A DOCUMENTATION SCHEME FOR OBJECT-ORIENTED SOFTWARE SYSTEMS

J. Sametinger, A. Stritzinger

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

Tel.: ++43-732-2468-9435, ++43-732-2468-9437
E-mail: sametinger@swe.uni-linz.ac.at, stritzinger@swe.uni-linz.ac.at

Abstract

The object-oriented programming paradigm improves the reusability of software considerably. Suitable documentation must be provided in order to facilitate software reuse, however. Extensive reuse of existing software components requires increased maintenance activity and hence enhances the importance of system documentation. An adequate scheme should help to achieve higher documentation quality.

In this paper we provide a documentation scheme that aims to fulfill the documentation needs of both reusing and maintaining personnel. This scheme distinguishes among overview, external view, and internal view of both static and dynamic aspects of software components. The various views are described and illustrated by examples. The goal is to provide guidelines for writing documentation of object-oriented software systems and for evaluating the completeness of existing documentation.

1. Introduction

Object-oriented programming is accompanied by new approaches and additional requirements for the documentation of extensive software systems. The reuse of existing software components is increasing with the object-oriented programming paradigm. This intensifies the need for precise interface descriptions to express the capabilities of reusable components. The question arises of how best to describe object-oriented software components (e.g., application frameworks) to ease both their reuse and their maintenance, and how to document software systems that reuse existing software and (partly) modify its behavior.

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 has two aims: 1) to describe interfaces to facilitate reuse, and 2) to describe implementation aspects for communication between system developers and maintenance personnel.

Documentation schemes for conventional software systems do rarely exist (e.g., [Pom86]), but they do not address the special needs of software reusers that arise with the wide-spread use of object-oriented programming. Reusability cannot be achieved solely by providing generalized, high-quality components; providing the reuser with detailed information about relevant aspects of potentially reusable building blocks is of acute importance. 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. But 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.

But the technique of object-oriented programming also confronts programmers with the question of how to describe software that is based on the concepts of inheritance, polymorphism, and dynamic binding. These concepts cause static descriptions (e.g., the source code) to fail in reflecting the dynamic behavior of a software system. Hence, dynamic aspects of object-oriented systems have to be provided explicitly in order to facilitate program comprehension.

We will describe a scheme for system documentation of object-oriented software systems which should help to provide a guideline for writing documentation and for evaluating the completeness of existing documentation. Section 2 introduces this documentation scheme, which consists of six parts. These parts are described in the succeeding sections.

2. Documentation Scheme

Object-oriented software systems are usually based on library components. It is not exaggerated to state that an object-oriented system is typically an extension to a class library or an application framework. This characterization usually holds 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—builds the base for the whole documentation.

The increasing reuse of existing software components necessitates a stricter separation between implementation descriptions (needed for maintenance) and reuser information (needed for reuse). By reuser information (or reuser documentation) we mean the part of the system documentation that is needed by programmers to reuse existing software components. (This is different from user documentation, which describes the user interface and the functionality of a software system.) The implementation description (or implementation documentation) concentrates on the internal details of the software, whereas the reuser documentation describes the software components from an external point of view.

	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. 1: Documentation scheme for object-oriented software systems

The dynamic behavior of object-oriented software systems is usually more complex than that of conventionally implemented systems. This leads to the distinction between static aspects of a system (its architecture) and its dynamic behavior (e.g., control flow). Finally, in order to provide an overview of a system, a general view is needed. Such an overview is important both to make a decision on whether to reuse existing software components and to ease the familiarization process for programmers (reusers and maintainers).

Our scheme results in six different documentation parts (see Fig. 1), 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. Additionally, an overview of class libraries is crucial for the decision of whether or not to reuse a library or components thereof.

In the following sections we describe each of the six documentation parts, their contents, their structure, and their intended readers. For clarity we also provide examples of each of the presented parts. The examples are taken from the documentation of the application framework *Smallkit*, which was developed by one of the authors (see [Str91]). Smallkit is an easy-to-learn-and-use application framework that attempts to make the advantages of frameworks accessible to less experienced programmers (e.g., students).

3. 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, etc.), the organization of the classes (e.g., class hierarchy, client relations), and short descriptions of all classes.

This part of the documentation is intended for readers who are to get familiar with the structure of the described software, i.e., both reusers and maintainers. But it also should clarify whether the collection of classes under consideration is suited for reuse in a particular case. General descriptions are very useful to address reusability questions.

