Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 97-064

Patterns of Object-Oriented
Software Component
Documentation

by: Jeffrey Kotula
(Ph.D. Thesis)

Patterns of Object-Oriented Software Component Documentation

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

JEFFREY JOHN KOTULA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

NOVEMBER 1997

Abstract

This thesis employs the basic theory of patterns to systematize the largely unexplored domain of software component documentation; it presents an original pattern language to describe successful documentation systems, and creates a framework to guide the development of CASE tools, further research, and theorizing. As an analytical tool, patterns describe solutions to problems existing within a given context in a given domain, and by developing a set of inter-locking patterns, a pattern language is formed, providing a synergistic solution to an entire class of problems. The pattern language presented in this thesis solves a class of problems arising from the needs of software engineers attempting to use or reuse an object-oriented software component. Each of the 39 patterns in the pattern language is defined by 1) carefully describing the exact problem it solves and the contextual forces that might influence the solution, 2) providing a solution to the problem, 3) identifying where, in an industrial setting, the solution can be seen, and 4) identifying other closely-related patterns in the language. In addition to the pattern language itself, a general classification system for component documentation patterns is presented. This multi-dimensional classification system serves not only as an aid in understanding the pattern language, but also as a framework for further research and identification of documentation patterns. In sum, this thesis describes a model of component documentation based in actual industrial practice, expressed abstractly as a pattern language, and organized through analysis and classification.

Dedication

For Kim

Without you, I could never have finished this.

Without you, I could never be who I am today.

For My Parents

The foundation of learning is in the joy and wonder

with which we experience life.

Thank you for the foundation.

Table of Contents

1. PATTERNS OF SOFTWARE COMPONENT DOCUMENTATION 1

1.1 INTRODUCTION 1

1.2 THE ROLE OF COMPONENT DOCUMENTATION 2

1.3 USING PATTERNS TO CAPTURE DOCUMENTATION TECHNIQUES 3

1.3.1 Proven solutions and successful documentation 4

1.3.2 Describing Documentation Patterns 5

1.4 THE COMPLETE PATTERN CATALOG 6

1.4.1 Generative Patterns 6

1.4.2 Content Patterns 7

1.4.3 Structure Patterns 8

1.4.4 Search Patterns 8

1.4.5 Presentation Patterns 9

1.4.6 Embedding Patterns 10

1.5 CONCLUSION 10

2. A CLASSIFICATION SYSTEM FOR DOCUMENTATION PATTERNS 11

2.1 CLASSIFICATION BY THE TYPE OF PROBLEM BEING SOLVED 13

2.2 CLASSIFICATION BY THE DEGREE OF INVOLVEMENT REQUIRED BY THE AUTHOR 15

2.3 CLASSIFICATION BY THE LEVEL OF VALIDATION FOR THE PATTERN 17

2.4 CONCLUSION 17

3. SOFTWARE COMPONENT DOCUMENTATION IN THE DEVELOPMENT PROCESS .. 19

3.1 BASELINE DEVELOPMENT PROCESS 20

3.2 DOCUMENTATION PRODUCTION 21

3.3 USING DOCUMENTATION DURING MAINTENANCE 23

3.4 USING DOCUMENTATION DURING COMPONENT REUSE 23

3.5 CONCLUSION 25

4. GENERATIVE DOCUMENTATION PATTERNS 26

4.1 ENVIRONMENT INDEPENDENCE 26

4.2 HYPERTEXT 29

4.3 EMBED CONTENT IN SOURCE CODE 31

4.4 AUTOMATE TEDIOUS TASKS 34

4.5 Leverage Programming Language Semantics 35
4.6 Centralize Document Design 37
4.7 Directed and Exploratory Navigation 40

5. CONTENT DOCUMENTATION PATTERNS 43

5.1 Tutorial Information with Examples 43
5.2 Diagrams and Illustrations 45
5.3 Overview Information 47
5.4 Generate Short Form 50
5.5 Generate Flat Form 53
5.6 Generate Inheritance Diagrams 54
5.7 Generate Call Dependency Diagrams 56
5.8 Generate Evaluation Metrics (Proto-pattern) 57
5.9 Derive Class Characteristics (Proto-pattern) 59

6. STRUCTURE DOCUMENTATION PATTERNS 61

6.1 Map Organization to Language Constructs 61
6.2 Hierarchical Libraries 64
6.3 Link to External Documents 66
6.4 Backtracking 68
6.5 Generate Hyperlinks 69
6.6 Reveal Lexical Scope 71
6.7 Typed Links (Proto-pattern) 73

7. SEARCH DOCUMENTATION PATTERNS 75

7.1 Keyword Cross-Reference 75
7.2 Inheritance Navigation 76
7.3 Full-text Search 79
7.4 Generate Index (Proto-pattern) 80
7.5 Generate Keyword Cross-reference (Proto-pattern) 82

8. PRESENTATION DOCUMENTATION PATTERNS 85

8.1 Maintain Consistency 85
8.2 Distribution via the Web 87
8.3 Icons, Headings, and Colors Mark Nodes 88
8.4 Readability through Typography 91

8.5 Typography-Encoded Details 93
8.6 Structure for Physical Management (Proto-pattern) 96

9. EMBEDDING DOCUMENTATION PATTERNS 98

9.1 Syntactically Scoped Documentation 98
9.2 Document in Small Chunks 100
9.3 Allow Typographic Control 102
9.4 Readable Embedded Documentation 105
9.5 Embedded Images (Proto-pattern) 108

10. REFERENCES AND FURTHER READING 110

10.1 References 110
10.2 Related Web Sites 113
10.3 Further Reading 114

APPENDIX A. DOCUMENTATION PATTERNS IN COCOON 121

APPENDIX B. THE SCIENTIFIC METHOD OF INVESTIGATION 125

List of Figures

FIGURE 1: THE BASIC MODEL 20
FIGURE 2: DOCUMENTATION PRODUCTION 21
FIGURE 3: INCLUDING SUPPLEMENTAL MATERIAL 22
FIGURE 4: USING DOCUMENTATION DURING MAINTENANCE 23
FIGURE 5: USING DOCUMENTATION DURING REUSE 24
FIGURE 6: SAMPLE INHERITANCE DIAGRAM 55
FIGURE 7: EXAMPLE OF A HYPERTEXT ORGANIZATION 63
FIGURE 8 : A DEEP INHERITANCE HIERARCHY 77
FIGURE 9: A WIDE INHERITANCE HIERARCHY 77

1. Patterns of Software Component Documentation

Abstract

High-quality documentation for object-oriented software components is critical to their success as usable/reusable components. Documentation is the primary means of communication between the author of the component and its client, providing insight into design intent, use cases, potential problems, etc. To date, no theoretical framework exists that can be used to evaluate the effectiveness of the design of component documentation or to guide authors in its production. However, patterns *can be used to describe the documentation techniques that are found in successfully deployed systems, offering a pragmatic definition for what is required of component documentation.*

Keywords: Patterns, documentation, reuse.

1.1 Introduction

Software components are parts of a software system, each encapsulated by a formal interface [Buschmann+96]. The interface can be specified in a variety of ways, but in the case of object-oriented components the implementation mechanism is the class or class library. A set of components, communicating through their interfaces, is used to build a complete working system. Thus, software components form the basic unit for software reuse. But it is important to add that, since reuse must be a planned engineering goal, software components must be *intended* to be reused from the very beginning [Faget+92]; the component author must design and implement it with its future users in mind.

This chapter explores some of the issues surrounding component documentation, a key aspect of the collaboration between component author and client. First, the critical role of documentation in the component use/reuse process is considered. Although developing an appreciation for the importance of documentation is vital, it is not enough—we must have some way of evaluating the effectiveness of documentation, and

of assisting component authors in producing it. To this end, *patterns* are proposed as a mechanism to capture effective documentation techniques. The final section contains a brief catalog of the 39 patterns in the documentation pattern language described in chapters 4 through 9.

1.2 The Role of Component Documentation

There are three attributes of software components that determine how successful clients are in reusing them: identifiability, understandability, and confidence [Faget+92]. The latter two attributes are achieved if not completely, at least primarily, through the component's documentation. Unfortunately, the attitude taken toward component documentation throughout the industry is generally lax, and often hostile or dismissive. A simple analogy will illustrate why we must work to elevate the role of documentation.

Consider the dilemma experienced by most people upon purchasing a complex piece of equipment like a VCR. While conceptually simple, the use of the device is typically obtuse and difficult to understand. So each VCR is packaged with a user's manual detailing its operation, providing troubleshooting tips, etc. This manual is the owner's sole path to understanding the use of the VCR.

Software components are certainly more complex, in general, than VCRs, although they may embody a simple concept. They are also considerably less standardized in their interface. Details of performance characteristics, sequences of function calls, pathological behaviors, etc. all must be understood to properly utilize the component *in the way in which it was intended to be utilized*. Anyone can try to use a VCR, but only those who understand it will be able to make it operate as they wish. Likewise with software components.

The author has certain usage scenarios in mind when designing and implementing the component—she may even have specified a set of use cases. But it is not enough that the author alone have an awareness of this information. If clients of the component do not understand its intent and how the author expects it to be used, they will misuse it. In the best of circumstances the component will work well enough despite this misunderstanding. In the worst, the client will grow dissatisfied with the component and

simply discard it, having no confidence in its ability to do what they need. In between these two extremes lie a wealth of daily irritations and confusions that face every software development project. Parnas suggests that this is one of the main reasons software reuse efforts fall short of expectations [Parnas+83].

Documentation, as a direct path for information from the author to the client, is an extremely efficient form of knowledge transfer. In fact, documentation is "one of the most important ways to improve program comprehension [and to reduce] software costs." [Sametinger+95] Despite the fact that one goal of component design is to reduce the information needed to use the component to a bare minimum, there will always be some details that must be understood by the client. Reading the actual source code—which may not even be accessible to the client—to acquire this knowledge is a very labor-intensive process.

So from the client's perspective, "documentation is arguably more important than programming." [Covington85] A clever implementation of an unusable component is of no value.

There are two contexts in which this component knowledge is required: during initial use/reuse of the component; and during maintenance, either perfective or adaptive. During initial use, it is likely that the client will need a top-down view of the component, proceeding from general to more specific information as they develop an understanding of it and are, ultimately, able to incorporate it into their system. In contrast, maintenance work may require quicker access to the details of what is happening at a point of failure or modification. While the information needed in both cases may be the same, the client will need to access it from different directions, placing even more of a burden on the author to provide adequate navigational paths through the documentation.

1.3 Using Patterns to Capture Documentation Techniques

A *pattern* is commonly defined as a proven solution to a problem within a given context. Exactly what comprises a *solution*, a *problem* and a *context* is still a somewhat contentious issue as the software engineering patterns community is still grappling with how to understand and apply patterns. However an intuitive understanding of this simple

definition should suffice. Complementing the notion of a pattern, a *pattern language* is generally considered to be a (possibly incomplete) set of inter-related patterns used in concert to solve a class of problems.

Patterns were introduced by Christopher Alexander to characterize solutions to architectural problems [Alexander+77]. The software industry "discovered" Alexander's work and began adapting it for its own purposes. In recent years there has been a flurry of activities, publications, conferences, etc. exploring the use of patterns in software.

The main work has been devoted to two areas: software design patterns and organizational patterns. Software design patterns attempt to characterize software solutions to both general [Gamma+95] [Buschmann+96] and domain-specific problems. Organization patterns attempt to describe the processes and structures used by successful development organizations [Coplien+95]. Both of these efforts have yielded new insights into the problems they address, and provided new tools for thinking about them.

This paper proposes the use of patterns to capture the traits, behavior, and characteristics of successful component documentation.

1.3.1 Proven solutions and successful documentation

The above discussion defined a pattern as a proven solution to a problem. One difficulty with this definition is in determining what makes a solution *proven*. Clearly a pattern which embodies an inadequate solution to a problem is of no particular value. Yet what criteria can be used to determine the trustworthiness and merit of a particular solution? For industry to accept and use new tools such as patterns, they must provide "simple and effective techniques...[that] address real issues..." [Onoma+95]

The patterns community has agreed upon a workable, pragmatic criterion: if a particular pattern can be found in some number of commercially deployed systems, that pattern is considered to be proven. The number of deployed systems required is generally two or three. While this may not be a statistically significant number, it is enough to provide protection against the proliferation of aberrant or harmful patterns. Proposed patterns that are not justified through deployed systems are often called proto-patterns to indicate their tentative or experimental nature.

Note also that we usually only consider patterns from commercially deployed systems; that is, systems whose overall value has been proven in the marketplace. This stipulation helps guard against patterns that are not feasible in commercial software.

These conventions are certainly not foolproof, but they do provide a context for understanding what would be considered a successful documentation system and from this, we can discover patterns of component documentation. In accordance with this accepted criterion, the patterns in the catalog all are validated by at least three industrial uses. Note, however, that not all referent systems are commercial: some freeware systems are referred to, as well as the documentation portion of larger software systems.

1.3.2 Describing Documentation Patterns

Every pattern language uses a uniform format for describing the patterns it contains. The basic elements of the patterns that must be described are the problem to be solved, the context and forces at work in the environment of the problem, and the solution represented by the pattern. The following items will describe each software documentation pattern:

Synopsis

A one line statement of the purpose of the pattern; the problem to be solved.

Problem: Context and Forces

A description, in reasonable detail, of (1) the situations where the need for the pattern arises, (2) the forces causing the dilemma, and/or (3) the forces influencing the choice of solution.

Solution

A description of the solution prescribed by the pattern.

Known Uses

A list of systems where this pattern can be seen.

1.4 *The Complete Pattern Catalog*

The pattern language presented here consists of 39 inter-related patterns, broken into six categories. This section provides the synopsis for the patterns not described above. The names of proto-patterns are displayed in italics.

1.4.1 Generative Patterns

Environment Independence:

The documentation system must be feasible and effective in any operating environment.

Hypertext:

Present the documentation as hypertext to enable effective navigation of the information.

Embed Content in Source Code:

Ensure that the component documentation is synchronized with the component source code.

Automate Tedious Tasks:

Automate as many documentation production and maintenance tasks as possible to relieve the workload of the author(s).

Leverage Programming Language Semantics:

Take advantage of all the information about the component that is inherent in the source code.

Centralize Document Design:

Centralize the layout and design of the documentation to ensure its consistency.

Directed and Exploratory Navigation:

Provide a means for both directed and exploratory navigation to support reader education and research.

1.4.2 Content Patterns

Tutorial Information with Examples:

Provide the client with information on how to use the component in specific, common use cases.

Diagrams and Illustrations:

Include diagrams and illustrations in documentation for clarification purposes.

Overview Information:

Provide high-level overview descriptions and summaries in the documentation for each component, library, and class.

Generate Short Form:

Automatically create a brief summary of the extensions that a derived class makes to its base class.

Generate Flat Form:

Automatically create a complete summary of all class members from all parent classes.

Generate Inheritance Diagrams:

Automatically generate diagrams depicting the inheritance structure of the classes.

Generate Call Dependency Diagrams:

Automatically generate diagrams depicting the inter-class dependencies in each library or component.

***Generate Evaluation Metrics*:**

Automatically generate object-oriented code metrics to help clients evaluate the components.

***Derive Class Characteristics*:**

Automatically list the descriptive attributes that characterize each class.

1.4.3 Structure Patterns

Map Organization to Language Constructs:

Structure the hypertext nodes and the links between them to parallel the programming language constructs used to implement the component.

Hierarchical Libraries:

In the documentation, provide a mechanism for grouping related classes together to form subsystem libraries.

Link to External Documents:

Join supplementary documents to the component documentation through hypertext links.

Backtracking:

Provide the user with links that backtrack to related, higher levels of documentation.

Generate Hyperlinks:

Automatically hyperlink each occurrence of a software artifact's name to its full documentation.

Reveal Lexical Scope:

Structure the documentation to reflect the lexical scope of the software artifacts within a component.

***Typed Links*:**

Classify hyperlinks by type to enable more directed, selective navigation through the component documentation.

1.4.4 Search Patterns

Keyword Cross-reference:

Provide a mechanism to help clients find classes in a component within categories of interest.

Inheritance Navigation:

Provide an efficient, convenient means to quickly navigate through the inheritance structure for a given class.

Full-text Search:

Provide utilities to search through the full text of the documentation.

***Generate Index*:**

Automatically determine which words and phrases in the component documentation are most significant and generate an index from them.

***Generate Keyword Cross-reference*:**

Automatically parse class names to create a minimal list of cross-reference keywords.

1.4.5 Presentation Patterns

Maintain Consistency:

Maintain consistency in the visual appearance of the documentation.

Distribution via the Web:

Use Web pages on the Internet to provide access to the most current version of the documentation.

Icons, Headings and Colors Mark Nodes:

Use pictorial icons, standard headings, and/or colors to mark the context and purpose of hypertext nodes in the documentation.

Readability through Typography:

Graphically represent the textual information clearly and effectively.

Typography-Encoded Details:

Use typographic conventions to encode syntactic details where programming language syntax is directly included in the documentation.

***Structure for Physical Management*:**

Design the physical repository of the documentation to enable its easy versioning and administration as a unit.

1.4.6 Embedding Patterns

Syntactically Scoped Documentation:

Use syntactic rules to associate embedded documentation with the programming language construct it documents.

Document in Small Chunks:

Structure the embedded comment text so that software artifacts are documented in small increments.

Allow Typographic Control:

Program the extraction tool to support typographic markup within the manually authored documentation text.

Readable Embedded Documentation:

Design a documentation-embedding syntax that enhances the readability of the source code.

***Embedded Images*:**

Centralize file management by embedding ASCII-encoded images within the source code for later extraction, decoding and inclusion in the final documentation.

1.5 Conclusion

Without adequate documentation we become component archeologists, digging through the source code to uncover its secrets. The pattern language presented in chapters 4 through 9 illustrate the way component documentation patterns can be used to capture and analyze individual strategies for creating effective documentation. The patterns work together, supporting a wide range of documentation design goals, and can be used to form a comprehensive pattern language. It is vital that we devote more effort to software component documentation if the industry is to increase the quality of multi-component systems and to develop them with reasonable efficiency and confidence.

2. A Classification System for Documentation Patterns

Abstract

The pattern language for object-oriented component documentation consists of 39 individual patterns. Their subject matter covers a wide range of contexts from generative patterns to patterns detailing the appearance and formatting of the documentation. The patterns are organized into small, manageable groups by using three classification criteria. The criteria consist of (1) the problem solved by the pattern, (2) the degree of the author's involvement, and (3) the validation provided for the pattern.

Keywords: Patterns, documentation, classification.

A pattern language is intended to be a framework for solving an entire class of problems, but without some type of classification or organization, it is reduced to a collection of solutions to loosely related problems. In an unorganized collection of patterns, finding one that solves a particular problem requires the user to sift through all the patterns blindly. While this may be acceptable for small pattern languages (5 to 10 patterns), for larger ones it is very inefficient. With a classification system, however, the pattern search can be directed. Additionally, in a language without organization it becomes difficult for the user to understand the relationships between the patterns, and to see how the language as a whole addresses the problem domain.

This chapter defines the classification system developed for the documentation pattern language. The system consists of three criteria, derived through an analysis of the patterns *after* they had been identified:

1. Classification by the type of problem being solved
2. Classification by the degree of the author's involvement
3. Classification by the level of validation

Each criteria is described in detail, along with a discussion of how it was derived, in sections 2.1, 2.2, and 2.3, respectively. The following table summarizes the classification system (patterns specified in *italics* are proto-patterns—see section 2.3):

Classification of Documentation Patterns

Types of Problems Solved	General Guidelines for Documentation System	Level of Author Involvement: Total (Manually Authored)	Partial (Assisted by Automation)	None (Fully Automated)
Generative	Environment Independence Hypertext Embed Content in Source Code Automate Tedious Tasks Leverage Programming Language Semantics Centralize Document Design Directed and Exploratory Navigation			
Content		Tutorial Information with Examples Diagrams and Illustrations	Overview Information *Generate Evaluation Metrics*	Generate Short Form Generate Flat Form Generate Inheritance Diagrams Generate Call Dependency Diagrams *Derive Class Characteristics*
Structure	Map Organization to Language Constructs	Hierarchical Libraries Link to External Documents Backtracking	*Typed Links*	Generate Hyperlinks Reveal Lexical Scope
Search		Keyword Cross-reference	*Generate Index* *Generate Keyword Cross-reference*	Inheritance Navigation Full-text Search
Presentation	Maintain Consistency *Structure for Physical Management*		Distribution via the Web Icons, Headings and Colors Mark Nodes	Readability through Typography Typography-Encoded Details
Embedding	Syntactically Scoped Documentation Document in Small Chunks Allow Typographic			

	Control Readable Embedded Documentation *Embedded Images*

2.1 Classification by the Type of Problem being Solved

The following step-by-step analysis of the patterns in the pattern language led to the selection of the first classification criteria: using the types of problems solved by the patterns to determine categories. The first step of the analysis was to identify a small set of criteria that could reasonably be used for classification. The definition of a pattern—"A solution to a problem in a given context"—was found to contain three such criteria: the solution to the problem; the problem to be solved; and the context within which the problem arose. Once a set of candidates was identified, the patterns in the language were examined using each of the criteria in turn, searching for commonality between them. Any characteristics shared between patterns identified a potential category. Thus, a categorization was developed for each candidate criteria. The final step of the analysis was to examine each categorization and determine which was best by asking the following questions:

1. Do all the patterns in the language belong to a category?
2. Are all the categories rationally justifiable, or just coincidental?
3. Are the categories reasonably exclusive of each other so that patterns can be assigned to categories unambiguously?

Categorizing the patterns based on the type of problem being solved created the strongest, most coherent classification consisting of six distinct categories: generative, content, structure, search, presentation, and embedding patterns. The categorization is strong because it is balanced and well-defined—each category contains approximately the same number of patterns, and the definition of each category is clear and unambiguous. More importantly, however, the categories are logically tied to the problem domain of documentation. To illustrate this, consider the following questions one must ask in designing a documentation system:

1. What information should be included in documentation?

2. How should the information be put together?
3. How should the information be displayed visually?
4. How can the reader find the information they need?

These questions—which are completely independent of patterns—correspond to the content, structure, search, and presentation categories, respectively, and provide assurance that the categorization is correct.

The remainder of this section will provide detailed descriptions of each of the six categories.

Generative patterns are the principles that form the basis of the documentation system. They are called "generative" because they are high-level patterns that can be used to derive other patterns in the language. (This standard terminology from the patterns community is somewhat backwards, as patterns are usually "discovered" rather than "derived.") Generative patterns prescribe the organization, attributes and characteristics of an effective documentation system. They are concerned with the overall usability of the documentation from the perspective of the reader and the author. For the reader, the documentation must be usable and effective. For the author, it must be a system that can be created and maintained efficiently with a minimal amount of unnecessary overhead, duplication of effort, and loss of information over time.

Content patterns describe the information needed in the documentation for its users to achieve their goals. Documentation for object-oriented software components requires very specific types of information, as well as the more general types needed in almost any domain. The content patterns specify both, and along with structure and search patterns they are broken into further sub-categories based on how the information is acquired.

Structure patterns are concerned with the way the documentation is put together, and especially with how the reader navigates through it. These patterns presume the use of hypertext for the documentation system as prescribed by the *Hypertext (4.2)* pattern and therefore focus mainly on the topology of the hyperlinks in the document. Users can

only navigate the hypertext document through the hyperlinks, so the usability of the document is dependent on the hyperlink topology.

Search patterns describe services in the documentation system which aid users in finding information of interest. Indices and tables of content are needed, of course, but since the documentation is in a hypertext format the search structures must go well beyond such simple lists. Also, when using documentation for a software component, the granularity of the needed information varies widely, from syntactic or programmatic details to high-level class library organization information. The search patterns prescribe successful methods for addressing these issues.

Presentation patterns specify how the documentation should look. Problems concerning the graphic design, typography and layout of documentation are considered, as well as other ergonomic issues.

Embedding patterns are concerned with how the documentation information can be embedded within the source code (see section 4.3). A wide variety of problems and opportunities arise when embedding documentation in source code. (In this context, the documentation text in the source code is concerned with a *client's* usage of the code, not with the component's internal implementation.) Most of the issues revolve around the needs of the documentation authors for structuring the content and avoiding duplication of information, while still allowing the efficient extraction of the content into final hypertext form.

Chapters 4 through 9 present the patterns of each of these primary classifications.

