Linköping Studies in Science and Technology
Dissertation No. 758

Library Communication Among Programmers Worldwide

by

Erik Berglund

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2002

Printed in Sweden by UniTryck, 2002

Abstract

Programmers worldwide share components and jointly develop components on a global scale in contemporary software development. An important aspect of such library-based programming is the need for technical communication with regard to libraries – *library communication*. As part of their work, programmers must discover, study, and learn as well as debate problems and future development. In this sense, the electronic, networked media has fundamentally changed programming by providing new mechanisms for communication and global interaction through global networks such as the Internet. Today, the baseline for library communication is hypertext documentation. Improvements in quality, efficiency, cost and frustration of the programming activity can be expected by further developments in the electronic aspects of library communication.

This thesis addresses the use of the electronic networked medium in the activity of library communication and aims to discover design knowledge for communication tools and processes directed towards this particular area. A model of library communication is provided that describes interaction among programmer as webs of interrelated library communities. A discussion of electronic, networked tools and processes that match such a model is also provided. Furthermore, research results are provided from the design and industrial evaluation of electronic reference documentation for the Java domain. Surprisingly, the evaluation did not support individual adaptation (personalization). Furthermore, global library communication processes have been studied in relation to open-source documentation and user-related bug handling. Open-source documentation projects are still relatively uncommon even in open-source software projects. User-related bug handling does not address the passive behavior users have towards bugs. Finally, the adaptive authoring process in electronic reference documentation is addressed and found to provide limited support for expressing the electronic, networked dimensions of authoring requiring programming skill by technical writers.

Library communication is addressed here by providing engineering knowledge with regards to the construction of practical electronic, networked tools and processes in the area. Much of the work has been performed in relation to Java library communication and therefore the thesis has particular relevance

for the object-oriented programming domain. A practical contribution of the work is the DJavadoc tool that contributes to the development of reference documentation by providing adaptive Java reference documentation.

Much human ingenuity has gone into finding the ultimate Before. The current state of knowledge can be summarized as thus:

In the beginning, there was nothing, which exploded.

Other theories about the ultimate start involve gods creating the universe out of the ribs, entrails and testicles of their father. There are quite a lot of these. They are interesting, not for what they tell you about cosmology, but for what they say about people.

–Terry Prachet

Lords and Ladies
Victor Gollancz Ldt.
1992

Acknowledgement

First and foremost I would like to thank my supervisor Henrik Eriksson for his dedicated support in this research venture. I am grateful for his constant availability and detailed supervision. Furthermore, I am also grateful for his keen interest in my project and the fascination of technology that we share. I would also like to thank my secondary supervisors Sture Hägglund, Kjell Olhsson, and Kristian Sandahl for their participation.

Magnus Bång, fellow Ph.D. candidate and close friend, deserves thanks for contributing to this thesis. Our constant discussions and his philosophical skill have contributed much to my thinking. Thank you Magnus!

Michael Priestley at IBM Toronto Lab deserves special thanks for fruitful discussions and for co-authoring Paper III. Furthermore, I would like to thank Ulf Magnusson, Peder Gunnbäck, and Martin Rantzer at Ericsson and Douglas Kramer and the Javadoc Team at Sun Microsystems.

Continuing, I would like to thank my colleges at the Department of Information and Computer Science at Linköping University, particularly past and present members of HCS and ASLAB. Moreover, thank you Ivan Rankin and Pamela Vang for improving my English.

The SSF (Swedish Foundation for Strategic Research) have been my primary financial supporter through ECSEL (Excellence Center in Computer Science and Systems Engineering in Linköping). Furthermore, my work has been supported by the Swedish National Board for Industrial and Technical Development (Nutek) under grant no. 93-3233 and the Swedish Research Council for Engineering Science (TFR) under grant no. 95-186.

Finally, I must also pay tribute to the guardians of my non-scientific life. Thank you family and friends! I would particularly like to mention Aseel, my parents Båge and Margareta, and the boys. In closing, a particularly warm thought of gratitude goes to Hoffman (a dog without plans for world domination).

List of Papers

Papers Included in this Thesis

I Berglund E. (in press) *Designing Electronic Library Reference Documentation.* Accepted for publication March 2002, Journal of Software and Systems

II Berglund E. (submitted 2002) *Helping Users Live With Bugs*

III Berglund E. and Priestley M. (2001) *Open-Source Documentation: in search of user-driven, just-in-time writing* In Proceedings of SIGDOC 2001, October 21– 24, 2001 in Santa Fe, NM

IV Berglund E. (2000) *Writing for Adaptable Documentation* In Proceedings of IPCC/SIGDOC 2000, September 24 – 27, Cambridge, Massachusetts

V Berglund E. and Eriksson H. (2000) *Dynamic Software Component Documentation* In Proceedings of the Second Workshop on Learning Software Organizations, in conjunction with the Second International Conference on Product Focused software Process Improvement June 20 2000, Oulu, Finland

VI Berglund E and Eriksson H (1998) *Intermediate Knowledge trough Conceptual Source-Code Organization* In Proceedings of the 10:th International Conference on Software Engineering & Knowledge Engineering, June 18-20 San Francisco Bay CA USA, pp 112 – 115

Other Publications by the Author

Eriksson H., Berglund E., Nevalainen P. (2002) *Using Knowledge Engineering Support for a Java Documentation Viewer* In Proceedings of The 14:th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), July 15-19, Ischia, ITALY

Granlund R., Berglund, E. and Eriksson H. (2000) *Designing web-based simulation for learning* in Journal Future Generation Computer Systems, special issue with the best papers from the International Conference on Web-Based Modelling and Simulation 1998, Elsevier.

Berglund E (1999) *Use-Oriented Documentation in Software Development* Linköping Studies in Science and Technology, Thesis no. 790, School of Engineering at Linköping University

Berglund E and Eriksson H (1998) *Distributed Interactive Simulation for Group-Distance Exercises on the Web* in Proceedings of the 1998 International Conference on Web-based Modelling & Simulation, January 11-14 1998 San Diego CA USA, pp 91 – 95

Contents

Abstract i

Acknowledgement iv

List of Papers vi

1 Introduction 1
- 1.1 Research Question . . . 3
- 1.2 Library-Based Programming . . . 5
- 1.3 Library Communication . . . 9
  - 1.3.1 Program Understanding . . . 10
  - 1.3.2 Language . . . 10
- 1.4 Literate Programming . . . 10
- 1.5 Electronic, Networked Medium . . . 11
- 1.6 Contributions . . . 12
- 1.7 Thesis Overview . . . 13

2 Research Method 15
- 2.1 Methods in Library Communication . . . 15
  - 2.1.1 Industry Laboratory . . . 16
  - 2.1.2 Iterative and Explorative Development . . . 17
  - 2.1.3 Subjective and Objective Data . . . 18
  - 2.1.4 Summarizing: Library Communication Research . . . 19
- 2.2 Methods Applied . . . 19
  - 2.2.1 Explorative and Iterative Development . . . 19
  - 2.2.2 Data Collection . . . 20
  - 2.2.3 Standard Tools or Bleeding Edge . . . 20
- 2.3 Future Considerations . . . 21
  - 2.3.1 Open-Source Explorative Development . . . 21
  - 2.3.2 Individual Data . . . 22

3 Library Communication **23**
3.1 Libraries and Programming 23
3.1.1 What are Libraries? 23
3.1.2 Who is the Library User? 26
3.1.3 What is Library-Based Programming? 27
3.1.4 Library UI 28
3.2 A Model of Library Communication 30
3.2.1 Current Model 30
3.2.2 Improving the Current Model 32
3.3 Conclusive Remarks 33
3.3.1 The standard design 33
3.3.2 The good examples 34

4 Designing for Library Communication **37**
4.1 Design Criteria on Library Communication 37
4.1.1 Source Code and Documentation Interruption 38
4.1.2 Publication Platform and Programming Environment Interruption 39
4.1.3 Feedback and Documentation Interruption 40
4.2 `new Javadoc()` 41
4.2.1 Source Code and Documentation Interruptions 41
4.2.2 Publication Platform and Programming Environment Interruption 42
4.2.3 Feedback and Documentation Interruptions 43
4.3 Programming Languages and Communication 44

5 Related Work **47**
5.1 Analysis of Library-Based Programming 47
5.2 Formal Approaches to Library Communication 48
5.3 Component Browsing 48
5.4 Exploring Functionality 49
5.5 Brief History of Electronic Reference Documentation 50
5.6 Summary of Related Work 51

6 Discussion **53**
6.1 Community Software Development 53
6.2 Understanding Valuable Adaptation 54
6.3 User-Driven Communication 55
6.4 Passive Reading 56
6.5 Summary of Discussion 56

7 Conclusion **59**

8 Summaries of the Papers **63**
8.1 Paper I: Designing Electronic Library Reference Documentation 63
8.2 Paper II: Helping Users Live With Bugs 64
8.3 Paper III: Open-Source Documentation: in search of user-driven, just-in-time writing 64
8.4 Paper IV: Writing for Adaptable Documentation 65
8.5 Paper V: Dynamic Software Component Documentation 65
8.6 Paper VI: Intermediate Knowledge through Conceptual Source-Code Organization 66

References **67**

Paper I:
Designing Electronic Library Reference Documentation **79**

Paper II:
Helping Users Live with Bugs **97**

Paper III:
Open-Source Documentation: in search of user-driven, just-in-time writing **112**

Paper IV:
Writing for Adaptable Documentation **132**

Paper V:
Dynamic Software Component Documentation **145**

Paper VI:
Intermediate Knowledge trough Conceptual Source-Code Organization **157**

A Appendix A:
Javadoc **169**

B Appendix B:
Dynamic Javadoc (DJavadoc) **179**

Chapter 1

Introduction

The *global sharing* of *software components* collected in *libraries* is the basis of contemporary software development, visible for instance in object-oriented programming languages. Global sharing has not only added to the programmer's toolbox; it has also introduced changes to the software-development process, perhaps representing a general transition from language-based development to library-based development. Traditionally libraries are viewed as managed collections of software assets, mainly reusable components (Atkinson and Mili 1999). However, in contemporary programming the library becomes the language for programmers, a foundation of programming based on thousands of components. For instance, Java programming is programming based on the Java standard development kit (SDK) and Visual C++ programming is programming based on the Microsoft foundation classes (MFC, Prosise 1999). These libraries are not reused in the traditional sense where the libraries are first selected, then adapted, and finally integrated into development and where retrieval is a major issue (Kruger 1992, Basili et al. 1996, Frakes and Fox 1995, Mili et al. 1995, Mili et al 1999). Instead the libraries become a programming language that consists of thousands of constructs that are less stable, less formally specified, and subject to more rapid growth and change compared to languages.

An important change that occurs in this transition from language to library is an increased need for technical communication in relation to libraries, that is, *library communication*. In library-based development, programmers handle large, complex, and evolving sets of programming constructs which it is neither possible nor relevant to learn or memorize. Rosson (1996) states that programmers spend considerable time communicating with others in their organization. Library-based programming leads to communication expanding and including a community outside the team or the organization, through library reference documentation, mailing lists, FAQs, and other channels of communication. Programmers communicate within technical communities that form

around the libraries they use.

Library communication, partly due to global sharing, places new requirements on technical communication compared to traditional language communication. In language-based development, resources are in some sense limited, stable, and non-evolving and the need for communication can be summarized as *tutorial* (reading to learn). However, in library-based programming, resources change, grow, and multiply even during relatively short periods which has become apparent from the development of Java core libraries first publicly release in 1995 during which time it has undergone 5 versions and grown from 3 to 135 libraries (see Table 1.1 on page 6). Libraries change their content, the grow, new libraries appear, and the development of some libraries is discontinued. As a result of the evolution of libraries, development and use of libraries sometimes become parallel or overlapping activities. Library specifications are released early to the public, sometimes even before an implementation exists, and the use of the library may therefore precede, run in parallel, overlap, or await the development of libraries. As an example, Sun Microsystems has on several occasions released Java library specifications without existing implementations or with implementations limited to specific operating systems (e.g. Java Speech API, Java TV API). Tutorial and *reference* communication becomes continuous communication needs. Furthermore, because libraries change it becomes necessary to *debate* libraries, that is to discuss issues concerning current implementation and the future of development. Bugs, features, design, and implementation issues become relevant to the library community as a whole and not just the core development team.

Communication becomes a central activity in library-based programming. Library-based programmers may even spend more time reading documentation and communicating within the technical community than they do actual coding with regards to time spent using library functionality. The purpose of the communication is to increase the speed and quality of development and decrease the cost and frustration. Efficiency in presentation and global distribution of evolving content are relevant aspects of this communication. Commonly, library communication is most commonly system-oriented, that is designed as encyclopedic descriptions of systems. As such, library communication provides little services facilitating the execution of programmers' communication tasks. Good quality library communication requires information design (e.g. Jacobson 1999, Rosenfeld and Morville 1998). The usability of communication tools and processes is also dependent on how well they correspond to readers' mental models of the communication (Norman 1990). Hence, information design needs to be based on knowledge about library-based programming and the programmer's mental model.

In this work, I have studied the design of communication tools and processes in library-based programming (sometimes referred to as literate programming see Section 1.4 [Knuth 1991]). In particular I have worked towards a use-oriented design of automated or user-driven communication processes in the

electronic, networked medium for this domain. My work has also resulted in a model of library-based programming from the perspective of technical communication. Though some of the papers presented in this thesis are written as general papers, they are clearly and particularly relevant for library communication (see Papers III and II).

Much of this work has been centered on the Java programming language domain (JAVA, Campione and Walrath 1998) and the Javadoc tool that provides automatic generation of reference documentation from Java source files (JAVADOC, Kramer 1999, Friendly 1995). I have studied the design of Javadoc documentation, for instance, requirements for individual adaptation and redesign of Java library reference documentation. The Java language domain is relevant to library communication because it is focused on the construction of libraries as a means of sharing software components. Moreover, it is currently one of the largest and most frequently used programming languages. Javadoc is also relevant to library communication since it represents state of the art in automated documentation generation and also produces state of the art online reference documentation.

Part of this work has been conducted though the development and evaluation of a practical documentation tool called Dynamic Javadoc (DJavadoc). DJavadoc produces adaptive documentation for Java source files by extending Javadoc. The interested reader can try DJavadoc at `http://www.ida.liu.se/~eribe/djavadoc` using Microsoft Internet Explorer (version 4 or higher). DJavadoc should be viewed as a practical result of my research.

1.1 Research Question

The overall research question addressed by this thesis can thus be summarized as:

> *How do we improve communication in library-based programming using the electronic, networked medium?*

The goal is thus to make it easier to develop programs using libraries and to develop libraries for global communities. In this context I have focused on the issue of communication within the library community (among developers and users concerning the use and development of libraries). Communication is in my opinion a highly relevant research issue in library-based programming that has received little attention in the past. Furthermore, I limit my work to the electronic, networked medium in which new possibilities appear due to the recent changes in electronic text and the software development activity brought about by the popularization of Internet.

We can further decompose the overall question based on a communication process division:

1. *How do we improve content production?* – Users and developers produce communication content, such as documentation. Concrete examples of research questions include: How do we maximize automation from on source files (from both developer and user)?

2. *How do we improve content publication?* – Produced content must also be published to its intended audience. In this area it is relevant, for instance, to ask how do we identify relevant receivers in a global community?

3. *How do we improve content acceptance?* – Published content must also be accepted by users, by which I mean that the information is actually used in some way. An example of a research question is: How do we determine which content is being used?

   (a) *How do we improve content to source transfer?* – A particular relevant sub-question that concerns the integration of communication content into library or project source files.

4. *How do we improve content debate?* – In the library communication process, discussion and feedback (debate) is often a cornerstone. An example of a research questions is: How do we automate routine debate content?

   (a) *How do we improve content error handling?* – An especially relevant sub-question because errors are unexpected and may be highly frustrating and costly.

The decomposition of the overall question provides a large research area that is beyond the scope of this thesis. Therefore I have identified a number of concrete sub-questions of the overall question and focused my work on them:

- *How should electronic reference documentation be designed?* – Reference documentation is an important library communication tool. What can the electronic, networked medium provide for reference documentation? What are programmers' requirements on electronic reference documentation? Mainly relevant to questions 2 and 3. (Papers I, VI, V, and Appendix B)

- *How should adaptation be used in electronic reference documentation*? – It is plausible that adaptation of reference documentation can provide improvements in library communication. At what level of individuality is adaptation relevant for programmers? Is it relevant on an individual level, an application-category level, or a general level? Relevant mainly to questions 2 and 3. (Paper I)

- *How should open-source documentation processes work?* – Open-source development is an electronic, networked development process currently exploited in the development of libraries and software. How do open-source documentation processes work? What type of open-source documentation projects exist today? How do open-source software projects treat documentation? How does the state of the art in open-source library communication match the requirements of open-source development? Relevant mainly to questions 1 and 4. (Paper III)

- *How should user-related bug handling be designed?* – Integration of users in the bug handling process is one element of the electronic, networked medium currently exploited in library communication. What is the state of the art in user-related bug handling? What are users' requirements for the bug-handing process and the distribution of bug knowledge? How well does the state of the art match the requirements? Relevant to question 4a. (Paper II)

- *How can adaptive authoring in reference documentation be supported?* – Library communication requires an authoring process. What are the elements of authoring in electronic reference documentation? How does writers' work with real-time redesign as a literary quality? What support do common web languages, such as HTML, have for adaptive authoring? Relevant to question 1. (Paper IV)

In closing, research questions that address the improvement of library communication also address the issue of understanding the library communication activity. This thesis therefore also address the question: *What is the library communication activity?* (Chapters 3 and 4)

1.2 Library-Based Programming

Library-based programming can be viewed as programming based on an evolving programming language with continuous creation of constructs that are not organized or specified by one organization. For a long time, programmers have shared software components on a global scale. Fortran II, released in 1958, enabled the use of separately compiled subroutines (Carver 1969). Today however, global sharing is not just a possibility but a foundation of programming. Languages such as Java are highly integrated with the library concept (called application programming interfaces or APIs in the Java world) and the development and sharing of libraries has increased dramatically over the last decade because of the popularization of Internet. The development of the Java language core library, Java SDK, points to this fact, see Table 1.1. Another example is the open-source language, Python, which in November 2001 had 252 global modules with over 2,200 functions (PYRD).

Table 1.1: Development of the Java standard development kit (Java SDK) so far (JAVA).

SDK version	Packages	Classes	Ref. Doc. (Mbytes)
1.0 (1995)	3	70	3
1.1 (1997)	22	600	8
1.2 (1998)	59	1,800	80
1.3 (2000)	76	2,150	97
1.4 (2001)	135	2,700	131

Library-based programming can also be viewed as one behavioral strategy that programmers apply to produce program executables. It is based on collections of externally built abstract data types (ADTs) that were not part of the programming language to begin with. Bruce (1996) considers ADTs to be perhaps the most important development of programming languages. However, in "No Silver Bullet" Brooks (1987) points out that much of the complexity of software comes from conformance to other software, that is, other ADTs.

As a programming activity, library-based programming and language-based programming are different, illustrated by Table 1.2. Of course, few programmers or programming projects are completely library-based.

Table 1.2: Some difference in behavior between language-based programmers and library-based programmers.

Language-based programmers	Library-based programmers
Constructs ADTs	Searches for ADTs
Implements algorithms	Uses implemented algorithms
Knows the language	Knows where to locate information
Defines the structure of programs	Defines the structure of programs
Builds his or her own ADTs	Shares ADTs
Builds new ADTs	Requests new ADTs
Programs by modelling	Programs by finding models

Changes

In library-based programming, many of the premises of programming change compared to language based programming:

- *Changing platform* – Language-based programming is based on a stable technology: the programming language. Library-based development, however, takes place within a technological environment that continuously evolves and expands, see Table 1.1.

- *Large amounts of constructs* – Language-based programming is based on relatively limited amounts of constructs. Library-based development, however, is based on very large sets of components that also continue to grow. As a comparison the Java language has about 50 reserved words but the Java core class library (Java SDK) has over 2,700 classes, see Table 1.1.

- *Community activity* – Language-based development can be regarded as process in which the development team is a well-defined unit (often from the same organization and sometimes including customers). Library-based development, on the other hand, is a community process in which independent groups without a common goal jointly develop the platforms which particular applications build upon and thereby also determine standards and structures through de-facto processes. In library-based development projects intertwine through the libraries they reuse and develop. Development is performed both by using available libraries and by participating in the development of libraries.

- *Less formal* – Language-based development is highly formal with grammatical definitions of languages. Library-based development is based on a foundation of loosely defined relations among components that can include many implicit structures. One example is the abstract window toolkit (AWT) in Java which requires `Components` to be placed in `Containers` to become visible even though this relation is not explicitly stated in library specification (i.e. a implicit library assumption).

As a result of the popularization of the Internet and as a result of the changes in programming behavior that global sharing of has brought about we today find an increased:

- *Openness* – Across organizations concerning technology and technological direction.

- *Standardization* – Global cooperation leads to (de facto) standards. Furthermore, an increased use of distributed, joint platforms of development lead to more similarities in development projects.

- *Publication* – Elimination of production and distribution costs makes publication of libraries and related documents and GUIs easier.

- *Communications flow* – Library specifications, debate, and publications.

The recent trend in open-source development illustrates this fact. Successful global development projects have arisen due to joint, global sharing of software, such as the Linux platform (today a common platform [LINUXO, Torvalds 1999]) and the Apache web server (currently the most common web server with

more than half the market [AP, NETCRAFT]). Another relevant example is the open-source development platform SourceForge, providing free tool support for open-source projects, which in October 2001 had 28,000 projects and 270,000 registered users (SF).

Reuse

A relevant question is whether or not library-based programming and global sharing constitute software reuse. In my mind, library-based programming is software reuse. However, I use the term global sharing instead to place focus on issue of communication among programmers that software reuse imply. Software reuse is most commonly defined as the process of creating software from existing software rather than building it from scratch (Kruger 1992, Basili et al. 1996, Frakes and Fox 1995, Mili et al. 1995). However, the term reuse also carries with it the idea that reuse is an engineering practice where reusable components are developed as part of application development through generalization. Beck (2000) argues against this, because generalization constitutes work spent on possible future benefits that may never materialize. Frakes and Fox (1995), however, showed that programmers like reuse as a basis for programming. Basili et al. (1996) showed significant benefits from reuse in software development in terms of reduced defect density and rework as well as increased productivity. However, the study was performed on 8 smaller student projects and does not necessarily represent industrial projects. Glass (1998) argues that reuse is not so commonplace as one may think and that in reality few components that are reused from collections such as the Java SDK. This is of course an empirical research questions that Glass does not answer. However, reuse is definitively an issues considered relevant to software engineering. For instance, Mili et al. (1999) state that software development cannot possibly become an engineering discipline so long as it has not perfected a technology for developing products from reusable assets in a routine manner on an industrial scale.

However, my view of library-based programming differs somewhat from the traditional view of software reuse. Software reuse is commonly described as programming using existing software components. Here library-based programming and therefore also reuse is characterized as a community activity where the use of libraries and the development of libraries are not clearly separated activities. (This community perspective does not require but includes open-source approaches to development.) Public beta release has become commonplace as well as to involve users in the development or libraries though beta testing, mailing lists, discussion forums, and features requests. In my mind, users are therefore participating in development rather than simply locating, adapting and integrating stable reusable components.

Component

Furthermore, I have chosen not to use the term "component-based programming" which could have been suitable in this context. In the software engineering community the term component is used to denote binary, independent software product with clear and defined purposes that can be directly deployed in development (Szyperski 1999, Brown and Wallnau 1998). Library-based programming is closer to a process using parts of more open and more general components. Also, using the library metaphor is relevant because development to a large extent requires going to the library, asking around for the right information, collecting it, studying it, applying it, and adding new information to a global library.

1.3 Library Communication

In the literature on software engineering and programming tools, communication within a technical community (such as reading reference documentation) is an aspect of programming that is often omitted or treated lightly (Pressman 2000, Schach 1997, Reiss 1996, van Vilet 1993, Sommerville 1989, Brookshear 1994). An underlying reason for overlooking such communication may be that programming traditionally involved limited sets of programming-language constructs that could be learnt by programmers. Currently, however, programmers base development on large collections of software component libraries.

I use the expression communication in library-based programming to denote activities taken by professionals in the act of transferring knowledge and code as part of programming using globally shared libraries. Mainly I refer to communication with the external technical environment and not so much within the project team. Such a definition also includes the actual transfer of source code, via humans, among programs (where humans initiate transfer but not necessarily perform the transfer). Because software is information itself, the transfer task can reach all they way to the application or all the way back to the library product in the electronic, networked medium.

Typical communication activities include writing documentation and example code, reading documentation and example code, participating in mailing lists, extracting code and including it in project source files, reporting bugs, requesting features, reading FAQs, searching for knowledge, locating people with skill, and explaining to others. Most commonly communication with the external technical community is performed in writing, but also by copying and pasting from web pages directly to source files. I also consider using code-completion functionality in development environment as acts of communication (the environment generates a context-specific documentation from which it enables code transfer by direct manipulation).