The following example illustrates part of an overview of *Smallkit*, its static structure (class hierarchy), and a short class description therefrom.

Introduction

Smallkit is an application construction set consisting of semifinished building blocks that implement standard behavior and provide a core structure of potential applications. In addition numerous frequently used user interface building blocks are provided. New applications can be constructed by modification and extension of the predefined components and by addition of custom components. The purpose of Smallkit is to factor as much common code as possible out of applications into predefined and reusable components.
Smallkit is implemented in Smalltalk/V and is based on the toolbox of the Apple Macintosh.
…

Class Hierarchy of Smallkit

The classes EvtHandler, Application, Document, View and Command form the most important components of Smallkit. The overall hierarchy is shown in Fig. 2.

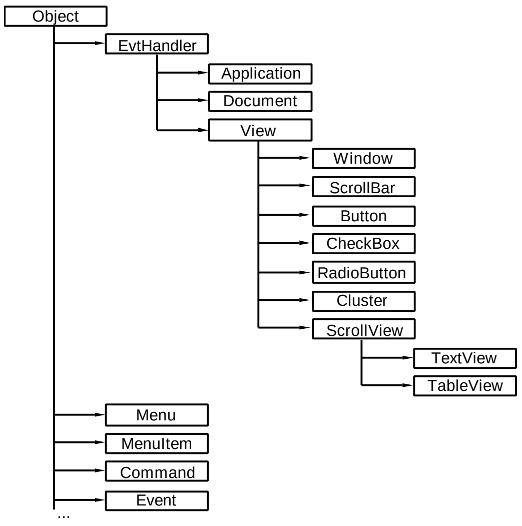

Fig. 2: Class Hierarchy of Smallkit

Each application constructed with the framework consists of a central building block, the application object, which is described by a custom subclass of Application. Applications often create and modify data objects that can be stored in and retrieved from a file. We call these application-specific data documents, which are instances of subclasses of Document. Documents are usually displayed somehow in views. Views, documents and the application are capable of processing events. Events which result from input operations are fetched and distributed to event handlers by the application object. User input that causes a change in a document should be undoable. Command objects serve for modelling such undoable commands.
...

Class EvtHandler

The class EvtHandler is an abstract class which defines a common protocol for all objects that are to handle events. The application programmer usually does not derive new direct subclasses of EvtHandler. The framework already provides three subclasses: Application, Document and View. The subclasses Application and Document are still abstract; the user has to subclass them for each application.

...

A more detailed scheme for the contents of the static overview cannot be provided as it heavily depends on the specific software system under consideration.

4. 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 are event handling and general control flow, but also process and interapplication communication models, the handling of undoable commands in applications, change propagation, and window and/or view updating policy.

Again this part of the documentation is intended for readers who are to get familiar with the described software, i.e., both reusers and maintainers. However, this part of the documentation is not necessary for a reuse decision. It is primarily intended to provide a deeper understanding of the existing components and their behavior.

Event Handling in Smallkit

At run time an application consists of a number of event handler objects that are responsible for handling certain incoming events in certain program states. Since an event handler is usually not responsible for handling all events, there is a mechanism to forward certain events to another event handler. Therefore event handlers are connected with each other by a link. The next handler is usually a more general handler than the previous one. For instance, the next handler of a button is its window, which is succeeded by the document to which the window belongs, followed by the application object itself. This means that all events which are not handled by a more specialized event handler will end up at the application object.

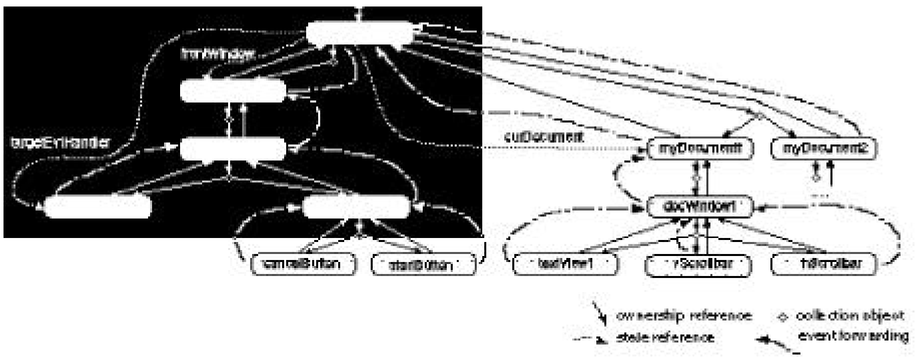

Fig. 3: Event Handler Hierarchy of a Smallkit Application

Figure 3 shows an event handler hierarchy, which is of crucial interest for understanding the dynamic behavior of the event handling application. …

Both the static and the dynamic overview of a software system should be written without presuming too much knowledge on the part of the reader. Due to the diversity of possible systems, however, it is impossible to provide a more exhaustive guideline for the internal structure of these documentation parts.