2.2 Classification by the Degree of Involvement Required by the Author

Further analysis of the characteristics shared by the patterns within each of the primary categories revealed a second classification criterion based on the degree of effort required by the author. Throughout the pattern language, various levels of automation are used to reduce the workload of the author, leading to the four categories for this criterion: general guidelines, manually authored, assisted by automation, and fully automated.

General Guidelines, the least automated of the four categories, prescribe (1) approaches to documentation, and (2) principles of document design, that the author must

bear in mind at all times. They are not directly achievable tasks or discrete items of required information, but principles that require constant attentiveness on the part of the author. For example, the *Maintain Consistency (8.1)* pattern specifies that consistency—in many forms—must be maintained in the visual presentation of the documentation. Some forms of consistency, such as the style and tone of the writing and the level of detail it provides, can only be achieved if author continually monitors their work to ensure it. So, the constant attention required by the patterns in this category leads to a high level of effort by the author.

In contrast, *Manually Authored* patterns only require the author to provide specific pieces of information that cannot be derived from any other source. For instance, one of the patterns (see section 5.1) requires the component documentation to contain tutorial, or how-to descriptions, of common ways of using the classes it contains. Modern programming language semantics are not capable of expressing this information, and so the author must provide it in natural language text.

Patterns in the *Assisted by Automation* category also require the author to provide certain information, but it is automatically augmented by information extracted from the source code (see *Leverage Programming Language Semantics (4.5)*). Consider the *Overview Information (5.3)* pattern, which requires the documentation to contain an overview description of each class in a component. Parts of the information needed for the overviews—lists of member functions, inheritance information, data type dependencies—can be determined automatically by examining the source code. However, this data is not enough to provide a client with a complete overview of the class. A human author must provide a natural-language description of it, describe its intent, where it should be used, and any other general information that will be helpful. Creating the part of the overview that can be automatically generated, however, should *not* be a responsibility of the author, but of an automation tool.

Some of the documentation patterns are *Fully Automated*, in that their use requires no additional effort on the part of the author. There are two ways in which a pattern can be fully automated. First, as for the *Assisted by Automation* patterns, information content

can be extracted from the source code and inserted into the documentation. Programming language syntax and semantics provide a rich source of information that can be leveraged. The second way to automate a documentation pattern is to modify the extraction tool (see *Embed Content in Source Code (4.3)*) to implement the solution described by the pattern. Since the problem solution is simply built into the extraction tool, the component author need not take any action.

2.3 Classification by the Level of Validation for the Pattern

The final criteria for classifying the patterns is borrowed from the patterns community, and is used as a gross indicator of the level of validation a pattern has had through actual use. One of the aspects of the patterns movement that has a strong appeal to engineers is that a pattern is, generally, a *proven* solution (see section 1.3.1), meaning that it has been tested and found to be effective in a variety of real systems. But there is a dilemma encountered by pattern writers who discover only a single use of what appears to be a very effective pattern. Since the pattern has not been validated through appearing in several systems it cannot justly be considered a *proven* solution. On the other hand, it seems wasteful to discard a promising idea because of a rule that is, after all, just a convention.

To preserve the value of these patterns as well as the integrity of the pattern language, the patterns are classified as *proto-patterns* to indicate that they are prototypes or experimental patterns and should not be given the full authority or consideration due more proven patterns. Of the 39 patterns in the documentation pattern language, about 18 percent are proto-patterns. Their names are italicized in the table in section 2.0.

Proto-patterns play an important role in a pattern language, broadening what might otherwise be a fairly conservative, and possibly stifling, tool. They allow for speculation and indicate areas of experimentation and exploration, setting the stage for the growth of the pattern language. At the same time, by segregating proto-patterns from other patterns, the user of the pattern language is made aware of their tentative nature and can deal with them accordingly.

2.4 Conclusion

By providing a rich categorization system that organizes the patterns around a variety of independent concerns, the multi-faceted relationships between the patterns in a pattern language are made apparent. The categorization system provides a framework within which the pattern language can be learned, but also provides a focus for pattern discovery activities by identifying ways in which the patterns manifest themselves.

The categorization system presented here is simple and elegant, but other categorizations may later be invented that add new insights into the pattern language. There is precedent for such an event from the field of software design patterns: the pattern language published in [Gamma+95] was integrated into a new classification scheme introduced by [Buschmann+96]. In addition, the [Gamma+95] patterns were re-classified by [Zimmer95]. Classification provides a valuable and flexible tool for understanding complexity.

3. Software Component Documentation in the Development Process

Abstract

Patterns in any given pattern language are dependent upon the context for which they are intended. In the case of the documentation pattern language, the overall context is the software development process, specifically that part of the process involving the production and use of component documentation. This chapter describes the role of component documentation within the development process, providing a model of the context shared by all the documentation patterns.

Keywords: Software documentation, development process, software engineering, reuse.

Patterns in any given pattern language share a common context because they all solve problems within a particular domain. In the case of the documentation pattern language, the overall context is the software development process, specifically that part of the process involving the production and use of component documentation. Understanding this common context is key to accurately interpreting the patterns because it provides an overview of how individual patterns fit into the pattern language by (1) giving insight into the reasons behind and consequences of the problem it solves, and (2) establishing a framework for developing solutions.

Accordingly, this chapter describes the role of component documentation within the development process, providing a model of the context shared by all the documentation patterns. The first section provides a simplified, baseline model of the overall software development process. The next section discusses the way in which documentation is produced in conformance with the pattern language. The final two sections outline the use of documentation within maintenance and reuse activities, respectively. Each section provides an incremental refinement of the process model, until

finally a complete model of the software development process (as it relates to documentation) is formed.

3.1 Baseline Development Process

A basic model of the development process consists of three main steps: requirements gathering and analysis, design, and coding.

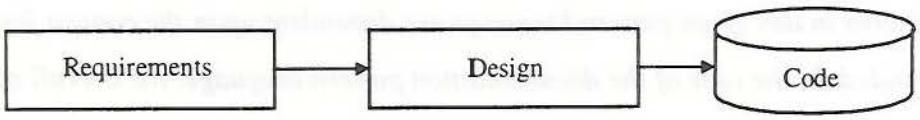


Figure 1: The Basic Model

During the Requirements step, the problem(s) to be solved by the component or application are defined in detail. The requirements, in turn, feed the Design step where a satisfactory solution is formulated. Finally, The Code step implements the solution.

Granted, this simplified version of the process does not reflect the iterative nature of development nor contain all the steps usually given in traditional process models. It was not intended to—the purpose of the model is to reflect the development process as it pertains to component documentation. Eliminating the process steps that do not impact documentation makes the model easily understood while still maintaining compatibility, at a high level of abstraction, with many methodologies.

3.2 Documentation Production

The *Embed Content in Source Code (4.3)* pattern asserts that documentation text should be physically stored in the source code, and then extracted and formatted by a tool to produce its final form. The following diagram illustrates the point in the process where the extraction tool is used to generate the documentation:

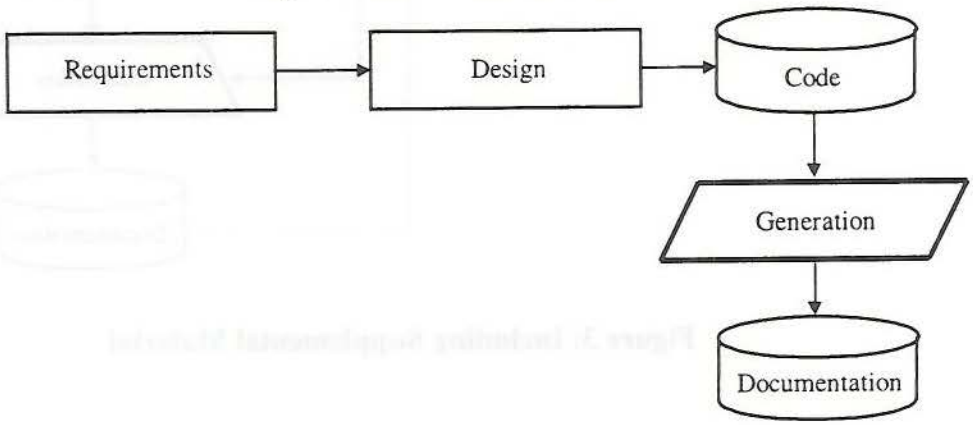


Figure 2: Documentation Production

Extracting documentation from source code is a fundamental concept of the pattern language, and is crucial to understanding most of the patterns. This approach is distinguished from the traditional view that documentation should be maintained as a separate artifact. By embedding the documentation in source code and extracting it for final production, documentation becomes *integral* to the process, and the process itself helps ensure that the documentation and source code remain in agreement over time.

The documentation embedded in the source code is the primary repository of the information required by the reader, but other sources may also provide important clarification or elaboration. For example, documents produced in the Design step may help the user to understand the rationale for the design of the component or to see why alternative designs did not succeed. Other external sources may provide descriptions of hardware, software, or algorithmic technologies that are used in the component.

Any supplemental material required can be included in the final form of the component documentation in one of two ways: (1) a verbatim copy can be directly included in the component documentation; or (2) it can simply be referenced through a

hypertext link such as a Uniform Resource Locator (URL). The first approach is represented by the solid line connecting the Design and Generation steps in the following diagram, and the second by the dashed line connection Documentation to the Design step:

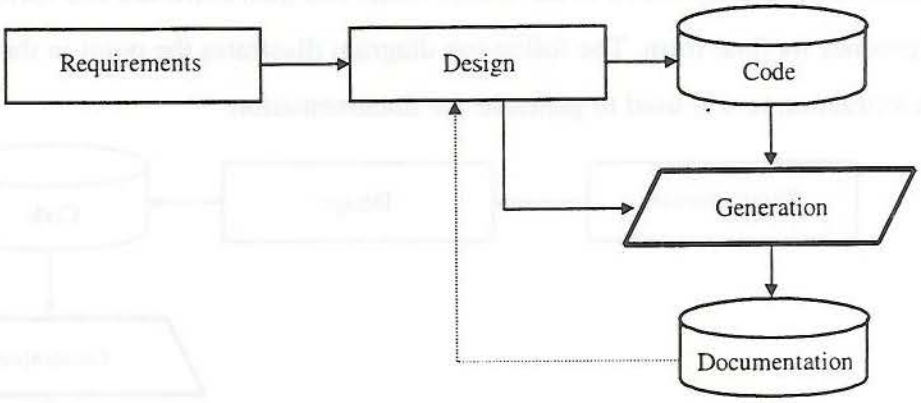


Figure 3: Including Supplemental Material

3.3 Using Documentation During Maintenance

Code will always undergo maintenance work during its lifecycle, either to correct problems or to extend its functionality. For either purpose, the engineer performing the work needs two sources of information:

1. Component documentation, which is used to understand the purpose and design of the component, as well as its behavior from the client's point of view.
2. The source code itself, which is used to understand implementation details, and to design and implement component modifications.

The following diagram illustrates how these maintenance activities fit into the development process:

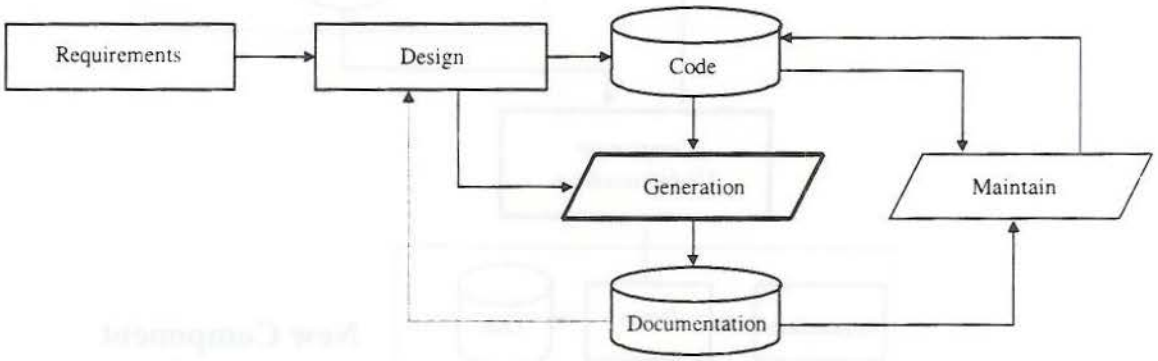


Figure 4: Using Documentation During Maintenance

Once the source code has been modified, the documentation must be re-generated by the extraction tool, ensuring that both remain synchronized with each other.

3.4 Using Documentation During Component Reuse

Software components, as defined in section 1.1, are intended for reuse in the construction of new components. In order to reuse components, the engineer(s) must have adequate documentation to fully understand the component or set of components involved in the process. Unlike the maintenance engineer, the reusing engineer may not have access to the source code, and therefore must rely *solely* on the component

documentation. So component documentation, being the only resource available, must provide complete enough information to enable the engineer to use the component correctly. This is illustrated in the final expansion of the software development process diagram:

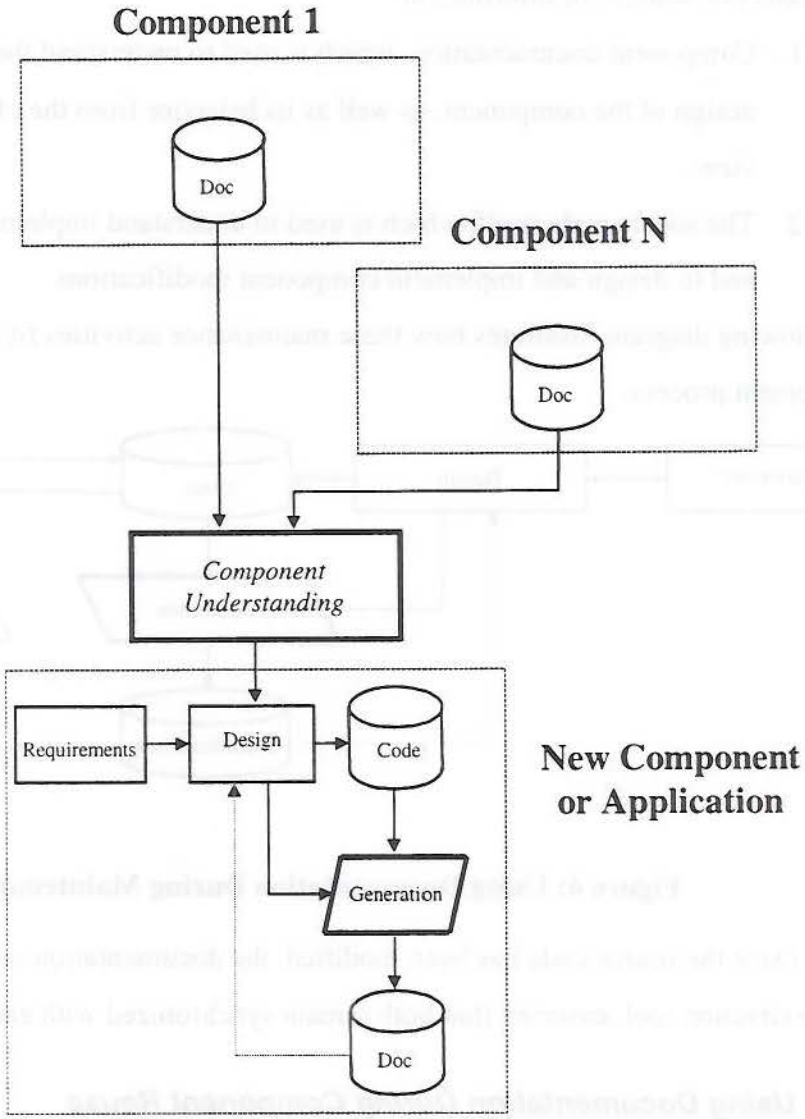


Figure 5: Using Documentation During Reuse

Access to the source code may not be allowed if the component was purchased from an external supplier: the licensing agreement may not include full access to its implementation details. (It is in the supplier's business interest to protect their trade

secrets or competitive advantages through such licensing constraints.) Such restrictions are not uncommon, but are themselves limited by the capabilities of the programming language used for the component. Depending on the language, the supplier may not be able to completely restrict the client's access to the implementation. In such cases, some suppliers will encode, or obfuscate the source code so that it is not easily readable, thus providing a minimal level of protection for their property. Unfortunately, this will still deprive the reuse engineer of a valuable source of information.

As the industry moves more and more toward a component-based model of software architecture, access to component implementations will likely decrease. The only remaining source of information for the reuse engineer is the component documentation, which they will simply have to rely on being complete and accurate.

3.5 Conclusion

By understanding how documentation is produced and used within the development process, the problems and solutions presented by the pattern language take on a deeper meaning as parts of a larger, synergistic whole. The problems addressed by the patterns can be seen in light of a specific enterprise rather than as isolated, solitary occurrences. Their solutions can be tailored to support the process and to reinforce the solutions given by other patterns in the language. Through this type of tightly interwoven interaction, the pattern language emerges from the patterns, and ultimately, from the basic problem it addresses.

4. Generative Documentation Patterns

Abstract

Generative patterns of documentation establish an overall framework for how the documentation system operates. They can be considered general constraints or design goals for the system. Some of the patterns, such as Environment Independence (4.1) and Automate Tedious Tasks (4.4), are not related to documentation per se but are nonetheless important to creating a useful system. Others, such as Hypertext (4.2) are specific to the documentation product itself. All the generative patterns work in concert to create a consistent, effective approach to documentation that unifies the other five categories of patterns.

4.1 Environment Independence

Synopsis

The documentation system must be feasible and effective in any operating environment.

Problem: Context and Forces

Environmental dependencies will create a barrier to the use and acceptance of both the component and its documentation; the dependencies are essentially limitations that restrict the possible clients for the component. The delivery system—the physical data, and the viewing infrastructure—for the documentation is the principle source of environmental dependencies for a documentation system. The physical information itself, the actual disk files, must be in a portable digital form so they can easily be produced and transferred to the client system. The infrastructure programs needed for viewing and/or navigating the documentation, if there are any, must also be portable. Components will be more successful—that is, they will have the largest number of clients—the fewer constraints they place on their operating environment.

The operating environment may vary in a number of ways, each affecting the documentation system in different ways:

- Operating system—This is the portability factor usually considered the most important since different operating systems generally require different formats for data and executable instructions. Whole market segments can be excluded as clients if the documentation system is not portable between operating systems.
- System resource (CPU speed, RAM, hard drive, etc.) available—Documentation systems that are supported by viewing or navigating software must take into account the raw computing capacities of the computer system. Minimum system requirements must be determined that will support the needed functionality with reasonable performance. However, if the minimum requirements are too high, many potential clients may be excluded.
- Networked or stand-alone systems—Documentation systems may be distributed across local-area networks or the Internet, or they may exist stand-alone on individual workstations. If networked, the documentation system must take into account the network speed, Internet response time, server loading factors, etc.
- Programming language—Software components can come in many languages, including C++, C, Java, Lisp, or Ada. Some of these languages are similar in syntax in structure, but others are very different. The documentation system for the component must be flexible enough to be applied to any of the common programming languages.
- Compiler used—Some compilers introduce modifications to the syntax of the programming language, generally through introducing special keywords into declarations. The documentation system must be able to handle this gracefully.

To further complicate the situation, clients may be working with several components simultaneously. Understanding the combined system requirements of all

components and ensuring they are met can be time-consuming and even expensive, particularly if more advanced hardware is required.

Solution

The documentation system must be usable in as wide a variety of operating environments as possible. Ideally, the documentation system would remain completely independent of the operating environment. The delivery mechanism would impose no extra requirements on the client's environment beyond those already imposed by the component itself. However, the benefit of achieving this ideal must be weighed against the cost of restricting what perhaps *could* be done to make the documentation system more effective. For instance, suppose a given software component only requires a command line interface such as provided by DOS or Unix. This restriction may be reasonable for the component, but severely cripples the documentation system if it conforms. For such situations there are two less stringent, yet workable, types of environment independence that can be achieved.

First, the documentation system could assume and use technologies that are ubiquitous in the targeted client environment. The client environment can be profiled in order to identify the types of capabilities and equipment that can be expected. For example, it may be reasonable to assume that a client system will always have, among other things, a high-resolution monitor, a printer, a mouse. For the example above, making this assumption removes the conflict for the documentation system; although the component doesn't require it, the documentation system can safely assume that more capable technology is available.

Even assuming such system capabilities however, some documentation problems may require the use of newer or more demanding technologies, leading to a second level of environment independence—the use of open standards. Open standards, whether for hardware or software technologies, should be preferred over proprietary standards. For instance, for representing hypertext documents an open standard such as HTML (Hypertext Markup Language) should be preferred over vendor-controlled standards such as the Microsoft Help format. Open standards are safe, being

controlled by industry consortiums or standards committees whose purpose is to promulgate them, as opposed to being controlled by a single vendor whose primary interest is in their individual corporate well-being.

Known Uses

One of the most common documentation delivery systems used is still the printed page. It has the advantage of being completely independent of the computing environment. Despite being seriously limited in other respects (see *Hypertext (4.2)*), this independence is often noted as a desirable property ([Shirk88] for example).

Ubiquitous technologies are also relied upon for documentation delivery. For some applications, particularly low-cost programs, the documentation is simply delivered as a plain ASCII text file. It is assumed that the client can both read this on-line and print it out if a hardcopy is needed or desired.

Still other applications deliver their documentation in a more complex, but standardized, digital format. Postscript and Adobe PDF formats are used quite often, since the software needed to read these formats is freely and easily available.

Related Patterns

See also the *Distribution via the Web (8.2)* pattern. Web-based Internet technology is a rapidly emerging open standard which can easily support all the delivery needs of documentation systems.

As an alternative to literate programming documentation paradigms, which require clients to use special software and software build processes, refer to the *Embed Content in Source Code (4.3)* pattern.

4.2 Hypertext

Synopsis

Present the documentation as hypertext to enable effective navigation of the information.

Problem: Context and Forces

Readers need a system that will allow them to navigate quickly and easily through the software component documentation. Depending on the type of information

sought, users will need to navigate in either a directed or exploratory manner [Dusink93].

Directed navigation is used most often when the reader is trying to learn in a highly structured way by following navigation paths laid out by the author. For instance, the chapters in a book are deliberately ordered by the author to define a navigation path. In contrast, paths in a hypertext document can be more complex than a straight linear flow because of the flexibility of hyperlinks. The navigation paths are designed by the author to help the user learn specific information. For software component documentation, the following types of questions would be answered through directed navigation:

- How is this component or class organized?
- How is a certain affect achieved with this class?
- What is the inheritance structure of the classes?
- What other components or classes does this component or class depend upon?

While directed navigation is generally used to orient the reader within a component or class, exploratory navigation is used to search through a web of loosely or incidentally related information. It is best suited for learning about particular details of a component or when the question is not very specific, as in the following:

- What is causing a particular behavior in the class or component?
- Where are resources being locked or reclaimed?
- Does the calling procedure or the called procedure maintain ownership of newly allocated memory?
- What are the legal values for this argument?

Both types of navigation must be supported by the documentation system since both types of questions will need to be answered by the user. The questions requiring directed navigation are of particular importance during initial development, where the user has yet developed a clear understanding of the component they are using.

The questions answered through exploratory navigation are more important during maintenance activities when the component is generally understood, but the user still lacks some understanding of its details.

Solution

Hypertext should be used to support the navigation styles needed by the user. Directed navigation is handled quite simply in a hypertext document through the author's use of tables of contents, lists of related topics, links to next and previous topics, etc. Hypertext's biggest strength, however, is in supporting an associative, opportunistic, exploratory style of learning [Foehr+86], being based upon the idea that learning occurs through association [Carlson88] as well as through more directed styles of learning. Written documentation can support exploratory learning, but the media is much more difficult to work with than hypertext.

Known Uses

Hypertext documentation is a very common part of the help systems of modern software applications. As examples, refer to any of the Microsoft Office products. In the domain of software components, the Microsoft Foundation Classes, RogueWave C++ components, and Intersolv Views tools provide excellent examples. [Creech+91] and [Biggerstaff+87] provide summaries of many hypertext software documentation systems including SEER [Latour+88], Neptune [Delisle+86], NoteCards [Halasz+87], and Planetext [Gullichsen+88].

Related Patterns

The impact of this generative pattern is seen throughout the pattern language, but especially in the Structure and Search patterns.

4.3 Embed Content in Source Code

Synopsis

Ensure that the component documentation is synchronized with the component source code.