1.3.1 Program Understanding

Library communication includes program understanding but not commonly on such a level of detail that is commonly addressed in relation to program understanding and software comprehension. Program understanding is the issue of making sense of programs (Birgerstaff et al. 1994, Woods and Yang 1996, Bohem-Davis 1988, Rugaber 1995). Often these issues are relevant in relation to software maintenance, and include work in reverse engineering (see, for instance, Tilley et al. 1992).

1.3.2 Language

A relevant aspect of library communication is the existences of two types of languages: *natural language* and *code language*. Both are used to support the knowledge transfer process with the difference that code can be directly used in coding but may also be less expressive compared to natural language. Another relevant aspect of library communication is that the vast majority of participants are programmers (both developers of products and the users of these products). The difference in technical competence between the developer and the user is not so distinct as in other areas.

1.4 Literate Programming

Literate programming, in a restrained view, is the combination of writing descriptions and writing code in the same process – an essayist view of programming. In an open view, literate programming is the aim for a programming process that supports the communication tasks at hand in relation to programming in which case my work can be regarded as work in literate programming. The open view does not necessarily require a changed coding activity, but rather a focus on support for the construction of efficient communication tools and processes. The term literate programming has been around since the 1980s and is accredited to Donald Knuth (Knuth 1984, Knuth 1991, Ramsey 1994, Østerbye 1995). Knuth's vision for programming was that programs should be considered works of literature for humans. Literate programming is a view of programming where the purpose of a program is to communicate to other humans what the author wants the computer to do (Knuth 1991). In this sense, literate programming addresses program readability. Ramsey regards literate programming tools as tools that allow parts of programs to be organized in any order and from which both documentation and code can be extracted (Ramsey 1994). Programming and documentation should be mixed into a literary activity where descriptions and code mix naturally (Knuth 1991). Knuth also implemented a system called WEB, which initially combined Pascal programming and TeX writing (Knuth and Silvio 1994). A number of literate programming systems have followed of which Javadoc can be considered the

most commonly known system (Friendly 1995, Kramer 1999, Østerbye 1995, Normark et al. 2000, Johnson and Johnson 1997).

1.5 Electronic, Networked Medium

The electronic, networked medium gives rise to new possibilities in communication. Electronic text has a history of less than half a century. According to Dillon (1994), electronic text arrived as late as in the 1980s and is still evolving. Screen resolution is still a major issue of electronic reading (Dillon 1994, Kahn and Lenk 1998). However, electronic text is not just text presented on the screen (Hackos 1997).

Compared with the art of writing, which is over 5,000 years old, and the art of bookmaking, which has been around since Gutenberg invention of the printing press in the 15^{th} century, electronic text is still in its infancy. As a result, I expect electronic text will continue evolve for quite a long time. However, currently electronic text provides new possibilities that include:

- *Expression* – Addition of time, interactivity, action, global connectivity, meta information, document relations, and so forth as a means of expression in text. Kahn and Lenk state that the most exciting characteristic of type on the screen is the added dimensions of time and focus on the resulting ability of electronic text to move (Kahn and Lenke 1998). Moving text is also related to exploration of animation as part of expression (Zellweger 2000, Lewis and Weyers 1999, Wong 1996). Adaptability in electronic media has also been addressed (Brusilovski and Vassileva 1996, Brusilovski 1996, Kantorowitz and Sudarsky 1989 Rutledge et al. 1997, White 1998, ADH&H).

- *Cross-referencing* – Unlimited cross-referencing within and among documents with near instantaneous access. During most of the latest decade the electronic text area has been focused on hypertext (Bush 1945, Nelson 1987, Bolter 1991, Dillon 1994, Nielsen 1995). On his homepage Nelson describes hypertext as a concept that is still misunderstood and misused (Nelson 2001). Though hyper linking is relevant, it may be more important to investigate other aspects of the electronic, networked medium (besides non-linearity).

- *Standards* – Development of a commonly used communication infrastructure that new solutions can be based on. In the web area, the World Wide Web Consortium (W3C) is a central organization in the global standard process of web languages.

- *Global aspects* – Global, joint cooperative editing of text and global development of resources. The open-source development paradigm that has

drastically evolved during the past half decade is one important example of the global aspects of electronic text (OSIWS, SF, DiBona et al. 1999, Raymond 1999a).

Based on these possibilities it becomes possible to construct, for instance:

- *Adaptation* – Changing communication in relation to user models.
- *Evolution* – Released material that include new content during the life cycle of a topic.
- *Reading books* – Books that search for information and include new information, that is, reading agents inside books.
- *Annotation-based discussion* – Global annotation as a means of discussion, debating directly in globally distributed documents.
- *Live communication* – Global scale, people-to-people communication.
- *Task integration* – Integration of actions in texts, for instance to issue commands to programs from text.

These new possibilities give rise to new types of services in global communication. Generally, the electronic, networked medium (e.g. online text) provides new possibilities requiring new authoring and design techniques in the field of technical communication (Hackos 1997, Baker 1997, Smart 1994). It is also likely that new communication patterns will emerge. For library communication, it is relevant to explore the new design spaces that have appeared to facilitate programming. However, it is important to remember that the electronic, networked medium is itself also still evolving.

1.6 Contributions

This thesis contributes to the software development process through an analysis of communication in programming and the construction of communication process and tools in the electronic, networked medium. Specifically, the individual scientific contributions are the following:

- A model of library communication, contributing to the understanding of the communication process in relation to programming based on globally shared libraries. Design criteria that such a model leads to are also provided. (Chapters 3 and 4.)
- An empirical analysis of user needs in library communication, studying Javadoc users in the industry laboratory, providing requirements on the design of electronic library reference documentation and uncovering deficiencies in the Javadoc design. In particular, I provide an evaluation of

individual adaptation in Javadoc documentation examining the adaptive dimensions of the electronic, networked medium. (Paper I)

- An analysis of open-source development of documentation, providing a framework for open-source development in technical communication, discussing the use of the global and evolving aspects of electronic, networked media. (Paper III)
- An analysis of the requirements of passive, global bug knowledge sharing, providing an architecture for use-oriented design of error communication throughout the lifecycle of products and discussing global aspects of the electronic, networked media. (Paper II)
- Evaluation of electronic authoring, studying the support for expression of electronic concepts on an authoring level and, in particular, the support in client-side web technologies. (Paper IV)

The DJavadoc system is also a practical contribution of my research, directly usable in programming projects, which have also been tested in a real-work situation as part of this thesis work. DJavdoc provides a concrete design alternative to Javadoc and traditional online reference documentation and also provides a testing platform for the evaluation of individual adaptation in the electronic, networked media. (Appendix B and Papers I, V, and VI)

1.7 Thesis Overview

This thesis is organized in the following way: Chapter 2 discusses the research methods in relation to research in library communication and the work presented in the thesis. Focus is placed on explorative, iterative systems development based on evaluation in the industry laboratory. Chapter 3 provides a model of library communication. Chapter 4 provides a discussion on design considerations called for by the model in chapter 3. Chapter 5 discusses work related to the research presented in this thesis, discovering a lack of studies of programmer behaviour in relation to library-based programming and a lack of evaluation of existing communication tools in this specific area. Chapter 6 provides a discussion of issues of relevance to this work, addressing issues such as individual adaptation, user-driven communication, and community software development. The conclusions of the thesis are provided in chapter 7. In chapter 8, summarises the six papers included in the thesis after which the papers are provided. Finally Appendix A describes background on technologies such as Javadoc and DHTML that are needed to understand the DJavadoc system described in Appendix B.

Chapter 2

Research Method

Finding and developing knowledge is a difficult task that requires scientific methods that deliver reproducible, reliable, and valid results. Many practical considerations must also be taken into consideration, because they affect the design of research experiments. In this chapter, I will discuss methodological issues of consideration in relation to software-component library communication and thereby propose methods for research in library communication. I will also describe what I have specifically done from a methodological perspective and finally address what I would have done differently given the knowledge and experience I have today.

There are two general points that currently set the stage for research in library communication:

- *Early phase of research* – In recent years much has changed both concerning the premises of programming and the possibilities of the communication (the social behavior of programmers and the possibilities of the electronic, networked medium).
- *Applied science* – The study of library communication is an applied science striving towards improved programming tools and programming processes rather than the discovery of general knowledge. General knowledge about programming behavior in relation to libraries however, is needed to accomplish applied scientific results.

2.1 Methods in Library Communication

Research in library communication should aim at uncovering how the electronic, networked media can be used to provide adequate tool and process support for library-based programming. This research should also focus on the practitioner,

uncovering the requirements of both the user of libraries and the producer of the libraries. In particular, it is relevant to address user-oriented design of library communication in contrast to the state of the art approaches, which are system-oriented (see Section 3).

2.1.1 Industry Laboratory

Laboratory studies have often failed to predict real-world usability. However, it is the lack of the correct context rather than laboratory experimentation per se that is responsible for this failure (Dillon 1994). Brooks (1980) argues that generalizing between student programmers and experience programmers is not justifiable. Therefore, in Computer Science and, in particular, for human-related areas such as library communication, relevance in research requires experimentation in real-work situations with experienced subjects. This is often discussed in terms of performing research in the *industry-as-laboratory* (Yin 1994, Basili 1996, Potts 1993, Glass 1994). At first glance the industry laboratory approach requires evaluation of academic work in real-work situations. Equally important however, is acquiring empirical problem definitions from industry. Potts argues that what researchers think are major practical problems often have little relevance to professionals, whereas neglected problems often turn out to be important (Potts 1993). Though Potts is more focused on empirical experimentation and analysis than technology development, the same principles are likely to apply to architectural-oriented research ventures. Industry-related problem definition also comes into focus in Davis' (1994) article on "Fifteen Principles of Software Engineering". These principles are proposed as (temporary) laws of physics for software engineering. Glass (1994) advocates the use of evaluation in the engineering model of research (where the value of models is also tested). Tichy et al. (1995) showed that research papers in Computer Science to a large degree failed to provide empirical evaluation.

A difficult part of research in the industry laboratory is gaining and maintaining access to data collection (Gummesson 1991). Industry may be reluctant to provide resources and to expose internal details, such as source code or processes. Many practical constraints may be placed on the controlled experimentation, requiring the researcher to bargain with the rigorous design of experiments. The publication of results may be in question and time-consuming negotiation of research contracts may be required to reach a settlement that both camps can accept. Even though contracts are developed, changes in staff and priorities for companies may also disrupt data collection. Nonetheless, the industry laboratory is essential to the study of library communication because the purpose of this applied science is to further support the professional programming activity in relation to libraries.

It is also important to remember that other communities may provide access to members of the programming profession. An increasingly relevant alternative is open-source communities, often consisting of professionals but based

on independent, cross-organizational groups (DiBona et al. 1999, Raymond 1999a). In recent years open-source activities have increased and, in practice, become a major development method in software engineering. Here restrictions are less demanding and access to data more open, particularly to source files and communication archives that are distributed under an open license policy.

2.1.2 Iterative and Explorative Development

The open exploration of design alternatives is relevant to produce new ideas, new architectures, and new concepts, particularly in early phases of research. Basili (1996) states that the software-engineering discipline requires a cycle of model building, experimentation, and learning to uncover or develop knowledge. In library communication the need for explorative development is focused on an open and broad exploration of the design space and the possible approaches to design. The search for knowledge can be compared to the search for requirements in software development. Learning from software development, the explorative development should be performed in an iterative manner, where requirements are generated in every step by evaluating developed tools in the industry laboratory (see Section 2.1.1). To conduct an iterative development in which systems are created in a design-evaluate-redesign loop is currently part of many development methods (Sotirovski 2001, Russ and McGregor 2000, Brooks 1987, Jacobson et al. 1999, RHP, Beck 2000). Sotirovski (2001) states that: "Practiced all along, often introduced by practitioners through the back door, iterative development methods are lately receiving their overdue formal recognition." Brooks (1987) advocates a *growing* perspective on software development rather than a *building* perspective.

In an iterative research process, the researcher must balance between cycles that are too short and too long. As a goal, iteration cycles should be short rather than long to get frequent feedback from the industry laboratory. The iterative process must, of course, start somewhere and the initial design should therefore be based on general knowledge from fields such as technical communication (SIGDOC, IEEEPCS), human-computer interaction (SIGCHI, Helander et al. 1997), and software engineering (SIGSOFT, SEWEB).

In human-computer-interaction research and design, prototyping is used to explore the design space and to visualize potential designs. Houde and Hill (1997) provide an in-depth discussion of prototypes. Prototyping, of course, is highly relevant for explorative development and should precede system building. However, it is relevant to go further in library communication to gain access to the industry laboratory. More complete system development also provides hands-on experience with technology, exposing relevant issues such as implementation feasibility, and system performance.

2.1.3 Subjective and Objective Data

In library communication, with exploration as a goal, it is highly relevant to address subjective data in relation to the desires and needs of professionals. Qualitative methods (which provide subjective data, e.g., results from interviews) are often criticized for a lack of strict control of research variables, questioning the validity and reliability of results. There is a continuing debate about the value of qualitative methods, addressed for instance in (Kvale 1989, Kvale 1996, Gummesson 1991). I acknowledge this debate but do not consider it further in this thesis. In my view, both qualitative and quantitative methods are valuable but imperfect tools that both have roles to play in Computer Science and library communication research.

In the industry laboratory, it may be difficult to collect large amounts of highly detailed and rigorous subjective data because of the difficulty of gaining and maintaining access. As a result, it can be risky to base research on larger subjective data collection because data collection can be interrupted or discontinued by subjects. For exploration, open methods, such as semi-structured interviews are appropriate (Kvale 1996). It is even useful to conduct informal discussions to gain some opinions from professionals whenever the opportunity arises.

Naturally, objective data is relevant to research on library communication. Industry may still view logging as potentially dangerous but once access is granted maintaining access is not a problem. Objective data can provide highly valid data, but also lacks the richness of the qualitative methods which questions the relevance of such data collection. Fundamentally, it is the interpretations related to objective data that is problematic in research (subjective interpretations prior to choosing what to collect and afterwards in creating meaning) (Kvale 1989). It is even argued that there is no such thing as objective data since subjective interpretation precedes or follows data collection (Kvale 1989, Gadamer 1989). Furthermore, Pfleeger discusses the limitation of measurement and how it may misdirect researchers. A more probabilistic than natural view on measurement is advocated combined with a design-evaluate-redesign approach to research (Pfleeger 1999). Once again, I acknowledge the discussion but do not consider it further in this thesis.

Objective data can be collected, for instance, by logging user interaction with tools. Another relevant area for objective data collection is system evaluation. Though the research area is in an early phase, there are many different systems developed in the practice of library communication. These systems represent both the state-of-the-profession and the state-of-the-art. Studying the design of existing tools, the data and knowledge they collect, how they treat issues such as copyright and so forth, enable the discovery of common or best practice. System evaluation may very well lead to the discovery of lack of support for arguable needs. Internet has increased the accessibility of software downloads for evaluating purposes, making it much less costly and time

consuming to conduct system evaluation. Yet another means of objective data collection is analyzing source code produced in programming projects. Though produced source code does not directly determine the communication needs of programmers, it does expose the results of the project and can also answer some questions about the related communication.

2.1.4 Summarizing: Library Communication Research

In my mind, it is relevant to conduct explorative, iterative development in design-evaluate-redesign cycles (similar to Basils building-experimentation-learning cycle [Basili 1996]). Evaluation should focus on professionals providing input to the general architectural process.

2.2 Methods Applied

In this section I will reflect upon the methods I have used in my research, the choices I have made, and the consequences of these choices.

2.2.1 Explorative and Iterative Development

I started my work in explorative development, first working with low-level prototypes, described in Paper VI and in Appendix B and later develop a system that could be applied in the real work situations, see Paper I, IV, and V. The DJavadoc system was the result of this work but the work I did with DJavadoc also inspired the analytic papers I produced: Papers IIand III. Originally I was inspired by general design knowledge from technical writing and HCI and also by my own personal experience of using Javadoc as a programming tool. An important source of inspiration has been the *minimalism* approach to technical writing. Minimalist instructional material should inspire action, support and encourage exploration, be brief, provide error information, and so on (Carroll 1990, Carroll 1998). These basic values can be transferred to the design of tool support in library communication, though the minimalist approach focuses on technical writing.

During the iterations, I started with less rigorous evaluation to keep iteration cycles short. I gradually increased the level of rigor in the cycles. To start with rigorous experiments was deemed inappropriate because little is known about library-based programming as an activity (see chapter 5) and relevant research questions are unknown for this particular area. Instead, the intention was to find questions, design systems that address these systems, and answer them more rigorously later on. Looking back I have completed one full circle in my process (described in Appendix B and Papers V, IV, and I).

2.2.2 Data Collection

I have collected data by:

- *User studies* – based on long real-work use experience by professionals and through semi structured interviews, described in Paper I.
- *System evaluation* – of existing tools and projects in the software community, in Papers II and III.

The combination of user studies and system evaluation provide a complementing data collection process.

2.2.3 Standard Tools or Bleeding Edge

In my work I have deliberately stayed close to standard tools by utilizing client-side web technology and have also adhered to the design of Javadoc. For explorative development and evaluation in library communication, it is relevant to stay within the bounds of standard development platforms, such as standard web browsers or common development environments. There are several reasons for developing within the standard tool space:

- *Relevance* – For practitioners, the relevance of tools is higher if they are integrated with standard tools. Using client-side web technologies professionals were also able to test my systems without installing programs on their computer, thus making it easier to overcome difficulties in evaluation.
- *Familiarity* – By expanding existing tools that users are familiar with questions of design that are not relevant to the study but required to create a working systems can be avoided. Users remember what to do and how to do it when they work with the tool; they have knowledge in their head and knowledge in their tools (Norman 1990). Introducing new tools with different functionality and visual appearance can cause complications because users lose their tool memory.
- *Ease of development* – Working within the standard tool set makes it easier to develop systems because more tools support is available (perhaps in the form of libraries). The development of the global community also continues to produce new tools along during research projects.
- *Testability* – Standard tools are more stable and therefore more reliable in experimental situations. Technical errors are likely to appear less often, which reduces the risk of research results being flawed because of technical deficiencies.

- *Attracting development resources* – It is easier to extract external development resources, for instance though open-source projects, for popular and common platforms than for uncommon. The popularity of technology is likely to be one of the major factors for independent participation in open-source projects.

The term "standard tools" seems to indicate old technology. For explorative development in research it may perhaps be argued that the "bleeding edge" technology should be applied (the latest and most advanced). However, standard tools are not necessarily old and low-tech. For instance, web technology has evolved at high speed during my thesis work. Furthermore, for research ventures aiming at use-oriented designs that provide solutions usable by professionals bleeding edge does not always provide the best solution. Standard tools may also provide underestimated and highly relevant features that can be further exploited to provide relevant development. However, standard tools must not stop researchers from exploring unconventional ways of design.

2.3 Future Considerations

After every research venture it is relevant to reflect on what could have been done differently.

2.3.1 Open-Source Explorative Development

I have been restrictive with the distribution of my explorative systems without clear indications of cooperation. In retrospect, I recommend more full-blown open-source projects for similar research ventures. From a research perspective, open-source is a new but relevant area of investigation with few in-depth analyses published (Feller and Fitzgerald 2000, Feller et al. 2001). There are also some publications written by key figures in the early days of open source (DiBona et al. 1999, Raymond 1999a, Perence 1999). Open-source development is based on massive parallel development (Raymond 1999a, Sanders 1998, Raymond 1999b, Feller and Fitzgerald 2000). It has resulted in notable software products such as Linux (LINUXO, Torvalds 1999), GNU software (GNUS, Stallman 1999), and the Apache web server (currently covering more than half the market [AP, NETCRAFT]). Open-source projects can be utterly decentralized where no authority dictates what who shall work on and how. Still, tremendous organization and cooperation emerges in this decentralized activity (Perkins 1999). Robustness is one of the benefits claimed for open source (Willson 1999, Perkins 1999). It is also a process driven by demand for the product in the programming community itself (Vixie 1999). In Paper III, open-source is discussed in more detail in relation to the documentation process and the ability to provide user-driven, just-in-time production of documentation.

As a method for research, open-source development combines peer review with peer collaboration. Users can freely contribute to projects, either by providing input or by becoming developers in their own right. Because the process is open to a global community, it can generate knowledge about user requirements and user attitudes. Granted that a project is able to attract a large and globally distributed community, it can bridge cultural gaps and provide a well-grounded exploration of functionality. We can also consider the development process itself as a research process that delivers valid results through open peer review and collaboration. In this case, global research projects should be that include large sets of independent research groups.

2.3.2 Individual Data

In my experience it has been difficult to maintain access of person resources in industry, in particular without support from upper management. One way to reduce the risk of losing access to data is to focus on empirical material such as source code, and discussion forums archives rather than subjective data collected directly from individuals. However, such empirical material should not be regarded as a substitute for person resources in terms of what knowledge it can uncover.

Chapter 3

Library Communication

In this chapter I describe a model of programming based on software component libraries focusing on the communication activities involved. The relevance of the model lies in its approach to communication in programming and the resulting design requirements it places on tools and processes, which are discussed in chapter 4. The model is not complete and relative values of different issues are not assessed.

3.1 Libraries and Programming

Library communication faces a completely different reality than traditional programming communication (based on the language model of programming). Both programming behavior and the communication medium have changed. Most approaches to technical communication and programming tools are designed for the traditional model. In order to address the specific needs of library communication it is necessary that we understand what library programming constitutes and what the electronic, networked medium can offer library-based programming.

3.1.1 What are Libraries?

Somewhat naively, software component libraries can be viewed simply as collections of reusable components. However, because of the frequent use of libraries in languages such as Java, the role of the library becomes more complicated. Libraries represent a technological framework that has been pulled down over the programming languages that we use to build software. They represent the joint efforts of global software-development communities but are also the joint boundaries within which these communities develop software. Compared to programming languages, libraries contain much larger amounts of constructs,

more implicit structures, are often released early and relatively untested, and are also sometimes published before being completed. To exemplify the extended meaning of the library in a global development perspective, I elaborate on what libraries may represent beyond being collections of reusable software components:

- *Extensions to programming languages* – What could be integrated in the languages can also be added without change to the language through the construction of libraries. One example of this is remote invocation of methods across networks in Java that the Java RIM library provide (JRIM). RIM is a typical language concept that has been placed in a library.

- *Attraction of platform value-providers* – Libraries acquire value-providers to platforms by opening platforms to external extension and facilitate the development of valuable applications. One example is the Java SDK itself that help users quickly develop Java programs and thereby helps create a demand for Java products. Another example is the rational extensibility libraries that open rational tools for external development (RSE).

- *Distribution formats for software components* – Libraries are a commonly used format for the distribution of software components. Many programming languages and operating systems provide some form of packaging construct to collect components into libraries. They may be called packages, modules, namespaces, or DLLs but they are all libraries.

- *Evolving base for programming* – Libraries constitute an evolving, changing, unstable base for development compared to programming languages. The development of the Java SDK exemplifies this, see Table 1.1 on page 6. In the transition between version 1.1 and version 1.2 of Java SDK the event model was completely changed (AWT). Evolution often takes the form of the introduction of new design and the gradual phasing out of old designs rather than the direct change (to handle backward compatibility).

- *Networks of interrelated libraries* – Together, different libraries form technological webs of globally dependant technologies. Most Java libraries are based on another Java library (even if we exclude the default `java.lang` dependence in Java). Java SDK itself is contains a few examples of this. For instance, `javax.swing` is to a based on `java.awt`, and `java.rmi` is based on `java.net` and `java.io`. Third party libraries, developed outside Sun Microsystems, also provide relevant examples. Apache, for instance, includes a number of Java projects that have use several libraries as their basis in implementation (JAPACHE).

- *De-facto standards* – Libraries are the working documents of future standards that provide the basis for technology. Popular libraries, in reality, form standard implementations that later may be formalized into

standards. One example of this is the SAX parser library specification for XML parsers that originally was released in May 1998 and that has been implemented in over 20 different parsers in 5 different programming platforms. SAX is not currently a formal standard but it is a de-facto standard because of the strong support it has in the XML community and because it has been built into much of the XML-related technology (SAX). Another type of libraries that become de-facto standards are libraries that act as *middleware* between applications and system types such as databases management systems. One such example is the Java-database-connectivity library (JDBC) for which many database providers develop implementations. Other similar examples are Java Speech and Java TV (JSPEACH, JTV). Middleware libraries have the ability to become programming de-facto standards because they simplify the process of using multiple systems of the same type. However, they also dictate the interface these system types must adapt to and may thereby also create standards for these systems types.

- *Boundary objects among independent groups* – Libraries are objects that connect may different groups developing software. The connection is often implicit and social rather than structured and defined. The different groups develop their own software but are also affected by each other through the communication they have in relation to the libraries. The SAX library is once again a good example of this. On the web site, 85 different individuals are given credit for having contributed to the design of SAX (SAX).

- *Implicit contracts* – Library developers and library users form implicit contracts for the design of business value. Libraries bind different providers together. Invested time and cost in learning and communication ensure that users of libraries continue to use the same libraries. Software applications based on libraries are bound to the libraries. Considering the fact that Java SDK 1.4 includes 20,000 methods distributed over 2,500 classes (JSDK1.4), users will be reluctant to make drastic changes.

Libraries are more complex than simple collections of software components, in particular through the social implications of library-based programming. Libraries represent a development continuum rather than distinct releases and are also more of a service than a product. Furthermore, the library is the primary connecting element for globally distributed independent developers.