5. 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 class library or a software system.

Each class has four interfaces that have to be described: to the superclass(es), to the subclasses, to the client classes (i.e., classes that use the class), and to the component classes (i.e., classes that are used by the class (see Fig. 4)).

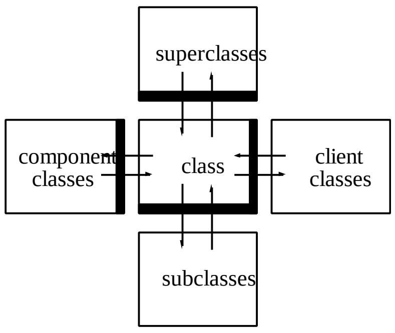

Fig. 4: Interfaces of a Class

The interface to the superclass(es) and to the component classes is sufficiently described by listings of the class names (the interfaces being described there). The list of component classes can even be omitted if all classes of the system are regarded as global (typically so in pure object-oriented systems).

In our scheme the interface to client classes and the interface to the subclasses are described together because the client interface is always a subset of the interface to the subclasses. We suggest first describing the provided functionality of a class in an informal manner. For each public method a method header, a short method comment, and further instructions about when to call and when to override the method are provided. Additional remarks or annotations can be appended whenever necessary. The name, functionality and argument types of each method are obligatory in each case. Class methods (as provided by Smalltalk) should be described separately from instance methods but treated the same way.

In addition to the interfaces already mentioned, classes may have hidden influences or hidden dependencies on their environment as well. Classes may use or modify global objects, manipulate files, or call arbitrary system functions. Such hidden interdependencies must be documented carefully. We suggest appending further descriptions of hidden dependencies just below the regular interface descriptions. Since such interdependencies need custom descriptions, we do not provide a pattern.

The following example demonstrates the interface description of class Application. Please note that some programming languages (e.g., Smalltalk) do not provide any possibility to distinguish between public and private methods/variables. In this case the distinction can (and should) only be recommended in the documentation.

Class Application

The class Application serves as an abstract superclass for each concrete application. It contains the main event loop, where user events are gathered and their proper handling is initiated. Application objects install application-related menus and handle the corresponding menu commands. Typical commands which are common to all applications and which are therefore part of the class Application are about, open, quit, etc. ...

Superclass: EvtHandler

***Class methods*:** none

Instance methods:

addAppWindow:aWindow
"Add aWindow to the application window list."
When to call:
after creation and initialization of an application window
When to override:
never
Annotation:
Application windows are windows which are not related to a document (e.g. tool palettes).
Windows which belong to a document must not be added as application windows.

doCommandKey:anEvent
"Process a command key and return a command."
When to call:
never
When to override:
when the application should respond to command keys (shortcuts)
The overriding method must respond to all application-specific command keys that are not already handled by preceding event handlers. Since there are some default command keys defined in the application class, the overridden method must be called at the end.
Annotation:
To get the character from anEvent, use anEvent characterCode.
...

Other interdependencies
The creation of the class object is done by the modified resume method of the DispatchManager. ...

When the methods provided by a class are very numerous, we suggest introducing categories to ease the reuser’s familiarization with the class. Examples of categories are archiving methods and general utility methods (see [GUI91] for examples).

6. Task Interface Description

It is extremely difficult for programmers to use existing classes when only their interface descriptions are available. Class libraries and application frameworks consist of hundreds of classes and thousands of methods. Thus the following questions arise: *What do we have to do to fulfill a certain task? Which classes have to be used? Can they be used directly, or do we have to derive subclasses? Which methods have to be overridden? Which messages have to be sent (and in which order)?* Like the class interface descriptions the task interface descriptions are intended for programmers who either reuse or maintain a class library or a software system.

Especially programmers beginning to reuse a class library get very frustrated by not being able to answer all these questions. Class interface descriptions are primarily used as references and do not provide any answers. Thus we additionally need descriptions of how to fulfill typical tasks (i.e., recipes) that are supported by a class library.