Problem: Context and Forces

Documentation is often stored in a physically separate location from the source code itself. This dual maintenance scenario is problematic in that the two become inconsistent over time [Sametinger+95]. Therefore, it is desirable to keep the source code and the documentation together in one physical location.

One force that must be considered in this problem is that the source code is *required* for building the software and so must always be present, while documentation is, for software compilation purposes, optional. If source code is kept in some alternate form—say a database or as an extractable part of some other document (as in literate programming systems and some CASE tool environments)—engineering and software build processes must be altered to work with the alternate form. Establishing such process modifications may not feasible, depending on the character of the organization, the existence of legacy source code, and so on.

Another force involved is the use of graphics in documentation. Figures and diagrams are key elements of good documentation, thus they must be supported by the format in which the documentation is stored.

Solution

Embed the documentation for the software component within its source code files and create a utility program to extract the documentation into a more easily readable form, such as Web pages.

All programming languages have some type of free-form commenting syntax that can be used as a container for documentation text within the source code. Simple formatting conventions can be applied within such commented text for structure. A detailed demonstration of how commenting text is formatted for processing by the Cocoon documentation system follows:

```
/*----------------------------------------------------------------------
CLASS
      Handler
      Base class for all file handling, sentinel recognition, etc.
KEYWORDS
      Cocoon, Handler, Section
DESCRIPTION
      Base class definitions for section handling.  The processing of a
      section is broken into 4 main steps.
            <ul>
            <li>  Search for some sentinel string that indicates
```

```
                the beginning or end of the section.
            <li> Perform some processing upon entry into the section.
                This is done in the constructor for the base class.
            <li> Process a line of text from the source code.  During this
                processing, the section can also look for some exit
                criteria.
            <li> Perform some processing when the section is exited.
            </ul>

    Handler's can be nested.  The set of possible handlers at any given
    time is dependent on what handler is active.  For instance, the
    handler for member function definitions can only be made active
    by the handler for processing a class declaration.
-------------------------------------------------------------------------*/
class Handler {
public:
            ////////////////////////
            //  The ctor should perform whatever processing is needed when
            //  entering the section.  This may include opening files,
            //  setting up state information or whatever.
            //  [in] xref       Cross-reference file data.
            //  [in] custom             Data for user customization of output.
    Handler( Xref& xref, Customize& custom ) :
            _xref( xref ), _custom( custom ), _subhandler( NULL )
            { }

            ////////////////////////
            //  Perform whatever processing is needed when the section
            //  is exited.  Should close any files that were open,
            //  or write any bracketing information such as list termination
            //  tags.
    virtual
    ~Handler();

            ////////////////////////
            //  This function is the main control algorithm for coordinating
            //  the nesting and creation of handlers.  This function should
            //  be called for each line of source text that is read while
            //  the handler is active.
            //  [in] buff       Buffer of source code.
            //  [out] surrendercontrol
            //          Set to TRUE if the controlling handler should
            //          immediately exit this section.
    void
    Control( StringBuff& buff, int& surrendercontrol );

            ////////////////////////
            //  Perform whatever processing is needed for source code lines
            //  within this section.
            //  [in] buff       Buffer of source code.
            //  [out] surrendercontrol
            //          Set to TRUE if the controlling handler should
            //          immediately exit this section.
    virtual void
    ProcessLine( StringBuff& buff, int& surrendercontrol );

            .
            .
            .
```

Note that all documentation appears within the natural commenting syntax of C++, *so no special compilation technology or development processes are needed.* The content of the comment text does have a formatting syntax, most easily visible in the documentation for function arguments. Also, the use of HTML syntax within the

comments allows the author more control over typography and presentation when the documentation is extracted into Web pages.

Although Cocoon will extract the documentation and format it into HTML hypertext, the documentation as it appears in the source code is actually quite readable. This is important because it allows the engineer working on the code easy and convenient access to the documentation, while at the same time enabling its extraction and formatting for wider audiences. It also helps eliminate the dual maintenance problem by putting the documentation in proximity to the code, making it more likely to be modified when the code is modified.

The example above shows only one possible set of commenting conventions that could be used. Once the documentation text is in place, it can be extracted and formatted in a variety of ways and for any number of purposes. For instance, one set of documentation could be generated for users internal to the authoring company, and another set for external users.

Known Uses

JavaDoc, c2man on UNIX systems, cxx2html, Cocoon and Doc++ all take the approach of embedding documentation with the source code commenting syntax.

Related Patterns

- *Centralize Document Design (4.6)*
- *Tutorial Information with Examples (5.1)*
- *Map Organization to Language Constructs (6.1)*
- *Maintain Consistency (8.1)*
- All patterns in the *Embedding* category (chapter 9)

4.4 Automate Tedious Tasks

Synopsis

Automate as many documentation production and maintenance tasks as possible to relieve the workload of the author(s).

Problem: Context and Forces

Many of the tasks involved in creating and maintaining good documentation are time-consuming and tedious. Manually creating an index, for example, has been aptly described as an "exercise in drudge work." [Simpson+88] The same can be said of other tasks such as creating a table-of-contents, typographic formatting, and inserting hyperlinks.

Solution

Whenever possible, tedious tasks should be automated during both the production and long-term maintenance of the documentation by modifying the extraction tool (see section 4.3). The extraction tool already has the basic functionality needed for implementing automation tasks: it processes both the source code and documentation text; and it produces the final form of the documentation in the proper format. Further, since the extraction tool is already a part of the documentation production process, the automation task requires no extra infrastructure. Were a separate automation tool to be introduced, additional modifications to the process would be required.

Known Uses

Word processing software such as Microsoft Word, Novell WordPerfect, or COMPANY Framemaker automatically generates indexes and tables-of-content which would have been exceedingly difficult to create and maintain manually. They also provide other specialized automation functions such as global search and replace utilities, and automatic document formatting.

Related Patterns

- *Centralize Document Design (4.6)*
- *Leverage Programming Language Semantics (4.5)*
- All the *Fully Automated* and *Assisted by Automation* patterns
- *Structure for Physical Management (8.6)*

4.5 Leverage Programming Language Semantics

Synopsis

Take advantage of all the information about the component that is inherent in the source code.

Problem: Context and Forces

Replicating information in both the source code and the documentation leads to extra effort for the author, and the dual maintenance problem. Unfortunately, completely avoiding this replication is not possible. Some types of information—such as data type dependencies, inheritance, call dependencies, argument passing semantics, exceptions, pre-conditions, and post-conditions—is legitimately needed in both source code and documentation. Within source code it is used for the implementation of the class or component, although exactly *which* pieces of information the source requires will vary depending on the programming language. In comparison, the documentation uses the information to assist the reader in understanding the correct use of the class or component. Both purposes are critical to the success of the component and neither can be omitted.

The replication of information results in two problems. First, more effort than is strictly necessary is expended to create the information in both source and documentation. Since engineers are willing to invest only minimally in reuse efforts [Sembugamoorthy+92], this added work will doubtlessly be unwelcome. The dual maintenance problem is the second result of replication. The basic issue is that when identical information is stored in two separate places, it will tend to become inconsistent over time. See the *Embed Content in Source Code (4.3)* pattern for a full discussion.

Solution

Keep the effort required by the code/documentation author to a minimum by using the extraction tool (see section 4.3) to capture shared information from the source code and use it during the creation of the documentation. Since the documentation version of the information is generated from the source code version, it cannot

become inconsistent, thus solving the dual maintenance problem. Further, since the documentation version is automatically generated, the author can avoid the effort that would be required to manually duplicate it.

Using the extraction tool for this purpose is reasonable since it must parse the source code in any event and will have access to any sharable information. Thus, no extra framework for the documentation process is required.

Known Uses

The code browsers commonly found with compiler IDE's give the user access to a great deal of information that is useful for component understanding such as call graphs, inheritance graphs, and metrics. All this information is derived from the source code.

Documentation systems such as Cocoon, JavaDoc, and cxx2html also take this approach. By processing the source code and extracting documentation text, they can synthesize the user-created documentation with information that is directly available in the source.

Related Patterns

All of the patterns in the Fully or Partially Automated categories are based at least in part on this general principle. Additionally, the following patterns are closely related:

- *Embed Content in Source Code (4.3)*
- *Automate Tedious Tasks (4.4)*
- *Map Organization to Language Constructs (6.1)*

4.6 Centralize Document Design

Synopsis

Centralize the layout and design of the documentation to ensure its consistency.

Problem: Context and Forces

When multiple authors are each working simultaneously on different parts of a component, consistency in the design of the documentation is difficult to achieve and maintain. Each decision point in the document design and authoring process provides

a chance for inconsistency to be introduced because different authors will invariably make different choices. Some of the many design questions facing authors are:

- What should the page/screen look like?
  - How should the page/screen be partitioned?
  - What fonts and text styles will be used?
  - Where should the headers be placed?
  - What role should color play in the presentation?
- How should the information be ordered on the page/screen?
  - Should class summary listings precede the overview text?
- What type of information should be included or excluded?
  - Should both short and flat summaries of classes be included?
  - Should sub-classing instructions have their own section, or simply documented through the virtual methods?

Each time an author(s) makes a decision that is inconsistent with a previous decision the effectiveness of the documentation system is reduced. Inconsistency causes unnecessary confusion for readers and interrupts their learning experience. As [Foehr+86] points out, "Consistency is crucial" to creating documentation with a harmonized appearance and an easily interpreted layout.

Inconsistencies could be reduced to some degree by manual means such as establishing documentation design standards that all authors must follow. However, this is an ineffective approach, one that requires more effort and unduly complicates the process by requiring the organization to:

1. Decide how to establish the standards.
2. Arrive at a consensus on the standards.
3. Initiate a review process for documentation to ensure that standards are indeed followed.
4. Teach the standards to engineers new to the component.

These tasks may be made even more difficult by an overall lack of enthusiasm on the part of the engineers.

Solution

Use the extraction tool to centralize the design of the documentation. The extraction tool provides a convenient framework both for embedding documentation content in the source code and creating the final presentation form of the documentation. This framework helps enforce consistency in two corresponding ways. First, since the documentation is represented within the source code in a form that is neutral to the document design, many decision points are removed from the authoring process. The author can thus work at a high level of abstraction and be free of concerns over the final presentation details of the document. Secondly, since the extraction tool, rather than the author(s), is producing the final form of the documentation, the work of multiple authors will look identical. During final production, the extraction tool can process the documentation in a variety of ways—reordering, modifying typography, checking for completeness—thus ensuring that the content produced by different authors looks the same.

Despite this enforced consistency, the documentation system maintains a high degree of flexibility. Since decision points are isolated to one central location—within the extraction tool—the system can easily customize the final form of the documentation according to user specifications. In response to user-supplied parameters, the extraction tool can vary the final presentation form of the document, including elements such as the color used for background or text, the fonts used, the way hyperlinks are presented. The user can vary these parameters between components, but not *within* a component.

This approach has an added advantage: mass modifications can be made to the document design without requiring effort on the part of the author. A simple two step process can be used to introduce a design change. First, the extraction tool is modified to output the documentation in accordance with the new design. Second, the extraction tool is re-executed, producing a new version of the documentation. Although this process will work for many design modifications, some changes will still require author intervention. For instance, changes that require new material to be

included will need the author(s) to first create the content before the documentation can be regenerated.

Known Uses

Cocoon, JavaDoc, cxx2html, Doc++, and c2man all use various extraction tools to produce consistent documentation. Most of them also allow at least rudimentary customization.

Related Patterns

- *Maintain Consistency (8.1)*
- *Readability through Typography (8.4)*
- *Map Organization to Language Constructs (6.1)*

4.7 Directed and Exploratory Navigation

Synopsis

Provide a means for both directed and exploratory navigation to support reader education and research.

Problem: Context and Forces

Without adequate navigational tools, readers will not be able to conveniently access the information and this makes the documentation system unusable. There are two broad categories of navigation tools: directed and exploratory [Dusink93, identified as "systematic" and "ad hoc"]. Directed navigation follows distinguished paths, or pre-planned routes through the documentation, leading from one topic to another. The topics follow a logical order that systematically develops a theme. Distinguished paths might encompass topics such as overviews, tutorials, or class inheritance trees. Exploratory navigation, on the other hand, is comparatively unstructured in that readers intuitively create their own paths. The links they follow are not dependent on the logical structure of the documentation, but rather on the structure of the readers' information needs at the time. Examples of exploratory navigation tools include hyperlinks, indices, keyword searches, and full-text searches.

Both categories of navigation tools must be provided to support the readers' needs. Directed navigation is relied upon heavily by novices while they learn the overall content and structure of a component, and exploratory navigation is used primarily by experts who are trying to locate specific pieces of information. A single reader may be a novice in some areas of the component, and an expert in others, so both types of navigation would be needed depending on his/her objective. The primary purpose of the navigation tools is to provide access to the information readers need when they need it, and depending on the circumstances, both types of tool will be required to successfully accomplish that purpose.

Solution

In the documentation, provide both directed and exploratory navigation tools that will aid reader education and research. Hyperlinks are the primary means of navigation in the documentation (refer to the *Hypertext (4.2)* pattern) and they easily support "the application of a variety of techniques for examining a [document], including free navigation, [and] distinguished paths..." [Creech+91]. Directed and exploratory hyperlinks can either be explicitly constructed by the author, or automatically created. The following table summarizes the types of links that are best suited to each method of creation:

	Directed Navigation	**Exploratory Navigation**
Manual Creation	Tutorials Overviews	Links to related topics
Automatic Creation	Inheritance trees	Keyword cross-reference Indices Other search tools

Links that are manually created must be directly inserted by the author during the writing of the documentation. Directed navigation links may not be needed for simple topics that can be adequately addressed in a single page, however, they will be required in order to adequately develop more complex topics. Exploratory navigation links pose a unique challenge to the author; s/he must anticipate readers' intuitive

paths to determine the most appropriate placement of links. Fortunately, most of the cross-linking needed for exploratory navigation can be accomplished through automated methods.

The extraction tool can be used to create both directed and exploratory navigation paths as long as the information required to insert the links is available either in the source code or in the documentation text. For example, the extraction tool could access the class' full inheritance information and create navigation paths (refer to the *Inheritance Navigation (7.2)* pattern).

Known Uses

Cocoon, JavaDoc, Microsoft on-line help, and other documentation systems provide examples of both categories of navigation tool in various forms. For details, refer to the discussions provided on the **Related Patterns** below.

Related Patterns

- *Hypertext (4.2)*
- *Map Organization to Language Constructs (6.1)*
- *Backtracking (6.4)*
- *Generate Hyperlinks (6.5)*
- *Typed Links (6.7)*
- *Keyword Cross-reference (7.1)*
- *Inheritance Navigation (7.2)*
- *Full-text Search (7.3)*
- *Generate Index (7.4)*

5. Content Documentation Patterns

Abstract

Content patterns describe what information the documentation must contain for the client to effectively use the component. There are two types of information that Content patterns provide: procedural and declarative [Simpson88]. Procedural information describes how to do *something, or how to achieve a certain aim. In contrast, declarative information describes an items properties or characteristics. Many of the Content patterns—such as Generate Short Form (5.4), and Generate Inheritance Diagrams (5.6)—can be fully automated by analysis of the source code. However, some of the most important patterns cannot be implemented through automation. Human authors must provide the content for such patterns as Overview Information (5.3), and Tutorial Information with Examples (5.1), or else severely compromise the completeness and effectiveness of the documentation.*

5.1 Tutorial Information with Examples

Synopsis

Provide the client with information on how to use the component in specific, common use cases.

Problem: Context and Forces

Component clients are typically task oriented. They expect components to be easy to use and are quickly frustrated when they are not, or when the intended usage is not clear. As a result they may either reject the component completely or use it incorrectly, creating errors in their system.

Procedural and reference information complement each other, and both are necessary to facilitate true understanding of the component [Simpson+88]. Reference material consists of raw, unembellished fact. Invaluable for understanding details of class operation, reference information can be found in, for instance, an inheritance diagram, or a short class listing. In contrast, procedural material consists of cookbook

or how-to information concerning the ways in which a class is used in different situations. It is generally not found in source code commenting.

Unlike reference information, procedural information is extremely difficult—if not impossible—to infer from source code and its authoring cannot be generally or reliably automated. Procedural information typically specifies a series of steps that must be performed to achieve some end goal. For instance, documentation for a class representing a text file should include a procedure for opening the file and reading it through until the end.

```
File Reading Procedure:
    1. Open the file by passing the file name into the constructor.
    2. Read a single line using the readLine() function.
    3. While the value returned by readLine() is not zero, read the next line.
```

This description is verbose and requires a translation into source code by the reader. Most software procedures can be specified more completely and succinctly with code samples:

```
MyFile          text(aName);
char*         buffer;

while ( text.readLine(buffer) != 0 )
{
      // perform processing on each line
}
```

Although a simple example, clearly there is no way that this sequence of operations could be reliably inferred from the raw, undocumented source code. The only piece of information that could be inferred from the code is that the constructor must be called before the (non-static) readLine() function.

Solution

Augment each class, class library, and each member function, with tutorial examples describing how to use them to achieve typical goals. For instance, for a class representing a file on disk, tutorial examples should be provided showing the code needed to open, read, and close a file. Similarly, an example should show how to write to a file. Also, to assist software engineers in creating derived classes, base classes should document the set of member functions that *must* be overridden by derived classes as well as those that only *might* be overridden, and how the derived classes will be used within the framework established by the base class.

Careful selection of use-cases will also reduce the need for routing—that is, for forwarding the reader to another section of the documentation. As [Simpson88] points out, "If the sections of the document follow the natural sequence of user tasks, there will be less need to route the reader elsewhere." Further, tutorial examples must be geared toward the audience; trivial examples will not be of much value to an intended audience of highly experienced engineers. The author of the tutorial documentation must exercise care judgment in selecting their examples.

Known Uses

Tutorial examples have a huge number of known uses. Consider virtually any book describing how to use a complex component—UNIX [Stevens92], the Microsoft Foundation Classes or the X Window System, to name a few. These books are filled with code samples showing in detail how to accomplish certain tasks. Ironically, example code is often omitted in the original documentation for components and is too frequently *only* found in companion volumes written by third parties.

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Overview Information (5.3)*
- *Document in Small Chunks (9.2)*

5.2 Diagrams and Illustrations

Synopsis

Include diagrams and illustrations in documentation for clarification purposes.

Problem: Context and Forces

The information needed to understand a software component is often difficult to grasp through textual descriptions alone. The amount of information in text is proportional to its volume, so a great deal of text is required to communicate a great deal of information. There is a danger that important details might be obscured or altogether lost, especially considering the large volume of information needed to understand software components. For example, describing the complex relationships

between the parts of a component would require a great deal of text that would, in turn, be difficult to read and quickly grasp. Important details regarding inheritance and call dependencies, or run-time object interactions could easily be missed. As another example, the client depends on high-level overviews to build an accurate mental picture of how a component might fit into the system as a whole. Subsystem layering, data flow, and the conceptual architecture of the component's environment contain many subtle details that are important to forming that mental picture.

Solution

As a supplement to textual documentation, support the use of diagrams and illustrations within the component documentation. [Simpson+88] articulates the value of diagrams by stating that:

> "...graphics provide many types of information more effectively than text... Graphics...can reduce the information load in text, and...provide more information in less space... [They] can be structured to the immediate task of the user, and they can also be designed to minimize information processing for certain cognitive tasks."

Diagrams and illustrations can come from a wide range of sources, including CASE tools, requirement specifications, photographs, and drawings created either by hand or by using a software tool.

The mechanism used to insert graphic illustrations into documentation will depend on the presentation system. For instance, when using HTML, images would be directly embedded into the documentation text via an HTML tag:

```
/*-------------------------------------------------------------------------
CLASS
      SomeClass
OVERVIEW
      The author can include a diagram in the documentation quite easily
      using the <img>tag, as in the following:

            <img src=someclass_overview.jpg>

      In the final documentation, the tag above would be replaced by
      the image store in the specified file.

-------------------------------------------------------------------------*/
class SomeClass{
public:
      .
      .
      .
```

By using diagrams and illustrations, the author can present concise, clear representations of important information that can be more easily grasped and remembered than equivalent text.

Known Uses

Virtually all printed documentation contains diagrams or illustrations. The Web-based documentation extraction systems Cocoon and JavaDoc allow embedding of images through HTML **<img>** tags.

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Generate Inheritance Diagrams (5.6)*
- *Generate Call Dependency Diagrams (5.7)*
- *Icons and Headings Mark Nodes (8.3)*
- *Embedded Images (9.5)*

5.3 Overview Information

Synopsis

Provide high-level overview descriptions and summaries in the documentation for each component, library, and class.

Problem: Context and Forces

Components, libraries, and classes can be very complex artifacts, and the user needs several types of summary and introductory documentation to successfully begin the learning process. To orient the reader to the purpose of the artifact and the documentation describing it, the following materials are required:

1. An overview description providing a brief introduction to the artifact and outline the scope of its responsibilities within the software. The overview includes any general usage guidelines and underscores any aspects of the artifact that need special attention.
2. A list of the elements comprising the artifact. For components, a list of the libraries and/or classes it contains. For libraries, a list of its classes, global functions, and global data. And for classes, a list of its members, in both short and flat forms.
3. A table of content that lists all of the documentation topics available for the artifact, including the items required for 1 and 2 above.

Without this contextual framework, the reader must first sort through the details of the component in order to understand its general purpose. As a result, the learning process will be slower and more difficult, as will searching for specific pieces of information that may needed.

Solution

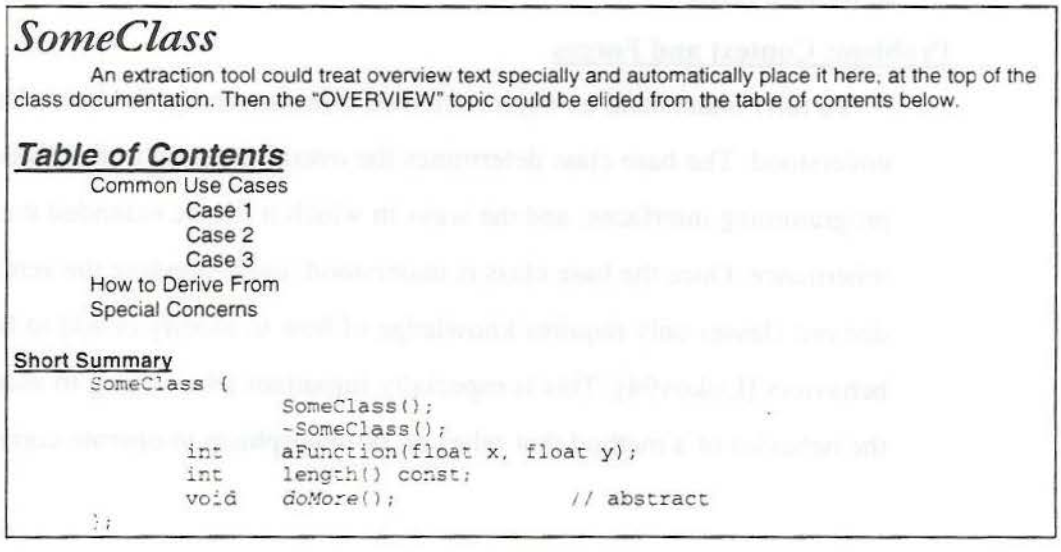
Both manual and automated techniques are required to provide adequate overview documentation for the component. Since overview descriptions do not exist within the source code, and cannot be automatically generated, the author must write them for each of the major software artifacts.

Once the overview descriptions are written, the extraction tool is used to automatically generate summary listings of the elements of each artifact, and create a table of content for the full documentation. Refer to the *Generate Short Form (5.4)* and *Generate Flat Form (5.5)* patterns for examples of generating summary listings. Before the tool can be used to create a table of content some structure must be

imposed on the documentation text so that the tool can distinguish between topic headings and the main body of the text. Once topics can be identified with an algorithmic rule, the extraction tool can create the table of content. The format used by the Cocoon system to identify topic headings is illustrated in the following code example:

```
/*-----------------------------------------------------------------------
CLASS
      SomeClass
OVERVIEW
      Some description text belongs here...
COMMON USE CASES
      Sub-topics are allowed...
COMMON USE CASES: FIRST CASE
      The first use case is described in detail...
COMMON USE CASES: SECOND CASE
      Followed by the second...
COMMON USE CASES: THIRD CASE
      And the third...
HOW TO DERIVE FROM
      A discussion about creating sub-classes...
SPECIAL CONCERNS
      Mention of any special problems or issues with this class...
-----------------------------------------------------------------------*/
class SomeClass{
public:
      .
      .
      .
```

The Cocoon extraction tool recognizes any string starting in the first column and consisting of all upper-case letters as a topic heading. Once the topic headings are identified, a table of content can be produced which includes both the topic headings and all of the automatically generated information. The format for the final documentation might look like the following:



Note that the entries in this table of content should be hyperlinked to the full topic text—refer to the *Hypertext (4.2)* pattern for more details.

