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

[Pom86] Pomberger G.: Software Engineering and Modula-2, Prentice Hall, 1986.

[Sam92a] Sametinger J., Pomberger G. A Hypertext System for Literate C++ Programming, Journal of Object-Oriented Programming Vol. 4, No. 8, pp. 24-29, Jan. 1992.

[Sam92b] Sametinger J. DOgMA: A Tool for the Documentation and Maintenance of Software Systems. VWGÖ, 1992.

[Sam93] Sametinger J. and Stritzinger A. A Documentation Scheme for Object-Oriented Software Systems. OOPS Messenger, 1993.

[Str91] Stroustrup B.: The C++ Programming Language (Second Edition), Addison-Wesley, 1991.

[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.