The task interface description depicts the implementation of important tasks from a dynamic and external point of view. Therefore the task interface description consists of a number of recipes of typical problem situations. Such a recipe collection is often called a cookbook. The cookbook is especially useful for reusers of a library, but also eases program comprehension for maintenance personnel.

The following example demonstrates a typical task supported by application frameworks: installing and responding to menu commands.

Installing and responding to menu commands

How can a certain view install a menu with a number of commands, alter these commands, and perform associated operations whenever the user invokes one of the commands?

1. When the view is initialized the view-related menus should be initialized, too. If there are multiple menus to be initialized or if the menu is rather complex, the introduction of a special initializeMenus method is suggested.

```
fileMenu:=Menu new:'File'.
fileMenu appendItemToMenu:''New/N' cmdSymbol:#new;
         appendItemToMenu:'(-' cmdSymbol:nil;
         appendItemToMenu:'Close/W' cmdSymbol:#close.
```

2. The view must override the method doInstallMenus.

```
super doInstallMenus.
fileMenu insertIntoMBar:2.
```

The message doInstallMenus is sent to the view whenever the target event handler switches and the view is present in the event handler chain, i.e. the view's window is the front window.

3. When some or all of the menu commands should be modified in order to mirror a certain state (e.g., disable menu command), the view has to override the doSetupMenus method.

```
super doSetupMenus.
theCurDocument isNil ifTrue:[
        Menu disableCommand:#save;
              disableCommand:#close].
```

   The message doSetupMenus is sent to the view whenever the user clicks into the menu bar and the menu commands become visible.

4. The view must override the method doMenuCommand:aCmdSymbol.

```
aCmdSymbol==#new
        ifTrue:[perform the associated operations]
        ifFalse:[super doMenuCommand:aCmdSymbol]
```

   The message doMenuCommand is sent to the view whenever the user initiates the command (by menu selection or by pressing the command key) and no preceding event handler has handled the command.

7. 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 any software engineer 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, the use of other classes (components), and the name, purpose and type of all methods and variables. Furthermore, private methods as well as methods which implement non-trivial algorithms should be described here. The instance and class variables of the class Application, whose interface was presented in the previous section, is described, below.

Class Application

Instance variables
isAppDone
  Boolean which controls the main event loop; is false while the application is fetching input events
theAppWindowCollection
  OrderedCollection which contains all windows that do not belong to a document (i.e., dialog windows)
theCurDocument
  Subclass of Document which denotes the current document or nil if no document is open
theDocumentClass
  Class which denotes the class used for creating new document objects
theDocumentCollection
  OrderedCollection which contains all open documents
theTargetEvtHandler
  EvtHandler which denotes the event handler object that is currently the center of interest; this handler receives, for instance, keyboard events

...

Class variables
CurrentAppClass
Class which denotes the class which is used for creating the application object
...

The following example demonstrates the description of a more complex method that requires a detailed description for the comprehension of its implementation.

storeBinaryOn:anObject **stream**:aStream
An important functionality of class Document is the generic mechanism for storing and retrieving complex objects. Objects usually have references to other objects which need to be stored and retrieved as well. The requirements of a generic mechanism are:

- It must be able to traverse an object web in a systematic order; hence it is necessary to have the (meta-)information about the internal structure of each object; in particular, it is necessary to know which variables are references to other objects.
- It needs to be able to detect whether the object was already processed and, if not, store the object; avoiding multiply stored objects is crucial for reestablishing the original structure.
- Upon processing an object which is already stored, only references to this object have to be stored. (Mapping concrete pointers to abstract references is necessary since it is usually impossible to reconstruct object structures at the same memory locations.)
- It must be able to read an object file and reconstruct the original object web.

When an object is written to a file, the class of the object is needed first. In case it is a so-called manifest object (nil, true, false and small integers), we simply have to write the object's name or the number in the file. ...
The following grammar describes the structure of binary object files generated by this method.

```
BOS=    {'['object']'}
object=   'nil'                                     undefined object
        | 'True'                                    True object
        | 'False'                                   False object
        | 'skippedObject'                           File, FileStream, Directory, Context, ...
        | aSmallInteger                             Small Integer object
        | className':'nOfIndexedInstVars':'
            ('C'objectName                          class object
            | 'M'objectName                         meta class object
            | 'E'methodClassName methodSelector     method object
            | 'G'globalSym                          other global object
            | 'S'string                             string object
            | 'B'bytes)}                            byte array object
            | {'|' (object | objectReference)})     normal object
```

The complete source code is not part of this documentation, though it is advantageous to interweave these descriptions with the corresponding parts of the source code in the sense of literate programming (see [Knu84]). However, this is practicable only when suitable tools are available (e.g., [Sam92]).