Known Uses

It is a standard technical writing practice to precede detailed discussions with overviews or introductions. Listings and tables of content are automatically produced by Cocoon, JavaDoc, and literate programming systems.

Related Patterns

- *Hypertext (4.2)*
- *Leverage Programming Language Semantics (4.5)*
- *Directed and Exploratory Navigation (4.7)*
- *Generate Short Form (5.4)*
- *Generate Flat Form (5.5)*
- *Generate Inheritance Diagrams (5.6)*
- *Generate Call Dependency Diagrams (5.7)*
- *Document in Small Chunks (9.2)*

5.4 Generate Short Form

Synopsis

Automatically create a brief summary of the extensions that a derived class makes to its base class.

Problem: Context and Forces

To fully understand an object-oriented class hierarchy, the base class must first be understood. The base class determines the overall purpose of the class hierarchy, its programming interfaces, and the ways in which it can be extended through inheritance. Once the base class is understood, understanding the semantics of derived classes only requires knowledge of how to modify or add to base class behaviors [Liskov94]. This is especially important when trying to assess or predict the behavior of a method that relies on polymorphism to operate correctly.

Consider one part of a software component for a GUI framework. Such frameworks usually have some base class representing all atomic interface widgets or controls. For instance:

```
class Widget {
public:
      Widget(Widget* parent);
      virtual        ~Widget();
      const char*    getName() const;
      XYPosition     getPosition() const;
      virtual void   draw() const;
      // etc...
}; // Widget
```

The Widget class declares a set of virtual and non-virtual functions that define the way in which all Widget sub-classes will be used and how they will interact with their environment. For example, all Widget sub-class constructors are required to accept a parameter indicating their parent Widget, and **char*** data is used to name widgets. Additionally, some portions of the interface are specified as virtual—such as the **draw**() function—to allow the sub-classes to modify the behavior, *but not the semantics*, established by the Widget base class.

Since the semantics of the base class are (or should be) invariant across all sub-classes, understanding the semantics of the sub-class requires one to first understand the semantics of the base class. Then, understanding the sub-class is simply a matter of learning how the sub-class extends the base class.

The information inherent in the source code of the base class and sub-class declarations—function signatures, data types used, functions that are virtual or overridden, **const**-ness, etc.—is crucial not only to understanding the classes, but also to correctly compiling them. Thus, the information must be syntactically embodied in the source code. Since much of the information needed to understand a sub-class is already present in the source code, it follows that the documentation system should take advantage of it rather than requiring manual duplication.

Solution

Create a tool to compare the declarations of the base class and the derived class, and create documentation describing the added and overridden methods. Present these methods to the reader as a simple tabular list.

Consider a C++ sub-class that inherits several functions *in toto* from the base class, overrides several others, and introduces several new functions. To understand this sub-class, we need to know how its behavior varies from the behavior of the base class. Specifically, which functions were redefined and which were added..

This information is found in the syntax of a C++ class declaration and can be easily extracted. It can then be reformatted (omitting comment text, normalizing indenting and the use of whitespace, etc.) and presented to the user:

```
Class MyClass inherits from SomeBaseClass
      overriding
            public virtual void    function1();
            public virtual void    function2();
      and adding
            public SomeValue              getValue() const;
```

The above information is inherent in the source code of a class declaration. All the documentation system needs is a tool to process the class declarations, extract it, and format it for easy consumption by a reader.

Known Uses

The Eiffel system [Meyer88] was the first to name this type of summary. Class browsers available in compiler IDEs from Microsoft, Borland, etc. also generate this information from source code. Web-based extraction systems such as JavaDoc from Sun Microsystems, cxx2html and Cocoon are also capable of producing this short form of class listing.

Related Patterns

- *Automate Tedious Tasks (4.4)*
- *Leverage Programming Language Semantics (4.5)*
- *Directed and Exploratory Navigation (4.7)*
- *Generate Flat Form (5.5)*
- *Inheritance Navigation (7.2)*
- *Typography-Encoded Details (8.5)*

5.5 Generate Flat Form

Synopsis

Automatically create a complete summary of all class members from all parent classes.

Problem: Context and Forces

A user must understand the complete behavior of an object-oriented class to use it correctly. An integral part of the behavior of a class is found in the combined effect of all of the class' parents. That is, the behavior of a class is the sum of the behaviors of its parents, in addition to any behavior the class itself adds. Each parent, beginning with the root of the inheritance tree, specializes the semantics and behavior of the class through adding or redefining member functions. So despite being a static structure of the software, the analysis of the behavior behind a particular function call can be complex, and the user requires a comprehensive summary of the totality of the class definition.

Solution

For each class, generate a summary that lists all members from all parent classes. The listing of members must be both complete and concise to enable the user to both rely upon it and easily scan it. An example of an effective style of presentation for the listing can be seen in the example given for *Generate Short Form (5.5)*. Ideally, the listing would indicate the class that introduced the member, as well as the most-derived class that redefined it.

Known Uses

The Eiffel programming system was the first to coin the term "flat form" for describing such listings [Meyer88]. The Cocoon and Doc++ documentation systems also generates this style of listing.

Related Patterns

- *Leverage Programming Language Semantics (4.5)*
- *Overview Information (5.3)*

- *Generate Short Form (5.4)*
- *Generate Inheritance Diagrams (5.6)*

5.6 Generate Inheritance Diagrams

Synopsis

Automatically generate diagrams depicting the inheritance structure of the classes.

Problem: Context and Forces

Information about the inheritance structure of an object-oriented component is generally spread out within the source code, making it difficult to browse or summarize. Understanding the overall inheritance structure of the component is a critical first step to correctly using it because inheritance is the nexus of an object-oriented design, embodying and constraining its most crucial elements. The architecture of the component, the solution it implements, its complexity, and its flexibility are all communicated through the inheritance hierarchy.

Manually forming a complete understanding of the inheritance structure through browsing the source code is a difficult, tedious, and error prone task. Parent classes are identified as part of a class declaration, and this results in the inheritance information being distributed throughout the source code. Since each class declaration contains only a small piece of the entire inheritance tree, the information from the scattered declarations must be combined in order to form a comprehensive view.

Solution

Present, in a graphic diagram, the inheritance structure of the classes within the component by using the extraction tool to automatically compile the information from the source code. The inheritance hierarchy can then be displayed as a simple tree diagram, using any standard graph layout algorithm:

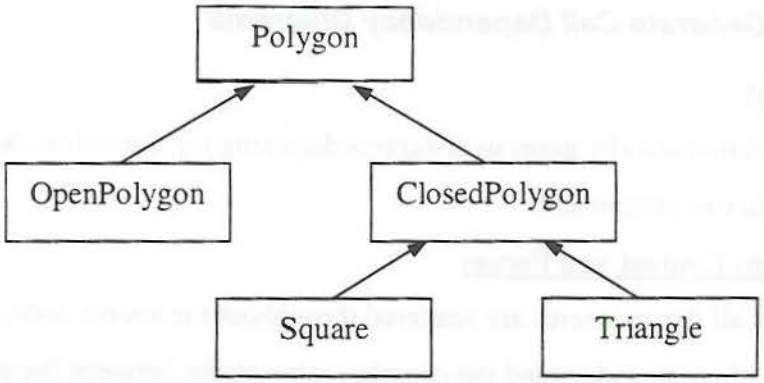


Figure 6: Sample Inheritance Diagram

The information in such diagrams is comprehensive—every class is listed and its inheritance relation to all other classes is apparent. As an added benefit, the class depictions in the diagram could be hyperlinked to its full documentation.

Known Uses

Both Cocoon and JavaDoc provide textual representations of the inheritance structure for each individual class. Although they do not create graphic representations, they do demonstrate the feasibility of extracting the information needed to create the diagrams, as well as attest to their usefulness.

Inheritance diagrams are also used by CASE tools such as Rational Rose and Select OMT (from Select Software Tools Ltd.) as both a design and a documentation tool. In the strictest sense, these CASE tools cannot be considered as a "Known Use" because they are not documentation systems. However, since design documentation is one their key uses, it does serve as a model of the usefulness of inheritance diagrams.

Related Patterns

- *Hypertext (4.2)*
- *Leverage Programming Language Semantics (4.5)*
- *Diagrams and Illustrations (5.2)*
- *Generate Flat Form (5.5)*

5.7 Generate Call Dependency Diagrams

Synopsis

Automatically generate diagrams depicting the inter-class dependencies in each library or component.

Problem: Context and Forces

Call dependencies are scattered throughout the source code, making it difficult for the client to understand the complex interactions between the parts of a component. For the component to serve its intended function, the classes and objects must become interdependent, or *couple*, by collaborating through function calls and object references to each other. Typically, components have numerous classes and many run-time objects, so the internal coupling between them leads to a high degree of complexity. The user must trace through chains of dependencies to understand the ways in which the component parts are coupled, and this is a difficult process when done manually because dependencies are generally strewn throughout the code.

Understanding call dependencies provides three types of vital information to the component client. First, it illustrates the sequence of control of the component calculations. Second, it clarifies points at which side-effects that change the system's state can be introduced. And third, it describes compile and build dependencies for the component. All of this information is important during the development process, especially during debugging and unless it is readily accessible the development work will be impeded.

Solution

To help the user understand the interactions between classes and objects within a component, automatically create diagrams depicting call dependencies using the extraction tool. The *Generate Inheritance Diagrams (5.6)* pattern has an analogous solution that can be referred to for details.

Known Uses

CASE tools, and compilers such as Microsoft Visual C++, provide graphic call dependency information. Although they are not, strictly speaking, documentation

systems, they illustrate the value of the information for documentation purposes. Similarly, call dependencies are an indispensable tool for reverse engineering analysis [Chen+94].

Related Patterns

- *Leverage Programming Language Semantics (4.5)*
- *Diagrams and Illustrations (5.2)*
- *Generate Inheritance Diagrams (5.6)*

5.8 Generate Evaluation Metrics (Proto-pattern)

Synopsis

Automatically generate object-oriented code metrics to help potential clients evaluate the components.

Problem: Context and Forces

Due to a lack of objective information, it is difficult for potential clients to assess the quality, stability, and reliability of software components. Currently, potential clients have only four seriously-flawed sources of evaluation information available to them. The first source is testimonials from existing clients of the software component. Unfortunately, the potential client may not know of or have access to any current clients of the component whose needs closely match their own. As a second source, the potential client can use the marketing materials prepared by the component supplier, however such materials are, understandably, heavily biased in favor of the component. Third, independent reviews of the component from trade magazines such as *Software Development* and *Unix Review* can be examined. However, depending on the application area, the component of interest may never be reviewed, and if it is the review is not likely to go into enough depth to make an adequate assessment. As a final source of evaluation information, the client may be able to obtain an evaluation copy of the component, allowing them to perform analyses, experiments, and tests to determine its suitability. Of the four sources, this is the most useful because it allows the clients to assess the component in their own environment for their specific needs. Unfortunately, not all vendors provide

evaluation copies. Nor is it guaranteed that problems that might arise in a full-scale implementation will be apparent through small isolated tests. Further simply performing the analyses and tests on the evaluation copy is a very time-consuming, and thus expensive, process.

Committing to a component is a long-term investment for a client. Without sufficient information, the client may not chose the right component and will later either have to replace it or adapt their design to an ill-suited component.

Solution

Use the extraction tool to calculate object-oriented metrics which potential clients can use to describe and assess the component. As measurements of quality and complexity, metrics provide an objective source of evaluation data. The appropriate set of metrics to use must be determined by the component vendor, who must then modify the extraction tool to calculate them. There are many possible metrics that can be readily calculated, including:

- Number of classes in the component
- Average number of members per class
- Average number of dependencies between classes
- Average cyclomatic complexity of member functions
- Lines of code (LOC)

Historically, there have been many heated debates over the most meaningful way to calculate certain metrics such as LOC. To avoid misunderstandings, the vendor must describe *precisely* how each metric is calculated.

Known Uses

This pattern is an experimental proto-pattern because it has no known industrial uses. Nonetheless, the problem as outlined is serious and may be addressed by the proposed solution. Therefore, since a feasible approach has been demonstrated, it should be attempted.

Related Patterns

- *Leverage Programming Language Semantics (4.5)*

5.9 Derive Class Characteristics (Proto-pattern)

Synopsis

Automatically list the descriptive attributes that characterize each class.

Problem: Context and Forces

Many attributes of object-oriented classes can provide a quick orientation to users but because they are buried in the source code, they are usually not easily accessible. These attributes describe both usage and performance characteristics of the class, indicating:

- If the class is abstract (cannot be instantiated) or final (cannot be sub-classed).
- If the class is a singleton.
- The size, in bytes, of individual objects in a class.
- If all the member functions are implemented in-line.

To illustrate how easily informative characteristics can be overlooked within the source code, consider the following simplistic example:

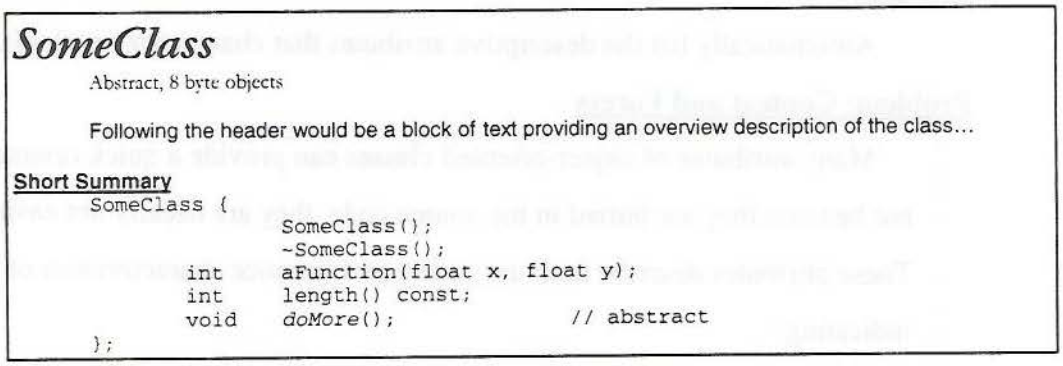
```
Class SomeClass
{
public:
      SomeClass();
      ~SomeClass();
      int    aFunction(float x, float y);
      int    length() const;

      virtual void   doMore() = 0;

private:
      float  _x;
      float  _y;
}; // SomeClass
```

The "**= 0**" in line 9 declares that this is an abstract class. This is vital usage information and the client should be alerted to it, since it limits the use of the class and indicates its role as an abstract concept. The syntax used by C++ in this case is for the compiler's benefit, not the reader's, and an engineer is unlikely to spot it while perusing the source code.

Solution

Use the extraction tool to analyze each class and then label it with descriptive attributes. Any attributes the tool has identified must be listed near the summary

information to supplement the overview documentation. Following is an example of a class' introductory documentation:



SomeClass

Abstract, 8 byte objects

Following the header would be a block of text providing an overview description of the class...

Short Summary

```
SomeClass {
                SomeClass();
                ~SomeClass();
        int     aFunction(float x, float y);
        int     length() const;
        void    doMore();               // abstract
};
```

The prominence of the attributes in this example sharply contrasts the original source code as seem in the problem statement above, and makes the information much more accessible to the user.

In the source code, the extraction tool will look for attributes in one of two basic forms. First, the attribute may be explicit, called out by a keyword or other special syntax. Examples of this would include declarations of *abstract* and *final* classes within Java. In its second form, the information may be implicit within the class declaration or implementation. For instance, the declaration of C++ classes lists the data members, which can be used to calculate the size of instantiated objects. Similarly, the class can be identified as abstract if it has only protected constructors.

Known Uses

Modern languages like Eiffel and Java make such class characteristics a prominent part of the class declaration syntax. For C++ source code, the Cocoon system indicates whether or not each class is abstract or in-line.

Related Patterns

- *Leverage Programming Language Semantics (4.5)*
- *Generate Evaluation Metrics (5.8)*

6. Structure Documentation Patterns

Abstract

Structure patterns describe how the software component documentation should be partitioned, and how its parts should be linked together to form the hypertext topology. These patterns complement the Content patterns: while the Content patterns describe what is available, the Structure patterns describe how the content is accessed and traversed by the reader. Patterns like Map Organization to Language Constructs (6.1) dictate aspects of the component documentation design by demanding that the final documentation produced by the extraction tool be structured in particular ways. Other patterns in this category—such as Generate Hyperlinks (6.5)—utilize the extraction tool to automatically produce the hypertext topology.

6.1 Map Organization to Language Constructs

Synopsis

Structure the hypertext nodes and the links between them to parallel the programming language constructs used to implement the component.

Problem: Context and Forces

When readers are confused by the organizational layout of the documentation, they will devote needless effort trying to understand the documentation, rather than focusing on the component. As [Friendly95] notes in discussing the Java documentation system, the system must "support users well enough that their focus never [has] to shift from understanding the API to navigating the documentation." To achieve this goal, the documentation must be modular, that is, it must be divided "into separate, stand-alone sections that allow the user to select and use those sections fitting his or her particular interests and needs." [Simpson+88] However, designing the modules of the documentation involves two primary difficulties: ensuring that the user is always aware of the purpose and significance of each

section; and ensuring that the relation between two hyperlinked sections is clear [Thuring+91]. A lack of clarity in the organization or linking of the documentation will cause readers confusion and frustration, making it difficult to understand and navigate through.

Solution

Since the reader is already familiar with the programming language constructs, structure the documentation to parallel them, thereby facilitating the reader's formation of a mental model of its organization. The modules of the documentation should be based upon the organizing constructs of the language, and the main navigation paths should reflect the semantic relationships between them. By leveraging the user's existing knowledge about the programming language in this way, their use of the documentation will be more intuitive and natural.

C++ is used to illustrate how the organization of the documentation can parallel programming language constructs. The basic modules of the documentation should be organized around the class, since that is the central abstraction used in C++. Documentation topics provided by the author should be broken out into sub-modules that are accessed from the main class module. Similarly, complete documentation for member functions and data members should be partitioned into individual sub-modules and accessed through a short form listing of the class given in the main class modules. Additional hyperlinks need to be provided to connect the class documentation to related classes, including links to the class' parent classes, and links from member functions to classes used as arguments. The following diagram depicts the hypertext structure as outlined:

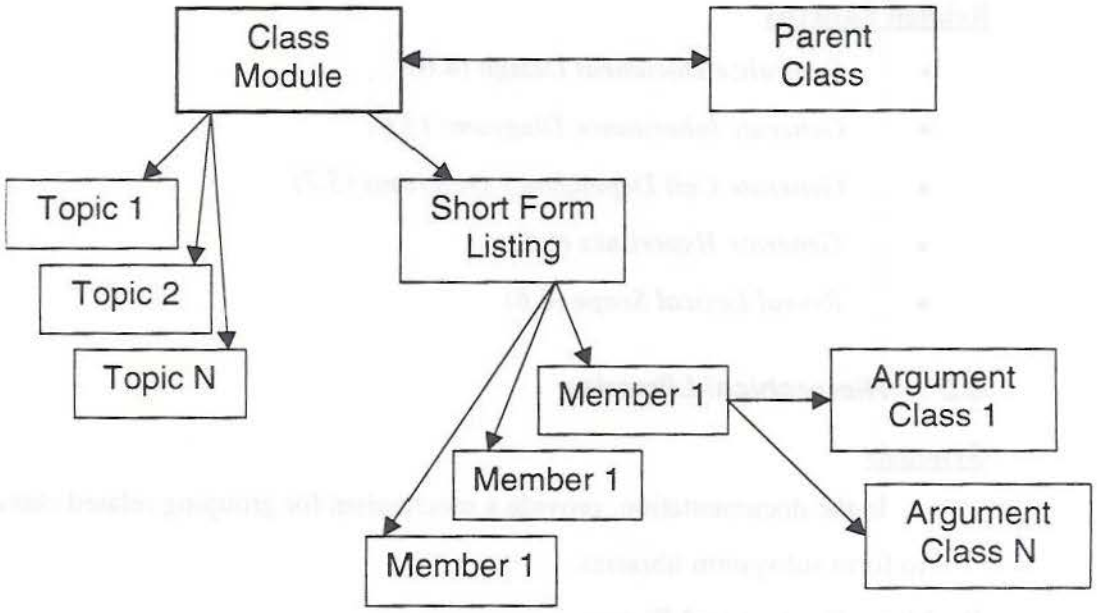


Figure 7: Example of a Hypertext Organization

Other language constructs fit neatly into this basic documentation organization, just as they fit neatly into the language. For instance, for documentation purposes, C++ **struct**s can be treated as mini-classes, and global functions and data can be treated analogously to class members.

This partial example illustrates how the semantics of the programming language are an integral part of the overall hypertext design, guiding the formation of both nodes and links. The resulting hypertext document has an intuitive structure for any engineer who is well versed in the language.

Known Uses

Cocoon, JavaDoc, and cxx2html all implement this basic organizational concept to greater or lesser degrees. Cocoon and cxx2html both use a short form listing as a navigation tool to access the full member function declarations. JavaDoc uses the language-support **package** construct to organize classes into subsystems and structures the documentation accordingly. All three provide linking to parent and argument classes.

Related Patterns

- *Centralize Document Design (4.6)*
- *Generate Inheritance Diagrams (5.6)*
- *Generate Call Dependency Diagrams (5.7)*
- *Generate Hyperlinks (6.5)*
- *Reveal Lexical Scope (6.6)*

6.2 Hierarchical Libraries

Synopsis

In the documentation, provide a mechanism for grouping related classes together to form subsystem libraries.

Problem: Context and Forces

Complex components contain numerous classes that are typically partitioned into libraries (i.e., modules, subsystems, packages) and the documentation should parallel this structure; unfortunately, not all programming languages directly support libraries, so the needed information is not available in the source code. To manage their complexity, components are broken down into libraries, thereby allowing for a separation of concerns between its parts. If a component is complex enough to warrant its subdivision into libraries, it follows that its documentation will certainly

be equally complex and require a similar subdivision. Components written in languages such as Java or Ada are directly supported by such hierarchical composition, so the documentation can simply parallel the organization specified in the source code. However, for languages such as C++ that lack such a facility, the author needs to superimpose a library structure from outside the source code.

Solution

Utilize the extraction tool to create a hierarchical library structure within the component documentation to help manage its complexity. Before the tool can be used, it must first be modified to read a specification of the library structure and to output the final documentation in accordance with it. The structuring information can be provided to the extraction tool through a number of different mechanisms. For example, the extraction tool can read a configuration file containing the structure data, or the data can simply be embedded in comment-directives within the source code. Once the extraction tool has the information, it can produce a hypertext page for each library, and within each page provide hyperlinks to each of the classes contained in the library.

The structure data required by the extraction tool describes a core hierarchy of software artifacts, with the component itself at the top, the individual classes at the bottom, and libraries in between. However, important information about the libraries is still missing. Full documentation for each library would include an overview, a tutorial or use-case descriptions, and some description of how this library fits into the design of the component as a whole. The reader needs to know each library's purpose, and how it is used within the component. And it is the author's responsibility to embed this documentation in the source code (see *Embed Content in Source Code (4.3)*) so the extraction tool can insert it into the final form of the documentation. For consistency, it is suggested that the format of the embedded library documentation be the same as that used for the class documentation. On the next page is an example of library documentation that matches the class documentation example in section 4.3.

```
/*-------------------------------------------------------------------------
LIBRARY
      MyLibrary

      This library is part of the MyComponent software package.

OVERVIEW
      This section describes the purpose and general characteristics
      of MyLibrary.

HOW TO USE
      This section provides use-case examples.

-------------------------------------------------------------------------*/
```

Known Uses

The Cocoon documentation system supports the addition of a library structure to C++ documentation through the use of an external configuration file. The elaborate compilation systems needed to support C and C++—which lack inherent library constructs—also make use of external configuration files called "makefiles". Complementing these uses of the pattern, the value and need for hierarchical library structuring is recognized by the Booch analysis and design methodology, which includes the concept of a class category for grouping related classes.