3.1.2 Who is the Library User?

At first glance, the library user is a programmer developing software using a library as a resource in development. However, the question of the library user is also more complex. In software reuse, user roles are commonly divided into

to component *locator*, *adapter*, and *integrator* (Kruger 1992, Basili et al. 1996, Frakes and Fox 1995, Mili et al. 1995, Mili et al 1999). However, this separation of roles is too limited in my view. Here I propose an expanded and more detailed categorization of user roles with regards to library communication specifically:

- *Locator* – Finds libraries, primarily from the Internet or by word of mouth (or email). For libraries of Java SDK's size, finding the library is not as problematic as finding valuable components within the library. However, for smaller libraries locaters locate libraries rather than components. Today, libraries are located mainly by searching the web or looking though specially designed web portals such as Jars (JARS) or SourceForge (SF). Mailing lists also provide a source of references for locators, as libraries sometimes are discussed here. So far, however, automated processes for the location of libraries have not been successful (Mili et al. 1998).

- *Announcer* – Places libraries where locaters are likely to find them. Announcers are required because automated processes for component retrieval are still not working satisfactory (Mili et al. 1998). At first glance, this seems to be a strict developer role but library users also act as announcers. In my mind, word of mouth or in this case word of email is still probably one of the more powerful announcement activities.

- *Examiner* – Studies a library as part of a feature evaluation. The examiner, for instance, must determine if a library can be used in development. Besides determining if and how a library can be used, the examiner needs to assess the current and future status of the library, for instance with regards to the probability of future support.

- *Learner* – Learns how to use the library, both in terms of what the library provides and how to integrate that functionality. This is the role most commonly addressed in library communication see Section 3.3.1.

- *Requester* – Makes requests for the future development of the library, including bug fixes. Public beta release of libraries is common practice today and many libraries have bug databases. Feature requesting is handled in much the same way and sometimes bug reporting and feature request are handled by the same system. Requesters work with library bug and feature databases. They need to make requests and to receive relevant information to avoid replicating other requesters work (particularly in relation to bugs). Paper II discuss this issue more in depth (focusing on the user-related bug handling).

- *Debater* – Actively discuss matters related to the library: a library lobbyist that influences the direction of library development. This role is linked to the requester role, but include other social aspects such as

networking. Many larger companies have discussion forums where libraries they release are discussed. Examples include, the Sun Microsystems' Java Discussion Forum (JAVADISC), IBM's Alphaworks Discussion Forum (ALPHADISC), and Microsoft's Developer Community web site (MICRODISC). The debater may also be passive and participate by listening to the debate to keep up to date with current events. Most users are probably passive debaters rather than active.

- *Anticipator* – Waits for the implementation of new features and bug fixes, hopeful that development will be completed within the timeframe of their own project. The anticipator role becomes increasingly relevant as the speed of development increases and it becomes more feasible to wait from new solutions. Compared to debaters, anticipators need progress signals rather than knowledge and, in my mind, the anticipator role needs to be treated as separate from the debater role in efficient communication tools and processes.

- *Developer* – Finally, users can become developers in the library development process that provides actual source code. Open-source development is entirely based on this user role, see Paper III. Users playing the developer role need to become members of the development team and communicate on the same premises.

These different roles view the libraries differently and require different things from library communication. Often, the same individual plays these roles at different times. For instance, an user is first a *locator* and then an *examiner* in relation to a new library. If the library seems useful the application programmer becomes a *learner*. If the application programmer discovers that the library is relatively new he/she may become an *anticipator*. Depending on personal interest or on the relevance of the particular library, the programmer may also become a *debater*.

To facilitate library use by these different roles, library communication must provide suitable process and tool support. What constitutes good design may differ for different roles or different users.

3.1.3 What is Library-Based Programming?

A library-based programming model must focus on several aspects of development that differ from the language based development perspective. Programming cannot simply be regarded as an activity involving the production of code based on a learnt set of language constructs. Such a library-based programming model contains aspects such as:

- *Multiplicity* – Programmers use multiple libraries from different sources that do not have a joint production system and that are not completely

integrated. The libraries may not even be from the same programming language.

- *Evolution* – Libraries are evolving elements in programming that continue their development throughout their use. First public release may be an alpha-release or more likely a beta-release. Libraries must also update themselves to include new user needs and to adhere to technological standards and frameworks to stay competitive. For instance, XML is currently one of the strongest standards that libraries should match.

- *Legacy* – Libraries will be accompanied by a legacy of older version and out dated information sources that will make drastic change difficult. If design concepts change in the version flow, the old solutions will remain in the background, making understanding and communication about the library more difficult. Furthermore, old books and web pages may also foster the belief that the old model is still in use.

- *Debate* – Library-based programming includes technical debate, that is the participation of library development in the user community. The needs and desires of library users end up in mailing lists, discussion forums, and also sometime as feature requests. Error reports also fall into this category. Library design must facilitate this activity by providing suitable support for participating and at the same time extracting value from that debate.

Compared to language-based programming, library-based programming includes changes in the technical environment and therefore also the continuous inclusion of new material and changed material. Communication becomes a more important aspect of programming. In essence, libraries should be considered as c*ommunity services* facilitating application programming. Library users from all around the world are interlocked socially through the library. An active community with a social structure of users and power users, where sharing of knowledge and actual code takes place, is relevant to the popularity of libraries.

3.1.4 Library UI

Generally libraries are considered to be non-graphical tools. However, libraries have user interfaces (UIs) through which users access their functionality (McLellan et al. 1998, Pemberton 1997). Online documentation is the most common example of a library UI through which programmers learn about the syntax and semantics of libraries. Discussion forums and mailing lists on the Web are other examples supporting the communication tasks that library users and library developers have in relation to the library. Also, bug-handling systems are often provided to handle error and feature requests in relation

to libraries. Moreover, code-completion functionality in development environments can be thought of as GUIs through which programmers can access the functionality of the library by direct manipulation.

It seems there are many different library services spread out over a large number of UIs. There is no common UI through which the user performs library-related tasks. Supplying a common UI would of course make library-use easier by automating the manual steps taken to cross UI boundaries but it may also counteract the role of libraries in other tools, such as the editor, and place too much emphasis on the library.

Let us make the basic comparison between library UIs and the UI of a common word processor (in this case examples are taken from MS Word). A word processor that works like reference documentation would require you to read about the *save* function in the manual and there find the correct syntax to enter at the prompt to issue the command. Available functionality is described in textual format and has to be copied manually into the source code. Copying is even hindered by the use of hyperlinks in documentation since hyperlinks are difficult to select as text (the pointer keeps shifting to the hand, which cannot be used to copy). Code-completion functionality, on the other hand, supports copying and syntax correctness. Furthermore, all commands are presented with equal relevance, that is without a meaningful structure. The time it takes to locate and issue the *open file* command is equal to or even longer than the time it takes to open a web page to the *office at the web*. Commands are organized without regard to the probability of use. Commands are listed alphabetically, which in the word processor case would place *Autotext* in the top and *Undo* at the bottom of the menu system.

The comparison with the common UI of a desktop application uncovers the lack of user-oriented design in library communication. However, in all fairness, it is also relevant to point out that library users are a particular brand of software users perhaps who require a different style of support than the general user. Library users are generally programmers and as such separate themselves from traditional users quite distinctly in some aspects:

- *Code-fluent* – Library users are or should become fluent in the programming language the library is based on. Therefore, natural language is not necessarily the main language of choice for the transfer of knowledge.

- *Commando-based* – Library users are accustomed to a commando-style interface of programming languages and perhaps more accustomed to a UI style (placing commandos) than a GUI style (direct manipulation) interaction models.

- *Expert computer users* – many library users are expert computer users competent enough to make use of flexible, open systems that allow them ample room for the construction of individually developed support sys-

tems. Designs that are too rigid may become obstacles in their work processes.

Though there is a need for use-oriented-interaction design for library users this does not necessarily translate to the type of design that suits the general desktop user.

3.2 A Model of Library Communication

The model of library communication presented here treats library communication as a community activity where independent developers and users with different goals use and develop libraries.

Library communication is the act of communicating as part of library-based programming. Compared to languages, libraries are much larger, continuously changing, seldom complete, less well tested, and less formally specified sets of constructs. Technical evolution during programming projects, legacy of published and inconsistent documentation as well as debate are elements of library communication. Programmers use multiple libraries from independent developers with little or no cooperation in the communication process. Distinguishing between developers and users is not so relevant, because developers are often users of other libraries. Members of library communities are simply programmers with varying technical competence. It is more important to distinguish between roles, such as examiners and learners, to help support programmer tasks.

Ultimately, programmers have the same need for use-oriented designs in library UIs as users do in general. However, programmers are also different from traditional users. They are code-fluent, used to command-based UIs, and are expert computer users. Supporting a programming model described here with the premises of the programmer as a user and the web of interconnected libraries and programmers is, in my mind, an important step in the development of global software engineering. In chapter 4, the design of electronic, networked tools and processes in relation to this model is discussed.

3.2.1 Current Model

Most commonly, contemporary library communication is conducted according to the model presented in Figure 3.1. In this model, developers write documentation based on source code and publish it on web sites. Users then obtain documentation on their own initiative and provide feedback though debate forums such as bug data-bases and mailing lists. In this model, there are three distinct interruptions in the flow of communication:

- *Source code and documentation* – An interruption between the documentation and source code that exists on both sides of the developer-user

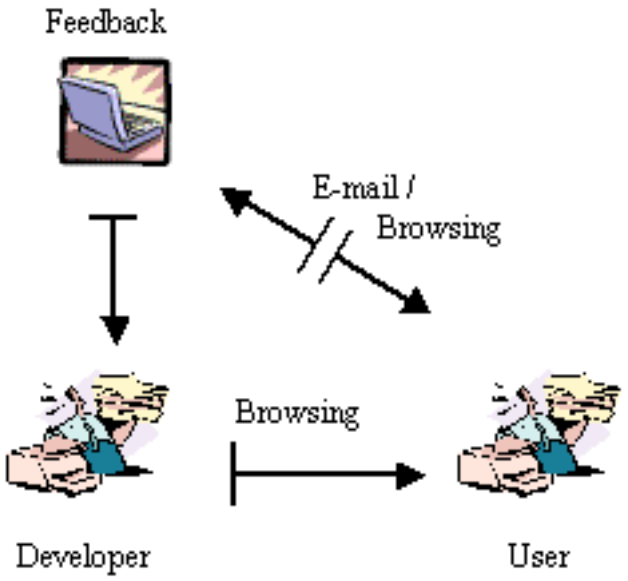

Figure 3.1: The contemporary communication model between developers and users of libraries. There are three distinct interruptions in the communication flow: (a) between source code and documentation, (b) between developers web site and users programming environment, and (c) between feedback database and developer documentation.

relation. Deviation between documentation and source code is a well-known problem in software engineering.

- *Publication platform and programming environment* – An interruption between publication platforms for documentation and programmers development environment. Users are required to perform an active search to acquire updates of developer documentation, which commonly reside outside their standard development environment on a web site they have little reason to frequently visit or in a set of books. Email distribution of documentation removes this interruption to a certain degree. However, emails are not always read even though the programmer's mailbox has received them.

- *Feedback and documentation* – An interruption between user feedback and the documentation it concerns. This interruption is both physical and organizational. The physical interruption comes from a lack of integration of debate into documentation. The organizational interruption comes form the difference of thread-based discussion and topic-based documentation where content has to be recompiled before being used.

In general, these interruptions can be difficult and time-consuming elements in library communication because they introduce stops in the process that require active transfer of content among different platforms. From the developer's side, they cause problems because information is published but not received. From the user's side, they cause problems because feedback is collected but not handled. For both sides, they cause problems because communication and source-code development is not synchronized.

3.2.2 Improving the Current Model

Interruptions in library communication could be removed by a more direct model of communication. Figure 3.2 presents such a simplified model. Here communication is truly bi-directional and communication potentially flows all the way from source files to source files. How to achieve this mode is discussed in Chapter 4. Parts of this model however, exist today in different formats. Javadoc, for instance, removes part of the interruption between source code and documentation (not back from documentation to source code, JAVADAOC, appendix A). Annotated manuals, see section 3.3.2, can handle the feedback interruption by introducing discussion inside manuals. The model in Figure 3.2 is essentially a single connection between two programmers in a library community. If the model is expanded to webs of interconnected programmers with relations across organizations and in some cases even across programming languages.

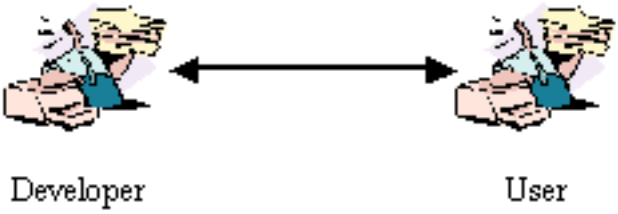

Figure 3.2: Removing the interruptions of Figure 3.1 creates a truly bi-directional channel of communication between developers and users.

3.3 Conclusive Remarks

The state of the art for library communication is provided here, by providing a description of the standard design and the highly relevant exceptions to the norm.

3.3.1 The standard design

The standard design of library communication varies of course, with respect to the different forms of communication. There are also varying degrees of different materials for different libraries, dependent both on size and the development organization. However to summarize, most libraries provide the following library communication pattern:

- *Tutorial material* – Most commonly libraries have so called "get started" or "hello world" texts. These texts are task oriented and provide valuable material for users of libraries. However, beyond the absolute basics tutorial material seldom provides detailed task-based instruction.

- *Reference documentation* – This is the majority of the communication produced in relation to libraries. Reference documentation is, with few exceptions, system-oriented in nature and works as alphabetic encyclopedias without connection to tasks or user profiles. Code completion functionality in development environments provides active support for the transfer of code but also provides extremely minimalist tutorial value. Sometimes libraries provide specifications documents that can be used as reference material, but they are often too complicated, provide too

detailed technical information, and do not provide enough concrete descriptions of how to use libraries.

- *Ongoing debate* – Mailing lists and discussion forums allow the library community to discuss both the current and the future implementation of libraries. Libraries must be relatively large to accumulate an active online debate. Smaller libraries use mailing lists more as a channel for announcements. Mailing lists and discussion forums provide task-oriented approaches to instruction and also facilitate people-to-people connections across organizations. Elements such as future requirements, misunderstandings, poor design, and use frequency could potentially be extracted from the archives of these forums. However, it can be very difficult to extract value from compilations of knowledge spread over multiple threads of conversation without a clear topic structure. Moreover, debate on bugs focuses mainly on collecting bug knowledge rather than the distribution of bug knowledge to users. Though bug databases are made available for search online, the active search required by users means that bug knowledge is poorly communicated (see Paper IIfor a discussion on why this is the case).

Hyper linking is generally the only feature that separates online reference documentation from printed reference documentation. In some cases hyperlinks match concepts in the programming domain, such as return values or parameters of subroutines. Javadoc, for instance, uses hyperlinks to illustrate relations among components. The hypertext network becomes a direct mapping of Java component relations. Though structures are relatively invisible in the documentation, programmers can understand them by browsing the documentation. Still, in many cases, hyperlinks are not used to represent relations in library communication.

3.3.2 The good examples

There are a few highly relevant examples of library communication systems representing the most advanced library communication system utilizing the potential of the electronic, networked medium. I will discuss these systems here to give a reflection of what can be accomplished.

- *Annotated manual* – The annotated manual is an example of how the user community can be included in the creation process (discussed in depth in Paper III). The annotated manual provides prototype documentation to start with. Users can then provide comments on the documentation and on comments written by other users. A debate can be held in the documentation which becomes a foundation for the continued development, error handling, and development of documentation. Examples include

the PHP annotated manual and the MySQL annotated manual (PHPAM, MYSQLAM).

- *Completely open manuals* – Even more flexible are the collectively written manuals that are completely open for rewriting. Users have complete freedom to edit the documentation. Examples include the Squeek manual (SQEM). Squeek is an implementation of the Smalltalk programming language.

- *Execution in manuals* – Manuals that include execution options can provide the user with the ability to try out library functions directly. This functionality is included in the Mathematica Help Browser (Wolfram 1996). JSP Explorer is another recent example system that allows users to test Java server pages code online (JSPEXP).

- *Adaptive manuals* – The adaptive manual changes its content and presentation in relation to the user. The Mathematica Help Browser and MSDN Web Workshop also include limited adaptive mechanisms (Wolfram 1996, MSDNWW). DJavadoc is a more advanced adaptive system that allows the user to define views of information (DJAVADOC).

Chapter 4

Designing for Library Communication

In this chapter I present design implications of the model of library communication presented in Chapter 3. There are three of design discussions: one general for library communication, one specific for Javadoc (see Appendix A), and finally one for programming languages. The discussions are focused on the communication interruptions presented in Section 3.2.1.

4.1 Design Criteria on Library Communication

In May 2001 Sun Microsystems reported having 2 million registered members to their developers connection web site (Nourie 2001), which is a minimal estimate of the number of Java developer worldwide. If at the same time libraries continuously grow and change, development teams will have difficulties keeping up with the demand for library communication that their user community requires. This problem has two solutions: automatic or user-driven approaches to communication. Being able to automate library communication from library source code is the aim of several scientific and commercial ventures (Knuth 1991, JAVADOC, SODARR). A reasonable guess however, is that automation will not be able to meet every need. Eventually human effort will be required to close the gap between automation and good, solid user-oriented library communication. The most plausible solution lies somewhere in-between, where automation both produces content and automatically develops tools for the continued user-driven production of content. This however, is a process that requires tool support built with a different frame of mind than which is currently produced. In this section, I focus on general design requirements for

removing the interruptions of Section 3.2.1 based on automated or user-driven communication.

4.1.1 Source Code and Documentation Interruption

This interruption refers to the manual work required to write documentation and to provide communication tools that are based on the content of library source files. It also refers to the manual work required to move feedback from feedback databases such as bug databases to source files. The following design issues are relevant to the removal of this interruption:

- *Communication-facilitating infrastructures* – Automating the construction of communication facilitating infrastructure from library source files is an important starting point for a library communication process. Generation of debate systems (such as mailing lists) is one example of what such automation can provide, where different communication systems are automatically set up from a library project. Today, automation in library communication is more focused on generating content that on providing communication tools.

- *Automated Writing* – The writing processes need automation both to save time and to provide a standard look and feel. Javadoc exemplifies this in the production of documentation from Java source codes, providing a straightforward method of producing standard Java reference documentation. However, Javadoc does not provide complete automation and still requires much manually written documentation. By analysing source code it could be possible to provide explanation of the use of methods. Test code written to verify the functionality of library components could also be used for this purpose. However it is perhaps not realistic to expect complete automation of the writing process (though desirable).

- *Example-code extraction* – Developing automatic example-code extraction strategies is highly relevant because users frequently request example code. Algorithms need to be developed that facilitate the extraction of useful code examples from source code. One example of such an algorithm extracts all occurrences of a construct (e.g. a method) and provides them in line-number order. Example code is manually constructed today.

- *Feedback Integration* – Integrating feedback into source files is relevant to facilitate the process of working with a user community that actively participates in development. Though filters may be required to determine whether feedback is relevant and correct, once feedback is deemed correct it should appear inside source files to reduce the amount of parallel tools developers use and thereby increase the visibility of feedback. For instance, bug reports should be added to the source files containing the bug. Feedback however, is handled by separate systems today.

4.1.2 Publication Platform and Programming Environment Interruption

This refers to format differences used in communication and in programming. For instance, documentation is expressed mainly in natural language where as programs are expressed in programming languages. Further more, documentation is increasingly web based (services as well as documents) but programming still is mainly based on local resources. The following design issues are relevant to the removal of this interruption:

- *Documentation-to-code transfer* – Writing should focus on facilitating the transfer of knowledge (ultimately source code) from the library specification and source files. Example code is one example of documentation designed for documentation-to-code transfer. Integrated development environments also provide support in this area, commonly though code-completion functionality. However, documentation as an activity is still focused on teaching the use of library concepts rather than providing documentation-to-code transfer services.

- *Living documents* – Documents need to be more living in the sense that they evolve and include new material as the community continues to develop its resources. Downloaded resources need to incorporate new content as well as illustrate its changes. Local copies of documentation must themselves track the evolution of their originals to help users discover when new information is available. In essence, this requires the production of documents that read other sources and integrate material. Today however, most documents are dead on arrival and do not develop new material. Annotated manuals are examples of living documents, see Section 3.3.2.

- *Communication portal* – It is essential to provide some form of common communication portal that allows users to easily participate in the communication processes that concern a library. A suitable candidate to the communication portal is library reference documentation. For users of development environments, the environment is perhaps a more suitable candidate. The important issue, regardless of platform, is to avoid spreading communication services (e.g. bug reporting) over a large amount of disconnected communication platforms, such as web sites and local copies of documentation which unfortunately is the common design today.

- *Programming-language communication* – For library users, it is relevant to focus on the programming language rather than the natural language. Today, documentation uses code snippets to explain descriptive texts. The opposite focus is desirable, where natural language is used to explain code. One way of focusing more on code is to provide example code as

illustrations of how a component is used rather than natural language explanations. Besides being directly usable in programming, code also teaches the syntactical specification of the programming language.

4.1.3 Feedback and Documentation Interruption

This refers to the separation of feedback resources and document resources where debate and requests are handled separately. Provided is a list of design issues with regards to this:

- *Just-in-time writing* – The speed of growth and change requires just-in-time approaches to content production that delivers content when desired and thereby avoid producing content that is not required by the user community. A stronger focus is put on online forms of communication such as chats and mailing lists. A majority of the communication work will be performed after product release, though an early documentation prototype may be required. Today however, the documentation process is assumed to be relatively complete on product release. Paper III discusses this issue in more detail.

- *Short release cycles* – Short release cycles will be required to quickly deliver content, in particular for early phases of the library communication. Such short cycles may require large staffs that cannot be matched by the development organization, advocating user involvement in the writing process. Today, documentation release cycles are too long because the production of documentation focuses on professional products such as books or official websites. One exception however, is library mailing lists where response time may be under 24 hours.

- *Request-response driven writing* – Just-in-time writing requires a request-response driven writing process where request are generated and responded to either automatically or by members of the library community (including both users and developers). Some requests could be automated, for instance by analyzing online documentation access logs. For more qualitative requests, members of the community have to provide request. Today, users are included in user-centered design but as subjects rather than peers in the production of content. As Paper III shows, today even open-source project have a restricted approach towards user involvement in document production.

- *Open-source approaches* – Essentially, an open-source approach to writing is required to accomplish efficient request-response driven writing. The major benefits of such a design are relevance and priority in the content production phase. In many cases, library users are competent enough to understand the inner workings of the library and can also contribute to

the communication flow by providing responses. Unfortunately, today one commercial drive in software development is documentation and support.

- *Structured writing* – To facilitate user involvement in the production of requests and responses, users need structured writing processes (exemplified by Javadoc that provides some structure though predefined tags, see Appendix A). Library users are most commonly programmers without training in technical writing. A controlled, well designed structure on a content level can support users both by providing dispositions of quality documentation and by automating typographical design. Today however, most user contributions are provided in raw text.

4.2 `new Javadoc()`

In its current form, Javadoc provides a straightforward generation process that produces streamlined documentation for Java libraries, see Appendix A. It uses a structured documentation process that integrates code and comments in the same format and produces batch HTML documentation. Javadoc is one of the strong features of Java that has also become something of a model for documentation generation tools for (object-oriented) programming languages. However, Javadoc also produces system-oriented, encyclopedic reference documentation and need to be further developed to facilitate library communication. The strength of the Javadoc structure is the generation process that provides disposition for the writing process, supporting communication and providing quality control through structured writing. Here I focus on how Javadoc can be further developed to help remove the interruptions described in Section 3.2.1.

4.2.1 Source Code and Documentation Interruptions

This interruption refers to issue of automation with regards to the source-to-documentation relation and to the feedback-to-source relation. The following design issues are relevant to the removal of this interruption:

- *Speed of navigation* – Reduce the time it takes to find information. Currently, Javadoc shows some bad design choices in this area. For instance, methods are listed last in the document although they are the most commonly used type of information in Java library documentation. The design should focus on users tasks, which may require multiple designs to accommodate different roles.

- *Component execution* – Include execution functionality in Javadoc documentation to provide direct exploration of component functionality. Many components can be visualized graphically and it is relevant to be able test

the components directly while learning about the components. For instance what happens to a graphical component when methods are applied can be visualized.

- *Internal documentation* – The internal documentation process during the development of libraries and of applications, could benefit from a separate documentation structure to save time in the writing phase and to visualize project progress in documentation directly. For instance, the comment structure could include tags that relate to testing, such as *unit-tested*, *user-tested*, and *integration-tested*.

- *User-roles* – Include user roles in the comment structure; see Section 3.1.2 for a discussion of library user roles. Providing different commenting structures for these different roles enables developers to address multiple users in the same source for different builds of the documentation.

- *Multiple sources* – Increase the integration of documentation from multiple sources. Users download libraries from multiple sources. For the user it is therefore relevant to integrate the communications flows, for instance by building joint reference documentation from multiple sources. Javadoc currently does not support multiple documentation sources.