8. Task Implementation Description

In contrast to the implementation descriptions of individual classes, this part of the documentation portrays implementation aspects that are spread over several classes and/or methods. Class implementation descriptions correspond to implementation descriptions of conventional software systems. They are still needed to document object-oriented systems, but their importance decreases due to the fact that methods usually are relatively short and rarely contain complex algorithms. The complexity of object-oriented software systems lies in the interweaving of classes and methods, which requires special attention in the documentation.

The task implementation description characterizes the implementation of important tasks from a dynamic and internal point of view. Its main purpose is to give answers to important questions like: *How does the system (not an individual class thereof) fulfill a certain task? 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.

The following example demonstrates the redrawing of a view, which involves several classes.

Redrawing of views

What happens when a part of a view is out of date and the program sends the message forceRedraw to the particular view?

1. The focus is set to the view (see also focusing).
2. The displayed rectangle of the view (displayedRect) is added to the update region of the window (by the toolbox procedure InvalRect).
3. The toolbox window manager generates an update event to have the area content drawn anew.
4. The update event is fetched by the method mainEventLoop of class Application and passed on to the handleEvent and handleUpdateEvent methods.
5. The method handleUpdateEvent of the class Application determines the affected window and sends the update message to it.
6. The windows update method signals a BeginUpdate to the toolbox, sends a drawContents message to itself and signals an EndUpdate. BeginUpdate and EndUpdate calls are required by the toolbox window manager.
7. The method drawContents of class View first sets the focus to the view. Then the draw message is sent to the view itself, which causes the view-specific draw method to be executed which performs all drawing operations. Next the drawContents method iterates over all subviews and (recursively) sends the drawContents message to all visible subviews.

9. Conclusion

We have presented a scheme that is intended to be used as a guideline for the documentation of object-oriented software systems that are to be reused. Although our documentation scheme does provide a useful guideline for creating adequate system documentation, a number of issues remain open. Some of these open points stem directly from contradicting or unresolved requirements of object-oriented programming languages.

- Explicit class interfaces for heirs and even for clients are lacking.

  In order to be able to distinguish strictly between external and internal views, a class's interface must be defined explicitly; hidden back doors should not exist. The problem of interfaces for heirs of a class has not yet been addressed clearly by most known languages. Some object-oriented languages do not even provide well-defined interfaces for client classes.

  The importance of well-defined and absolutely binding interfaces can hardly be overestimated. Whenever a language has insufficient interface concepts, the system documentation has to describe all interfaces extremely carefully.

- Public operations have varying level of importance.

  Reusable classes should offer a rather general interface. For instance, a class `Text` could offer operations for manipulating the characters, changing the text fonts and styles, and perhaps changing the line break character set. As a consequence some operations always have to be known, others have to be known often, and some only rarely have to be known in order to be able to reuse the class.

  We suggest dividing the interface description into several sections of varying degrees of importance if the interface complexity justifies such a solution.

Classical system documentation of programs written in an imperative style often emphasizes the description of procedures and functions. Sometimes even flow charts are used to illustrate the functioning of single procedures. In addition, data structure descriptions and module interface descriptions can be found in classical system documentation.

In contrast our scheme does not dedicate particular attention to the description of simple methods; instead, we try to explain interdependencies of system components very carefully. Furthermore, the presented scheme attempts to provide information aimed especially at software reusers. If software reuse is to achieve the importance that it deserves as an implementation technique, which we do hope, documentation has to address the special needs of reusers.

10. References

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

[GUI91] GUI_Master (Class Tree for C++): Class Reference, Vleermuis Software Research bv, The Netherlands, 1991.

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

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

[Sam92] Sametinger J., Pomberger G.: A Hypertext System for Literate C++ Programming, Journal of Object-Oriented Programming, Vol. 4, No. 8, pp. 24-29, January 1992.

[Str91] Stritzinger A.: Smallkit: A Slim Application Framework, Journal of Object-Oriented Programming, Vol. 4, No.6, pp. 11-18, 1991.