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Leverage Programming Language Semantics (4.5)*
- *Directed and Exploratory Navigation (4.7)*
- *Tutorial Information with Examples (5.1)*
- *Overview Information (5.3)*
- *Map Organization to Language Constructs (6.1)*

6.3 Link to External Documents

Synopsis

Join supplementary documents to the component documentation through hypertext links.

Problem: Context and Forces

Providing the user with complete and easily accessible documentation is difficult with traditional documentation techniques (the printed page), particularly when

integrating external documentation sources. External sources of important information include design or requirements documents (see section 3.2), ANSI or ISO standard specifications, documentation for the programming language, and documentation of any third-party software used within the component. In traditional documentation, referring to such sources requires the reader to first find the correct manual and then look up the needed information. This is time-consuming and interrupts the learning process.

Solution

Use hypertext links to merge external documentation with component documentation, creating a seamless information space for the reader. From the reader's point of view, the external documentation will be fully integrated with the component documentation, and accessing it will not require any interruption of their problem-solving activities. Using hypertext links in this way has two limitations, however. The primary limitation is that the external documentation must be accessible and in an on-line format compatible with the component documentation. For example, if, as is suggested by the *Distribution via the Web (8.2)* pattern, the component documentation is presented as HTML Web pages, the external documentation must also be accessible from the Web, ideally in HTML format as well. The second limitation is that the external documentation must be topic-addressable: that is, the hypertext linking mechanism must be able to refer not only to an entire document, but also to individual topics within a document. Suppose the external documentation from the previous example is in PDF format; because of the way the HTML hyperlinking mechanism operates, the external documentation will only be able to be referenced in its entirety. Thus, the reader would have to sort through the contents of the whole document, rather than being conveniently directed to the precise topic of interest. These limitations are serious, but considered short-term because of the growing popularity of the Web.

Known Uses

Cocoon, JavaDoc, cxx2html all allow the author to insert HTML URLs directly into the documentation text, seamlessly integrating the component documentation with all the information available on the Web.

Related Patterns

- *Hypertext (4.2)*
- *Directed and Exploratory Navigation (4.7)*
- *Distribution via the Web (8.2)*

6.4 Backtracking

Synopsis

Provide the user with links that backtrack to related, higher levels of documentation.

Problem: Context and Forces

In searching for particular pieces of information within a hypertext document, readers will sometimes pursue unproductive navigation paths, and it is difficult for them to retrace their steps when the documentation system lacks efficient backtracking mechanisms. The documentation system is composed of the hypertext browser and the hypertext document itself, and either of these elements may contain backtracking tools. Unfortunately, hypertext browsers typically provide inadequate tools, such as: history lists which generally require the reader to backtrack through each individual step; and bookmarks which are tedious for the reader to manually administer. And there is no guarantee that the backtracking hyperlinks within the document itself will be consistently available since they will only exist if the author manually creates them. Without adequate mechanisms, readers are forced to start their search over at the top of the documentation each time they wish to backtrack—a tedious and time-consuming method.

Solution

Use the extraction tool to insert, within each module, hyperlinks to higher levels of the documentation, thereby providing the reader with convenient reference points

for backtracking. The extraction tool must be modified to consistently add to each page some standard design element containing backtracking links. These links will connect to programming language constructs higher up in the software hierarchy of the component. As an example, for a member function page, the links should step up through the hierarchy tree as follows: to the class containing the function, the library containing the class, and the component containing the library. Links can be offered through a navigation menubar appearing at the same position on each page, but connecting to the class, library, and component that are appropriate for the function. Through the consistent use of such backtracking links, the documentation system supports the reader by making document exploration faster and more convenient.

Known Uses

Cocoon and JavaDoc both provide backtracking links on all class and function (logical) hypertext pages. These links connect to the class, and the library or package containing the class. Similarly, Microsoft's standard on-line help browser provides the reader with constant access to a standard set of pages that are ideal for starting an information search, thus supporting easy backtracking.

Related Patterns

- *Hypertext (4.2)*
- *Centralize Document Design (4.6)*
- *Directed and Exploratory Navigation (4.7)*
- *Map Organization to Language Constructs (6.1)*

6.5 Generate Hyperlinks

Synopsis

Automatically hyperlink each occurrence of a software artifact's name to its full documentation.

Problem: Context and Forces

Creating the hypertext links for each of the numerous connections between the software artifacts in a component is extraordinarily tedious. Speaking generally about the process of hypertext authoring, [Jordan+89] states:

> "One disadvantage of building... structures in hypertext is the high cost, or 'cognitive overhead', the [author] pays in performing tasks such as creating... links..."

For software component documentation, the problem is exacerbated by the large number of interconnections between artifacts [Sametinger+95] leading to a correspondingly high number of hyperlinks. Additionally—and unfortunately—tedious tasks such as creating hyperlinks are often indefinitely postponed by the time-pressed engineer authoring the documentation.

Solution

Utilize the extraction tool to automatically insert hyperlinks connecting each appearance of an artifact's name to its full documentation. The extraction tool must be modified to output the links—a task it can easily achieve since it must already be aware of the name of each artifact as well as the location of its full documentation in order to create the final documentation.

The automatic hyperlinking can be done for any of the named software artifacts in a component, including classes, functions, packages, libraries, data members, member functions, global variables, and arguments. For small artifacts like data members and member functions, the extraction tool should only insert hyperlinks within the scope of the documentation for their containing structure. For example, hyperlinks to member functions and data members should only be inserted within the context of the documentation for the class declaring them. It is unlikely that the members will be referenced outside of this scope, and by eliminating the external links the overall number of hyperlinks remains manageable.

Fully automating the production of these links will relieve the author of menial work without sacrificing the navigation needs of the reader.

Known Uses

Cocoon, cxx2html, JavaDoc, and Doc++ all perform automatic hyperlinking based on the names of artifacts.

Related Patterns

- *Hypertext (4.2)*
- *Automate Tedious Tasks (4.4)*
- *Directed and Exploratory Navigation (4.7)*
- *Map Organization to Language Constructs (6.1)*

6.6 Reveal Lexical Scope

Synopsis

Structure the documentation to reflect the lexical scope of the software artifacts within a component.

Problem: Context and Forces

Programming languages define rules for lexical scoping which determine when artifacts are accessible, or visible, to other artifacts, and if the documentation does not accurately reflect this scoping, the reader will be unaware of important usage restrictions and will make design and coding decisions based upon incorrect assumptions. Usage restrictions result from many different lexical scoping rules, including:

- Whether class members are public (available to all external clients of the class), protected (available only to functions of derived classes), or private (available only to the class implementation).
- Whether a plain function is global (available to any other function) or static (available only to functions defined in the same source code file).
- Whether one class is nested within another, and thus subject to the public/protected/private scoping rules.
- Whether a Java class is private to a package or is available to any external client.

Misunderstanding the scope of an artifact can lead to serious consequences, from design errors in the inheritance hierarchy, to syntax errors that can only be fixed by detailed analysis of the component source code, to class visibility problems in the

component distribution. It is vital that the documentation clearly indicate the scope of each artifact to avoid such errors and the time required to correct them.

Solution

Make the visibility of each artifact clear to the reader by grouping together items with the same scope, and/or by explicitly labeling artifacts to indicate their scope. The grouping technique presents the information in a concentrated, but easily understood format. The following example illustrates one way that lexical scoping can be clarified through grouping:

```
class Stack
{
public:
                        Stack();
                        ~Stack();
      StackItem&        pop();
      void              push(StackItem& newItem);
      void              clear();

protected:
      void              growStack(int newsize);

private:
      StackItem*        _data;
      int               _currentSize;
      int               _currentAllocation;
}; // Stack
```

The labeling technique, in contrast, is more explicit and forceful, as illustrated by the following sample of the documentation for the *pop()* function:

> ***StackItem& Stack::pop()*** Publicly available
>
> This function pops the most recently *pushed* item from the stack and returns it to the caller. If the stack is empty, calling this function causes an exception to be raised. The caller is responsible for deleting the returned object when they are finished with it

Either of these techniques makes an artifact's scope clear, but, as has been shown, they are not mutually exclusive. The author of the documentation extraction tool must carefully decide on the strategy to be used for representing the lexical scope of each artifact as part of the overall documentation design: the presentation of scoping information must be consistent with the style and layout of the entire document.

Known Uses

Cocoon, cxx2html, and JavaDoc all clearly present lexical scoping information in the documentation. (Cocoon chooses to represent the scope of private class members

by omitting them altogether, since the external component clients will not have access to them.)

Related Patterns

- *Leverage Programming Language Semantics (4.5)*
- *Generate Short Form (5.4)*
- *Generate Flat Form (5.5)*
- *Map Organization to Language Constructs (6.1)*

6.7 Typed Links (Proto-pattern)

Synopsis

Classify hyperlinks by type to enable more directed, selective navigation through the component documentation.

Problem: Context and Forces

When engaged in browsing a hypertext document, readers often know the specific type of information that they are searching for, but must manually sift through all of the hyperlinks available to them even though only a handful of the links are relevant to their specific search. Additionally, since hyperlinks are typically displayed on the screen with a typographic marker such as an underline or a different color, when many hyperlinks are visible the screen can become visual cluttered. When the reader is searching for a *specific type* of information—for example, links to documentation for any third-party software used by the component—the visual clutter can easily occlude the links being searched for. Without some way to manage the complexity of the displayed hyperlinks, the reader may never recognize the hyperlinks they are seeking.

Solution

Tag each hyperlink with data indicating the type of information it connects to, so, based on the user's needs, the hypertext browser can filter out unnecessary links. The idea of classifying hypertext links was first raised in [Parunak89]:

> "If links are *classified* by different types, the topology induced by links of any one type may be much simpler than the overall

> topology of the entire system... The insight for hypermedia is that a hyperbase structured as a set of distinguishable hierarchies will offer navigational and other cognitive benefits that an equally complex system of undifferentiated links does not..."

By locally simplifying its topology, the hypertext will better facilitate directed searches, and reduce screen clutter.

Unfortunately, not all hypertext specification systems currently allow hyperlinks to be tagged with type information (HTML for instance), so the applicability of this pattern is limited.

Known Uses

Despite its currently limited applicability, this proto-pattern solves problems in a way that is similar to other, more established patterns. The idea of filtering information is also found in the *Generate Short Form (5.4)* pattern; the data in the short form is filtered from the complete data available in the flat form to provide the reader with easy access to a specific, limited type of information. This precedent illustrates the potential value of this pattern.

Related Patterns

- *Hypertext (4.2)*
- *Directed and Exploratory Navigation (4.7)*
- *Generate Short Form (5.4)*
- *Generate Flat Form (5.5)*
- *Generate Hyperlinks (6.3)*

7. Search Documentation Patterns

Abstract

Search patterns describe services in the documentation system which aid users in finding information of interest. Most of the patterns in this category are meant to support navigation that is not clearly directed and more intuitive, where the user's knowledge of exactly what they are searching for may be not be clear. Examples of these patterns include Keyword Cross-reference (7.1), and Full-text Search (7.3).

7.1 Keyword Cross-Reference

Synopsis

Provide a mechanism to help clients find classes in a component within categories of interest.

Problem: Context and Forces

Complex components may consist of many classes, each with a well-defined purpose and limited scope of responsibilities and interactions with other classes, and finding a particular class within such a large set can be difficult. For instance, GUI frameworks can easily have 50 to 100 classes or more. While clients may not recall or know the exact name of the class, they usually know something of its purpose or how it fits into the component. What they require is a mechanism to help them use this general knowledge to identify the class of interest.

Solution

Categorize each component by using keywords that describe it. The keywords can be combined in boolean expressions by the client to identify a set of classes of interest through the use of a dynamic search engine. Simpler, yet still effective, methods can also be used. A straightforward index, like those found in virtually all textbooks, can be a particularly effective approach for hardcopy documentation. In a

hypertext document, the index entries can be hyperlinked to the complete class documentation.

In keeping with the "Embed Content in Source Code" pattern in section 4.2, the keywords describing a class should be specified as part of the in-line commenting text for the class and extracted by the cross-reference query mechanism. Following is an examples of how such keywords might be specified:

```
/*-------------------------------------------------------------------------
CLASS
      BezierSurface

      Objects of this class represent a 3D Bezier surface.

KEYWORDS
      Surface, free-form, geometry, face, Bezier

DESCRIPTION
      .
      .
      .
```

Known Uses

Cocoon produces a keyword cross-reference Web page for each library of classes it processes, and supports simple boolean queries. On-line help systems and most reference manuals typically contain index listings.

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Generate Keyword Cross-reference (7.5)*

7.2 Inheritance Navigation

Synopsis

Provide an efficient, convenient means to quickly navigate through the inheritance structure for a given class.

Problem: Context and Forces

Usage information for a class is often spread throughout the classes in its inheritance hierarchy, and unfortunately navigating up and down an inheritance tree one link at a time can be slow and cumbersome, especially for deep and/or wide trees. For deep inheritance trees, many parent classes must be visited, as illustrated by the following sample tree:

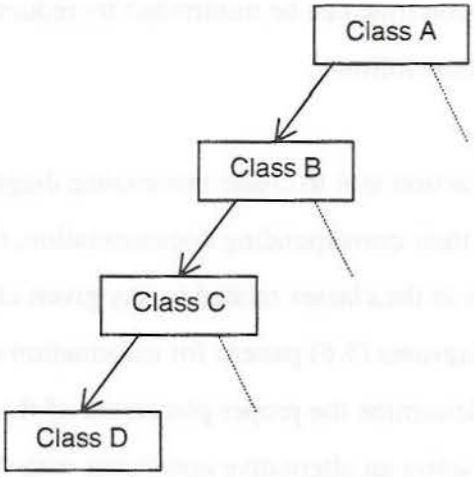


Figure 8 : A Deep Inheritance Hierarchy

In this example, a reader beginning with class D must move three steps up the chain of inheritance before accessing the root parent, a common need especially when the root parent is an abstract class. In contrast, wide trees have a high fan-out rather than many-leveled inheritance:

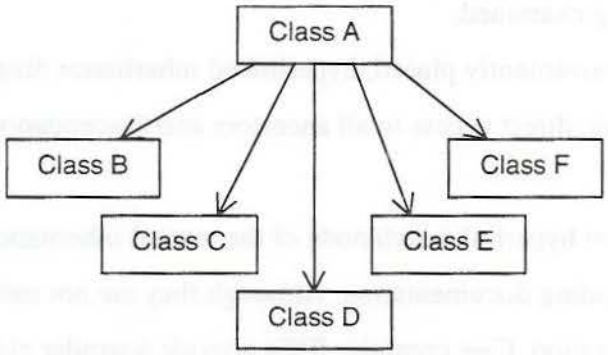


Figure 9: A Wide Inheritance Hierarchy

Suppose that the reader is trying to contrast the behavior of class B against that of its sibling classes: for each switch between siblings two links must be followed, the first to the parent, and the second down to the new sibling.

In both these scenarios, and also in more moderate cases, many links must be followed to move within the inheritance tree. The total time readers spend navigating—an activity that does not directly increase their knowledge or understanding of the classes—is proportional to the number of links they must

follow. Navigation time can be minimized by reducing the number of individual links the user must follow.

Solution

Use the extraction tool to create inheritance diagrams whose nodes are hyperlinked to their corresponding documentation, thus providing the reader with one-step access to the classes related to any given class. Refer to the *Generate Inheritance Diagrams (5.6)* pattern for information on creating the inheritance diagrams. To determine the proper placement of the diagrams, the documentation designer must select an alternative consistent with the overall design. Viable placement alternatives include:

- Place the relevant diagram within the class' main documentation page.
- Provide a hyperlink on the main documentation page to access a different page containing the diagram.
- Use a separate frame to continually display the inheritance diagram appropriate to the class being examined.

With a conveniently placed, hyperlinked inheritance diagram, the reader has immediate, direct access to all ancestors and descendants.

Known Uses

Cocoon hyperlinks each node of the textual inheritance tree it creates to the corresponding documentation. Although they are not meant explicitly for documentation, C++ compiler IDEs provide a similar class browsing capability. Likewise, object-oriented CASE tools such as Rational Rose provide graphical means of directly navigating through an inheritance hierarchy.

Related Patterns

- *Directed and Exploratory Navigation (4.7)*
- *Generate Inheritance Diagrams (5.6)*
- *Map Organization to Language Constructs (6.1)*
- *Generate Hyperlinks (6.5)*

7.3 Full-text Search

Synopsis

Provide utilities to search through the full text of the documentation.

Problem: Context and Forces

The structured search tools typically provided with component documentation—such as keyword cross-references, indices, and tables of content—are of little use when the reader needs low-level details that are either in an unpredictable location or spread throughout the documentation. For instance, suppose the reader is trying to determine which classes throw a particular exception. Structured search tools cannot find this information: a cross-reference can only find a *single* class, and it is unlikely that the exception of interest is listed as a keyword; and indices do not necessarily list *all* occurrences of the item. Or suppose the reader needs to know where a particular file-name is used within the component, but has no clear notion of where to begin looking or what topic to search for.

These examples show that not all reader search needs can be adequately addressed by directed, high-level search mechanism such as indices. So, lacking a general, flexible tool, the reader will be forced to resort to tools and techniques outside the scope of the documentation system.

Solution

In the documentation system, include a mechanism for performing an exhaustive search of both the documentation text and source code of the component. The mechanism used to perform the search can be selected from any number of existing technologies—for instance the Unix *grep* utility, or the WAIS tool. The documentation designer must choose a mechanism that is consistent with the overall design. A full-text search utility provides three important advantages to the reader:

1. The ability to perform specific searches that are not otherwise directly supported by the documentation system.
2. The ability to find candidate locations for beginning a more directed search.

3. And the ability to directly answer important questions.

General tools such as the full-text search are important because they can accommodate user needs that are not foreseen by the designer of the documentation system.

Known Uses

Although they are not currently integrated with component documentation systems, the Unix *grep* utility and the WAIS tool are popular, general tools used for full-text searches. On-line help systems, such as that provided by Microsoft, typically provide the same basic capabilities but are tightly integrated with the documentation system.

Related Patterns

- *Directed and Exploratory Navigation (7.x)*
- *Keyword Cross-reference (7.1)*
- *Inheritance Navigation (7.2)*
- *Generate Index (7.4)*

7.4 Generate Index (Proto-pattern)

Synopsis

Automatically determine which words and phrases in the component documentation are most significant and generate an index from them.

Problem: Context and Forces

A document index is a vital, heavily relied-upon tool, but unfortunately for the documentation authors creating it, it is as [Simpson+88] describes it, "an exercise in drudge work." The author must explicitly identify each individual word or phrase that is to be included in the index, and then manually mark each occurrence—throughout the entire text—of the word or phrase that should be referenced by the index entry. Finally, the author must compile all the occurrences into a completed index. Although word-processing systems (when available) typically can automate this last step, the preceding steps cannot be automated, so the preponderance of the effort must be carried out manually. While an index is an expectation of, and an

indispensable tool for the reader, the author may dramatically scale back its scope, due to the amount of effort it requires.

Solution

Use the extraction tool to analyze the full text content of the documentation, determine which words and phrases are significant, and then automatically create an index. The extraction tool always has complete access to the full text of the documentation so it can easily be modified to perform an analysis of its contents. The difficulty is in identifying the words and phrases that are important enough to belong in an index.

To overcome this difficulty, [Sembugamoorthy+92] describes a statistical technique that automatically indexes a document called Latent Semantic Indexing (LSI). In the LSI analysis of the document text, every word in the document becomes a keyword and is weighted

> "by the amount of information it conveys. Thus, commonly occurring terms receive very low weights relative to infrequent, but more precise terms. [LSI] adds power to full-text retrieval by combining it with a statistical technique that automatically constructs a similarity space in which words with similar meanings are in close proximity. Thus, the end-user's query terms and the terms in the...document do not have to match for the artifact to be found... What is needed to use LSI for software reuse, is to build some relatively straightforward text preprocessors designed to handle...source code, requirements, and design documents."

Although the LSI technique has only been applied to documentation for reuse efforts, the principles can be applied to any documentation, including component documentation. The extraction tool must be modified to preprocess the documentation text as it is being compiled for its final output form, and to then perform the statistical analysis of the LSI method. These modifications will automate an important, yet onerous, task for the documentation author.

Known Uses

The LSI indexing technique is not used by any industrial component documentation system as yet, but it has been tested in code-reuse environments and found to be very useful [Sembugamoorthy+92]. Although this pattern is experimental, because of its potential for saving the author significant effort and because its implementation is relatively straightforward, it is valuable and worthy of implementation.

Related Patterns

- *Automate Tedious Tasks (4.4)*
- *Full-text Search (7.3)*
- *Generate Keyword Cross-reference (7.5)*

7.5 Generate Keyword Cross-reference (Proto-pattern)

Synopsis

Automatically parse class names to create a minimal list of cross-reference keywords.

Problem: Context and Forces

Keyword cross-references are very valuable tools (see *Keyword Cross-reference (7.1)*), but when retro-fitting them to a large, existing code-base, specifying all the keywords for each individual class is tedious and time-consuming work. The need for retro-fitting is widespread, due to the general lack of emphasis placed on documentation. These poorly documented code-bases may contain hundreds of classes, each of which must be examined and described with keywords.

Solution

Use the extraction tool to automatically create cross-reference keywords by deconstructing class names. This is a limited solution in that it will certainly not yield *all* of the best keywords for the class. However, class names, if chosen carefully, do convey a great deal of important information. Consider the example used for the *Keyword Cross-reference (7.1)* pattern:

```
/*-------------------------------------------------------------------------
CLASS
      BezierSurface

      Objects of this class represent a 3D Bezier surface.

KEYWORDS
      Surface, free-form, geometry, face, Bezier

DESCRIPTION
      .
      .
      .
```

In this simple example, the class name contains the two most descriptive of the five keywords.

The extraction tool can easily parse the class names, but the author of the tool must first decide on a reasonable set of parsing rules. The author should examine the code-base(s) being processed to see if the names of the classes follow any convention or scheme from which parsing rules could be devised. As an example, the following three parsing rules work reasonably well for one of the common class naming convention:

1. Changes in case between consecutive letters mark word boundaries. This was the main rule used in the previous example to derive the keywords "Bezier" and "Surface" from the class name "BezierSurface."
2. Words must contain more than three letters, at least one of which must be a vowel. This rule acts as a filter to reduce the number of keywords incorrectly identified, and is especially effective in screening out prefixes that are used to avoid namespace collisions ("rwStack", "mfcString", or "oglPolygon").
3. Words consisting only of upper-case letters are considered to be acronyms, not keywords.

These simple rules are not foolproof, and are dependent on the consistent use of a naming convention throughout the code-base, as well as on the careful, thoughtful naming of the classes. However, with a small amount of effort invested to modify the extraction tool so it can perform the name parsing, a useful, though minimal, keyword cross-reference can be created.

Known Uses

Cocoon implements the parsing rules presented above to produce keywords from class names. It has been successfully used to automatically create reasonably populated cross-references for legacy code-bases.

Related Patterns

- *Automate Tedious Tasks (4.4)*
- *Keyword Cross-reference (7.1)*

8. Presentation Documentation Patterns

Abstract

Presentation patterns specify how the documentation should look with regard to its graphic design, typography, and layout. Patterns like Maintain Consistency (8.1) are similar to the generative patterns in that it describes general principles, but dissimilar because it is restricted to issues of presentation. In contrast, most of the presentation patterns are very specific, and offer concrete guidance, such as the Distribution via the Web (8.2) and Readability through Typography (8.4) patterns.

8.1 Maintain Consistency

Synopsis

Maintain consistency in the visual appearance of the documentation.

Problem: Context and Forces

Documentation with an inconsistent appearance will disorient and confuse readers, diverting their attention from their learning and research tasks. Inconsistency results when two pages are documenting similar items but differ in appearance with respect to issues such as the layout and ordering of elements like headers within each page, or the way in which typography or color are used. As an example, suppose that the class short form summary is included as part of the main class page for all but a small set of classes. The documentation differences would erroneously imply a corresponding difference between the classes themselves which the reader would then pursue, possibly even hypothesizing incorrect distinctions between them.

[Foehr+86] underscores the importance of visual consistency when he states that:

> "To augment the format's analytical organizational structure, a visual framework is needed. Well-executed documentation has a 'consistent look'... Online documentation...requires that formats be designed in such a way that there is resemblance among all of them... Consistency is crucial."

Inconsistency distracts readers from their primary task and may even mislead them, thereby reducing the effectiveness of the documentation.