4.2.2 Publication Platform and Programming Environment Interruption

This refers to issue of similarity in formats between the communication platform and the programming platform. The following design issues are relevant to the removal of this interruption:

- *Flow of communication* – Reduce the amount of information that users must handle for instance by providing multiple builds from the same documentation and allowing categorization in the structured writing process. Many users could benefit from a basic profile that only lists the most basic and useful information.

- *Task-orientation* – Increase the coupling between the information organization and user tasks. By providing clearly task-related content the library communication both decreases the amount of information and helps teach the use of libraries. Categorizations that are useful in this respect include application types (e.g. database applications, 3D games and so forth) and internal or external components (for application developers and library developers).

- *Directly usable code* – Users ultimately want to fill their source files with code, part of which comes from the library specifications. Being able to cut and paste various forms of code from the communication flow

into the source files increases the speed of development and provides syntax correctness. This is what code-completion functionality provides. However, example codes and larger, more complex pieces of code are also desirable. Finding ways to automate the production of code in the documentation is desirable, but manual processes can also be supported by the automatic generation of tool support.

4.2.3 Feedback and Documentation Interruptions

This refers to the separation of feedback resources and document resources where debate and requests are handled separately. Provided is a list of design issues with regards to this:

- *Debate infrastructure* – Just as Javadoc provides a straightforward process for the production of reference documentation, it should also produce debate infrastructures such as mailing lists, bug handling, and feature request systems. An automatic production process would decrease the time needed to produce such systems and also enable more detail-level discussion directly coupled to the source files (to organize discussion forums inside the source files or at least match the elements in the source files).

- *Integrate debate access* – If Javadoc produces the debate infrastructure it can also directly connect reference documentation to the debate and thereby make reference documentation a portal to the ongoing communication flow. Unnecessary steps in the communication could be removed (e.g. locating web sites with these debate structures and providing version and component information). Distributing the debate directly though the documentation would also support passive search by the user, which is particularly relevant for the distribution of bug knowledge (see Paper II for an in depth discussion of why this is the case). Integrating debate access would result in annotated Javadoc documentation.

- *Request and response functionality* – To support just-in-time production of content, users must be able to request information and also to respond to previously issued requests for instance using an annotated manual, see Section 3.3.2. Both the request and the response need to be integrated into copies of the documentation and the original source files (i.e., the documentation must be able to upload new content). There may also be a need to propagate requests and responses directly into the source code from which the Javadoc documentation has been built.

- *Reorganization functionality* – To support just-in-time production of task-based documentation organization, users should be allowed to change the organization of documentation, for instance by adding bookmark structures or by rearranging indices. In a global user community, this can

enable dissemination of task-related knowledge among users and developers. It also means that the user community can jointly improve the organization of the documentation.

- *Open-source infrastructure* – In its extreme, just-in-time production needs to become an open-source development process. Javadoc could help generate the technological infrastructure required to generate and support that process from source code directly. Open-source places other requirements on the generation process such as building social structures in the user community (e.g. though rating mechanisms). See Paper III for a framework of open-source documentation.

- *Disposition* – Increase disposition support in the comment structure. Continue the development of the comment structure to include a more detailed disposition of what constitutes quality documentation. Part of the benefit of structured writing is that users do not themselves have to define what constitutes quality documentation. Javadoc currently provides little disposition beyond the general description tag. For instance, for parameter descriptions the comment structure should support the description of meaning for different input values.

4.3 Programming Languages and Communication

Human Readability has been a factor in the development of programming languages previously, for instance, leading to the abandonment of GOTO-statements (Djikstra 1968, Louden 1993). With respect to the large number of libraries that have been developed and shared during the last decade, the design of programming languages should perhaps also focus on the support that is provided for automatic generation of communication tools and process.

The design of programming language fundamentally affects the development of communication; in particular automatic generation of communication infrastructure and content. By analyzing aspects of programming languages it is possible to develop languages that are more adapted to library-based programming. Mainly programming languages can help remove the interruption between source code and documentation, see Section 3.2.1. Here I provide some examples of how programming languages can be further developed for this purpose.

- *Typing* – Strong typing is beneficial in communication. Java, which is strongly typed, can automatically generate relations among components in the documentation and thereby allow for cross component browsing and structural browsing in libraries. Non-typed languages cannot enable

this automatic generation based on type. Consequentially, documentation has less ability to illustrate the implicit relations that do exist in the implementation.

- *Inheritance* – In principle, from a communication point of view there is nothing wrong with inheritance. It can even be beneficial, because manually written documentation can also be inherited. However, inheritance, in practice, leads to large structures. The Java SDK is an example of this (forinstance, the `javax.swing.JPasswordField` class has 6 levels of inheritance and over 300 methods [JSDK1.4]). As a result, it becomes a practical necessity to differentiate between declared and inherited methods in the documentation. However, inherited methods are often more useful than declared methods (which is why they are collected in an abstract class to begin with). As a result, the documentation organization does not illustrate the usefulness of method. In Javadoc, often less useful declared methods are more visible than more useful inherited methods. In essence, inheritance is useful for the development of software libraries but not necessarily for the use of libraries and library documentation.

- *Scope Modifier* – Modifiers are used among other things to describe the scope of method calls (subroutines) for instance. In Java the private, public, and protected modifiers are used to determine the external usability of a class member, such as a method. As an example, all other classes can use a public method. In practice, however, the scope modifiers in Java give rise to many public methods because large structures require public methods for internal cooperation across packages (collections of classes). The Java language thereby externalizes many methods that are not intended for library users. That this is a real problem can be seen from the many requests for *internal* and *external* documentation comments. The amount of such requests has been so high that the Javadoc Team has reserved words for the implementation of such comments in the future (JAVADOC). An alternative programming language solution is a more explicit definition of scope modifiers (e.g. similar to the C++ friends concept).

Chapter 5

Related Work

Library communication is a wide area of research involving many different aspects of human-to-human communication in programming. In my research I have covered some of these issues. In this chapter I address systems and studies in relation to my work and to library communication. Communication within development teams is not addressed because I work mainly with communication within the external technical environment.

5.1 Analysis of Library-Based Programming

Studying library-based programming is a requirement of developing communication tools and processes. Uncovering programmers' work habits in relation to libraries and what they need from a communication perspective is imperative to the development of library-communication tools and processes. It is also relevant to study professional programmer's behavior because the behavior of students may not be representative of professionals (Brooks 1980). Unfortunately, little work has been performed specifically in this area. Frakes and Pole (1994) studied representation models of reusable software components and search effectiveness for different representations and found no significant differences or no clear preferences from programmers. The study contained 3000 keywords which is similar to the number of classes in Java SDK 1.4 (see Table 1.1). The study was performed using professional programmers as subjects. Shull et al. (2000) performed a study sing reading techniques for object-oriented framework learning and found that example-based techniques are well suited for beginners but that hierarchy-based techniques are not. The Shull et al. (2000) study was performed on students.

Programming behaviour has been studied at a more general level. The studies often differentiate between expert and novice behavior (e.g. Soloway

and Ehrlich 1984, Altmann et al. 1995, Fix et al. 1995). For instance, Fix et al. (1993) studied display navigation by expert programmers. They reported that navigation is aided by long-term memory, working memory and the display itself. There has also been a lot of performed on software comprehension and program understanding (e.g. Bohem-Davis 1988, Rugaber 1995). Studies of expert programmer behavior, however, mainly focus on language-based programming, which may limit the relevance of the findings to this thesis.

5.2 Formal Approaches to Library Communication

A fair amount of work has been reported on formal approaches to library communication, mainly regarding component retrieval in relation to libraries (e.g. Mili et al. 1997, Fisher et al 1994, Penix and Alexander 1996, Yoëlle et al. 1991, Fischer 1998, Michail and Noitiks 1999). Mili et al. (1998) surveyed the state of the art of component storage and retrieval and came to the conclusions that though many solutions have been proposed, no solution has offered a breakthrough in software reuse, being either too inaccurate or too untraceable to be useful. The fact that new technology such as Internet keeps opening up new opportunities for packaging and organizing software was considered partly the reason why reuse storage and retrieval methods are still is not successful.

General to software reuse and component retrieval is the idea of software reuse being performed by locating and adapting software components to existing projects. Most approaches ignore the human programmer as an asset in the process (Maiden and Sutcliff 1993).

The development of formal methods that enhance software reuse is highly relevant. However, my work is more directed towards the social aspects of global cooperating programmers and their communication needs in relation to library-based programming. Formal approaches are complementary to the work presented here and will not be discussed further.

5.3 Component Browsing

Browsing nformation sources such as reference documentation, source files, and tutorials is relevant to library communication. The electronic, networked media provide new possibilities to improve efficiency, quality, cost, and frustration in this area. Hypertext browsing is common in library communication, see Section3.3, of which Javadoc is one of the most developed examples in common use (JAVADOC, Kramer 1999, Friendly 1995). The study I have performed in relation to DJavadoc involves information browsing based on adaptability in reading; see Papers IV, V, and I.

Adaptability in documentation can also be used to limit documentation resources and thereby decrease the information users handle. Mathematica and MSDN Web Workshop provide limited adaptability by allowing users to open and close sections of information on an individual page basis (Wolfram 1996, MSDNWW). Another common way to adapt information is to filter based on modifiers where components are removed depending on their type, such as public and abstract in the Java domain (e.g. Stanchfield and Mauny 2001). This technique was used in the prototype systems discussed in Paper V.

There are other relevant examples in this context. For instance, command based browsing that allows programmers to browse systems by using keyboard short cuts and by writing names of known components. Examples include the Info system for GNU documentation (Chassell and Stallman 1997) and code-completion functionality in development environments (e.g. VC and Stanchfield and Mauny 2001). For interpreted languages such as LISP, documentation can also be integrated into the components and called on from the command line (Steele 1990). Though users may not program interactively, they may still call documentation functions directly. For experienced users, command-based interaction provides browsing efficiency.

Context-dependent browsing also enables form for efficient browsing where the context of the programmer facilitates information browsing. Code-completion functionality provides context-dependent browsing to a highly limited documentation set related to the components the programmer is currently using. More generally, some development environments allow documentation browsing from the source code the user is working on (e.g. VC, Stanchfield and Mauny 2001). In this sense, use-oriented browsing of documentation is available but only works for components that have already exist in the source code and is perhaps most relevant for software maintenance and program understanding. Even more general, documentation and code can be interrelated into a combined information browsing process. This combination is possible in the Elucidator, which allows cross browsing in separate windows of both documentation and source code (Normark et al. 2000). Another relevant example is the HyperPro system for Smalltalk hypertext browsing which allows the splitting of code and documentation into several information nodes thereby enabling simultaneous browsing of several parts of a long literate program without splitting the program into subroutines for documentation purposes (Østerbye 1995).

5.4 Exploring Functionality

The electronic, networked medium provides the ability to test components directly in the documentation (e.g. JSPEXP, Wolfram 1996). The documentation includes a runtime environment that can execute code and visualize the result. One of the major advantages of trying out code directly is saving time designing test programs and enabling more detailed exploration. The useful-

ness of exploring functionality in documentation, however, is dependent on component types. For graphical components, for instance, the availability of interactive component exploration is easily appreciated. However, the same may not always be said about all abstract data types. Similarly, in interactive environments for interpreted environments, functionality can be tested directly, for instance in LISP (Steele 1990).

5.5 Brief History of Electronic Reference Documentation

The DJavadoc system, developed and evaluated as part of this work (see Appendix B and Papers Iand IV), is an electronic reference documentation system. It has been developed as part of a long tradition on electronic reference documentation for programming (documentation read from screens). Here I address that history by discussing how electronic reference documentation has evolved.

Early examples of electronic reference documentation include LISP documentation functions, which later developed into Common LISP (Steele 1990). The documentation was called from the command line. Unix man pages work in a similar fashion (UNIX). These examples came into use in the 1970s. In these approaches non-typeset documentation is printed in shells as output.

Later hypertext elements were introduced into online reference documentation systems. Examples include the Texinfo system (described in Chassell and Stallman 1997) which came into use in the mid-1980s. The Texinfo system provides navigational hypertext functionality among pages and documentation elements. Graphical style was originally not as elaborate as it is today due to hardware limitations. The Texinfo system also contains hyper linking for the page or book metaphor where concepts such as previous and next exist (i.e., independent of the content of the manual). Online manuals that use the page metaphor are still relatively common on the web.

Today most electronic reference documentation consists of online hypertext manuals that provide descriptive system-oriented, encyclopedic list of components in a relatively traditional book format that are distributed through the web. Elaborate graphical style is commonly used. Still reference documentation is often downloaded and read locally rather than directly from web sites.

The most recent contributions to electronic reference documentation are the evolving or annotated hypertext online manuals, which provide annotation functionality and manuals, with direct exploration of functionality. For a discussion on this topic see section 3.3.2

5.6 Summary of Related Work

Library communication is a relatively open field of investigation where most of the scientific work has been performed on formal methods. Little is known about the behavior of library-based programmers and about the usability of existing tools. Unfortunately, I have found nothing published on open-source documentation or on bug handling in relation to library-based programming (and nothing in general about bug handling). On the bright side, the field is open for investigation, providing many relevant problems for research in relation to programming.

Chapter 6

Discussion

This thesis addresses the issue of how we should use the electronic, networked medium as a platform for library communication? In general, the work illustrates the great potential for development in electronic communication tools and processes for library-based programming. However, it also illustrates the need to incorporate knowledge specifically related to the area of library communication. In this chapter, I discuss aspects of design in electronic library communication related to the thesis. The first issue is the global behavior of contemporary programming (community software development) and the relation to software development methods. Furthermore, I discuss continued studies on the value of adaptation in reference documentation. Moreover, user-driven documentation is discussed with regards to its value and to the possibility of such processes working in practice. Finally, passive reading as a component in library communication is discussed.

6.1 Community Software Development

This thesis illustrates that the electronic, networked medium gives rise to a new type of software development. I model library communication and library-based programming as a community activity, software, even at a project level, is developed within technical communities of independent organizations. These organizations have completely different goals but they still jointly determine the design of their independent applications through the design of their shared components. In contemporary development, this trend can be seen in the way component libraries are handled, in the community activities that these libraries give rise to such as discussion forums, web sites, and conferences, and how technical standards are developed and accepted by the communities.

On the extreme end of this community axis we find the ideal open-source projects, in which sharing and co-operation is present in all aspects of devel-

opment (e.g. design, development, and management). New levels of speed, quality, standardization, cost and efficiency are attainable in application development, as open-source projects illustrate (DiBona et al. 1999, Raymond 1999a). Commercial projects are currently moving towards the open-source end of community axis, illustrated by the fact that multinational corporations such as Apple, IBM, Intel, Nokia, and Sun have their own open-source licenses (OSIWS).

At the same time, contemporary industrial-strength software-development methods model development as a single-organization activity. Most development methods, such as the rational unified process (Jacobson et al. 1999, RHP) and extreme programming (Beck 2000) take little notice of the technical environment in which development is performed. Models of development generally focus on the customer and the development team but ignore the technical community. This community however, to an increasing degree develops components, dictates technical standards, and drives technical development.

Development models need to include the technical environment in their processes to adhere to the reality of global sharing. Taking advantage of the technical environment in an ad-hoc fashion has the same type of problems as ad-hoc development processes in general. Development models need to include steps that address technical communities outside the development team and describe how the team should interact with such communities. In turn, library communication tools can communicate such methodological development through their implementations and they way users change their behavior in relation to tools.

6.2 Understanding Valuable Adaptation

A question that remains open for investigation in relation to this work, is the matter of what constitutes valuable adaptation in library communication. A large part of this work has been directed towards the development and evaluation of individual adaptation in an industry setting (see Paper I, IV, and Appendix B). Common sense tells us that individual adaptation islikely to provide efficiency in communication by trimming the communication flow in relation to users' contexts. My research however, does not support individual adaptation (see Paper I). Further studies are naturally required to determine the value of individual adaptation. In addition to continuing experimenting with individual adaptation, there is a need to determine how to address adaptation in library communication.

In order to determine what constitutes valuable adaptation, I propose a study on the use profiles of a large and frequently used set of libraries such as the Java SDK. Java SDK has been used in many projects across many application domains and by studying how different projects use such a common set of libraries it becomes possible to explore the value of adaptability. For instance, it

is possible to determine if adaptability is required (a) at all, (b) on an individual level, (c) on an application-category level, (d) or if the solution is a combination of the above for Java SDK. It also becomes possible to create categories of projects based on their use profiles rather than on their application type. Such a study would also provide design knowledge for adaptive communication tools.

If a study of use profiles shows that individual adaptation is required then documentation needs to incorporate contextual data into the presentation and content production phase to achieve optimal communication. This, in essence, requires reasoning by the communication tools and access to personal data. On the other hand, if adaptation is required on an application level, communication tools can be developed on a global level and distributed throughout the library community.

6.3 User-Driven Communication

This work also addresses global user-driven communication and just-in-time production of content (see Paper III). User-driven documentation takes the stance that documentation is incomplete in nature and that the communication processes start when products are released. A central issue of the electronic, networked medium is the increased possibilities of user-driven process. The value is relevance in the production phase, through a global request-response structure involving users reducing cost and unnecessary content and thereby increasing efficiency and quality of the communication flow.

In a paper entitled "Nobody Reads Documentation", Rettig (1991) points out that learners want to get started quickly, rarely read software manuals completely, are discouraged by large manuals, and are best motivated by self-initiated exploration. An interpretation of these statements is that documentation is complementary to products and that their purpose is not to be complete but to support the products they describe. Rettig (1991) also points to a perception of documentation as part of the product interface rather than a separate product. User-driven processes seem suitable for the readers of software documentation.

User driven processes also provide the development process with feedback by illustrating what is: (a) difficult to understand from the software product and (b) not used. In both cases the user community are sending messages that need to be incorporated into future development. Apart from this, user-driven processes can avoid delivering large manuals before users actually have a need for them. In this sense, user-driven processes have the potential to deliver minimal but relevant documentation.

However, content production depends on the user community's ability and willingness to participate. Experiences from open-source development provide many success stories but the process has not yet been tested as a common development method. Open source has mostly been based on highly skilled

technical staff contributing in highly technical, global projects building tools for everyday use. A completely different challenge remains for the open-source method in relation to ordinary users with multiple goals. This is both a technical and a social issue and it remains to be seen whether or not easy enough tools and processes can be developed to facilitate and enable global participation by users in general.

6.4 Passive Reading

The thesis examines direct signalling inside applications as a means of achieving passive reading in relation to bug handling. The term "passive reading" is used here to denote reading that happens as users perform every day activities without direct intent. Communication tools themselves discover and integrate evolving material in the standard work environment. Passive reading is a highly relevant aspect of the electronic, networked medium because it can improve efficiency, quality cost, and frustration in communication. For user-related bug handling, passive reading is a requirement. In many situations, passive reading can be a valuable means of trimming large communication flows because users find only what is relevant to the actions users take.

In essence, documentations and applications become reading agents that handle part of the search and reading tasks involved in keeping up with large and constantly evolving information webs. In Paper II, passive reading is accomplished through an information distribution architecture that signals knowledge inside applications. For more heterogeneous information sources, agents would need to work independent of the information source and much more elaborate reasoning mechanisms would be required. The design of passive reading mechanisms is a highly relevant but also complex research topic.

In programming, passive reading becomes relevant for many different areas. Component publication is one example in which development organizations need to announce new components that have become available for particular tasks. In many cases, programmers use other related components. These components indicate which new components programmers are interested in.

6.5 Summary of Discussion

This thesis is wide in its nature and addresses several related issues that concern the design of electronic, networked tools and processes in library communication. In this discussion, I address issues of particular concern to library communication. The construction of software development methods that address the evolving technical environment as well as the programming team and the customer can help provide solid process support in library communication. Discovering the value of adaptation in library communication is relevant to providing tools that support programming. Developing user-driven processes in

library communication is relevant if synchronization between developers' work effort and user needs is to be improved. Finally, exploring the possibilities of passive reading can help users and developers communicate more efficiently through direct signaling and intelligent documentation that integrates material on its own initiative.

Chapter 7

Conclusion

Contemporary programming is a library-based activity in which collections of reusable software-component libraries constitute the basis of programming. This *library-based programming* rests on a foundation of thousands of rapidly evolving components with many implicit dependencies controlled by several independent development organizations. These components are typically less well tested or less formally defined than programming languages. As a result, much communication among programmers worldwide is required to locate, learn, use, and participate in the development of these libraries. The work presented in this thesis addresses how the electronic, networked medium should be used to support library-based programming.

A general conclusion of the work presented here is that potentially substantial benefits can be gained from developing communication tools and processes that use the electronic, networked media. Unfortunately, few studies have addressed the library-based programming *activity* and its requirements on communication *processes* and *tools*. Although there are many communication tools, few experimental evaluations of these tools have been performed and little is therefore known about their usability. As a consequence, developing electronic library communication requires a combined exploration of tool support and an examination of the library-based programming activity.

On a number of specific points, this research results in unexpected conclusions. The first surprise relates to the value of individual adaptation (personalization) in library communication. A common-sense assumption is that individual adaptation of content and presentation leads to improvements of efficiency, quality, cost, and frustration in communication. However, the research presented here does not support this assumption. Though the electronic, networked medium can provide individual adaptation the study did not shown that it was found to be desirable (by users). More thorough evaluation is needed to be conclusive, but the result is relevant nonetheless because it does not support the often-assumed value of individual adaptation. Instead, the study supported

general features of the electronic, networked medium without relation to user context.

Another surprising discovery relates to documentation approaches in open-source communities. Open-source development is a property of the electronic, networked media and is claimed to provide increased efficiency and robustness in software development. It seems natural that open-source approaches to documentation should work equally well. Open-source approaches to library communication could provide dramatic improvements in quality and efficiency in library communication and help remove costs and alleviate frustration. However, the research presented here shows that open-source documentation still is relatively uncommon even in open-source projects (that often take a more restrictive approach to sharing of documentation than to sharing of source code). If open-source documentation can provide these values is therefore uncertain. However, the electronic, networked medium provides the necessary technical architecture for open-source documentation and projects are starting up.

Moreover, the design of communication architectures for user-related bug handling is also surprisingly non-user oriented. Unfortunately, contemporary bug-handling systems expect users to search actively for bug knowledge by browsing online databases. (The design for library developers has been directly applied to library users.) Instead, communication architectures designed for users must adapt to the passive reading behaviour of users by signalling the existence of knowledge inside the standard communication flow, for instance inside reference documentation. Direct signalling leads to more efficient distribution of knowledge among users worldwide, disseminating knowledge without requiring active search. Passive reading is a highly relevant aspect of the electronic, networked medium that needs to be further explored in library communication.

Furthermore, electronic writing, surprisingly enough, is still directed towards the hardcopy medium. Though commonly used on the web, few electronic concepts, such as collapsible lists, can be used directly in authoring. To a large degree, any expression of the electronic, networked dimensions requires programming skills. The development of web-languages such as HTML and XML, which currently form the basis of electronic writing, provide little support for the expression of electronic concepts on an authoring level.

More generally, the work presented in this thesis shows that library communication is a community process where both library developers and library users contribute to a joint development effort of their joint programming platform. Because library-based programmers combine resources from independent organizations in their work, adherence to such community thinking is imperative. The design of communication tools and processes must adhere to this fact. Currently, the baseline for electronic library communication is hypertext documentation, which has added substantially to the area. The possibilities of the new medium however, provide more than non-linear organization and document cross-referencing. The work presented in this thesis contributes knowl-

edge of how we should use electronic, networked media to support library-based programming.

Chapter 8

Summaries of the Papers

Included in this thesis are six papers that address the use of electronic, networked media in library communication. On a more detailed level the papers can be divide into two groups: one addressing global communication and user contribution to the library communication process (Papers II and III) and one discussion the design of electronic reference documentation (Papers I, IV, V, and VI).

8.1 Paper I: Designing Electronic Library Reference Documentation

Accepted for publication March 2002, The Journal of Software and Systems.

Research question: How should electronic reference documentation be designed?

This paper provides one loop in a design-evaluate-learn research process for the development of electronic library reference documentation. It presents an evaluation of electronic reference documentation in an industrial setting as well as a brief presentation of the DJavadoc system (further discussed in Papers IV and V and in Appendix B). The evaluation is based on 2 experience programmer using DJavdoc for a period of 4 months in real work after which they where interviewed both with regards to DJavadoc and electronic reference documentation in general. The result of the evaluation is a set of requirements of for the future development of electronic reference documentation. A surprising conclusion from the paper is that no support was found for individual adaptation (personalization) with regards to the implemented adaptation in DJavadoc. A stronger focus was placed on support the task of moving source code from documentation to source files.

Relation to other papers: The paper adds an empirical evaluation to the technical discussion of electronic reference documentation held by Papers IV, V, and VI.

8.2 Paper II: Helping Users Live With Bugs

Submitted 2002.

Research question: How should user-related bug handling be designed?

This paper provides an architectural discussion of user-related, post-release bug handing in contemporary software development, with focus on global bug-related knowledge sharing among users. User-related bug handing is particularly relevant for library communication because public beta releases of libraries have become common practice. The paper is based on an analysis of bugs from a user perspective and on a survey of contemporary user-related bug-handing systems. The paper comes to the conclusion that contemporary bug-handing systems provide an insufficient communication structure for user-related bug handing because it is designed for active search for knowledge. As a result, bug-related knowledge is collected by the user community but not efficiently shared within the community. The paper suggests a solution to this problems in which bug knowledge is directly presented in the standard work environment by integrating bug signals directly in the graphical user interface (e.g. inside library reference documentation). Being able to discover bug knowledge as part of normal work routines is considered a requirement for user-related bug handling.