Solution

Maintain consistency throughout the documentation, including that produced automatically by the extraction tool and that produced manually by the author. The documentation produced by the extraction tool sets the standard for the presentation since it must be deliberately designed to create consistent documentation (refer to the *Centralize Document Design (4.x)* pattern for more details). It is thus incumbent upon authors to be vigilant in ensuring that the documentation they produce is consistent with what is produced by the extraction tool.

Authors can examine the output of the extraction tool or read its documentation to become familiar with its presentation conventions, and then follow those conventions while they write documentation. For example, if the extraction tool systematically represents programming language keywords in bold typeface, the author must do likewise in the documentation they create, and further, must avoid the use of bold typeface for other, conflicting purposes.

The effort devoted to consistency by the author(s) of the documentation and the extraction tool is required to ensure the effectiveness and clarity of the documentation.

Known Uses

Consistency is a cardinal rule of document design, and is apparent in every effective hardcopy manual and on-line help system available today.

Related Patterns

- *Centralize Document Design (4.6)*
- *Icons, Headings and Colors Mark Nodes (8.3)*
- *Readability through Typography (8.4)*
- *Typography-Encoded Details (8.5)*
- *Allow Typographic Control (9.3)*

8.2 Distribution via the Web

Synopsis

Use Web pages on the Internet to provide access to the most current version of the documentation.

Problem: Context and Forces

Without an adequate presentation mechanism, the documentation system cannot provide convenient and prompt access to the current version of the documentation. An adequate presentation mechanism meets a set of four stringent requirements:

1. *Hypertext Support*

   The documentation must be structured as a hypertext document to support reader navigation needs. Refer to the *Hypertext (4.2)* pattern for a full discussion of this requirement.

2. *Portable*

   The presentation mechanism must be available on all major hardware and operating system combinations to accommodate clients' different environments. In particular, the hypertext browser must operate in all environments because it is the main tool used to read the documentation. Refer to the *Environment Independence (4.1)* pattern for more information.

3. *Accessible Source Material*

   The text of the documentation (i.e. the actual physical representation of it) must be immediately available to the reader. Although an obvious requirement, accessibility can be hindered when not in the forefront of the author's mind. Hypertext files that are not available at a client's workstation have as little value as paper documentation that has been lost.

4. *Current Information*

   To make the maintenance of the documentation a manageable task for the component provider, and to ensure that it is kept current, the presentation mechanism should rely on a single, or very small number of locations for the documentation to reside. Refer to the discussion concerning the dual

maintenance problem in the *Embed Content in Source Code (4.3)* pattern for more details.

These four non-functional requirements constrain the choice of presentation mechanism by defining its operational parameters. Thus, the mechanism will ensure users' accessibility to the documentation, which, to the documentation system as a whole, is as important as providing a graphic layout tool.

Solution

Distribute the documentation as HTML Web pages to ensure its universal availability. The HTML delivery format satisfies all four of the requirements identified above. First, HTML Web pages support hypertext linking. Second, HTML is a portable format that is neutral to the operating systems and machine architectures. Additionally, Web browsers are available on all major platforms. Third, with a reliable, high-speed Internet connection, the content is easily accessible to the Web browser. Fourth and finally, the Web pages can be stored at a single site, or on a small set of mirror sites, ensuring that the material can be easily maintained.

Known Uses

Cocoon, cxx2html, and JavaDoc all use HTML Web pages to distribute the documentation they produce. The industry, overall, tends to be using the Web as the *de facto* repository for all documentation.

Related Patterns

- *Environment Independence (4.1)*
- *Hypertext (4.2)*
- *Structure for Physical Management (8.6)*

8.3 Icons, Headings, and Colors Mark Nodes

Synopsis

Use pictorial icons, standard headings, and/or colors to mark the context and purpose of hypertext nodes in the documentation.

Problem: Context and Forces

Readers can easily become lost or confused in large hypertext documents unless the various organizational elements of the documentation are easily identifiable. Elements that are easily identifiable "can be identified through cues independent of their content (that is, without reading the words that make them up)." [Shirk88] While elements like inheritance diagrams have a unique appearance, many others do not. Without visual markers, distinguishing between elements such as short and flat form summaries, individual class topics, or main class and library pages, would require readers to closely examine the content of the pages in order to determine their exact purpose; their work would be interrupted and an unnecessary level of uncertainty would be introduced into the learning process. The reader may even be left with incomplete and faulty information if, for example, s/he misinterprets a class short form as a flat form.

In order to assist the reader in quickly identifying the page they are viewing, it is imperative that the visual cues provided in the documentation be immediately and easily visible at a casual glance. However, it is equally important for the visual cues to be unobtrusive—something that the reader can easily ignore once they have been seen and noted. As an example of an obtrusive cue, suppose that all headings blinked rapidly in order to make them easily visible, yet the blinking was so distracting to the reader that s/he could not concentrate properly on the content of the page.

Solution

Use icons, headings, and/or colors as locality cues to differentiate between visually similar organizational elements in the documentation. The author can select any one of a number of cueing schemes that use highly visible and unobtrusive cues. The following two examples illustrate some ways in which icons, headings, and colors can be used as locational cues.

One way to distinguish between short and flat form summaries is to clearly label each with prominent headings and present them using unique colors. Headings are one of the best overall tools for orienting the reader within a document and are

equally effective in differentiating between visually similar elements of documentation. Summarizing the purpose and use of headings, [Simpson+88] says:

> "A key contextual cue is the selection and placement of...headings. [Headings] should be explicit and used liberally. This helps the user stay oriented during reading and provides locational cues for browsers who are...in search of a specific subject."

An additional tool for orienting the reader in this example is color—different colors can be used for either the summary text or the background color of the pages. Color should not be relied upon exclusively, but can be successfully used to augment other location-cueing mechanisms.

Individual topics of documentation can be distinguished from each other through the use of icons. A particular iconic image can be used to label a topic throughout the documentation. The icon and the topic heading should always appear together in order to 1) create the association between the icon and the topic in the reader's mind, and 2) provide a redundant locational cue. Icons enhance text headings in that they "attract attention to specific areas of the page or screen and provide 'controlled emphasis' to the document." [Simpson+88]

Since the extraction tool creates the final form of the documentation, most of these formatting conventions can be implemented with it, to ensure their consistent application throughout the documentation. In fact, the extraction tool can allow the author to selectively customize some aspects of the locational cues, and guarantee that the customizations are applied consistently. Authors may want to control, for instance, which iconic images appear with which topics, or what colors are used. The author may specify the details of the locational cueing scheme as long as the design of the document makes the reader's location easily, constantly, and consistently known to them.

Known Uses

Both Cocoon and JavaDoc make extensive use of icons and headings as locational cues. In addition, Cocoon allows the author to customize page background colors on

a library per library basis. Most well-written hardcopy documentation uses icons to mark items such as examples and important sidebar comments, as well as headers which identify not just sections, but also each individual page.

Related Patterns

- *Hypertext (4.2)*
- *Centralize Document Design (4.6)*
- *Tutorial Information with Examples (5.1)*
- *Overview Information (5.3)*
- *Generate Short Form (5.4)*
- *Generate Flat Form (5.5)*
- *Generate Inheritance Diagrams (5.6)*
- *Maintain Consistency (8.1)*

8.4 Readability through Typography

Synopsis

Graphically represent the textual information clearly and effectively.

Problem: Context and Forces

Some of the most important information found in component documentation is represented in the syntax of the programming language, including examples and short form listings. The special punctuation and syntax of programming language presents some special problems and concerns for the graphic design of the documentation. The visual presentation of documentation text must be amenable to quick understanding by the reader. "Visual form matters. Effective representation and presentation aids thought, the management of complexity, problem solving, and articulate expression." [Baecker+90]

One problem with programming language syntaxes is that most are not sensitive to whitespace between lexical elements, but human readers most certainly are. Another concern is that while a computer can read the symbol *SouthernCalifornia* as easily as *SOUTHERNCALIFORNIA* or *southern_california*, human readers typically

find either the first or third easier to read than the second. These and other ergonomic considerations are all issues to be considered in the graphic layout of the documentation. Complicating matters further is the fact that source code is simple ASCII text and has no convenient provision for any kind of typographic specification.

Solution

As part of the "Embed Content in Source Code" pattern, the documentation text will be extracted or generated from information in the source code. Graphic design and typography can be used in two ways to clarify and strengthen the documentation.

First, the documentation produced by the extraction tool should have an overall design that facilitates easy reading, including the liberal use of whitespace, a simple layout grid, and headers and separators where appropriate.

Second, typography can be used to aid understanding, especially for information expressed in the syntax of the programming language. Font type, size, special effects. color, and other design elements can all be used to advantage. For instance, typography can emphasize important or especially interesting features (such as whether or not a function is virtual or inline) and de-emphasize less important details. By using consistent typography, the documentation can be made easy to scan for information of a known type. Compare the readability of the following function signature

```
void* MyClass::someFunction(const char* arg1, double** array, int size) const;
```

with a typographically enhanced version:

```
void*
MyClass::someFunction(
                        const char*     arg1,
                        double**        array,
                        int             size)
const;
```

Both of these signatures contain the same text, they differ only in typography. However, in the enhanced version, the use of a more pleasing font, judicious use of point size, italics, bolding, as well as imposing a simple three column grid layout creates easier to read, more visually appealing documentation. A great deal of

empirical work has been done to examine which typographic conventions will yield the most readable source code [Baecker+90].

Yet some aspects of typography are beyond the control of the documentation system. For instance, the use of case in the syntax of the programming language (the *SouthernCalifornia* example above) will influence the readability of the documentation. Thus, it is wise for engineering organizations to understand the importance of typography and establish coding standards to help avoid problematic practices such as the use of all upper-case names.

Known Uses

Cocoon, JavaDoc, cxx2html, c2man, and most hardbound reference manuals all use typographic conventions to clarify the documentation.

Related Patterns

- *Automate Tedious Tasks (4.4)*
- *Leverage Programming Language Semantics (4.5)*
- *Typography-Encoded Details (8.5)*
- *Allow Typographic Control (9.3)*

8.5 Typography-Encoded Details

Synopsis

Use typographic conventions to encode syntactic details where programming language syntax is directly included in the documentation.

Problem: Context and Forces

Programming language syntax included within documentation can obfuscate important data because the notation does not differentiate between primary and secondary information. Most programming languages are designed to communicate with the compiler which reads all information as equally important. However, when the programming language syntax is used in documentation, its purpose becomes communicating with a human reader, who must distinguish between items of more or less importance. For instance, items of primary importance to the reader would be:

- The name of the function.
- The function's return type.
- The names and types of the function's arguments

These items identify the basic information needed to use the function. Items of secondary importance augment or enhance the primary items. This type of information includes:

- Whether or not each argument is **const** (**const** only refines the argument's basic type).
- Whether or not the function is virtual (most functions in an object-oriented design will be virtual).
- Whether or not the function is pure virtual (only relevant when subclassing).

To illustrate the difficulty in distinguishing between primary and secondary information in programming language syntax, consider the following:

```
virtual const SomeClass& myFunction(const int argc,
            char* argv[], const aModeEnum mode) = 0;
```

In this declaration, elements are only distinguished by their position and surrounding syntax, and therefore the reader must carefully inspect the code to locate the primary information. Once the primary information is found, the secondary information can be examined as well. In this simple example of a single function declaration, the visual clutter created by the secondary information is not overwhelming. However, when viewing a block of many function declarations, as in a short or flat class listing, the secondary information can easily obscure the most vital information.

Solution

Encode secondary information in programming language documentation text using typographic conventions. To encode information, simply establish a convention that equates a typographic feature—like italicizing, bolding, or point-size—with a specific word or piece of syntax, and then remove the syntax from the language text. A set of encoding conventions might include:

- Italicize the names of virtual functions
- Bold argument types for **const** arguments
- Omit "= 0" syntax and tag the entire class as abstract

Applying these conventions to the function declaration introduced in the previous section yields the following:

```
SomeClass& myFunction(int argc, char* argv[],
                      aModeEnum mode);
```

Since some of the syntax has been replaced by simple typographic differentiation, the declaration is simpler and easier to visually scan. (Note that applying this solution should be done in concert with applying the *Readability through Typography (8.4)* pattern.)

The use of such typographic conventions is most beneficial for class summaries generated by the extraction tool (refer to the *Generate Short Form (5.4)* and *Generate Flat Form (5.5)* patterns for more information on these summaries). The extraction tool must be programmed to produce syntax in accordance with these conventions.

Additionally, the documentation system must ensure that the reader understands the conventions used. This can be accomplished any number of ways, such as by including a legend with the class summaries, or by simply providing a description of the conventions in the introduction to the documentation system.

Known Uses

Cocoon and cxx2html both use italicized function names to indicate virtual functions in class summaries. JavaDoc uses bullet symbols of different colors to indicate whether a function is public, protected, or private.

Related Patterns

- *Leverage Programming Language Semantics (4.5)*
- *Centralize Document Design (4.6)*
- *Generate Short Form (5.4)*
- *Generate Flat Form (5.5)*

- *Readability through Typography (8.4)*
- *Readable Embedded Documentation (9.4)*

8.6 Structure for Physical Management (Proto-pattern)

Synopsis

Design the physical repository of the documentation to enable its easy versioning and administration as a unit.

Problem: Context and Forces

When the documentation's physical repository is distributed throughout the file system, file management tasks are overly complicated and time-consuming because they must be replicated at each storage location. There are three primary management activities that are complicated by the presence of distributed documentation files: physical management, versioning, and updating.

- Physical management involves the simple copying, deleting, moving, or archiving of files that is endemic to administering computer systems. With distributed files, however, the operations must be repeated at each location.
- Versioning is required in order to support different versions of the documentation for different released versions of the component—the documentation side of the well-known configuration management problem. To support versioning with distributed files, the component version must somehow be added to the names of the documentation files in order to avoid name collisions.
- And updating of the documentation files is required when either the component is modified (within a given release) or the documentation text itself is amended. Prior to updating, the out-of-date material at each distributed location must first be found and removed.

All this additional work increases the effort required for documentation as a whole, and yet does not benefit either the author or the reader.

Solution

Centralize the documentation files produced by the extraction tool in a standalone, dedicated location (directory) to enable them to be versioned and administered as a unit. Making the documentation standalone simply means copying all auxiliary files—such as source code header files or diagrams—to the dedicated location, thereby isolating and encapsulating the full content of the documentation. A single directory can be used to encapsulate the entire set of files needed, and can be treated as a unit during physical management activities such as copying, deleting, or archiving. Other management tasks are also simplified; documentation for different component versions can be stored in different dedicated location, and updating can be accomplished by deleting the out-of-date directory and then re-creating the files by using the extraction tool. The simple expedient of storing all documentation in a single dedicated location greatly simplifies all the mundane file management tasks.

Known Uses

The Cocoon utility provides explicit support for managing the documentation files as outlined above.

Related Patterns

- *Automate Tedious Tasks (4.4)*
- *Centralize Document Design (4.6)*
- *Distribution via the Web (8.2)*

9. Embedding Documentation Patterns

Abstract

The patterns in this category are used to embed documentation text in source code for later extraction, an idea introduced in the Embed Content in Source Code (4.3) generative pattern. Most of the patterns, such as Syntactically Scoped Comments (9.1) and Document in Small Chunks (9.2), are concerned with how the author should structure the information within the source code comments so it can later be extracted into the final form of the documentation. All the documentation discussed in these patterns describe the external view of the component or class rather than its internal implementation.

9.1 Syntactically Scoped Documentation

Synopsis

Use syntactic rules to associate embedded documentation with the programming language construct it documents.

Problem: Context and Forces

If embedded documentation is not clearly and unambiguously associated with a specific programming language construct, the extraction tool will not be able to determine its role within the final form of the documentation. During production of the final documentation, the extraction tool processes the classes, functions, and other language constructs (see *Leverage Programming Language Semantics (4.5)*) found in the source code, including the comment blocks that contain the documentation content. However, since comment blocks in most languages are not syntactically bound to any other language construct, the extraction tool cannot determine which software artifact the documentation describes. This is a serious problem that will make the task of producing the final documentation virtually impossible.

A related issue is raised by the need to keep the documentation for an artifact synchronized with its corresponding source code (see *Embed Content in Source Code (4.3)*). It is critical to the long-term effectiveness of the documentation that it be updated when the source code is modified.

Solution

Develop syntactic rules to associate comment blocks containing embedded documentation with the primary constructs of the programming language. Object-oriented documentation will concentrate on constructs such as classes, functions and variables, all of which have clear, explicit, and unambiguous syntax in most programming languages. Syntactic rules layered over the top of the programming language rules can be used to create associations between the syntax for these constructs and comment blocks adjacent to them. An example of such a rule would be stated informally as:

- Unless there is an empty line between them, a contiguous comment block refers to the class, function, or variable declaration that directly follows it in the source code.

Following are examples illustrating how the rule would be applied to create associations between documentation and declarations.

```
        //////////////////////
        //
        //  This comment block would satisfy the example rule
        //  and be associated with the function foo().
        //
int
foo();
```

```
        /*
        This comment block would not satisfy the example
        rule because of the blank line between the end of
        the comment block and the declaration of bar().
        */

int
bar();
```

This rule is not only easy to understand and implement in the extraction tool, but will correctly create associations with all of the software artifacts of interest in most popular programming languages. Other rules could be constructed, as long as they co-locate the documentation with its corresponding artifact to help ensure that they

remain synchronized; when authors are making modifications to the source code for an artifact, they will see the related documentation and update it accordingly.

The comment text may either precede or follow the declaration—if the rule can be programmed into the extraction tool it will suffice—although preceding declarations with documentation is a more widely accepted practice. Once the extraction tool can identify the construct associated with a block of documentation it can create the final documentation.

Known Uses

Cocoon, and cxx2html both use variations of the sample association rule given above to pair documentation with software artifacts. JavaDoc takes advantage of syntax that is built into Java, called "doc tags," to create the associations [Friendly95]. ([Kaelbling88] posits that indeed *every* construct of a programming language should have a syntactic placeholder for its documentation, and the Java "doc tags" syntax seem to be a move in that direction.)

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Leverage Programming Language Semantics (4.5)*
- *Map Organization to Language Constructs (6.1)*
- *Document in Small Chunks (9.2)*
- *Readable Embedded Documentation (9.4)*

9.2 Document in Small Chunks

Synopsis

Structure the embedded comment text so that software artifacts are documented in small increments.

Problem: Context and Forces

When the documentation for one logical concept or idea does not fit on one screen, the reader will be forced to scroll through the information—an inefficient and time-consuming operation. *One screen* of information is not a universally defined quantity, because it is dependent on a number of factors including the size of the

display window, the font used, and the layout of the non-text portion of the page. As part of its design, however, a documentation system will establish a canonical size for a single page, and it is when this size is exceeded that scrolling is required. Scrolling has two disadvantages. First, it is time-consuming because readers must either reposition their hands on the keyboard, or use their mouse to scroll the window. Second, scrolling is not a desirable form of navigation in a hypertext document since it does not support the quick and intuitive access to information that hyperlinks do. Requiring readers to scroll reduces the efficiency of the documentation system, and distracts them from the task at hand.

Solution

Structure the embedded comment text to document software artifacts in small increments "which do not require scrolling for understanding of a single segment of information."[Shirk88] Both the manually and automatically generated documentation must be structured into small "chunks of information." [Shirk88]

For manually created documentation, the *Syntactically Scoped Comments (9.1)* pattern states that the documentation should be structured around the constructs in the programming language, and the size of documentation chunks must be controlled within this framework. Some of the constructs, such as individual member functions, are conceptually simple enough that they can usually be documented in a single screen. Constructs such as classes, on the other hand, may require many pages of information to adequately describe them, so the author must manually break the documentation down into logical topics and sub-topics that are not too large. One way to do this is to provide a separate topic for each use-case [Jacobson+92] to document each main function of the class (refer to the *Tutorial Information with Examples (5.1)* pattern for more information). This may create a great deal of documentation, but when creating hypertext, the author should not worry about including too much detail, since "...each user will create an individualized path through the information." [Shirk88]

Automatically created documentation must likewise be structured to fit on a single screen, especially for items such as the class flat form summary (see *Generate Flat Form (5.5)*). To avoid screen overflow in this case, the extraction tool can be modified to place the private, protected, and public members of the class on individual hypertext pages.

Not only does breaking the documentation into small, manageable chunks obviate the need for scrolling, it also encourages modularity in the documentation—that is, the logical partitioning of individual concepts or ideas [Simpson+88].

Known Uses

Cocoon, JavaDoc, and cxx2html all allow the author to document in small increments, everything from individual member functions to classes. Cocoon provides explicit support for the nesting of topics and sub-topics in class documentation, while JavaDoc requires more manual effort to achieve the same effect.

Related Patterns

- *Hypertext (4.2)*
- *Map Organization to Language Constructs (6.1)*
- *Syntactically Scoped Comments (9.1)*
- *Readable Embedded Documentation (9.4)*

9.3 Allow Typographic Control

Synopsis

Program the extraction tool to support typographic markup within the manually authored documentation text.

Problem: Context and Forces

Without control over the typography used to display documentation text, the author must resort to formatting conventions that convey less information with more visual disruption. The author is restricted to using the case of letters and special characters such as "*" to signify intensity, stress, or other connotative information. For example, italics are often used to *emphasize* certain words. However, when

limited to NORMAL typography, *other* conventions must be used. Unfortunately, these alternate techniques have several disadvantages: they are visually distracting and can disrupt the reader's attention; there are no standard meanings associated with non-alphabetic characters such as "*"; they will be inconsistent with the typography of the rest of the documentation; and the variety of meanings that can be embedded into such conventions is severely limited unless the author creates an elaborate syntax. In contrast, typographic effects like italics, bolding, underlining, point size, and color preserve the visual harmony of the text, and can be used to easily and unambiguously add meaning to it. Without access to typographic effects, the author's expressive power is limited.

Having access to such effects, however, does not give the author license to use them without constraint. The document design embodied in the extraction tool (see *Centralize Document Design (4.6)*) establishes typographic conventions that must remain consistent throughout the documentation, including the text created manually by the author. It is incumbent on the author to use typographic effects in a manner consistent with the overall document design.

Solution

Program the extraction tool to support typographic markup within the manually authored documentation text. The programming required will vary depending on the presentation format selected for the final documentation. If the format is directly editable—that is, the author can simply insert typographic directives into the documentation text—as with HTML or nroff, the extraction tool must be programmed to copy the text verbatim to the final presentation form. So, using HTML as an example, the embedded documentation might contain text such as:

```
/*
CLASS
      SomeClass

      This is a <em>very</em> important class!
*/
class SomeClass
{
public:
      .
      .
      .
```

The author uses the appropriate HTML tags (such as the **** tag for **em**phasis) to control typography.

However, if the presentation format is not directly editable, the extraction tool must be modified to recognize and translate some special-purpose syntax into the correct final format. For instance, suppose the final documentation is output as a **.pdf** file, and the extraction tool recognized the keywords BOLD and ENDBOLD in the following embedded documentation:

```
/*
CLASS
        SomeClass

        This is a BOLD very ENDBOLD important class!
*/
class SomeClass
{
public:
        .
        .
        .
```

While producing the final form of the output, the extraction tool would need to translate the text and special formatting directives into the appropriate **.pdf** form so the text is displayed as intended.

Either solution achieves the desired effect, but require different amounts of effort in modifying the extraction tool.

Known Uses

JavaDoc and cxx2html both allow the author to directly use HTML in the embedded documentation text. Cocoon supports a hybrid approach: the author may directly use HTML, but Cocoon also supports a small set of conventions that modify the documentation text by inserting HTML tags and replacing keywords with tags. For example, Cocoon will insert a paragraph separator tag (**<p>**) in place of an entirely blank line so the author need not clutter the embedded documentation with them. Also, Cocoon recognizes the keywords EXAMPLE and END as bracketing pre-formatted code examples, and replaces them with the appropriate HTML tags during output. Features such as this are supported so authors are allowed some measure of formatting control, but are not *required* to learn HTML. However, if they do know it, they can take advantage of its full expressive power.

Related Patterns

- *Centralize Document Design (4.6)*
- *Distribution via the Web (8.2)*
- *Readability through Typography (8.4)*
- *Typography-Encoded Details (8.5)*
- *Readable Embedded Documentation (9.4)*

9.4 Readable Embedded Documentation

Synopsis

Design a documentation-embedding syntax that enhances the readability of the source code.

Problem: Context and Forces

Documentation embedded in the source code which lacks visual organization, uniformity, clarity, and distinctiveness cannot effectively convey information until it has been extracted into its final form. The documentation embedded in the code will be needed by the maintenance engineers (see section 3.3), and its legibility is dependent on the coherence of its visual presentation as reflected in the four attributes identified. For the documentation to be visually organized, its physical structure should reflect the logical or topical structure of the documentation text. As an example, documentation that is indented to reflect the nesting level of the topics and sub-topics is easier to comprehend than documentation that is evenly left-justified. For the documentation to be visually uniform, it must follow the same syntactic conventions regardless of the language constructs being documented. As an example, using the same style of C++ comment for all embedded documentation will create a visually consistent presentation. For the documentation to have clarity, it must be as free of artificial syntactic constructs as possible. As an example, documentation text that indicates paragraph breaks through the use of a blank line is more natural to read than text that uses a special character or string to signal the end of a paragraph. And finally, for documentation to be visually distinctive, it must be distinguishable from source code text at a glance. As an example, documentation

using syntactic symbols that are also used in the programming language will be more difficult to notice in the overall flow of the source code text than if it used symbols that did not appear in the language. All of these attributes must be present to create easily comprehensible source code documentation text.

A related and complementary constraint on the syntax of the embedded documentation is that it must be as simple and natural as possible. This constraint arises out of the needs of the author rather than the needs of the reader. Typically, software engineers are somewhat reticent to create documentation, but if the author's task is made as simple as possible, it is more likely to be done. In this case, that means the documentation system must use a syntax that is easy to understand and remember, one that can be followed once a few simple rules are known.

Solution

Design the documentation embedding syntax to maximize its readability by using formatting conventions in place of special characters or keywords, and/or using keywords in place of special characters wherever possible. This hierarchy of techniques is presented in order of their effectiveness:

1. **Formatting Conventions**—Formatting the text using whitespace and columns is the simplest, most unobtrusive, and therefore effective, syntactic tool. This same technique is commonly used by software engineers to organize the source code. As an example of how it can be used for documentation, topic headings can begin in column 1, and then the topic body itself can be indented from the left margin. Or blank lines can be used to signal paragraph breaks. While not an appropriate tool for programming language syntax, formatting conventions are ideally suited for documentation syntax since the use of whitespace and columns effectively creates a layout grid which enhances uniformity, clarity, and distinctiveness.
2. **Keywords**—Reserved keywords that signal syntactic elements have the advantage of blending directly into a block of natural-language text, while still being easy to locate. For example, keywords such as "EXAMPLE" and

"END" can be used to enclose a source code example in the documentation text. Keyword syntax enhances the organization, uniformity, and distinctiveness of the embedded documentation.

3. **Special Characters and Symbols**—If neither formatting conventions nor keywords are practical for a particular syntactic need, special character sequences can be used. For instance, suppose that syntactic markers were needed caution the reader regarding possible adverse effects of a function call, and that simple formatting or keyword techniques were determined to be too inconspicuous to function as a warning. In such a case, special character syntax such as "!!!" could be used to mark the warning text. The main advantages of special characters are that they can be used to create a very compact notation and they stand out in contrast to the rest of the documentation text. The main disadvantage of using special characters, however, is that when used too often the syntax of the documentation will begin to look more and more like the syntax of the source code, diminishing its distinctiveness. Given a software engineer's expertise with programming languages, it might seem natural to use such terse syntactic conventions, but in the interests of the readability of the documentation they should generally be avoided in favor of formatting conventions and keywords.

By designing the embedding syntax with these techniques the documentation will be both easy to read and easy to write within the source code.

Known Uses

Cocoon, cxx2html, and JavaDoc successfully use all three of these techniques in varying degrees. Cocoon and cxx2html primarily use formatting conventions to delineate syntactic units in the documentation. Keywords are used in Cocoon mainly as substitutes for HTML tag syntax. JavaDoc and Cocoon both make limited and effective use of special character syntax. JavaDoc mixes the use of special characters and keywords by using an "@" character to signify that the next word is a keyword

describing a class member, thereby making extensive use of a *single* special character but in a highly uniform manner.

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Document in Small Chunks (9.2)*
- *Allow Typographic Control (9.3)*

9.5 Embedded Images (Proto-pattern)

Synopsis

Centralize file management by embedding ASCII-encoded images within the source code for later extraction, decoding, and inclusion in the final documentation.

Problem: Context and Forces

When diagrams and illustrations created by the author are stored in individual binary files, the author is forced to manage them, in addition to the source code. The documentation in the source code and the auxiliary images used in the documentation form one logical, cohesive unit. However, the binary nature of most graphics formats require the images to be separated from the source code; unlike word processing files, source code is in plain ASCII format and cannot accommodate the embedded binary data. Thus, the author must physically manage not only the source code files, but also any files containing auxiliary diagrams and illustrations, doubling the effort required to move, copy, delete, or perform other configuration management tasks.

Solution

Using standard tools, encode images into ASCII form and embed them within special comment blocks that the extraction tool can use to create individual image files for the final documentation. The process is relatively straightforward.

1. To create diagrams and illustrations, authors use some form of graphics editing tool and then store the images in its corresponding binary form (e.g. GIF or JPEG).

2. Using a tool such as **uuencode,** the binary files are encoded into ASCII format.
3. The encoded image data is placed into the appropriate source code file and delimited by a specially formatted comment, as illustrated in the following example:

```
/* IMAGE example.gif
WKDKKWKLWLEISKKALALKEKEKEKSSIEKWQLKQKEKRJGOSIIKAKA.E.E.KK3IWKWK
SWLWLI34I3O2KWKLWK.,SLSLOAOA9E9E948KSKSLQLQLI33K2L2L3K3K2L2K3L3
KDSKKWKLK4TNNRNMJFDJDF8DF88ERRJ4J3JIRE8E83K4JTHDJKSKWI3U4U5J5J3
        .
        .
        .
*/
```

4. As a part of the source code files, the image data is now managed automatically along with the source.
5. The extraction tool detects the special comment syntax, decodes the image data, and creates the standalone image files that can be referenced by (or directly included in) the final documentation.

By embedding the images in the source code, the author is relieved of the burden of managing extra files, and also of ensuring that the image files are available to both the extraction tool and the final documentation.

Known Uses

This pattern has not yet been implemented in any documentation system. However, because it automates a tedious task and is relatively straightforward to implement, the potential value of this pattern justifies its use.

Related Patterns

- *Embed Content in Source Code (4.3)*
- *Automate Tedious Tasks (4.4)*
- *Diagrams and Illustrations (5.2)*
- *Structure for Physical Management (8.6)*

10. References and Further Reading

10.1 References

[Alexander+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[Baecker+90] Ronald M. Baecker, Aaron Marcus. *Human Factors and Typography for more Readable Programs*. ACM Press. Addison-Wesley. 1990.

[Biggerstaff+87] Ted Biggerstaff, C. Richter, "Reusability framework, assessment, and directions", *IEEE Software*, 4(2), March 1987.

[Buschmann+96] Frank Buschmann, Regine Meunier, Hans Rohnert Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Chichester, 1996.

[Carlson88] Patricia Ann Carlson. "Hypertext: A Way of Incorporating User Feedback into Online Documentation," in *Text, ConText, and HyperText: Writing with and for the Computer*. Edward Barret, editor. The MIT Press, Cambridge MA, 1988.

[Chen+94] X. P. Chen, W. T. Tsai, J. K. Joiner, H. Gandamaneni. "A Software Maintenance Model for C++ Programs." Technical report TR 94-32. Department of Computer Science, University of Minnesota.

[Coplien+95] J. Coplien and D. Schmidt. *Pattern Languages of Program Design*, "A Generative Development-Process Pattern Language". Addison-Wesley, Massachusetts, 1995.

[Creech+91] Michael L. Creech, Dennis F. Freeze, Martin L. Griss. "Using Hypertext in Selecting Reusable Software Components," *Hypertext '91 Proceedings*, San Antonio Texas, December 1991.

[Covington85] Michael Covington, "Documentation That Works," *PC Tech Journal*, January 1985.

[Delisle+86] Norman Delisle, Mayer Schwarz. "Neptune: A hypertext system for CAD applications," Tektronix Laboratories technical report no. CR-85-50, 1986.

[Dusink93] Liesbeth Dusink. "Insight in the Reuse Process?" Sixth Workshop on Institutionalizing Software Reuse. Owego, New York, 1993.

[Faget+92] Jean Faget, Jean-Marc Morel. "The REBOOT approach to the concept of a reusable component." The Fifth Workshop on Institutionalizing Software Reuse. Palo Alto, California, 1992.

[Foehr+86] Theresa Foehr, Thomas B. Cross. *The Soft Side of Software: A Management Approach to Computer Documentation*. John Wiley & Sons, new York, 1986.

[Friendly95] Lisa Friendly. "The Design of Distributed Hyperlinked Programming Documentation." International Workshop on Hypermedia Design. Montpellier France, June 1995.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[Gullichsen+88] Eric Gullichsen, Dilip D'Souza, Patrick Lincoln, Khe-Sing The. The PlaneTextBook, MCC technical report STP-333-86, 1986. Republished as MCC technical report STP-206-88, 1988.

[Halasz+87] F. G. Halasz, T. P. Moran, R. H. Trigg. "NoteCards in a nutshell," in *Proceedings of the 1987 ACM Conference on Human Factors in Computer Systems* (CHI+GI '87). Toronto, Ontario, April 1987.

[Jacobson+92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Massachusetts, 1992.

[Jordan+89] Danial S. Jordan, Danial M. Russell, Anne-Marie S. Jensen, Russell A. Rogers, "Facilitating the Development of Representations in Hypertext with IDE,"

in *Hypertext '89 Proceedings*, Association of Computing Machinery, New York, New York, 1989.

[Kaelbling88] Michael Kaelbling. "Programming Languages Should NOT Have Comment Statements," SIGPLAN Notices, Vol. 23, No. 10, 1988.

[Latour+88]Larry Latour, Elizabeth Johnson. "SEER: A graphical retrieval system for reusable Ada software modules." *Third International IEEE Conference on Ada Applications and Environments*, May 1988.

[Liskov94] Barbara Liskov, Jeannette M. Wing. "A Behavioral Notion of Subtyping." *Transactions on Programming Languages and Systems*. Volume 16, Number 6, 1994.

[Meyer88] Betrand Meyer, *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Onoma+95] A. K. Onoma, W. T. Tsai, F. Tsunoda, H. Suanuma, S. Subramanian. "Software Maintenance—An Industrial Experience," *Software Maintenance: Research and Practice*, Volume 7, pages 33-375, 1995.

[Parnas+83] D. L. Parnas, P. C. Clements, D. M. Weiss. "Enhancing Reusability with Information Hiding", *ITT Proceedings of the Workshop on Reusability in Programming*, Newport RI, 1983.

[Parunak89] H. Van Dyke Parunak, "Hypermedia Topologies and User Navigation," in *Hypertext '89 Proceedings*, Association of Computing Machinery, New York, New York, 1989.

[Sametinger+95] J. Sametinger, S. Schiffer. "Design and Implementation Aspects of an Experimental C++ Programming Environment," *Software--Practice and Experience*, Volume 25, Number 2. John Wiley & Sons, February 1995.

[Sembugamoorthy+92] Vel Sembugamoorthy, Lynn Streeter, Bill Keese, Mary Leland. "Igrep: A Real World Perspective on Locating Software Artifacts for Reuse." Fifth Workshop on Institutionalizing Software Reuse. Palo Alto, California, 1992.

[Shirk88] Henrietta Nickels Shirk, "Technical Writers as Computer Scientists: The Challenges of Online Documentation", in *Text, ConText, and HyperText: Writing with and for the Computer*, Edward Barret editor. The MIT Press, Cambridge MA, 1988.

[Simpson+88] Henry Simpson, Steven M. Casey. *Developing Effective User Documentation*. McGraw-Hill, New York, 1988.

[Stevens92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading MA, 1992.

[Thuring+91] Manfred Thuring, Jorg M. Haake, Jorg Hannemann. "What's Eliza doing in the Chinese Room? Incoherent hyperdocuments--and how to avoid them," *Hypertext '91 Proceedings*, San Antonio Texas, December 1991.

[Zimmer95] Walter Zimmer. "Relationships between Design Patterns," in *Pattern Languages of Program Design*, James Coplien and Douglas Schmidt, editors. Addison Wesley, 1995.

10.2 Related Web Sites

Patterns
http://st-www.cs.uiuc.edu/users/patterns/patterns.html
Organizational Patterns
http://www.bell-labs.com/people/cope/Patterns/Process/index.html
Cocoon
http://www.cs.umn.edu/~kotula/cocoon/cocoon.htm
JavaDoc
http://java.sun.com/products/jdk/javadoc/index.html
cxx2html
http://info.cv.nrao.edu/aips++/RELEASED/cxx2html
Doc++
http://www.zib.de/Visual/software/doc++/index.html
Literate Programming FAQ
http://shelob.ce.ttu.edu/daves/faq.html
PERCEPS: The Perl C++ Surveyor
http://friga.mer.utexas.edu/mark/perl/perceps

10.3 Further Reading

Adar, M., R. Field, W. Kubalski, T. Chou. "The Objectworks Browsing Model – Understanding Complex Software." Technology of Object-Oriented Languages and System TOOLS 11, Proceedings of the Eleventh International Conference. Santa Barbara, CA, 1993.

Aho, A. V., R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

Arnold, Ken, James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

Basili, Victor R., Lionel Briand, Walcelio L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators," Technical report UMIACS-TR-95-40. University of Maryland, Department of Computer Science, 1995.

Beck, Kent. "Patterns and Software Development." *Dr. Dobb's Journal*. Volume 19, Number 2, February 1994.

Beck, Kent, Ralph Johnson. "Patterns Generate Architectures." *European Conference on Object-Oriented Programming*. Springer-Verlag, 1994.

Biggerstaff, Ted. "Hypermedia as a tool to aid large scale reuse," Technical report STP-202-97, MCC, July 1987.

Biggerstaff, Ted, Alan J. Perlis. *Software Reusability: Volume I: Concepts and Models*. ACM Press, New York NY, 1989.

Biggerstaff, Ted, Alan J. Perlis. *Software Reusability: Volume II: Applications and Experience*. ACM Press, New York NY, 1989.

Booch, Grady. *Object-oriented Analysis and Design with Applications*, second edition. Benjamin/Cummings Publishing Company, Redwood City California, 1994.

Brooks, Frederick, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, Reading, MA, 1975.

Casonova, Marco A., Luiz Tucherman, Maria Julia D. Lima, Jose L. Rangel Netto, Noemi Rodriguez, Luiz F.G. Soares. "The Nested Context Model for Hyperdocuments," *Hypertext '91 Proceedings*, San Antonio Texas, December 1991.

Catlin, Karen Smith, L. Nancy Garret, Julie A. Launhardt. "Hypermedia Templates: An Author's Tool," *Hypertext '91 Proceedings*, San Antonio Texas, December 1991.

Cavaliere, Michael J. "Reusable code at the Hartford Insurance Group", *ITT Proceedings of the Workshop on Reusability in Programming*, Newport, RI, 1983.

Chao, David. "Software Reuse: Major Issues Need to Be Resolved Before Benefits Can Be Achieved." Sixth Workshop on Institutionalizing Software Reuse. Owegeo, New York, 1993.

Chen, Patrick Schicheng, Rolf Hennicker, Matthias Jarke. "On the Retrieval of Reusable Software Components," in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

Coad, Peter. "Object-oriented patterns." *Communications of the ACM*. Volume 35, Number 9, September 1992.

Coad, P., J. Nicola. *Object-Oriented Programming*. Yourdon Press, 1993.

Coplien, J. O. "Generative Pattern Languages: An Emerging Direction of Software Design." *C++ Report*, SIGS Publications, July-August 1994.

Curtis, Bill. "Cognitive Issues in Reusing Software Artifacts", in *Software Reusability, Volume II, Applications and Experience*, Ted J. Biggerstaff and Alan J. Perlis, ed. ACM Press, New York, 1989.

Dershem, H. L., M. J. Jipping. *Programming Languages: Structures and Models*. Wadsworth Publishing, 1990.

Deutsch, L. Peter. "Design Reuse and Frameworks in the Smalltalk-80 System", in *Software Reusability, Volume II, Applications and Experience*, Ted J. Biggerstaff and Alan J. Perlis, ed. ACM Press, New York, 1989.

Dyer, M. *The Cleanroom Approach to Quality Software Development.* John Wiley & Sons, Inc., 1992.

Edwards, Stephen. "Good Mental Models are Necessary for Understandable Software." The Seventh Workshop on Institutionalizing Software Reuse. St. Charles, Illinois, 1995.

Ellis, C. A., S. J. Gibbs, G. L. Rein. "Groupware: Some issues and experiences," *Communications of the ACM*. Volume 34, Number 1, January 1991.

Ellis, Margaret, Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

Foster-Johnson, Eric. *Unix Programming Tools*. M & T Books, New York, 1997.

Fraser, Steven D., Clifford S. Saunders. "Enhanced Reuse with Group Decision Support Systems," in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

Fugini, M. G., S. Faustle. "Retrieval of Reusable Components in a Development Information System," in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

Gabriel, Richard. *Patterns of Software*. Oxford University Press, New York, 1996.

Goldberg, A., D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983.

Goodell, Mike. "What Business Programs Do: Recurring Functions in a Sample of Commercial Applications", *ITT Proceedings of the Workshop on Reusability in Programming*, Newport RI, 1983.

Hayes, Phil, Jeff Pepper. "Towards An Integrated Maintenance Advisor", in *Hypertext '89 Proceedings*, Association of Computing Machinery, New York, New York, 1989.

Hecht, M. S. *Flow Analysis of Compute Programs*. North Holland, New York, NY, 1977.

Heisler, K. G., P.E. Johnson, W. T. Tsai, Y. Kasho, J.R. Snyder. "Software Representation to Support Change," Ph.D. Thesis, Department of Computer Science, University of Minnesota, 1993.

Hopcroft, J. E., J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

Horowitz, Ellis, John B. Munson. "An Expansive View of Reusable Software," *IEEE Transactions on Software Engineering*. Volume 10, Number 5, September 1984.

Jette, Christina, Reid Smith. "Examples of Reusability in an Object-Oriented Programming Environment," in *Software Reusability: Volume II: Applications and Experience*, Ted Biggerstaff, Alan Perlis, editors. ACM Press, New York, 1989.

Johnson, Ralph. "Documenting Frameworks Using Patterns." *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*. ACM Press, 1992.

Joiner, J. K., W. T. Tsai. "Ripple Effect Analysis, Program Slicing, and Dependence Analysis." Department of Computer Science, University of Minnesota, 1993.

Jones, T. C. "Reusability in programming: A survey of the state of the art." *IEEE Transactions on Software Engineering*, Sept 1984.

Jones, W. P. "On the applied use of human memory models: The memory extender personal filing system," *International Journal Man-Machine Studies*, Vol. 25, 1986.

Karlsson, Even-Andre. "A Cleanroom approach to object oriented development for reuse", The Fifth Workshop on Institutionalizing Software Reuse, Palo Alto, CA, 1992.

Katz, Schmuel, Charlie Richter, Khe-Sing The. "PARIS: A System for Reusing Partially Interpreted Schemas," *9th International Conference on Software Engineering*, Monterey CA, March 1987.

Leijter, M., S. Meyers, S. P. Reiss. "Support for Maintaining Object-Oriented Programs." IEEE Transactions on Software Engineering, Volume 18, Number 12, 1992.

Lippman, S. B. *C++ Primer*, second edition. Addison Wesley, 1991.

Liskov, B. Oral presentation at OOPSLA 87. Orlando, FL, October 1987.

Litvintchouk, Steven D., Allen S. Matsumoto. "Design of ADA Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification," *IEEE Transactions on Software Engineering*. Volume 10, Number 5, September 1984.

Maguire, Steve. *Writing Solid Code*. Microsoft Press, Redmond, WA, 1993.

Malcolm, Kathryn C., Steven E. Poltrock, Douglas Schuler. "Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise," *Hypertext '91 Proceedings*, San Antonio Texas, December 1991.

Matsumoto, Yoshihiro. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels", *IEEE Transactions on Software Engineering*, Vol. 10, Number 5, September 1984.

Meyer, Bertrand. *Eiffel: the Language*. Prentice Hall, NJ, 1992.

Meyer, Bertrand. "Reusability: The case for Object-Oriented Design", *IEEE Software*, Vol. 4, Number 2, March, 1987.

Meyer, Bertrand. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, NJ, 1994.

Morel, Jean-Marc, Jean Faget. "The REBOOT Environment," in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

Neighbors, James M. "DRACO: A Method for Engineering Reusable Software Systems", in *Software Reusability: Volume I: Concepts and Models*. ACM Press, New York NY, 1989.

Nielsen, Jakob. "The Art of Navigating Through Hypertext", *Communications of the ACM*. Volume 33, Number 3, March 1990.

Ning, Jim. "Module Interface Specification and Large-Grain Software Reuse," *Sixth Workshop on Institutionalizing Software Reuse*. Owego, New York, 1993.

Parnas, D. L. "Information distribution aspects of design methodology." Technical Report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1972.

Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*. Volume 15, Number 12, December 1972.

Patel, Sukesh, Alan Stein, Paul Cohen, Kich Baxter, Steve Sherman. "Certification of Reusable Software Components," *The Fifth Workshop on Institutionalizing Software Reuse*. Palo Alto, CA, 1992.

Prieto-Diaz, Ruben. "Classification of Reusable Modules," in *Software Reusability: Volume I: Concepts and Models*. ACM Press, New York NY, 1989. Based on a paper published in *IEEE Software*, Vol. 4. No. 1, 1987.

Rice, John, Herb Schwetman. "Interface Issues in a Software Parts Technology", in *Software Reusability: Volume I: Concepts and Models*. ACM Press, New York NY, 1989.

Rubens, Philip, Robert Krull, "Designing Online Information", in *Text, ConText, and HyperText: Writing with and for the Computer*, Edward Barret ed. The MIT Press, Cambridge MA, 1988.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

Shlaer, S., S. Mellor. *Object-Oriented Systems Analysis*. Yourdon Press, 1988.

Selby, Richard W. "Quantitative Studies of Software Reuse", in *Software Reusability, Volume II, Applications and Experience*, Ted J. Biggerstaff and Alan J. Perlis, ed. ACM Press, New York, 1989.

Soerumgaerd, Lars Sivert, Frode Stokke. "Experiences from application of a Faceted Classification Scheme," in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

Soloway, Elliot, Kate Ehrlich. "Empirical Studies of Programming Knowledge." *IEEE Transactiosn on Software Engineering*. Volume 10, Number 5, September 1984.

Sommerville, I. *Software Engineering*, third edition. Addison Wesley, 1989.

Tsai, W. T., H. Gandamaneni, J. Sun, X. P. Chen, S. Subramanian, J. K. Joiner. "Software Maintenance for Object-Oriented Programs." Technical report TR 94-07. University of Minnesota, 1994.

Van Wyk, Christopher J. "Literate Programming: An Assessment," *Communications of the ACM*. Volume 33, Number 3, March 1990.

Wasserman, A. I., S. Gutz. "The future of programming", *Communications of the ACM*, 1982.

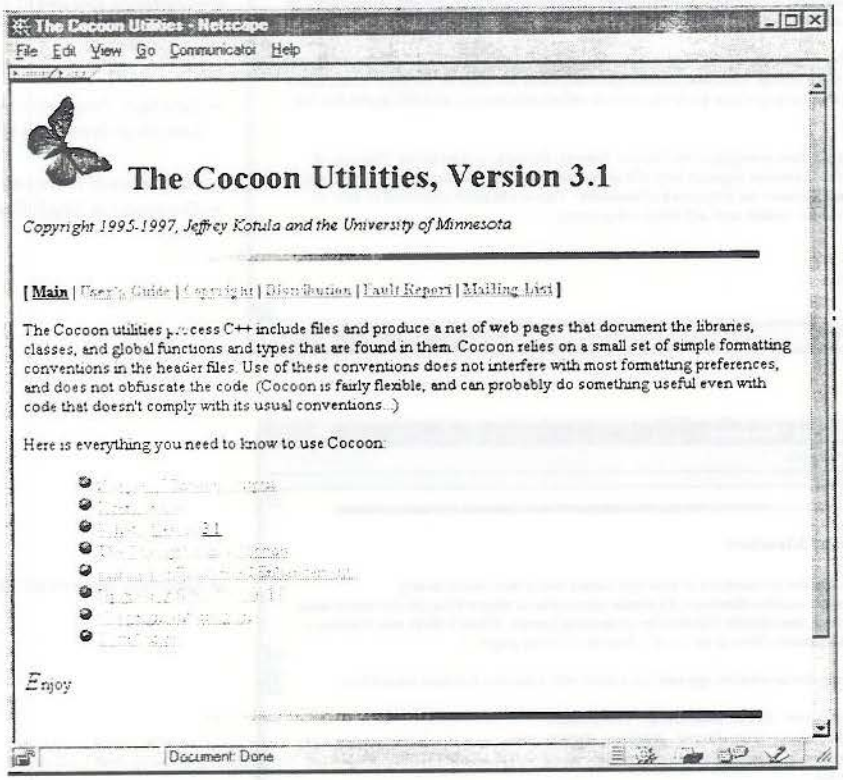
Webster, Bruce. *Pitfalls of Object-Oriented Development*. M & T Books, New York, 1995.