Relation to other papers: The paper is related to Paper III as it addresses the global aspects of library communication as well as user contribution in the library communication process.

8.3 Paper III: Open-Source Documentation: in search of user-driven, just-in-time writing

In Proceedings of SIGDOC 2001. October 21- 24, Santa Fe, NM.
Co-authored by Michael Priestley IBM Toronto Lab, Canada, email: `mpriestl@ca.ibm.com`

Research question: How should open-source documentation processes work?

This paper provides an architectural discussion of open-source development in the documentation process, with focus on user-driven, just-in-time writing. It is based on an analysis of the open-source development process (that still relatively unknown scientifically) and a survey of contemporary open-source

documentation process. A surprising finding is that even open-source projects shows evidence of a more closed approach to documentation than to source code. There are few really open documentation projects in library communication today. The paper also provides a framework for the open-source documentation process based on open-source development as it is characterized today and discuss how such a writing process can provide user-driven, just-in-time writing in general. Though the paper is general in nature, it is particularly relevant for library communication where development speed is rapid and users and developers are both technically competent.

Relation to other papers: The paper is related to Paper II in that it addresses global aspects of library communication and user contribution in the writing process.

8.4 Paper IV: Writing for Adaptable Documentation

In Proceedings of IPCC/SIGDOC 2000. September 24 - 27. Cambridge, Massachusetts.

Research question: How can adaptive authoring in reference documentation be supported?

This paper provides a discussion of what constitutes an adaptive authoring process and of what authoring support the web-language domain provide for the expression of adaptive texts. It takes stance in the DJavadoc project (further discussed in Papers I and V and Appendix B) and discusses what the authoring process must add to the general adaptation mechanism of electronic text. From a writing perspective, adaptive documentation is considered to consist of *change strategies* (what to change), *change mechanisms* (how to change), and *change constraints* (what not to change). The paper shows that adaptive writing, in the web-language domain, provides little support for the expression of adaptation and that programming skill is required to express adaptation in electronic text.

Relation to other papers: This paper describes the adaptive authoring process in electronic reference documentation from the DJavadoc projects perspective and thereby add to the technical discussion held by this paper and Papers V and VI.

8.5 Paper V: Dynamic Software Component Documentation

In proceedings of the Second Workshop on Learning Software Organizations. June 20 2000. Oulu, Finland.
Co-authored by Henrik Eriksson, Department of Computer and Information Science,

Linköping University email: her@ida.liu.se

Research question: How should electronic reference documentation be designed?

This paper provides a discussion of the management of knowledge in software development project though the redesign of documentation. It takes stance in the DJavadoc system and a prototype Javadoc system developed using Protégé and discusses how library knowledge can be captured through documentation systems for a learning software organization. In the paper, the redesign (adaptation) of documentation through view building and tailoring of indices is suggested as a way of capturing knowledge about software development projects that can be efficiently shared among programmers.

Relation to other papers: The address the knowledge capturing potential of an adaptive documentation platform and thereby add to the technical discussion held by this paper and Papers IV and VI.

8.6 Paper VI: Intermediate Knowledge through Conceptual Source-Code Organization

In Proceedings of the 10:th International Conference on Software Engineering & Knowledge Engineering. June 18-20. San Francisco Bay CA USA.
Co-authored by Henrik Eriksson, Department of Computer and Information Science, Linköping University email: her@ida.liu.se

Research question: How should electronic reference documentation be designed?

This paper addresses on the issue of multiple views in library documentation. Automated documentation generally generates one view on the relations among component. However, from a documentation perspective, the paper argues that multiple views are needed to accurately describe components from the different perspectives needed to match different information needs. In order to produce several views, tool support that simplifies the organization of components is required, in particular if users are considered potential view developers. The paper provides a prototype design of such a system and discusses the relevant aspects of this system.

Relation to other papers: The paper represents the starting point of the technical discussion held by this paper and Papers V and IV.

References

ADH&H, *Adaptive hypertext & hypermedia* web site. `http://wwwis.win.tue.nl/ah/`.

ALPHADISC , *IBM's Alphaworks Discussion Forum* `http://www.alphaworks.ibm.com/discussion`.

Altmann E. M., Larkin J. H., and John B. E. (1995) *Display Navigation by an Expert Programmer: A Preliminary Model of Memory.* In Proceedings of the 1995 Conference on Human Factors in Computing Systems. pp. 3–10.

AP, *Apache* (open-source web server). `http://www.apache.org`.

Atkinson S. and Mili A. (1999) *Software Libraries.* In Encyclopaedia of Electrical and Electronic Engineering. Ed. John Webster. John Wiley & Sons.

AWT, *Java abstract window toolkit (AWT)* web site. `http://java.sun.com/products/jdk/awt/vel`.

Barker M. (1997) *From Document Design to Information Design.* In Proceedings of the 15th Annual International Conference on Computer Documentation. October 19–22, 1997, Snowbird, UT USA. pp. 7–10.

Basili V. R. (1996) *The Role of Experimentation in Software Engineering: Past, Current, and Future.* In Proceedings of the 18th International Conference on Software Engineering. 25–30 March. Berlin, Germany. pp. 442–449.

Basili V. R., Briand L. C., and Melo W. L. (1996) *How Reuse Influences Productivity in Object-Oriented Systems.* Communications of the ACM. vol. 39. no. 10. pp. 104–116.

Beck K. (2000) *Extreme programming explained: embrace change.* Addison-Wesley.

Biggerstaff T. J., Mitbander B. G., Webster D. E. (1994) *Program understanding and the concept assignment problem.* Communications of the ACM. vol. 37 no. 5 pp. 72–83.

Boehm-Davis D. (1988) *Software Comprehension.* In Handbook of Human-Computer Interaction. Helander M. ed. North-Holland.

Bolter D. J. (1991) *Writing Spaces: The Computer, Hypertext and the History of Writing.* Lawrence Erlbaum Associates.

Brooks F. P. Jr. (1987) *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer. April. pp. 10–19.

Brooks R. E. (1980) *Studying Programmer Behaviour Experimentally: The Problem of Proper Methodology.* Communications of the ACM. vol. 23 no. 4. pp. 207–213.

Brookshear J. G. (1994) *Computer Science: An Overview.* Benjamnin/Cummings.

Brown A. W. and Wallnau K. C. (1998) *The Current State of CBSE.* IEEE Software. Septermber/October. pp. 37–46.

Bruce K. B. (1996) *Progress in Programming Languages.* ACM Computing Surveys. vol. 28 no. 1 pp. 245–247.

Brusilovsky P. (1996) *Methods and Techniques of Adaptive Hypermedia.* User Modeling and User-Adapted Interaction. no. 6. pp. 87–129, 1996.

Brusilovsky P. and Vassileva J. eds. (1996) *Special Issue on: Adaptive Hypertext and Hypermedia.* User Modeling and User-Adapted Interaction no.6.

Bush V. (1945) *As We May Think.* Atlantic Monthly, July, reprinted in Interactions March 1996 pp. 35–46.

Campione M. and Walrath K. (1998) *The Java Tutorial: Object-Oriented programming for the Internet.* Addison-Wesley (`http://java.sun.com/docs/books/tutorial/`).

Carroll J. M. (1990) *The Nurnberg Funnel.* MIT Press.

Carroll J. M. ed. (1998) *Minimalism Beyond the Nurnberg Funnel.* MIT Press.

Carver D. K. (1969) *Introduction to FORTRAN II and FORTRAN IV programming.* New York.

Chassell R. J. and Stallman R. M. (1997) *Texinfo: The GNU Documentation Fomat.* Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111–1307, USA (`http://www.delorie.com/gnu/docs/texinfo/texinfo_toc.html`).

Davis A. M. (1994) *Fifteen Principles of Software Engineering.* IEEE Software. November. pp. 94–101.

Deitel H., Deitel P. and Nieto T.R. (2000) *Internet and World Wide Web: How to program.* Prentice Hall.

DiBona C., Ockman S., and Stone M. eds. (1999) *Open Sources: Voices from the Open Source Revolution.* O'Reilly.

Dijkstra E. W. (1968) *Goto statement considered harmful.* Communication of the ACM. vol. 11 no. 3 pp. 147–148.

Dillon A. (1994) *Designing Usable Electronic Text: Ergonomic Aspects of Human Information Usage.* Taylor and Francis.

DJAVADOC, *DJavadoc* web site. `http://ida.liu.se/~eribe/djavadoc`.

DOM, *document object mode standard* at w3c`http://www.w3.org/DOM/`.

ECMA, *ECMA* web site. `http://www.ecma.ch/`.

ECMASCRIPT, *ECMAScript Language Specification*`http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM`.

Eriksson H., Puerta A., and Musen M.A. (1994) *Generation of Knowledge-Acquisition Tools from Domain Ontologies.* International Journal of Human Computer Studies. vol. 41 pp. 425–453.

Feller J., and Fitzgerald B. (2000) *A Framework Analysis of the Open Source Software Development Paradigm.* In Proceedings of the 21st International Conference on Information Systems 2000, Brisbane pp. 10–13.

Feller J., Fitzgerald B., and van der Hoek, A. (2001) *(W18) 1:st Workshop on Open Source Software Engineering, position paper for the workshop.* In Proceedings of the 23rd International Conference on Software Engineering, 2001 pp. 780–781.

Fischer B. (1998) *Specification-Based Browsing of Software Components.* In Proceedings of the 13th IEEE Conference on Automated Software Engineering, Honolulu, Hawaii.

Fix V., Wiedenbeck S., and Scholtz J. (1993) *Mental Representations of Programs by Novices and Experts.* In Proceedings of the 1993 Conference on Human Factors in Computing Systems. pp. 74–79.

Flanagan D. (2001) *JavaScript: The Definitive Guide, 4th Edition.* O'Reilly.

Frakes W. B. and Fox C. J. (1995) *Sixteen Questions about Software Reuse.* Communications of the ACM. vol 38. no. 6. pp. 75–87.

Frakes W. B. and Pole T. P. (1994) *An Empirical Study of Representation Methods for Reusable Software Components.* IEEE Transaction on Software Engineering. vol. 20. no. 8. pp 617–630.

Friendly L. (1995) *The design of distributed hyperlinked programming documentation.* In proceedings of the 1995 International Workshop on Hypermedia Design.

FUNNELWEB, *FunnelWEB* web site. `http://www.ross.net/funnelweb/`.

Gadamer H.-G. (1989) *Truth and Method.* 2d ed. New York: Crossroad Publishing.

Glass R. L. (1994) *The Software-Research Crisis.* IEEE Software. November. pp. 42–47.

Glass R. L. (1998) *Reuse: What's Wrong with This Picture.* IEEE Software. March/April. pp. 57–59.

GNUS, *GNU Software* (original free software initiative, origin of open-source). `http://www.gnu.org`.

Gummesson E. (1991) *Qualitative Methods in Management Research.* SAGE Publications.

Gunther C., Mitchell J., and Notkin D. (1996) *Strategic Directions in Software Engineering and Programming Languages.* ACM Computing Surveys. vol. 28 no. 4.

Hackos J.T. (1997) *Online Documentation: The Next Generation.* In proceedings of 1997 ACM Conference on Systems Documentation, Snowbird, Utah, USA. pp. 99–104.

Helander M., Landauer T. K., and Prabhu P. eds. (1997) *Handbook of Human-Computer Interaction.* Elsevier Science.

Houde S. and Hill C. (1997) *What do Prototypes Prototype.* In Handbook of Human-Computer Interaction. Helander M., Landauer T. K., and Prabhu P. (eds.) Elsevier Science.

IEEEPCS, *IEEE Professional Communication Society.* `http://www.ieeepcs.org/`.

Jacobson I., Booch G., and Rumbaugh J. (1999) *Unified Software Development Process.* Addison-Wesley.

Jacobson R. (1999) *Information Design.* MIT Press.

JAPACHE, *Apache Java project* web site. `http://java.apache.org/`.

JARS, *Java Review Service* web site. `http://www.jars.com/`.

JAVA, *Java* web site. `http://java.sun.com`.

JAVADIC, *Java Discussion Forum* `http://developer.java.sun.com/developer/community/`.

JAVADOC, *Sun Javadoc* web site. `http://java.sun.com/j2se/javadoc/index.html`.

JAVADOCMEM, *FAQ note on Javadoc memory requirements* `http://java.sun.com/j2se/javadoc/faq.html`.

JAVADOCTAGS, *Proposed Javadoc tags*`http://java.sun.com/j2se/javadoc/proposed-tags.html`.

JAVASCRIPT, *Netscape Javascript Developer Central* web site. `http://developer.netscape.com/tech/javascript/index.html`.

JDBC, *Java Database Connectivity* web site. `http://java.sun.com/products/jdbc/`, `http://java.sun.com/docs/books/jdbc/`.

Johnson A. L., and Johnson B. C. (1997) *Literate programming Using Noweb.* Linux Journal. 42:64–69.

JRIM, *Java remote method invocation (RMI) library* web site. `http://java.sun.com/products/jdk/rmi/index.html`.

JSDK1.4, *Java Standard Development Kit version 1.4.* `http://java.sun.com/j2se/1.4/`.

JSPEECH, *Java speech library* web site. `http://java.sun.com/products/java-media/speech/`.

JSPEXP, *JSP Explorer* (online test platform for Java Server Pages scripts). `http://www.mslinn.com/jspExplorer/`.

JTV, *Java TV library* web site. `http://java.sun.com/products/javatv/`.

Kahn P. and Lenk K. (1998) *Principles of typography for user interface design.* Interactions, pp. 15–29.

Kantorowitz E. and Sudarsky O. (1989) *The Adaptable User Interface.* Communications of the ACM, No. 31 Vol. 11 1989.

Knuth D. E. (1984) *Literate Programming.* Computer Journal. vol. 27. May. pp. 97–111.

Knuth D. E. (1991) *Literate Programming.* Center for the Study of Language and Information, Leland Stanford Junior University.

Knuth D. E. and Silvio L. (1994) *The CWeb System of Structured Documentation.* version 3.0. Addisson Wesley.

Kramer D. (1999) *API Documentation for Source Code Comments: A Case Study of Javadoc.* In proceedings of the Seventeenth Annual International Conference of Computer Documentation (SIGDOC'99), New Orleans, September 12–14, 1999.

Krueger C. W. (1992) *Software Reuse.* ACM Computing Surveys. vol. 24. no. 2. pp. 131–183.

Kvale S. (1996) *Interviews: an introduction to qualitative research interviewing.* Sage.

Kvale S. ed. (1989) *Issues of Validation in Qualitative Research.* Studentlitteratur.

Lewis J. E. and Weyers A. (1999) *ActiveText: a Method for Creating Dynamic and Interactive Texts.* In proceedings of the 12th annual ACM Symposium on User Interface Software and Technology. November 7–10, 1999, Asheville United States.

LINUXO, *Linux.org* (central source of Linux information). `http://www.linux.org`.

LIT_PROG_OVERVIEW, *Literate Programming Tools overview web page.* `http://www.desy.de/user/projects/LitProg/HTML.html`.

Louden K. C. (1993) *Programming Languages: Principles and Practise.* PWS Publishing Company.

Madien N. A., and Sutcliff A. G. (1993) *People-Oriented Software Reuse: the Very Thought.* In Proceedings of the Second International Workshop on Software Reuse. IEEE Computer Press. pp. 176–185.

McLellan S. G., Roesler A. W., Tempest J. T., and Spinuzzi C. I (1998) *Building More Usable APIs.* IEEE Software. May/June. pp. 78–86.

MFC, *Microsoft Foundation Classes,* web site. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/vcmfchm.asp`.

Michail A. and Notkin D. (1999) *Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships.* In Proceedings of 21st International Conference on Software Engineering. Los Angeles, CA. pp. 463–472.

MICRODISC, *Microsoft's Developer Community* web site. `http://msdn.microsoft.com/community/`.

Mili A, Yacoub S., Addy E., and Mili H. (1999) *Toward an Engineering Discipline of Software Reuse.* IEEE Software. September/October. pp. 22–31.

Mili A., Mili R., and Mittermeir R. (1997) *Storing and Retrieving Software Components: A refinement Based System.* IEEE Transaction on Software Engineering. vol. 23. no. 7.

Mili A., Mili R., and Mittermeir R. (1998) *A survey of Software Component Storage and Retrieval.* Annals of Software Engineering. vol. 5. pp. 349–414.

Mili H., Mili F., and Mili A. (1995) *Reusing Software: Issues and Research Directions.* IEEE Transaction on Software Engineering. vol. 21. no. 6. pp. 528–562.

MSDNWW, *Microsoft Developers Network Web Workshop.* `http://msdn.microsoft.com/workshop/entry.asp`.

MYSQLAM, *MySQL online annotated manual.* `http://www.mysql.com/doc/`.

Nelson T. H. (1987) *Literary Machines.* South Bend.

Nelson T. H. (2001) *Opinion about hypertext by one of the founding fathers,* viewed in October 2001 `http://ted.hyperland.com/whatIdo/`.

NETCRAFT, *Netcraft Web Server Surveys,* viewed June 2001.`http://www.netcraft.com/survey/`.

Nielsen J. (1995) *Multimedia and Hypertext: the Internet and Beyond.* AP Professional.

Norman D. A. (1990) *The Design of Everyday Things.* Basic Books.

Normark K., Andersen M., Christensen C., Kumar V., Staun-Pedersen S., and Sørensen K. (2000) *Elucidative Programming in Java.* In Proceedings of IPCC/SIGDOC 2000, September 24–27, Cambridge, Massachusetts.

Nourie D. (2001) *JDC Registers Over Two Million Users.* Online Article. `http://developer.java.sun.com/developer/technicalArticles/Interviews/2milmikenoel/`.

ODBC, *Microsoft Open Database Connectivity* web site `http://www.microsoft.com/data/odbc/default.htm`.

OSIWS, *Open Source Initiative (OSI)* web site. `http://www.opensource.org`.

Østerbye K. (1995) *Literate Smalltalk Programming Using Hypertext.* IEEE Transaction on Software Engineering. vol. 12 no. 2 pp. 138–145.

Pemberton S. (1997) *Programmers are Humans Too, 2.* SIGCHI Bull. vol. 29. no. 3 pp. 64.

Perence B. (1999) *The Open Source Definition.* In Open Sources: Voices from the Open Source Revolution. Eds. DiBona C., Ockman S., and Stone M. O'Reilly.

Perkins (1999) *Culture Clash and the Road to World Domination.* IEEE Software January/February 1999 pp. 80–84.

Pfleeger S.L. (1999) *Albert Einstein and Empirical Software Engineering.* IEEE Computer. October. pp. 32–37.

PHPAM, *PHP online annotated manual.* `http://www.php.net/manual/en/`.

Potts C. (1993) *Software-Engineering Research Revisited.* IEEE Software, September pp. 19–28.

Pressman R. S. (2000) *Software Engineering: A Practitioner's Approach.* McGraww-Hill

Prosise J. (1999) *Programming Windows With MFC.* Microsoft Press.

PROTEGE, *Protege* web site. `http://protege.stanford.edu/index.shtml`.

PYRD, *Python reference documentation,* viewed in November 2001 `http://www.python.org/doc/current/modindex.html`.

Ramsey N. (1994) *Literate Programming Simplified.* IEEE Software. Septermber. pp. 97–105.

Raymond E. S. (1999a) *The Cathedral & the Bazaar.* O'Reilly. Sebastapol CA, USA.

Raymond E. S. (1999b) *A Brief History of Hackerdom.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

Reiss S. P. (1996) *Software tools and environments.* ACM Computing Surveys, vol.28 no.1 281–284.

Rettig M. (1991) *Nobody Reads Documentation.* Communications of the ACM. vol. 34 no. 7 pp. 19–24.

RHP, *Rational Home Page,*`http://www.rational.com`.

Rosenfeld L. and Morville P. (1998) *Information Architecture for the World Wide Web.* Sebastopol: O'Reilly.

Rosson M. B. (1996) *Human Factors in Programming and Software Development.* ACM Computing Surveys vol.28 no.1 193–195.

RSE, *Rational Suite Extensibility* web site. `http://www.rational.com/leadership/initiatives/extensibility.jsp`.

Russ M. L. and McGregor D. (2000) *A Software Development Process for Small Projects.* IEEE Software September/October. pp. 96–101.

Rutledge L., van Ossenbruger J., Hardman L., and Bulterman D. (1997) *A Framework for Generating Adaptable Hypermedia Documents.* Proceedings of the Conference on Multimedia '97 November 9–13, 1997, Seattle, WA USA.

Sanders J. (1998) *Linux, Open Source, and Software's Future.* IEEE Software September/October 1998 pp 88–91.

SAVANNAH, *GNU Savannah open-source licensed, web project platform,* viewed January 2002`http://savannah.gnu.org/`.

SAX, *The Simple API for XML* web sites. `http://www.megginson.com/SAX/` `http://www.saxproject.org/`.

Schach S. R. (1997) *Software Engineering with Java.* Irwin.

SDK, *Java Standard Development Kit.* `http://java.sun.com/j2se/`.

SEWEB, *IEEE Computer Society Software Engineering Web.* `http://www.computer.org/SEweb/`.

SF, *SourceForge,* online open-source project web site. `http://sourceforge.net`.

Shull, F., Lanubile, F., and Basili V. R. (2000) *Investigating reading techniques for object-oriented framework learning.* In IEEE Transactions on Software Engineering, vol. 26. no. 11.

SIGCHI, *ACM Special Interest Group on Human-Computer Interaction.* `http://www.acm.org/sigchi/`.

SIGDOC, *ACM Special Interest Group on Software Documentation.* `http://www.acm.org/sigdoc/`.

SIGSOFT, *ACM Special Interest Group on Software Engineering.* `http://www.acm.org/sigsoft/`.

Smart K. L. and Whiting (1994) *Reassessing the Documentation Paradigm: Writing for Print and Online.* In proceedings of 1994 ACM Conference on Systems Documentation, Banf, Canada pp. 6–9.

SODARR, *SoDA from Rational Rose* (automated documentation in Rational Rose's suite of products). `http://www.rational.com/products/soda/index.jsp`.

Soloway E. and Ehrlich K. (1984) *Empirical Studies of Programming Knowledge.* IEEE Transactions on Software Engineering. vol. 10. no. 5. pp. 595–609.

Sommerville I. (1989) *Software Engineering.* Bath Press.

Sotirovski D. (2001) *Heuristics for Iterative Software Development.* IEEE Software May/June pp. 66–73.

SQEM, *Squeek online editable manual* (Squeek is a Smalltalk implementation). `http://squeak.cs.uiuc.edu/documentation/index.html`.

Stallman R (1999) *The GNU Operating System and the Free Software Movement.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

Stanchfield S. and Mauny I. (2001) *Effective VisualAge(r) for Java.* John Wiley. (see also `http://www-4.ibm.com/software/ad/vajava/`).

Steele JR., G. L. (1990) *Common LISP.* Digital Press.

Szyperski C. (1999) *Component Software: Beyond Object-Oriented Programming.* Addisson-Wesley.

Tichy W. F., Lukowicz P. Prechelt L., and Heinz E. A. (1995) *Experimental Evaluation in Computer Science: A Quantitative Study.* Journal of Systems and Software. 28:9–18.

Tilley S. R., Müller H. A., Orgun M. A. (1992) *Documenting Software Systems with Views.* In Proceedings of the 10th Annual International Conference on Systems Documentation. pp. 211–219s.

Torvalds L. (1999) *The Linux Edge.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

UNIX, *The Open Groups Unix.*`http://www.unix-systems.org/`.

van Vilet H. (1993). *Software Engineering Principles and Practice.* John Wiley.

VC, *VisualCafé* (development environment) `http://www.visualcafe.com/`.

Vixie P. (1999) *Software Engineering.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

W3C, *World Wide Web consortium* `http://www.w3.org/`.

White M. (1998) *Designing Dynamic Hypertext.* In proceedings of the 2nd Workshop on Adaptive Hypertext and Hypermedia HYPERTEXT'98 (`http://wwwis.win.tue.nl/ah98/`), Pittsburgh, USA, June 20–24, 1998.

Willson (1999) *Is the Open-source Community setting a Bad Example?* IEEE Software January/February 1999 pp. 23–25.

Wolfram S. (1996) *The Mathematica Book.* Wolfram Media, Cambridge University Press.

Wong Y. Y. (1996) *Temporal typography: a proposal to enrich written expression.* In Proceedings of the 1996 Conference on Human Factors and Computing Systems. pp 408–409.

Woods S. and Yang Q. (1996) *The program understanding problem: analysis and a heuristic approach.* In Proceedings of the 18th International Conference on Software Engineering. pp. 6–15.

Yin R. K. (1994) *Case Study Research: Design and Methods.* Sage.

Zellweger P. T., Regli S. H., Mackinlay J. D., and Chang B.W. (2000) *The Impact of Fluid Documents on Reading and Browsing: An Observational Study.* In Proceedings of the 2000 Conference on Human Factors and Computing Systems.

Paper I.
Designing Electronic Library Reference Documentation

Accepted for publication in Journal of Software and Systems, 2002

Abstract

Contemporary software development is based on *global sharing* of *software component libraries*. As a result, programmers spend much time reading reference documentation rather than writing code, making library reference documentation a central programming tool. Traditionally, reference documentation is designed for textbooks even though it may be distributed online. However, the computer provides new dimensions of change, evolution, and adaptation that can be utilized to support efficiency and quality in software development. What is difficult to determine is how the electronic text dimensions best can be utilized in library reference documentation.

This article presents a study of the design of electronic reference documentation for software component libraries. Results are drawn from a study in an industrial environment based on the use of an experimental electronic reference documentation (called Dynamic Javadoc or DJavadoc) used in a real work situation for 4 months. The results from interviews with programmers indicate that the electronic library reference documentation does not require adaptation or evolution on an individual level. More important ly, reference documentation should facilitate the transfer of code from documentation to source files and also support the integration of multiple documentation sources.

I.1 Introduction

Software-components libraries reused in programming are today shared on a global scale on the Internet. In this article software component libraries are refereed to simply as *libraries*. For a long time, programmers have shared software components on a global scale. As an example, Fortran II which was released in 1958, enabled the use of separately compiled subroutines (Carver, 1969). More generally, libraries are externally built collections of abstract data types (ADTs). Bruce (1996) considers ADTs to be perhaps the most important development of programming languages. However, in "No Silver Bullet" Brooks (1987) points out that much of the complexity of software comes from conformance to other software, such as external ADTs. Today, global sharing is not just a possibility but a foundation of programming. The library concept is highly integrated into languages such as Java (called application programming interfaces or APIs in the Java world). The development of the Java language core library, Java SDK, points to this fact, see Table I.1. Another example is the open-source language, Python, which in Novemeber 2001 had 252 global modules with over 2,200 functions (PYRD).

Table I.1: Development of the Java standard development kit (Java SDK) so far.

SDK Version	Packages	Classes	Ref. Doc. (Mbytes)
1.0 (1995)	3	70	3
1.1 (1997)	22	600	8
1.2 (1998)	59	1,800	80
1.3 (2000)	76	2,150	97
1.4 (2001)	135	2,700	131

Global sharing of libraries is essentially an act of software reuse. However, in this article the term global sharing is used to place focus instead on the issue of communication among programmers that software reuse implies (since attempts to automate component retrieval have not been successful [Mili et al., 1998]). Software reuse is something the Software Engineering community assumes valuable but which has not been substantially supported by empirical evidence. Basili et al. (1996) showed significant benefits from reuse in software development in terms of reduced defect density and rework as well as increased productivity. Frakes and Fox (1995) however, showed that programmers like reuse as a basis for programming. Glass (1998) on the other hand argues that reuse is not so commonplace as one may think and that in reality few components are reused in industry from collections such as the Java SDK. Yet, reuse is definitively an issue considered relevant to software engineering. Mili et al. (1999) for instance, state that software development cannot possibly become an engineering discipline so long as it has not perfected a technology for developing products from reusable assets in a routine manner on an industrial scale.

Libraries provide programmers with functionality and thereby remove part of the coding activity involved in the development of software applications. However, libraries also increase the amount of information gathering and learning - reading -

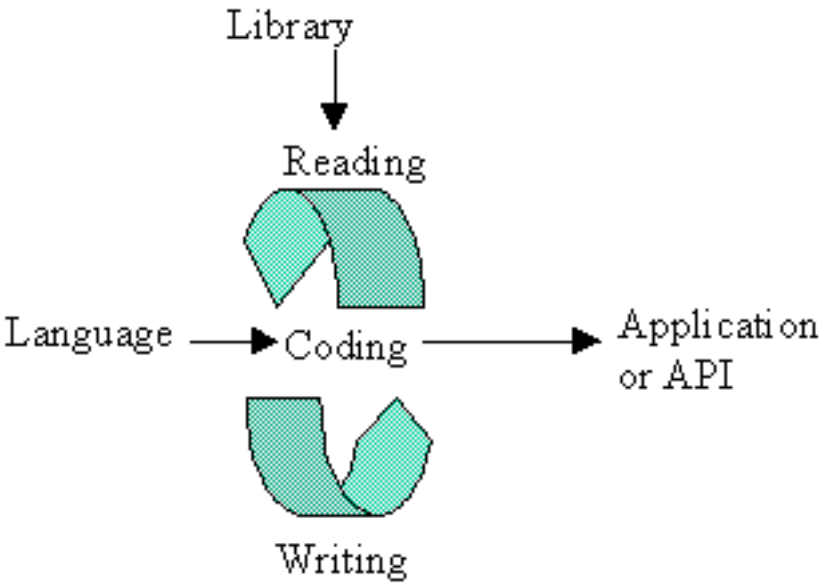

Figure I.1: Software component libraries relieve programmers of development tasks but, on the other hand, also require reading, understanding, and searching of library documentation. In this sense, library-based development transforms programming from a coding activity to an iterative reading-coding activity.

required by programmers. This comes from the problem of cognitive distance, that is the intellectual effort required by programmers to use libraries in development. In practice however, it has proven difficult to reduce the cognitive distance (Krueger, 1992). This issue can also be linked to Brooks (1987) refers to as the complexity of conformance to other software. As a consequence, reference documentation becomes a central tool in using libraries that facilitates the process of overcoming the cognitive distance [Krueger, 1992] and the complexity of conformance to other software [Brooks, 1987]. Programming is transformed from a coding activity to an iterative reading and coding activity, illustrated by Figure I.1. Libraries must be located, chosen, studied, and understood for use. Furthermore, libraries grow large and continuously evolve, see Table I.1. Having in-depth knowledge about thousands of classes is not realistic and not even relevant when new classes are added on a regular basis. Developers must continuously update themselves and therefore the need to read reference documentation exists throughout development projects.

Efficiency in reading is a relevant issue in the design of library reference documentation. The time it takes to collect knowledge, such as how to reuse a class, and syntax, such as the signature of a method, from documentation is a cost in software development. Rosson (1996) stated that programmers spend considerable

time communicating with others in their organization. Library-based programming causes communication to expand outside the organization through reference documentation. Electronic text can overcome some fundamental limitation of the static textbook to further support efficiency. For instance, electronic documentation has the ability to adapt content in relation to context variables (e.g., a programmer's development project). Commonly, hyperlinks are the only sign of a medium that provides additional dimensions such as time, interactivity, change, evolution, and the third dimension. Potentially, electronic documentation can support programming further by making use of these dimensions. The potential is visible in contemporary web technology and in adaptive hypertext research area (Brusilovsky and Vassileva, 1996; AH).

What is difficult is to tailor the general mechanisms of electronic documentation to library-based programming (Berglund, 2000). The electronic medium requires new authoring and design techniques in the field of technical communication (Hackos, 1997; Baker, 1997; Smart, 1994). This article presents a study aimed at uncovering design knowledge for electronic reference documentation for software component libraries. The study represents a design-evaluate-learn loop where an electronic documentation system, called Dynamic Javadoc (DJavadoc), has been evaluated in an industrial setting by subjects having 4 months real-work experience using DJavadoc. DJavadoc extends the standard Java library reference documentation (JR, see Section I.2) by adding dimensions of individual adaptation and evolution. The interested reader can test DJavadoc at `http://www.ida.liu.se/~eribe/djavadoc/` using Microsoft Internet Explorer (version 4 or newer). Judging form the study, individual adaptation of documentation is not highly relevant. Stronger focus should be placed on documentation's ability to help complete programming tasks, such as transferring code between documentation and source files and to combine reference documentation from multiple sources. This research is further discussed in Berglund (1999; 2000) and Berglund and Eriksson (2000).

The article is organized according to the following: Section I.2 provides background information on library reference documentation and Java standard library reference documentation (Javadoc). Section I.3 describes and motivates the study presented in this paper. Section I.4 describes the DJavadoc systems, presenting its features and discussing what can be learnt from evaluating these features. Section I.5 presents a study based on DJavadoc in an industrial setting. Results are presented as general comments on electronic library reference documentation. Section I.6 describes work related to the study of electronic reference documentation for software component libraries. Section I.7 provides a discussion of the future design of electronic documentation in the programming domain. Finally, Section I.8 summarizes and concludes the article.

I.2 Background: Library Reference Documentation

Library reference documentation is a software engineering tool that programmers use to transfer knowledge and syntax from documentation to programs. The content and typography of the documentation therefore become important to software de-

velopment. Ways to achieve reading support for library reference documentation, particularly from a use-perspective, is an important software engineering issue. In the literature on software engineering and programming tools, library reference documentation is often omitted or treated lightly (e.g., Reiss, 1996; van Vilet, 1993; Sommerville, 1989; Brookshear, 1994; Schachs, 1997). This omission of a more in depth discussion is particularly unfortunate since programming is increasingly becoming library-based (illustrated by the rapid growth of the Java SDK, see Table I.1). An underlying reason for overlooking library reference documentation may be that programming traditionally involved a limited set of programming-language constructs that could be learnt by programmers. Currently, however, programmers base development on large collections of software component libraries.

The traditional library reference documentation is designed to be a component catalogue that lists available components and present syntactical specifications. Most software component libraries provide online reference documentation of this kind, see for instance (AR; PR; JR). In general, library reference documentation provides:

- brief descriptions of components and component relations (that teach the use of components)
- navigational indices (that help readers access information in the documentation, often alphabetic in organization)
- syntactical specifications (that help programmers write syntactically correct code)

Java reference documentation or Javadoc is a typical example that fit this description. The name comes from the Javadoc program that produces documentation in batch from Java source files. Javadoc documentation consists of homogenous class documents. (Note that in this article the term class denotes Java interfaces, classes, exceptions, and errors.) Initially class information such as inheritance and subclasses are presented. Following, a textual description of the class is provided. Later on summary tables present class members, such as constructors and methods. Further down descriptions of the member summary entries are provided, which the reader reaches by following hypertext links in the summary tables. Figure I.2 presents a screen shot of a Javadoc class document. The interested reader should, however, visit the Java web site for an online demonstration (JR; Javadoc; Kramer, 1999; Friendly, 1995)

Hyper linking is generally the only feature that separates online reference documentation from printed reference documentation. In many cases hyperlinks match concepts in the programming domain, such as return values or parameters of subroutines. Javadoc use hyperlinks to illustrate relations among components. The hypertext network becomes a direct mapping on Java component relations. Though structures are relatively invisible in the documentation, programmers can understand them by browsing the documentation.

I.3 Method

This article reports on a technical study with the aim of uncovering design criteria for electronic reference documentation. It is based on two concepts: *industry as laboratory* and *iterative development*, discussed below. The first loop in such a research

Figure I.2: Screenshot of the standard Java reference documentation. The class documents are presented in the right frame. Further below method details are provided (not visible in the screenshot).

process is presented in this article. The evaluation is based on semi-structured interviews (Kvale 1996). Semi-structured interview was chosen because this is the initial loop in a research process and a wider perspective is considered (compared this to more focused and controlled but also more limited method of evaluation, such as experimentation based on predefined tasks). The strength of the study are that the developed tool builds on an already existing and well established tool (Javadoc, see Section I.2) and performs an evaluation in an industrial setting with expert programmers that have long real-work experience with the developed tool. The weakness of the study is the relative small number of subjects and the uncontrolled semi structured interview approach. As is the case with all studies, this study has to be supported by future research.

I.3.1 Industry-as-laboratory

Laboratory studies have often failed to predict real-world usability. However, it is the lack of the correct context rather than laboratory experimentation per se that is responsible for this failure (Dillon 1994, Brooks 1980) Therefore, in Computer Science in general and for human-related areas such as programming tools in particular, research requires experimentation in real-work situations with experienced subjects. This is often discussed in terms of performing research in the *industry-as-laboratory* (Yin, 1994; Basili, 1996; Potts, 1993; Glass, 1994). At first glance the industry-as-laboratory approach requires an evaluation of academic work in real-work situations. Equally important however, is to acquire empirical problem definitions from industry. Potts (1993) argues that what researchers think are major practical problems often have little relevance to professionals, whereas neglected problems often turn out to be important. Though Potts focuses more on empirical experimentation and analysis than technology development, the same principles are likely to apply to technical modelling. Industry-related problem definition also comes into focus in Davis' (1994) article on "Fifteen Principles of Software Engineering". These principles are proposed as (temporary) laws of physics for software engineering. Furthermore, Glass (1994) advocates the use of evaluation in the engineering model of research (where the value of models is also tested). Contrary to these indications, Tichy et al. (1995) showed that research papers in Computer Science largely failed to provide empirical evaluation.

I.3.2 Iterative Development

Basili (1996) states that the software-engineering discipline requires a cycle of model building, experimentation, and learning to uncover or develop knowledge. To conduct an iterative development in which systems are created in a design-evaluate-learn loops is currently part of many development methods (Sotirovski, 2001; Russ and McGregor, 2000; Brooks, 1987; Jacobson et al., 1999; Beck 2000). Sotirovski (2001) states that: "Practiced all along, often introduced by practitioners through the back door, iterative development methods are lately receiving their overdue formal recognition." Furthermore, Brooks (1987) advocates a growing perspective on software development rather than a building perspective.

I.4 The DJavadoc System

Electronic documentation can provide additional dimensions of functionality that support efficiency in the reading process. Adaptation and evolution are concepts brought about by time and interactivity that the electronic medium provides. The temporal aspects of the information web are sometimes regarded as a drawback (changes can be confusing) but can also be utilized as a means of efficiency. Adaptation and evolution are examples of electronic media abilities that potentially can support efficiency. According to Brusilovsky (1996), adaptive systems model their users and adapt various visible aspects of the system according to this model. Kantorowitz and Sudarsky (1989), on the other hand, define adaptable systems as systems supporting several different dialog modes. In this article, adaptation is defined as the process of changing into better- suited forms in relation to context. When the context variable is time, adaptation is synonymous with evolution.

Electronic text provides adaptation and evolution but not necessarily value for software development processes. Domain knowledge and user preferences must be combined to harness the raw expressiveness of the electronic media. Ultimately this requires information design (Jacobson, 1999; Rosenfeld L. and Morville P., 1998) based on domain knowledge established through empirical studies of programmer behavior. The study presented in this article is such a study, based on an experimental documentation system called Dynamic Javadoc (DJavadoc). DJavadoc realizes adaptation and evolution in the Java programming domain (`http://www.ida.liu.se/~eribe/djavadoc`; Berglund, 2000; Berglund and Eriksson, 2000; Berglund, 1999). In addition to the standard online Java library reference documentation (JR, see Section I.2), DJavadoc provides:

- redesign of information content
- temporary manipulation of redesigned content
- evolving index in a bookmark fashion for individual fast-access browsing

The following subsections examine these features and discuss what design questions may be answered by evaluation.

I.4.1 Redesign

DJavadoc provides programmers with means to further specialize the design of the reference documentation. Programmers can also easily redefine their settings at any time to change their focus.

Value

Redesign is applied to reduce time-consuming, repetitive manual searching for well-defined information types by removing what programmers consider excessive information types. For instance, programmers may keep only a summary of the methods in a class document. The redesign thereby allows for differences in design among individuals and throughout use of the documentation.

Study Goal

By studying the use of redesign in DJavadoc the need for adaptation in program documentation can be examined. For instance, programmers may change their settings every now and then to adjust for changes in their information need. On the other hand, different programmers may use different settings but not change their individual settings over time. Finally, everyone may be content with the same settings. Studying how programmers redesign reference documentation can uncover the knowledge of what constitutes appropriate adaptation.

Implementation

Programmers change their setting by checking on or off elements in an information model, see Figure I.3. Documents are changed in accordance with the new model by removing sections matching the information type.

I.4.2 Temporary Manipulation

After having defined a setting for DJavadoc, programmers may still want to explore removed information without changing the settings. These types of changes are temporary and need not be remembered for subsequent visits to the same document.

Value

The manipulation provides the programmer with the ability to check small information nodes without leaving the context of the current page. One example is that the programmer expands a method to check the detailed description and then collapse the description again (in this example method descriptions have been previously removed in the setting). It also enables a more compact design with easy access to information that is temporarily relevant. In essence, DJavadoc programmers continuously manipulate the text while reading.

Study Goal

Performance issues of temporary manipulation are relevant to investigate because deficiencies can disable the concept of manipulation in reading. Waiting for small subsections to expand may not be acceptable even though the programmer accepts a few sections to download new documents. Whether or not temporary changes are relevant for subsequent visits to documents is also relevant to examine.

Implementation

Hyper-links for certain key-elements elements are used to initiate an expansion of collapse by direct manipulation, see Figure I.4.

I.4.3 Evolving Index

Naturally, in relation to the thousands of classes available in the Java core libraries, reader can benefit from focused indexes in the documentation to provide fast-access

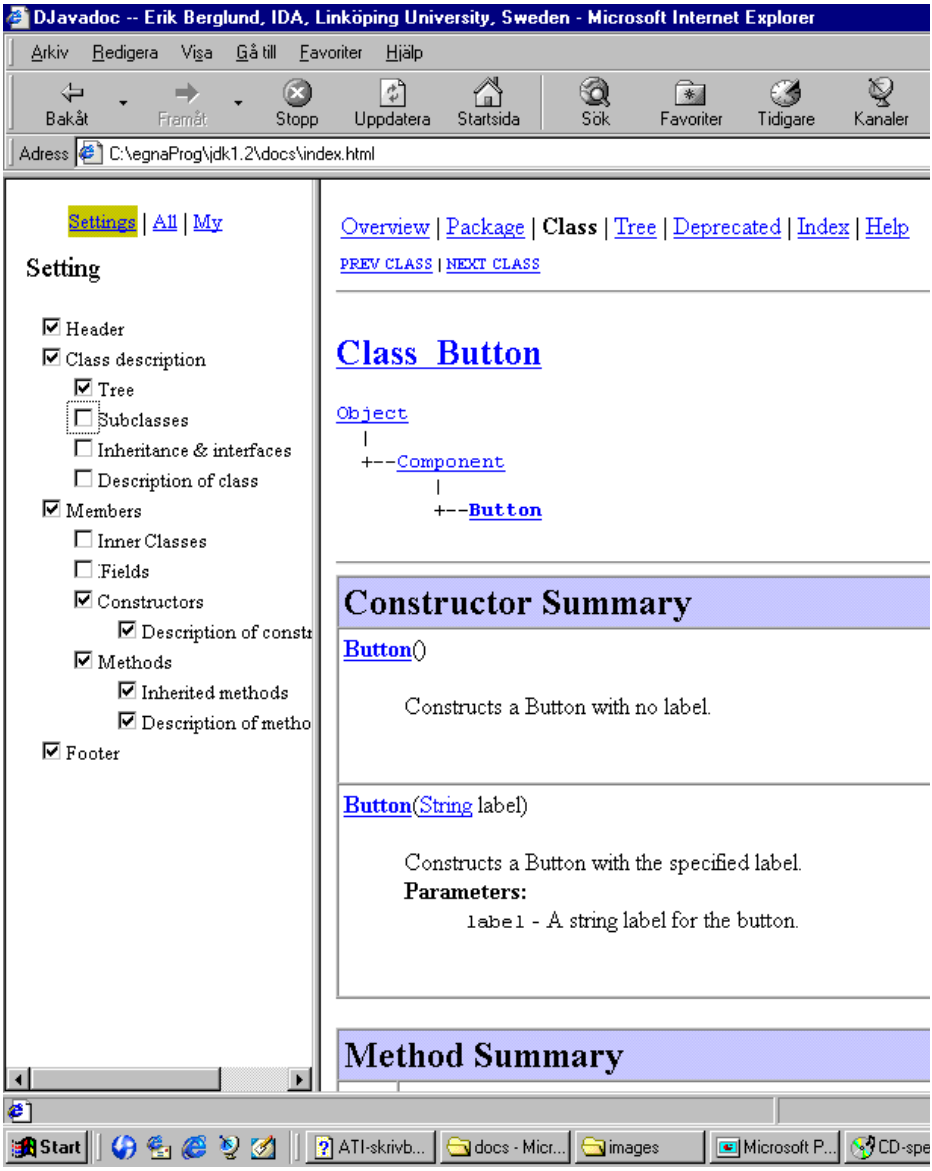

Figure I.3: The reader controls the visibility of different information types by checking elements on or off. As the reader browses the documentation, documents are redesigned in accordance with the view.

to relevant documents. In DJavadoc this need is supported by the construction of a individual navigation index.

Value

The idea behind the DJavadoc bookmarks is to reduce access time for particularly relevant documents through an evolving index. The programmer is allowed to build a representation of the most valuable documents in a special-purpose index. For instance, the classes used in a project can be collected.

Study Goal

By examining the use of evolving indices, the need for individual evolution may be discovered. A relevant question is whether or not programmers need project-related evolution (that relate to their individual situation) or if more general evolution principles suffice (e.g., relating to application types). If project-related evolution is required the documentation must adapt on an individual level.

Implementation

The programmer creates a personal navigation index in a bookmarks fashion, see Figure I.5.

I.4.4 Related Systems

Other systems also make use of the electronic presentation medium to further support programming. The most relevant related systems in this category are Microsoft Developers Network Online Web Workshop (MSDN) and the Mathemathica Help Browser (MM; Wolfram, 1996). Both these documentation systems have sections that can be collapsed or expanded in the documentation. Both systems also remember temporary changes as pages are revisited, treating these as definitions of future information needs (contrary to DJavadoc). The Mathemathica Help Browser, furthermore, allows the reader to execute statements directly in the documentation. Moreover, development environments, such as Visual Café (VC) and JBuilder (JB), provide related features. In Visual Cafe it is possible to browse the documentation directly from the source code. The source code becomes an evolving index to the documentation.

Compared to these systems DJavadoc goes further in the aspects of adaptation and evolution, to the point of being very open. DJavadoc is a research vehicle rather than a product and the openness is aimed at the exploration of adaptation and evolution rather than representing an optimal design.

I.5 Study

DJavadoc was studied in an industrial setting. The aim of the study was to find general design criteria on electronic library reference documentation rather than to evaluate the individual value of DJavadoc features. DJavadoc was, thus, used as an example system for electronic documentation. This section presents the evaluation and its results. Related studies are discussed in Section 5.

I.5.1 Subjects and Setting

Two programmers working at Ericsson Radio Systems in Linköping, Sweden, used DJavadoc instead of Javadoc during 4 months of normal work. They used DJavadoc documentation for Java core libraries version 1.2 which contains over 1,800 classes (see Table I.1). The programmers then participated in a semi- structured interview addressing both DJavadoc specifically and electronic reference documentation in general. One of the programmers had 2-3 years of education before working 20 years as a professional programmer (3 years with Java). The other programmer had 7 years of education before working 7 years as a professional programmer (2 years with Java). During the study the programmers worked in the same project at the same department. Both had extensive experience using Javadoc.

I.5.2 Results

The results of the study are present here as general comments on electronic reference documentation.

Redesign

DJavadoc provides redesign by the definition of content visibility (see Section I.4.1). The programmers both, independently, defined the same settings, which was different from the default setting, and never changed their settings during the study period. Their settings removed detailed descriptions of class members. The behavior of the programmers indicates that adaptation is not needed on an individual level.

Temporary Manipulations

Apart form the setting, some sections of the documents can be temporarily manipulated in DJavadoc, see Section I.4.2. The programmers generally appreciated the temporary manipulation. In particular, they liked the ability to temporarily open up more detailed descriptions of class members. This is not surprising since it is consistent with expert reading behavior (Hackos, 1998; Carroll, 1998; Redish, 1989).

Generally, the type of manipulation used in DJavadoc (stretch text, [Nelson, 1987]) worked well and the graphical performance was sufficient even for relatively small changes. For index sections, the collapse and expand functionality was less appropriate due to the relatively long lists found in standard Java library reference documentation (JR, see Section I.2). The programmer said that they had opened more than one package list at a time but never more than two. Based on this, collapse and expand works may work best for smaller sections and alternative graphical solutions to the navigation issue is required.

In DJavadoc temporary manipulation is not remembered for subsequent visits to particular document. This is, however, the case in some related systems, see Section I.4.4. When questioned about this issue, the programmers felt that remembering temporary changes were only interesting within short time frames. Consequentially, a time-invariant electronic layout may suffice to support programming.

Evolution

DJavadoc provides an evolving index in the form of a bookmark list (see Section I.4.3). The programmers had not used the bookmark list because initially the feature did not excite them. Without using it, they reasoned that few very common documents would be useful in such a list. However, they had no real use experience with the feature.

Another evolution issue was brought to attention in the interview. The programmers stated that they wanted support for the addition of new sections to downloaded documentation. These comments point out that programmer use libraries from different source and/or use libraries that are added at different times. Javadoc currently produces documentation that is internally cross-referenced, which allows programmers to browse relations among components, see Section I.2. Such cross-reference is, however, not possible among documentation from multiple library providers. Consequentially, the current production of documentation is in conflict with the design for cross-reference browsing.

Task Integration

During the interview it became apparent that the programmers used the documentation to copy code, both to increase programming speed and to ensure syntactical correctness. Documentation should therefore support the task of transferring code from the documentation to source files. Essentially, this points to the fact that documentation should be viewed as a programming tool rather than a text and that it should actively support a number of related tasks. Moving code from the library reference documentation to the source code is one such task.