Whittle, Ben. "Models and Languages for Component Description and Reuse," *Software Engineering Notes*. Volume 20, Number 2, 1995.

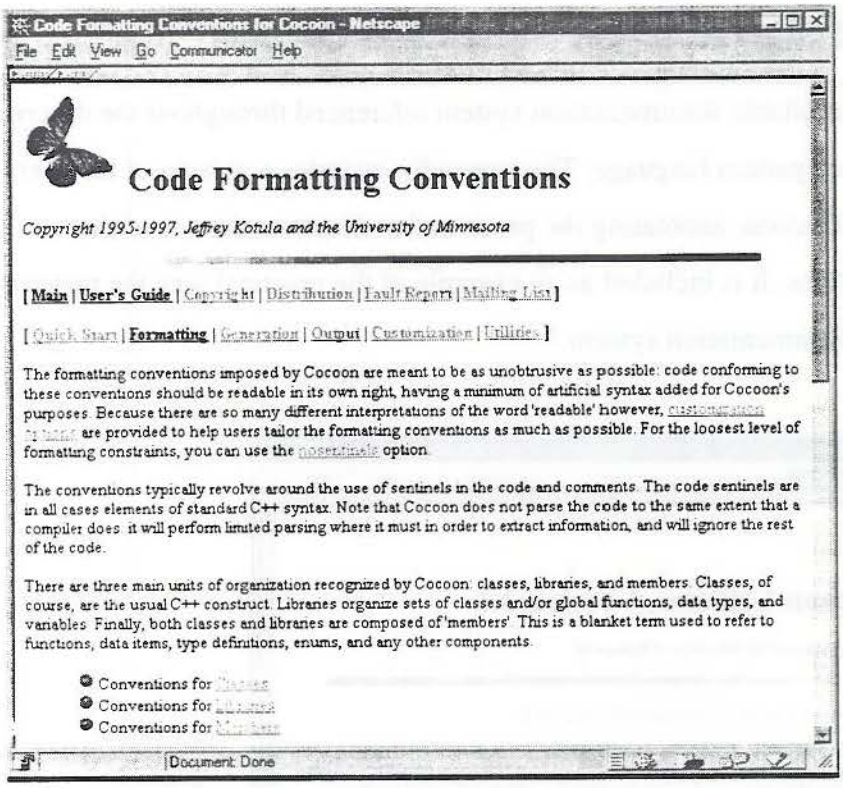
Wirfs-Brock, R. J., B. Wilkerson, L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, NJ, 1990.

Appendix A. Documentation Patterns in Cocoon

Cocoon is a freely available documentation system referenced throughout the description of the documentation pattern language. This appendix includes a portion of the user documentation for Cocoon, annotating the portions that illustrate the use of the documentation patterns. It is included as an example of the practical way the patterns are manifest in a real documentation system.



← *Embed Content in Source Code (4.3)*
← *Readable Embedded Documentation (9.4)*

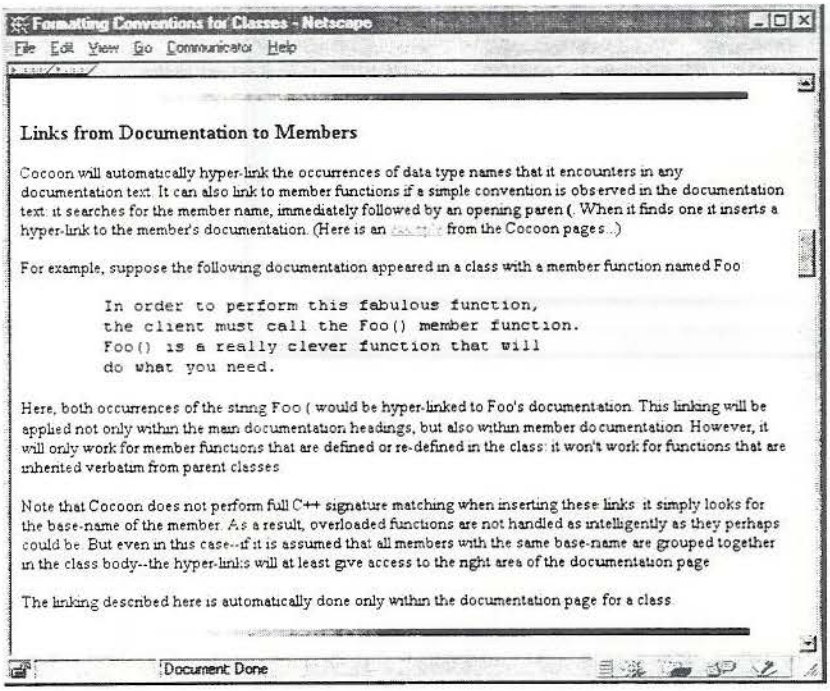


← *Centralize Document Design (4.6)*

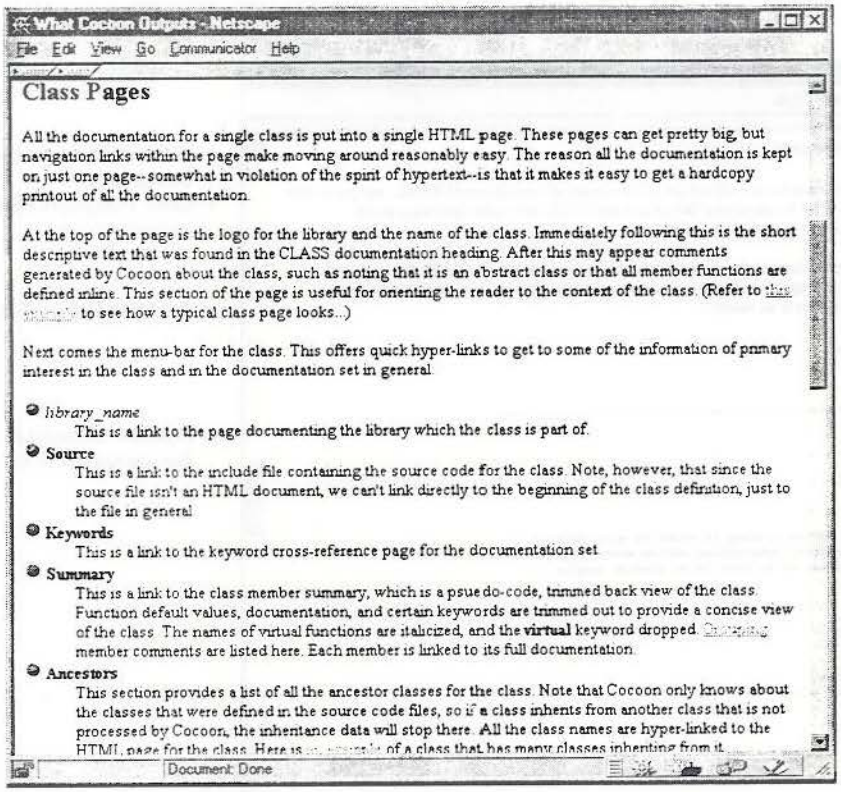
← *Leverage Programming Language Semantics (4.5)*

← *Hierarchical Libraries (6.2)*

← *Document in Small Chunks (9.2)*



← *Generate Hyperlinks (6.5)*



← *Icons, Headings and Colors Mark Nodes (8.3)*

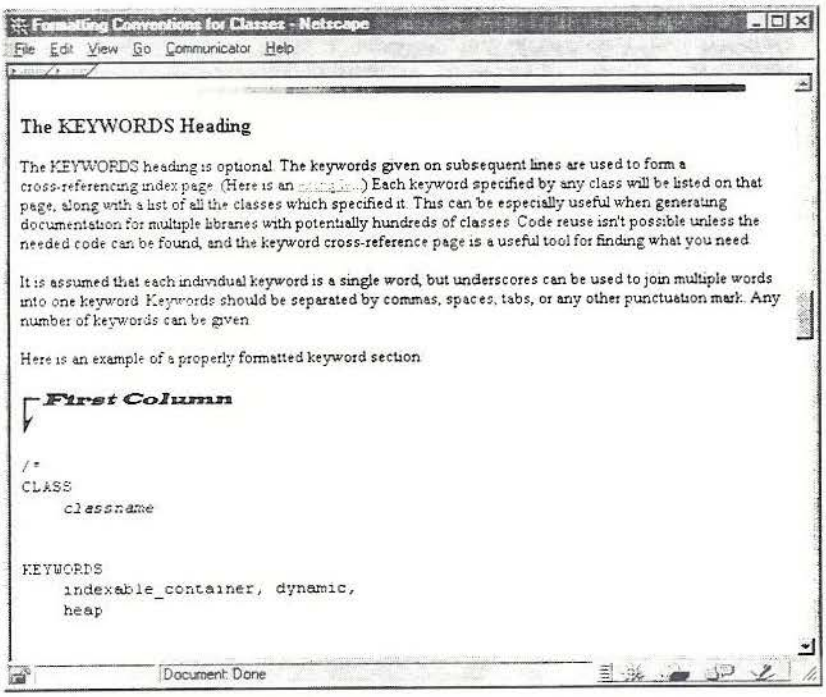
← *Backtracking (6.4)*

← *Keyword Cross-reference (7.1)*

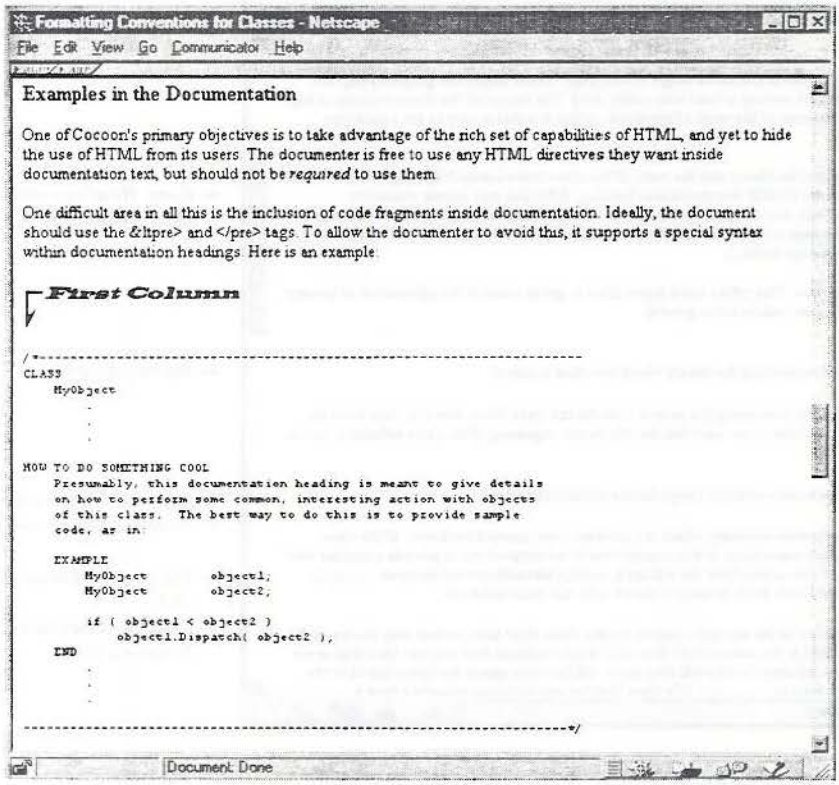
← *Generate Short Form (5.4)*

← *Typography-Encoded Details (8.5)*

← *Generate Inheritance Diagrams (5.6)*



← *Generate Keyword Cross-reference (7.5)*



← *Tutorial Information with Examples (5.1)*
← *Allow Typographic Control (9.3)*

← *Automate Tedious Tasks (4.4)*

← *Readable Embedded Documentation (9.4)*

Appendix B. The Scientific Method of Investigation

This section begins by describing the common, general methods of scientific investigation. These methods are then related to the usual, accepted research methodologies employed in the field of computer science, followed by discussions of the method used in this research, its limitations, and an explanation of why the method was chosen.

There are five common, primary methods of scientific investigation: experimentation, quasi-experimentation, logical reasoning, case studies, and surveys.

Classical experimentation is the best known and most time-honored method of scientific investigation. Experiments are conducted to prove or disprove a hypothesis. A researcher first forms a hypothesis, extrapolating, or deducing it from existing data or theories. The logical consequences of the hypothesis are examined and some *observable* experimental data are identified. An experiment is then designed to generate and gather data. The experiment is designed to isolate the effect under investigation into a dependent variable in the experimental program. If the gathered data is in agreement with the consequences predicted by the hypothesis, the hypothesis is considered verified by the experiment. Statistical methods can be employed in both experiment design and analysis of the resulting data, particularly to assess experimental error or confidence, and establish the correlation between variables. [Babbie89, Box+87]

Quasi-experimentation differs from classical experimentation with respect to the way in which the populations (test subjects) are selected. In a classical experiment, test subjects are assigned randomly to the test and control groups. (Control groups are used to establish that the observed effects of the experiment cannot be due to random chance.) Quasi-experimentation is often used for evaluation research, or program evaluation, which evaluates the effects of particular programs or practices. Such evaluations typically involve

working in the field, observing real-world situations rather than scenarios manufactured and controlled in laboratories. [Babbie89]

Logical reasoning is the process of building arguments to reach a conclusion. The researcher can start from existing conclusions created in other research, or establish a set of lemmas, or assumptions, that their reasoning is based upon. Deductive reasoning is used to proceed step by step from one conclusion to another that is logically dependent on it. Inductive reasoning is used to establish general truths from specific cases that can be proven. Logical reasoning is, historically, most applicable in such fields as mathematics where a rigorously defined notation, along with the rules of logic, are the primary method for extending theory. However, logical reasoning is also heavily relied upon to evaluate and interpret research results.

Case studies are one of the oldest methods of field research. Unlike experimental techniques that yield quantitative data, case studies usually provide qualitative observations, and often are used in the creation of theories. By direct observation of phenomena as it occurs naturally, a broad perspective is afforded the researcher. Case studies can be organized to study one individual situation, or a group of related or similar situations. [Babbie89, Yin94]

Survey research polls the sample population to discover prevailing opinions, beliefs, or attitudes held by people about the subject of investigation. Sample populations, consisting of a representative cross-section of the entire population in statistically significant numbers, are identified to receive the survey. A carefully designed, objective questionnaire is administered to the sample population and the results collected and examined. Common answers to related questions, or restatements of the same question, usually indicate a fundamental assessment or strong belief held by the respondent.[Babbie89]

These four basic methods—as well as their many variations—are used and combined throughout all the scientific disciplines. The next section will discuss the common ways these methods are applied in the computing sciences.

Commonly Accepted Methods of Scientific Research in Computer Science

The method of inquiry commonly used in the computing sciences varies according to the topic of study: basic computational theory and numerical analysis, artificial intelligence, and systems and software.

In the formation and study of basic computational theory, as well as for numerical analysis problems, logical or mathematical reasoning is the primary method of inquiry. Proofs are constructed through deductive as well as inductive methods to create a body of theory. Theories are often supplemented by the simulation, or actual execution, of programs which illustrate or model them.

In artificial intelligence, the most common research method is to build working prototypes of systems to demonstrate the feasibility of an approach to a problem. The problems range from expert systems for decision making, to game-playing, to artificial vision.

In the study of computer systems and software, a larger variety of scientific methods are employed. Software psychology is used to develop surveys, experiments, or quasi-experiments. Systems are prototyped to demonstrate their feasibility, sometimes supplemented by experimentation or quasi-experimentation. In some forms of system research, especially in evaluating algorithmic or system performance, researchers use mathematical techniques such as queuing theory, program traces, and simulation to analyze systems. Hybrid combinations of these methods are also common; for instance, using tracing data within a queuing model.

In the software engineering sub-field, methods of experimentation and experimental design are often hotly disputed. For example, in a recent issue of *IEEE Software* Beizer debated the long-established experimental validation used to support the cleanroom method, arguing against the validity of the experimentation done by other researchers [Beizer97]. The difficulties of experiment design, of conducting field experiments, and the high cost of experiments dealing with the development of large software systems may explain the prevalence of the use of surveys or logical reasoning in software engineering research.

The Method Used in This Research

The research for this thesis was carried out in five steps:

1. Current component documentation practices in the field were investigated. A wide variety of documentation systems were examined including printed manuals and books, online help systems, code browsers, the documentation facilities built into the Eiffel and Java languages, documentation utilities such as cxx2html, Doc++, and c2man, and literate programming systems.
2. Individual documentation patterns, and their contexts, were identified. Since patterns differ somewhat in their embodiment in different reference systems, this step was more qualitative than quantitative (see the discussion of limitations below).
3. The usefulness of the individual patterns was demonstrated. Two methods were used for this. The first method was to construct a logical argument describing why the pattern was effective in solving the problem it seemed to be addressing. A second method was applicable for many of the patterns: citing existing, relevant research results. For example, for the *Readability through Typography (8.5)* pattern, research by [Baecker+90] was cited to both justify the pattern and as a reference to more detailed explanations and empirical data. Many of the other patterns (*Full-Text Search (7.3)*, *Hypertext (4.2)*, *Directed and Exploratory Navigation (4.7)*, etc.) also referenced existing research.
4. Each pattern was validated by identifying at least three instances of its use within the reference systems. Each pattern must meet some definition to be considered a *valid*

part of a pattern language. In computer science pattern languages, this definition is rooted in the rationale that broad-based usage of a solution indicates its value and effectiveness; it is presumed that ineffective solutions would not be found multiple times in widely-used systems. While this assumption may be incorrect, it is certainly reasonable. The patterns community within computer science has agreed to a nominal definition of "broad-based usage": the use of a solution within three or more industrial systems. So to be considered valid, a pattern must be found in at least three industrial systems.

5. Finally, the patterns were combined into a full pattern language. This step involved developing a classification system for the patterns, and identifying the inter-connections between individual patterns. Through this step, the pattern language formed a framework describing not just individual solutions to individual problems, but an entire operational documentation system.

The research process employed for this thesis is analogous to that used to construct other computer science pattern languages, such as [Gamma+95].

Limitations of the Research Method Used

Pattern languages have two serious limitations: the subjectivity of pattern identification, and the variability of solution implementation.

Subjectivity is inherent in identifying patterns because researchers will inevitably work from different sets of observations about the reference systems, and therefore might easily identify different sets of commonalities between the systems. However, due to the nomothetic (non-exhaustive) nature of pattern languages, the differing results of multiple researchers do not negate their individual or collective value—they can legitimately stand alone and could also be combined to form a more complete pattern language.

The effects of this subjectivity can also be seen in the degree of acceptance of particular patterns from an established pattern language such as [Gamma+95]. For instance, while some of the patterns, such as Strategy, Factory Methods, and Proxy, are widely used and often cited, others such as Decorator are rarely used or discussed. There are two possible reasons for this discrepancy. First, the rarely-used pattern, while meeting the established criteria for broad-based use, may not have a broad enough context to allow its use in many systems. Second, other, more widely-used solutions may be better but not yet identified as patterns. Despite this phenomenon, the Gamma pattern language has, in general, met with success and acceptance within the software engineering community.

The second limitation of pattern languages is caused by the tension between the need for solution descriptions that are both generic and specific. Generic solution descriptions are more widely applicable, but more open to interpretation. Specific solution descriptions are easier to apply uniformly, but the contexts in which they apply are more limited. Either extreme will limit the effectiveness of the pattern, so the pattern author must choose a reasonable middle ground between genericity and specificity. In describing a particular solution, pattern authors tend to err on the side of genericity, placing a higher value on the breadth of applicability rather than the ease of application. This preference was adopted in the development of the documentation pattern language.

The variability in pattern implementations is seen in many pattern languages, for instance the software design patterns of [Gamma+95]. The Gamma patterns are specified very concretely through the use of source code examples, but nonetheless, each pattern has a set of suggested design modifications that can be used in certain contexts, leading to many legitimate variations in implementation. Such variations do not invalidate the pattern. The details that are varied do not substantively change the solution used.

Variations that do not substantially alter the pattern solution are also seen in the documentation pattern language. Consider, for example, the *Readability*

throughTypography (8.5) pattern. The example used to illustrate the solution technique used the following C++ function prototype:

```
void* MyClass::someFunction(const char* arg1, double** array, int size) const;
```

in contrast to a typographically enhanced version of the same text:

```
void*
MyClass::someFunction(
                        const char*     arg1,
                        double**        array,
                        int             size)
const;
```

The particular font and typographic effects used in the second version, or its grid layout could be modified substantially and still yield text that is much more readable than the original. For instance:

```
void* MyClass::  someFunction(
                 const char*     arg1,
                 double**        array,
                 int             size)
const;
```

The implementation details are less important than the general solution. The patterns provide *guidelines*, rather than requiring particular implementations.

It should be noted however, that the solutions to individual patterns *are* amenable to refinement through both experimentation (or quasi-experimentation) and surveys. [Babbie89] asserts that experimentation is best suited to areas where the problem being studied is relatively limited and well-defined. The pattern context establishes a relatively limited area of study, and the specified solution is usually defined enough to be testable, so experiments could be conducted to test the efficacy of variations of the solution. Thus, over time and through empirical methods, the pattern solutions can be made more specific and the implementation variability reduced. Surveys are best suited to inquiry when the unit of study is an individual person [Babbie89], such as in evaluating a software engineer's preference of one solution over another. So surveys can be used in conjunction with experimentation to evaluate pattern solutions.

Rationale for the Research Method

There are several reasons why the method of research used to create the documentation pattern language is justified:

1. First, this area of research (both pattern languages and component documentation) is very young, and is still in a broad exploratory stage. At this time, effort is probably best spent in discovering patterns and developing a broad framework for further research, as opposed to rigorous, traditional experimentation with individual patterns.
2. Given the difficulties faced by industry in creating high-quality, reliable software, focusing on discovering effective patterns and creating implementation tools to facilitate pattern usage is important.
3. While the actual implementation of individual patterns may differ, violations of the patterns often produce obvious and visible deficiencies in documentation. For instance, imagine an online documentation system that was not hypertext based as required by the *Hypertext (4.2)* pattern. Such a system would be immediately recognized as less effective than a corresponding system *with* hypertext.
4. The logical arguments that establish a pattern's usefulness are often powerful and convincing, primarily through their appeal to common sense. For example, consider the *Embed Content in Source Code (4.3)* or *Environment Independence (4.1)* patterns. The rationale used to support each of these patterns is relatively straightforward and unarguable. Disagreements with the rationale are most likely to arise from contradictions with other documentation methods, such as literate programming, that are established, but no more scientifically *proven*.
5. Experimentation, quasi-experimentation, case studies, and surveys can be conducted as the field matures. Further research into individual patterns should refine the solution descriptions and provide stronger validation. This thesis presents a first step to enable such research.

Additional References

[Babbie89] Earl Babbie. *The Practice of Social Research*. Wadsworth Publishing Company, California, 1989.

[Beizer97] Boris Beizer. "Cleanroom Process Model: A Critical Examination." *IEEE Software*, Volume 14, Number 2, 1997.

[Beveridge50] W. I. B. Beveridge. *The Art of Scientific Investigation*. Vintage Books, New York, 1950.

[Box+78] George E. P. Box, William G. Hunter, J. Stuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. John Wiley and Sons, New York, 1978.

[Glaser+67] Barney Glaser, Anselm Strauss. *The Discovery of Grounded Theory*. Aldine, Chicago, 1967.

[Yin94] Robert Yin. *Case Study Research: Design and Methods (Applied Social Research Methods, Volume 5)*. Sage Publications, 1994.