Example Code

During the interview the issue of example code also surfaced frequently. Example code is valuable both to explain code and to provide programming samples. The task of moving code from documentation to source files was, once again, the reason for using example code. The programmers wanted to copy and paste example code into the source files to increase development speed.

Information Priority

When asked about what piece of information is most valuable in the Javadoc documents, the programmers conclude that information about methods is most valuable. The current Javadoc layout, however, places methods at the end. Thus, simply reorganizing Javadoc based on user information priority would improve the efficiency in reading.

I.5.3 Study Summary

DJavadoc was studied in an industrial setting consisting of 2 programmers using the system for 4 months. Based on the study, it seems that a time-invariant electronic layout with temporary manipulation functionality suffice to support programming. The use of electronic presentation techniques to hide descriptions and create focused

views works well. Furthermore, documentation should actively integrate tasks beyond reading, in this case moving code from the documentation to the source files. Unfortunately the evolution aspects of the DJavadoc system (the individual index, see Section I.4.3) had not been utilized in this study and could therefore not be evaluate based on use experience. Evolution was, however, regarded as important but mainly in relation to the integration of material from different library providers (rather than evolving individual navigation indices).

Some weak points in the design of Javadoc tool were also discovered in the study. The placing of method information does not reflect their importance (as being the most commonly used information) and the production of documentation works against the desire to integrate documentation from multiple sources.

I.6 Related Work

There is a relatively limited set of studies addressing reading behavior and reading needs of programmer in relation to software component libraries (in particular with regards to electronic reference documentation). One of the most relevant examples is the Shull et al. (2000) study of reading techniques for object oriented frameworks (similar to class libraries but more advanced and specialized). Among other things, Shull et al. found that example-based learning was well suited for beginning learners. Compared to the study presented in this article, the Shull et al. study is a larger and more thorough but it is also based on student projects and therefore lacks the relevance of an industry setting. Frakes and Pole (1994) produced a study on the subject in relation to component retrieval. They compared four retrieval methods and found no difference in search effectiveness from the different methods. Supporting several methods were also found to be worth wile because no method found all components. Information retrieval, which Farkes and Pole address, is also addressed by Maarek et al. (1991) in a study on retrieval-based construction of software libraries. Maarek et al. illustrates how software libraries can be constructed in relation to the needs of a user through the means of information retrieval methods. On a more general note, Hertzum and Pejtersen (2000) performed another relevant study on the subject of information-seeking practices of engineers with a focus on the search for people to complement documents. The use of people as information sources is relevant in the design of electronic reference documentation. Compared to the work presented here, these studies do not address the reading issue in the electronic medium such as adaptation and evolution.

I.7 Discussion

The DJavadoc study presented in this article, based on real work experience in an industrial setting, provides a foundation for further experimentation and development of electric reference documentation. In this section, central issues uncovered by the study are discussed.

I.7.1 Application-Type Adaptation

The study indicates that programmers do not require adaptation or evolution on an individual level. The flexibility appreciated by the two long-term users had little to do with a redesign of documentation in relation to their individual task. They both use the same settings and never change them. Little reason to further focus on adaptation based on the individual programmer's situation has been found. However, the study was based on a small number of developers (though they had extensive experience) and cannot be considered conclusive on this issue.

Instead of focusing on the individual level, it is relevant to examine adaptation at an application-type level. Applications fall into categories: such as web services, database applications, and 3D games. Documentation could be designed to adapt their content and presentation style to such categories.

I.7.2 Task Integration

To a large degree, the programmers in the study focused on their desire to transfer syntax as well as knowledge from the documentation. The task they were interested in essentially evolved around the task of creating source code from a pool of code resource. To further support this need, an area of investigation is the design of prepared copy/paste strings on a class or method level for object- oriented languages. Development environments, to some degree, handle this issue through code-completion functionality but it can be further extended to provide even more support. Multiple types of cope/paste strings could, for instance, be relevant.

I.7.3 Multiple Source

The study also revealed the need for integration of documentation from multiple sources. An intermediate format is required in the process of generating documentation that allows for integration after release. Javadoc is currently produced in batch and delivers static HTML pages that are difficult to integrate (in particular to keep cross-referencing intact across multiple sources). Using more structured formats such as XML could increase the ease of integration. Fundamentally, this need illustrates the fact that user combines multiple source across library development organizations and that documentation tools should adapt to such a reality. The development of documentation tools should better support the integration of source from organizationally independent development projects with varying degree of detail publicity (e.g., open or closed source).

I.7.4 Using Code as Documentation

Apparent from the study is also the usefulness of code as documentation, for instance, example code and syntactical specification. Often, tutorial material uses code snippets of to illustrate what is described in the text. Code is an explanatory language that can both explain relations and be used directly in programming. Furthermore, programmers have a lot of training writing and reading code files. It is relevant to call further attention to this relation between the code and the documentation. Test code could, for instance, explain the proper use of components.

I.7.5 Communication vs. Documentation

Fundamentally, developers of reference documentation should stop thinking in terms of documentation and start thinking more in terms of communication for library reference documentation. The speed with which libraries change, grow, and multiply makes documentation inherently unstable and commonly out-of-date. Writing documentation before releasing libraries may, in fact, not be necessary in a global, networked development environment. User-driven, just-in-time production of documentation becomes highly relevant; particularly since the user community commonly is much larger than the development team (e.g. Java had 2 million registered Java developers at the Java Developers Connection web site in May 2001 [Nourie 2001]). This makes user involvement highly relevant to overcome a lack of resource, for instance though open-open-source approaches to documentation creation. That this is a feasible approach to writing can be seen from existing online annotated manuals where users today contribute to the writing process by providing comments, see for instance the PHP annotated manual (PHPAM). For a more in-depth discussion on open-source documentation, see Berglund and Priestley (2001). Another relevant alternative is indexing of people rather than text; see Hertzum and Pejtersen (2000).

I.8 Summary and Conclusion

The article presents a study of design criteria on electronic reference documentation for software component libraries. The study was performed in an industrial setting and based on real work, long-time experience of an experimental electronic documentation, called Dynamic Javadoc (DJavadoc). DJavadoc adds individual adaptation and evolution to traditional online reference documentation in the Java programming domain. By adding knowledge about the domain and about programmers' tasks into the electronic documentation, reading can become more efficient.

The study found indications pointing towards a design of electronic library reference documentation without adaptation on an individual level. Instead, documentation should focus on time-invariant focused views that can temporarily be manipulated (to find more detailed sections). Furthermore, the need for evolution is mainly focused on the integration of documentation from multiple sources. Moreover, to support the task of transferring code from documentation to source files was found relevant. Additionally, the study uncovered some weak points in the general Javadoc tool design.

The development of electronic reference documentation that provide efficient reading and coding support for programming purposes has become more relevant and will perhaps become increasingly relevant in the global programming era. The electronic medium provides useful functionality in this area. Studies like the one presented in this article can discover the design knowledge needed to adequately use the electronic medium in the programming domain. Automated documentation systems, such as Javadoc, have shown that it is possible to automate large parts of the documentation process for library reference documentation. This study help fine-tune that automation to produce the right type of electronic documentation.

References

AH, *Adaptive hypertext & hypermedia web site*, `http://wwwis.win.tue.nl/ah/`

AR, *Ada 95 reference manual*, `http://www.adahome.com/rm95/`

Baker M. (1997) *From Document Design to Information Design.* In Proceedings of the 15th Annual International Conference on Computer Documentation, October 19–22, 1997, Snowbird, UT, pp. 7–10.

Basili V.R. (1996) *The Role of Experimentation in Software Engineering: Past, Current, and Future.* In Proceedings of the 18th International Conference on Software Engineering, 25–30 March, Berlin, Germany, pp. 442–449.

Beck K. (2000) *Extreme programming explained: embrace change.* Addison-Wesley.

Bell D., Morrey I., and Pugh J. (1987) *Software Engineering A Programming Approach.* Prentice Hall.

Berglund E. (1999) *Use-Oriented Documentation in Software Development.* Linköping Studies in Science and Technology, Licentiate Thesis no. 790, School of Engineering at Linköping University ISBN: 91-7219-615-7. PDF version online, `http://www.ida.liu.se/~eribe/lic/berglund.pdf`.

Berglund E. (2000) *Writing for Adaptable Documentation.* In Proceedings of IEEE Professional Communication Society International Communication Conference & ACM Special Interest Group on Documentation Conference, IPCC/SIGDOC´2000, Cambridge, Massachusetts, September 24-27.

Berglund E. and Eriksson H. (2000) *Dynamic Software Component Documentation* In Proceedings of the Second Workshop on Learning Software Organizations, in conjunction with the Second International Conference on Product Focused software Process Improvement, June 20 2000, Oulu, Finland.

Berglund E. and Priestley M. (2001) *Open-Source Documentation: in search of user-driven, just-in-time writing.* In Proceedings of SIGDOC 2001, October 21-24, 2001 in Santa Fe, NM.

Brooks F.P. Jr. (1987) *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer, April, pp. 10–19.

Brooks R.E. (1980) *Studying Programmer Behaviour Experimentally: The Problem of Proper Methodology.* Communications of the ACM, 23(4), pp. 207–213.

Brookshear J.G. (1994) *Computer Science An Overview.* Benjamnin/Cummings.

Bruce K.B. (1996) *Progress in Programming Languages.* ACM Computing Surveys, 28(1), pp. 245–247.

Brusilovsky P. and Vassileva J. Eds. (1996) *Special Issue on: Adaptive Hypertext and Hypermedia.* In User Modeling and User-Adapted Interaction, 6.

Carroll J.M. Ed. (1998) *Minimalism Beyond the Nurnberg Funnel.* MIT Press.

Carver D.K. (1969) *Introduction to FORTRAN II and FORTRAN IV programming.* New York.

Davis A.M. (1994) *Fifteen Principles of Software Engineering* IEEE Software, November, pp. 94–101.

Glass R. L. (1994) *The Software-Research Crisis.* IEEE Software. November. pp. 42–47

Glass R. L. (1998) *Reuse: What's Wrong with This Picture.* IEEE Software. March/April. pp. 57–59.

DJavadoc, Dynamic Javadoc home page, `http://www.ida.liu.se/~eribe/djavadoc`.

Frakes W. B. and Pole T. P. (1994) *An Empirical Study of Representation Methods for Reusable Software Components.* IEEE Transaction on Software Engineering. vol. 20. no. 8. pp 617–630.

Friendly L. (1995) *The design of distributed hyperlinked programming documentation.* In proceedings of the 1995 International Workshop on Hypermedia Design.

Hackos J.T. (1997) *Online Documentation: The Next Generation.* In proceedings of 1997 ACM Conference on Systems Documentation, Snowbird, Utah, USA. pp. 99–104

Hackos J.T. (1998) *Choosing a minimalist approach for expert users.* In Carroll J.M. (Ed.), Beyond the Nurnberg Funnel. MIT Press.

Hertzum M. and Pejtersen A.M. (2000) *The information-seeking practices of engineers: searching for documents as well as people.* Information Processing and Management 36, pp. 761–778.

Jacobson R. (1999) *Information Design.* MIT Press.

Java SDK, *Java Standard Development Kit*, `http://java.sun.com/j2se/`.

Javadoc, *Javadoc home page*, `http://java.sun.com/products/jdk/javadoc/`.

JB, *JBuilder*, `http://www.borland.com/jbuilder/`.

JR, *Standard Java reference documentation* (for Java SDK 1.4), `http://java.sun.com/j2se/1.4/docs/api/index.html`.

Kramer D. (1999) *API Documentation for Source Code Comments: A Case Study of Javadoc.* In proceedings of the Seventeenth Annual International Conference of Computer Documentation (SIGDOC'99), New Orleans, September 12–14, 1999.

Krueger C. W. (1992) *Software Reuse.* ACM Computing Surveys. vol. 24. no. 2. pp. 131–183.

Kvale S. (1996) *Interviews: an introduction to qualitative research interviewing.* Sage.

Maarek Y.S., Berry D.M., and Kaiser G.E. (1991) *An Information Retrieval Approach For Automatically Constructing Software Libraries.* IEEE Transactions on Software Engineering 17(8) 800–813.

Mili A, Yacoub S., Addy E., and Mili H. (1999) *Toward an Engineering Discipline of Software Reuse.* IEEE Software. September/October. pp. 22–31.

Mili A., Mili R., and Mittermeir R. (1998) *A survey of Software Component Storage and Retrieval.* Annals of Software Engineering. vol. 5. pp. 349–414.

MM, Mathematica programming language, `http://www.mathematica.com/`.

MSDN, Microsoft developers network online web workshop, `http://msdn.microsoft.com/workshop/`.

Nelson T. H. (1987) *Literary Machines.* South Bend.

Norman D. A. (1990) *The Design of Everyday Things.* Basic Books.

Nourie D. (2001) *JDC Registers Over Two Million Users.* Online Article. `http://developer.java.sun.com/developer/technicalArticles/Interviews/2milmikenoel/`

PHPAM, *PHP online annotated manual*, `http://www.php.net/manual/en/`

Potts C. (1993) *Software-Engineering Research Revisited.* IEEE Software, September pp. 19–28.

PR, *Python library reference*, `http://www.python.org/doc/current/lib/lib.html`.

PYRD, *Python reference documentation*, viewed in November 2001, `http://www.python.org/doc/current/modindex.html`.

Redish J.C. (1989) *Reading to Learn to Do.* IEEE Transaction on Professional Communication 32(4) 289–293.

Reiss S. P. (1996) *Software tools and environments.* ACM Computing Surveys, vol.28 no.1 281–284.

Rosenfeld L. and Morville P. (1998) *Information Architecture for the World Wide Web.* Sebastopol: O'Reilly

Rosson M. B. (1996) *Human Factors in Programming and Software Development.* ACM Computing Surveys vol.28 no.1 193–195

Russ M. L. and McGregor D. (2000) *A Software Development Process for Small Projects.* IEEE Software September/October. pp. 96–101.

Schach S. R. (1997) *Software Engineering with Java.* Irwin.

Shull, F., Lanubile, F., and Basili V. R. (2000) *Investigating reading techniques for object-oriented framework learning.* In IEEE Transactions on Software Engineering, vol. 26. no. 11.

Smart K. L. and Whiting (1994) *Reassessing the Documentation Paradigm: Writing for Print and Online.* In proceedings of 1994 ACM Conference on Systems Documentation, Banf, Canada pp. 6–9.

Sommerville I. (1989) *Software Engineering.* Bath Press.

Sotirovski D. (2001) *Heuristics for Iterative Software Development.* IEEE Software May/June pp. 66–73.

Tichy W. F., Lukowicz P. Prechelt L., and Heinz E. A. (1995) *Experimental Evaluation in Computer Science: A Quantitative Study.* Journal of Systems and Software. 28:9–18.

VA, *Visual Age for Java* (development environment), `http://www-4.ibm.com/software/ad/vajava/`.

van Vilet H. (1993). *Software Engineering Principles and Practice.* John Wiley.

VC, *Visual Café* (development environment), `http://www.visualcafe.com/`.

Wolfram S. (1996) *The Mathematica Book.* Wolfram Media, Cambridge University Press.

Yin R. K. (1994) *Case Study Research: Design and Methods.* Sage.

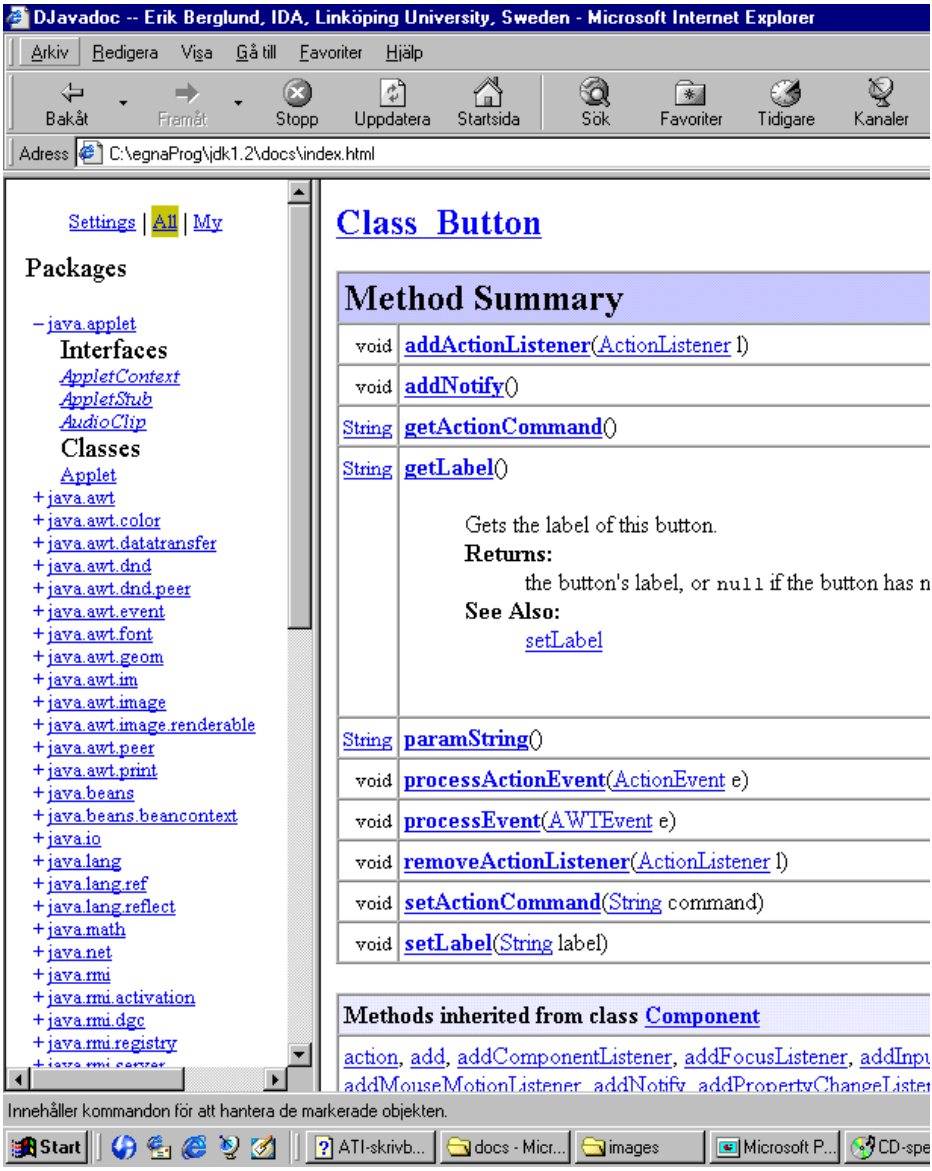

Figure I.4: By direct manipulation, the programmer may temporarily collapse and expand individual section of particular interest. Changes are not remembered for subsequent visits to the same document.

Figure I.5: The DJavadoc Bookmarks represents an individual view of the entire table of contents. Links to documents are saved to and removed from the list by the programmer.

Paper II.
Helping Users Live with Bugs

Submitted February, 2002.

Abstract

Bugs are everywhere: if there is one thing we have learnt over the years this is it. Although, as developers, we try to minimize bugs we must also help users live with the bugs that appear after releases. Today, online bug databases are commonly used to communicate bug knowledge to users. Users from all around the world provide bug reports and collect knowledge in a shared repository. However, online databases do not effectively distribute knowledge to users. Users need a communication architecture that actively distributes bug-related knowledge and presents it when relevant.

II.1 Bugs are Unexpected

The story of the moth stuck on the Mark II computer at Harvard University in 1947 is a well- known tale in software engineering. The engineers described it as the first actual case of a bug being found (Kidwell 1998). This story uncovers a fundamental problem with bugs: their unexpected nature. For users (and developers alike) bugs are not supposed to exist and the Harvard engineers certainly did not expect a real bug when troubleshooting the Mark II. Bugs are among the last things users suspect when things go wrong. Moreover, bug symptoms are not always immediately visible and users may therefore experience considerable costs both in time and money. Furthermore, bugs are difficult to verify if suspected. To confirm bugs, users must devise tests or gain access to the source code to inspect the implementation. Many users are not even proficient enough to complete such tasks.

Internet has completely revolutionized bug handling in the sense that a global user community can accumulate knowledge about bugs. In theory, every user can benefit from the discoveries of other individuals and thereby avoid related costs and problems.

Ultimately, users struggle with the lack of knowledge rather than the actual bug. Once bugs are discovered, users either continue to use systems or stop completely. (The term system is used here in its broadest sense for such diverse systems as software programs, desktop programs, software libraries, media streams and web sites). To help users live with bugs, developers need to help users discover the knowledge which is being accumulated in online databases by other users and developers worldwide. A fundamental requirement in designing usable systems is that systems correspond with users mental models and address the user tasks (Norman 1990, Shniederman 1998, Helander et al. 1997). Consequentially, bug handlings systems must take into account how users act in relation to bugs, how users perceive bugs, and what users need to accomplish in relation to bugs.

This article provides an analysis of post-release bug handling from a user perspective and pinpoints a fundamental flaw in contemporary design: the design for active search. The article is based on a study of 35 contemporary bug-handling systems, presented in detail in the sidebar: "Contemporary Bug Handling". Bug handling starts with the issuing of bug reports and ends with the distribution of solutions to users. Today users are involved in this process, mainly to provide bug reports. Bug databases are also made available online for users. Unfortunately, online databases do not communicate bug knowledge to users. This article explains why and what to do if you are serious about helping your users live with bugs.

II.2 Sharing Bug Knowledge Worldwide

The first and most relevant step to help users live with bugs is to:

Effectively share knowledge throughout the user community.

When someone discovers knowledge about a bug, for instance the existence of a bug or a bug workaround, everyone affected should also be given this knowledge and be able to apply the knowledge at the right time and place. The entire worldwide user community can benefit from the discoveries of individuals, illustrated by Figure II.1C. For the majority of the users however, online bug databases serve only to collect information, visualized by Figure II.1B. Most users are left oblivious to the collected knowledge.

Bug handling systems have evolved from systems intended for internal processes for system developers to systems aimed at both users and developers, illustrated by Figure II.1. However the design has not yet been adapted to the requirements of users. A fundamental difference between developers and users in relation to bugs is that:

Users are passive readers that will not search for knowledge actively.

Many of them will never realize that their problems may be related to a known bug reported by someone else and stored on a distant web site. Even though users may learn that knowledge is available, expecting users to routinely scan bug databases

as a proactive measure is unrealistic. Not even highly mature computer users can be expected to benefit from such solutions (because it takes to much time).

Consequently, to help users bug handling systems must:

> *Actively and efficiently distribute bug knowledge to affected users.*

II.3 Bugs are Communication Deficiencies

The design of post-release bug handling rests on an assumption that bugs are errors. In Webster's Collegiate Dictionary a bug is defined as "*...an unexpected defect, fault, flaw, or imperfection*" (Webster). However, for users bugs represent elements that cause them to fail at completing tasks they believe themselves able to perform. Whether or not the bug is caused by an actual error is of less relevance. From a user perspective:

> *(To users) bugs are communication deficiencies; gaps between the believed nature of systems and the true nature.*

Bugs appear when users have erroneous beliefs about systems caused by implementation errors and inconsistencies, poorly written and erroneous documentation, or misunderstandings. Only informing users about these erroneous beliefs can handle all possible situations. Building a new release of the software to solve the problem does not help if users are not made aware of such releases. Not even automatic distribution of bug fixes will work, sometimes simply because the development organization decides not to fix the bug.

II.4 Tasks, not Software, Fail

The design of post-release bug handling also rests on an assumption that software fails as a result of bugs. For users, however, tasks fail and fixing the problem does not necessarily translate into fixing the software. If a feature in a system is malfunctioning, completing the task may simply require using another feature. Workarounds can be just as good or even better than new software downloads.

To rout out all bugs and release defect-free systems is a software-engineering dream. Methods to support the development of defect-free systems have been created. One example is the Cleanroom software development method (Poore and Trammell 1996). Admirable as this dream is, in reality it can even be argued that the amount of bugs have increased in recent year (not as a result of Cleanroom development, of course). Today it is common practise to make wide-spread, public releases of beta versions via Internet. Yesterday's practice was beta testing by select groups of testers. As a consequence, more users are using unstable software. Perhaps bugs are an unavoidable property of the complexity of software; a property Brooks calls essential (1987). Every single system seems likely to contain bugs and it just a matter

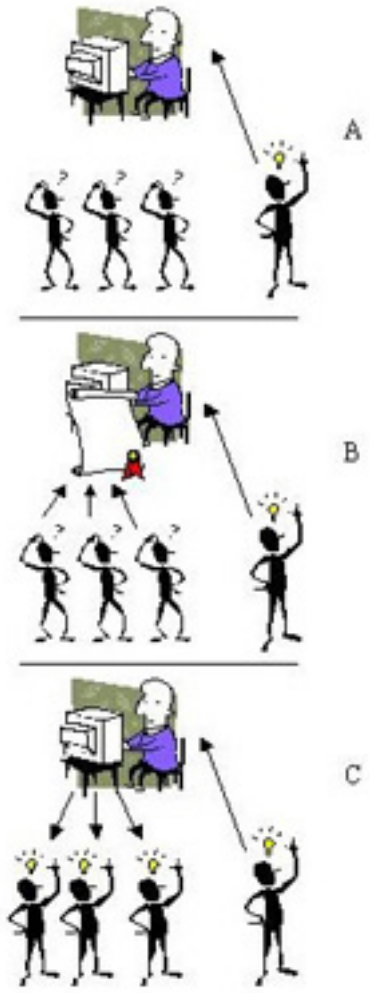

Figure II.1: Originally bug knowledge was collected (A). Today bug knowledge is also often available via the web (B). However, the efficient distribution of bug knowledge requires active distribution and timing (C).

of time before bugs surface unexpectedly somewhere. Fry claims that bugs are created because human memory capacity is limited (1997).

Regardless of why bugs exist, users still have to live with them. In fact, eliminating every possible bug may not be the ultimate goal for users of software systems, in particular for non-critical systems. To a certain degree users may accept bugs as long as the tasks they require can be completed in some form.

II.5 Designing for Passive Readers

Bug-handling systems aimed at supporting users must be designed for passive readers. As such, users will assimilate knowledge only if presented when relevant. Therefore, sending emails with bug reports does not work well even though user receives knowledge passively. Users may be generally interested in bug metrics or extremely critical bugs, but the relevance of individual bugs depends on the situation and must be presented at the right time and place to be efficiently communicated (Suchman 1987). This distinction becomes important in particular, once the communications flow grows beyond the first few bugs. Once individual bugs become relevant, users they still do not need full knowledge, they need *bug signals*.

II.5.1 Bug signals

Singling the existence of accumulated knowledge from a global user community is the first step in helping users live with bugs. Bugs signals, in short, provide:

- *Awareness*: a signal of the existence of the bug.
- *Impact*: a description of the how the bug affects the user. A documentation bug may, for instance, only require the user to read additional documentation.
- *Delay*: an estimate of the time left before a solution to the bug exists. Solutions may be workarounds.
- *Hyperlink*: a hyperlink that users can follow to full knowledge of the bug.

Signals should provide the knowledge needed to make course-of-action decisions. Users must determine how to proceed, for instance study a workaround to complete the task.

II.5.2 System Integration

Bug signals must also be presented at the right time and place to be effectively communicated. One way of achieving this requirement is direct integration of bug signals into systems GUIs (overlaying GUIs with signals) and in this sense continuously updating systems with bug-related knowledge. Consequently, normal work procedures guide users in a passive search of online bug databases. The actions users take will uncover knowledge relevant to them.

On first appearance, system integration of bug signals sounds like a job for an intelligent agent that analyses user behavior and submits signals whenever a bug-related action takes place. It also sounds like another annoying "clip". Bug signals

must avoid becoming obstacles to using systems efficiently. The signals should be non-invasive to avoid disturbing users. Consequently signals cannot contain more than a few words or one sentence of information. The use of underlining in the automatic spell checker in MS Word is an example of a non-invasive signaling method. Figure II.2 illustrates how to integrate bug signals into systems. Of course, for highly critical and potentially dangerous bugs, intervening and disturbing the user or even disabling the feature is motivated.

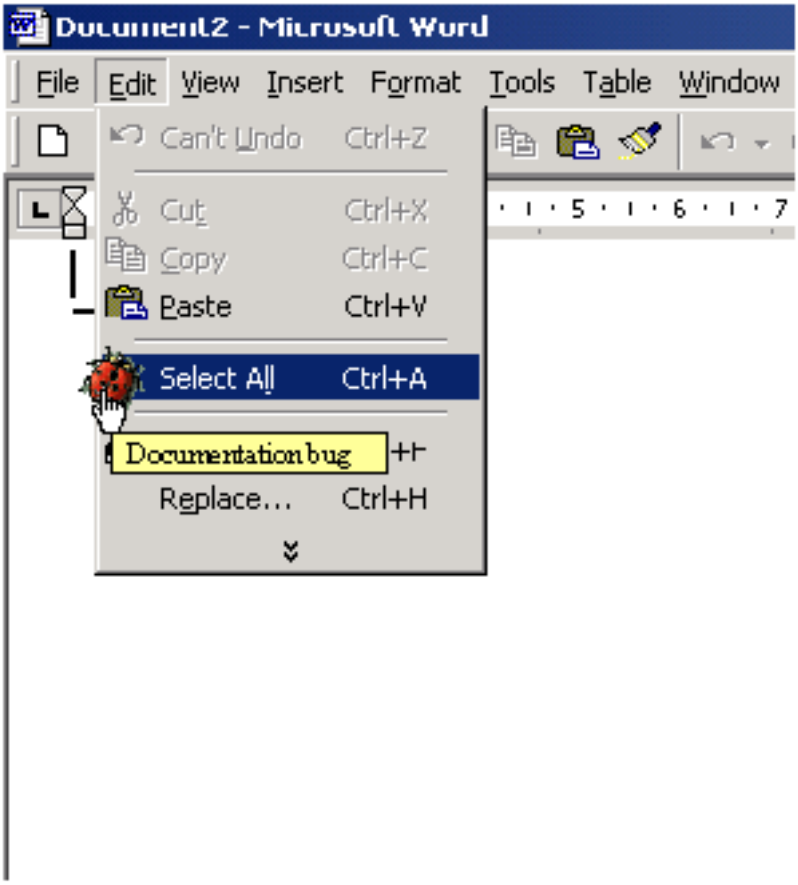

Figure II.2: Bug signals should be directly integrated into applications. Using the application, the user will discover relevant bug knowledge.

II.6 Designing for Active Writers

Contrary to the lack of support for passive readers, the software community acknowledges the user as an active writer. User beta testing is today a common approach product testing. For systems whose target group is software developers the user group is, moreover, highly technically competent and can verify bugs and provide software solutions.

Still, users commonly must overcome unnecessary obstacles to reporting bugs. Users must locate web sites and provide information about the malfunctioning component. Before issuing the report, users are often requested to search the database for similar reports to avoid multiple reports. In this sense, bug reporting is centred on the development organization making it easy for system developers rather than users.

The obstacles in the writing process are not equally critical to the problems of passive readers. However, bug report obstacles stand in the way of users contributing and should be minimized. Application integration for the active writer can remove many obstacles by providing context- dependent interaction with the bug database.

II.7 Open-Bug Communication

In essence, this article is advocating an open communication channel, between users of systems worldwide (on the topic of bugs). The system becomes a messaging system for users rather than a knowledge collection mechanism for the development organization. Finding a suitable implementation for such as design is a delicate process that requires gate keeping to avoid unrelated or malicious use and a non-invasive graphical design to integrate smoothly with users normal work routines. For systems such as software-component libraries for programmers it is highly relevant to include this type of design in development environments or reference documentation. For consumer products, the design may be more far fetched. In general however, the design should address the current problematic relation between users and online bug databases.

II.8 Effects on User Confidence

Why, you may ask yourself, should you draw attention to the bugs that are discovered in your systems. One positive aspect is that an open approach will help users live with bugs. However, users will also see all the bugs that never affect them while working with the GUI.

In "The Pocpcorn Report", Faith Popcorn reports on consumer reactions to problem in relation to bottled water (1991). It is clears from that example that users can excuse bugs, but that they will not be as forgiving towards irresponsible organizations that do not take care of their problems. Of course, users will be disappointed when bug frequency surpasses a certain level. Within limits, however, users will react positively to organizations that treat problem issues openly. In fact, a bug may also represent an opportunity to show good quality customer service. Ask yourself this question: "do you remember companies that have no problems or companies that take care of the few problems that arise swiftly and efficiently?"

Your first gut reaction may be that users are unforgiving and best left oblivious of the inadequacies of the systems they use. However, reacting that way you are also letting the limitations of one system reflect poorly on your entire organization. Helping users live with bugs is a sound business choice for mature companies with a focus on long term relationships.

II.9 Design Criteria on Bug Handling

In summary, to help users live with bugs you should implement the following design criteria:

1. *Bug knowledge should be self- distributing.* As soon as bugs are detected, distribute the knowledge to your systems, illustrated by Figure II.1C.
2. *Bug signals should surface inside systems.* Integrate bug signals non-invasively in your systems, overlaying the GUI with signals or as responses to user actions, illustrated by Figure II.2. Let the signal act as a hyperlink to full knowledge.
3. *Bug signals should help users make course-of-action decisions.* Construct minimal signals that provide the knowledge needed to decide how to proceed (e.g., to ignore the function or to try a workaround).
4. *Bug signals should include the nature of the bug and the bug handling process.* Design signals in relation to the user's task and supply estimates of the delay before solutions become available.

Bug handling is a relevant topic, in particular for users of beta products and early releases but also users in general. Removing obstacles for bug handling increases the speed with which problems and misunderstandings are solved. Helping users live with bugs is good-quality consumer service and may even help promote your company as a responsible, user-friendly organization.

References

Brooks F.P. Jr. (1987) *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer, vol. 20, no. 4. 1987, pp. 10–19.

Fry C., (1997) *Programming on an Already Full Brain.* Communications of the ACM, vol. 40, no. 4. 1997, pp. 55–64.

Helander M., Landauer T. K., and Prabhu P. eds. (1997) *Handbook of Human-Computer Interaction.* Elsevier Science.

Kidwell P.A. (1998) *Stalking the elusive computer bug.* IEEE Annals of the History of Computing, vol. 20, no. 4. pp. 5 –9.

Norman D. A. (1990) *The Design of Everyday Things.* Basic Books.

Poore J.H. and Trammell C.J. (1996) *Cleanroom Software Engineering: A Reader.* Blackwell Publishers.

Popcorn F. (1991) *The Popcorn Report: Faith Popcorn on the Future of Your Company, Your World, Your Life.* Doubleday: Bantam Doubleday Dell Publishing Group Inc.

Suchman L.A. (1987) *Plans and Situated Actions: The problem of human- machine communication.* Cambridge: Cambridge University Press.

Shniederman B. (1998) *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* 3:e edition. Addison-Wesley.

Webster's Collegiate Dictionary (1994) 10th ed.

II.10 SIDEBAR: Contemporary Bug Handling

Bug handling involves both users and developers in contemporary software development. The scientific community has focused solely on debugging and development method in relation to bugs. No work has been reported that treat bug handling as a separate phenomenon. Therefore, I have examined over 35 commercial and open source bug handling systems (see the Tigris web site for a listing of available bug handling system, `http://scarab.tigris.org/bug_trackers.html`). Here I provide an analysis of contemporary bug handling in relation to users. In general, bug- handling systems are not designed for users but for development organizations and their internal bug handling process.

II.10.1 Bug Handling Systems

The following categorization of bug handling systems describes the state of the art today:

- *Stand-alone bug-tracking systems*: provide support for the process of reporting, assigning, and tracking bugs and defects in general. These systems are developed as stand alone products that are not integrated in to a larger development environment. Many systems have web interfaces to facilitate distributed bug handling. Using a stand-alone bug-tracking system is particularly relevant for distributed groups that use multiple platforms. Examples include: BugBase, BugCentral, Bugzilla (used, for instance, in the Mozilla project, in the Apache web-server project, and in the Red Hat linux project), Debian bug tracking system, elementool, FogBUGZ, GTBug, TrackWise, and tTrack. A relevant example is the Debian bug tracking system that uses email for the distribution of bug knowledge and thereby enables passive distribution. However, bug mails are sent on a regular basis rather then when relevant to particular users.
- *Integrated stand-alone bug-tracking systems*: provides stand-alone bug tracking but also provide limited integration with other work tools. Integration, in these cases, constitutes accessibility from other tools or the use of a common underlying information structure. However, these tools are not completely integrated into development environments. These types of bug-handling systems are relevant particularly for groups that perform development within a specified framework. Examples include BugTalker, DevTrack, ClearQuest, Tigris, Visual Intercept, and Whups.
- *Online project management systems*: provide stand-alone bug handling integrated with general web-based project management systems. In these systems,

bug handling is integrated in online project management systems that are accessed via the web. The usefulness of these online development platforms depends on the security they provide and the functionality of the management site in general. Examples include Devx and Source Forge (28,000 projects with over 270,000 registered members, October 2001). SourceForge provides limited support for active distribution of general-bug metrics. However, SourceForge does not support any system integration of bug-related knowledge.

Most organizations still communicate their bugs to users via web pages. Generally speaking, such solutions are covered by the other categories with regards to functionality and content.

II.10.2 Bug Knowledge

The systems examined collect the following knowledge about bugs as part of their bug-handling process:

- *Bug state*: describes different steps that bugs go through in the bug-handling process (e.g. assigned, in verification, in testing, in implementation, and fixed).
- *Bug result*: refers to the outcome of the bug handling process (e.g. fixed, will not fix, is not valid, duplicate report).
- *Bug nature*: represents the bug type (e.g. code errors, issues, design, cosmetic, and misunderstandings).
- *Bug ordering*: represents priority among multiple bugs in the bug-handling process (a number range is often used to define ordering).

For users, bug knowledge represents something different. First, users need to be made aware of relevant bugs. Secondly, they need to make course- of-action decisions concerning what to do next and how to accomplish tasks. For course-of-action, users also need time estimates until a solution will be found or at least until more information is available. Time estimates are needed for the user to determine whether or not to wait for a solution. Unfortunately, the measurement of process time is missing from the knowledge collected in bug handling systems. Thirdly, users may need more detailed knowledge to accomplish their task in spite of the bug, for instance the contents of a workaround. Thus, for users, bug knowledge is:

- *Bug awareness*: the existence of the bug. Awareness also requires timing in presentation, in particular when the number of bugs grows beyond the first few reports. Expecting users to remember more than a few bug reports over time is unrealistic.
- *Bug impact*: the type of impact the bug will have on tasks. Bugs may be critical and thereby render it impossible to perform the desired task. On the other hand, bugs may also be non- critical and only result in a minor flaw that a user can live with. A documentation bug may only require the user to read new documentation to understand how to complete a desired task.
- *Bug delay*: the time remaining before the bug has either been dealt with or at least until more information will be available. Users need to know how long they can expect to wait for a solution.

- *Bug Knowledge*: the full collected knowledge concerning the bug, collected by users and developers worldwide. Generally this constitutes the content of online databases today with exceptions for internal bug-handling data such as information regarding who has been assigned to handle a bug. The knowledge may also need to be expressed with the users intention in mind, focusing on overcoming problems and completing task rather than on describing problems and finding source code solutions.

II.10.3 Distribution Initiative

Generally speaking, the distribution initiative is in the hands of the user, who must actively search bug databases. In some cases, users may subscribe to email services that distribute bug knowledge on a regular basis. Email distribution, however, is not really different from online databases apart from the fact that the user need not continuously check for updates on their own.

Some of the systems integrate with other tools. Potentially these tools could integrate bug knowledge into their GUIs. However, no example of direct bug-knowledge integration exists. Users must still access the bug knowledge directly.

II.10.4 Contemporary Bug-Handling Model

In summary, the contemporary model of bug handling is focused on the development organization. Many system do allow users to browse databases, illustrated by Figure II.1B. Knowledge about bugs is only actively communicated to users in a few cases and never directly integrated into systems. The vision presented in this article, illustrated by Figure II.1C, is not represented by any of the systems.

Users need help to live with the bugs that appear after the release of systems. Helping users live with bugs is not the same as handling bugs in the development process. Even though the ultimate solution to bug handling with respect to users is an open bug communication channel through systems, organizations can move towards a bug handling process that supports users. The sidebar "HOWTO: take the first step" illustrates how.

II.11 SIDEBAR: HOWTO: take the first step

The ultimate solution to helping users live with bugs (advocated in this article) requires direct communication of bug signals, preferably though non-invasive integration of bug signals in systems. However, this solution may be too costly or too time consuming to start off with, as it may require a complete work over of your systems. This sidebar shows how you to take the first step towards the goal of providing efficient distribution of bug- related knowledge to users.

II.11.1 Attitude Changes

- Realize that the vast majority of users are likely to have limited means of discovering bugs on their own. In most cases, even users that have technical

skill and competence will behave like users in general, simply because they do not have the time or interest to search actively.

- Recognize that bugs are unexpected and that users will search for bugs only after having suffered problems trying to get things working. Few users will suspect bugs at first and look for an error in their own understanding or use of the system.
- Recognize that users suffer from a lack of knowledge rather than the actual bug. Though bugs in critical systems can have disastrous effects, many bugs can simply be avoided.
- Recognize that users do not have the time to search for bug knowledge and need to be actively supported in finding what is relevant to them.
- Recognize that users initial need is for knowledge to help them discover bugs and to determine their course-of-action.

II.11.2 Preparing your Bug Handling Process

- From your internal bug-handling process, collect information about delays in the process (time estimates).
- Develop automatic download features in your systems that enable downloading of bug knowledge from the web.

II.11.3 Providing Solutions Fast

- Tell users about bugs: this is the first solution.
- Focus on solutions that do not require changes to the system, that is, workarounds or directions to alternative features. This will provide solutions faster.
- Include links to your bug database from within the system, typically under a help menu. Make sure that identification of both the system and version are included in the link.
- Consider including updates to the system GUI about the amount of reported bugs even though you do not include overlaid bug signals.

II.11.4 Bottom Line

The fundamental issue to help users live with bugs is to distribute knowledge. Rather than trying to develop bug free systems, focus on informing users and on dealing with bugs swiftly.

Paper III. Open-Source Documentation: in search of user-driven, just-in-time writing

Co-authored by Michael Priestley IBM Toronto Lab, Canada, email: mpriestl@ca.ibm.com

Published in the proceedings of SIGDOC 2001, October 21– 24, 2001, 2001 in Santa Fe, NM

Abstract

Iterative development models allow developers to respond quickly to changing user requirements, but place increasing demands on writers who must handle increasing amounts of change with ever-decreasing resources. In the software development world, one solution to this problem is open-source development: allowing the users to set requirements and priorities by actually contributing to the development of the software. This results in just-in-time software improvements that are explicitly user-driven, since they are actually developed by users.

In this article we will discuss how the open source model can be extended to the development of documentation. In many open-source projects, the role of writer has remained unchanged: documentation development remains a specialized activity, owned by a single writer or group of writers, who work as best they can with key developers and frequently out-of-date specification documents. However, a potentially more rewarding approach is to open the development of the documentation to the same sort of community involvement that gives rise to the software: using forums

and mailing lists as the tools for developing documentation, driven by debate and dialogue among the actual users and developers.

Just as open-source development blurs the line between user and developer, open-source documentation will blur the line between reader and writer. Someone who is a novice reader in one area may be an expert author in another. Two key activities emerge for the technical writer in such a model: as gatekeeper and moderator for FAQs and formal documentation, and as literate expert user of the system they are documenting.

III.1 The Problem

Over the years, the software industry has accepted that changing requirements are simply part of the software development process. An allowance for client requirements change, even an expectation of change, is at the foundation of most software development methodologies. The Rational Unified Process (RUP) illustrates this, and Extreme Programming (XP) exemplifies it. Taken to the extreme, as it often is in open- source development, the functionality of the product may not be determined until the day it is completed.

Continuous requirements change makes traditional methods of software documentation difficult. Measured from the last change, production lead-time is effectively nil. While some projects do incorporate documentation requirements into their production schedule, in many cases writers simply have to make the best of an impossible situation, and produce what documentation they can under the circumstances.

Writers cannot simply adhere to a pre-existing plan: they have to quickly assess the relevance of each change and assign priorities to each affected area. Throwing more writers at the problem is a solution with a rapidly diminishing return on investment: more writers typically require more coordination and planning, not less, and this compounds the risks posed by a volatile information domain. The problem cannot be solved with more planning or more reviewing. The writer simply has to make the most of what resources are available, and aim to produce something useful at the end of it.

Applying software development methods to the writing process may sound like a plausible solution to the problem (Utt and Mathews 1999). However, the solution falls short when documentation departments lack the resources and influence that would allow them to negotiate changes after the manner of development departments. While process, and especially integration of process (Priestley and Utt 2000), can help writers track changes, it doesn't help them find the resources or time to make changes. Application of processes and integration of processes provide only half the answer: they provide knowledge, but not the opportunity to apply it.

So the problem, finally, is that when we have the understanding, we have it too late; and regardless of how well we plan or how hard we work, the best we can hope for is an incomplete manual and help set that have a minimum of errors.

III.2 The solution

There are various ways to address this problem, innovations in how we write (in small reusable units), how we process (using various singlesourcing technologies), and how we ship to the customer (incrementally over the web, through a knowledge base, and so on and so forth). These solutions are useful, and make the most of what resources are available.

But a bolder solution is to simply accept that what we are shipping is incomplete, that documentation is in fact inherently incomplete, and then move on to the larger problem: how can we provide our customers with the answers to their questions?

Software documentation has been trending to the minimalist for quite some time. As software becomes more usable, it often picks up document-like attributes (from GUIs to embedded text to wizards), and becomes to some extent self-documenting, lifting some of the burden of completeness from the documentation. There's no need to document the obvious: when the software is self-explanatory (would that it were more often), the documentation can afford to be mute.

Unfortunately, as the same explanation will not serve all users, the same piece of software may be self-explanatory for some and completely opaque to others. This would seem to put the burden of completeness back onto documentation: even if a feature is obvious for one user it isn't for all, therefore document all features. While this conclusion is valid enough when we consider documentation as a static, published entity (something produced with the product for the product), the situation becomes more complex when we think of documentation as a networked and evolving entity, a larger world of information resources in which static documentation provides only a starting point.

In other words, shipping incomplete documentation may be acceptable if the information gaps can be filled in some other way, after the shipping date, as the answers become needed. This is a step beyond print-on-demand, to write-on-demand. Such user-driven, just-in-time production of content would also strengthen relevance in content production and foster communities building on a global scale.

How would write-on-demand processes work? User-driven, just-in-time documentation depends first on the availability of a community of users who can request and receive documentation. You cannot provide the answers without the ability to hear the questions. Users may be prepared to wait for an answer, if they know one is forthcoming. Further, a user may be prepared to collaborate in the answer, providing parts they do know if only to help speed up the writer's research time. In fact, herein lies the heart of our solution. The burden of completeness is derived from the fact that different users require explanations of different features: obviousness is subjective. But this same fact in a networked world implies the opposite: for every user who is confused by a feature, there is another user who understands it and can explain it. The corollary of partial confusion is partial understanding. The users themselves can fill in the holes. In fact, this is how mailing lists and discussion forums work. The role of the writer, in a situation like this, is to be in effect a sort of super-user: someone who is articulate and knowledgeable and regularly available to the community.

In software development, there is already a methodology that is based on such processes: open source development. In recent years, the open- source approach to software development has resulted in notable success stories: Linux (Linux.org), Mozilla or Netscape (Mozilla), and the Apache web server (over 50% of the market)

(Apache, Netcraft) are all large, global products in fast moving technical areas. Open-source development, in its purest form, is an ecological process with a focus on user-driven just-in-time production of content. The community develops what it needs when it needs it bad enough. Software grows from the needs, desires, and work of the community. Given the success of open-source development as a response to these problems in software development, it may be worth considering how the same methodologies can be applied to software documentation.

III.2.1 Open-source documentation and technical writing

In this paper we will discuss open-source documentation as a user-driven, just-in-time documentation process that delivers the documentation users want when they want it. In a sense, open-source development of documentation is practiced continuously today. Evolving content in mailing lists and FAQs are both the result and fodder for ongoing discussions that help develop a community's understanding of software products. Mailing lists and FAQs represent *technical debate* in user communities, which both answer questions about products and also discuss future development of products.

This paper addresses how technical debate can be turned into formal support for software products. We present an open-source documentation method focusing on debate and dialogue as the engines of content creation. Content extraction and debate moderation are also regarded as means for directing and transforming the tacit knowledge of the group into the explicit support for a technology. We will also address contemporary technical writing techniques in relation to the vision of open-source documentation, and discuss the changes that open- source documentation processes may bring about for the writing profession.

III.2.2 Organization

The paper is organized as follows. Section 3 analyses and describes open source development from the experiences of open-source software development. It also describes why open source development results in just-in-time, user-driven production of content. Section 4 provides a framework for open-source documentation projects and discusses how to achieve documentation through user contributions. Section 5 examines writing techniques in search for open-source processes. Section 6 discusses the state of the profession on open-source documentation projects. Finally, Section 7 summarizes the paper and discusses whether open-source documentation would work.

III.3 Open-Source Development

Open-source projects have received a fair bit of attention in recent years, with successful projects such as the Linux (Linux.org) operating system, the Apache web server (Apache), the Mozilla web browser (Mozilla), and the Perl and Python programming languages (Perl, Python). According to the open-source initiative (OSI), a non-profit corporation dedicated to managing and promoting an open-source definition:

> The basic idea behind open source is very simple. When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.
>
> – Open Source Initiatie web site (OSI)

Many open-source projects are developed as freeware but this is not a necessity of open-source projects. Though open source has its roots in freeware initiatives such as the GNU projects (GNU Software) of which the Emacs editor (GNU Emacs) is the most famous application (Stallman 1999), open source does not necessarily mean non-profit.

> Teaching new users about freedom became more difficult in 1998, when a part of the community decided to stop using the term "free software" and say "open-source software" instead.
>
> "Free software" and "Open Source" describe the same category of software, more or less, but say different things about the software, and about values. The GNU Project continues to use the term "free software," to express the idea that freedom, not just technology, is important.
>
> – Stallman 1999

The OSI definition of open-source does not exclude sales of open-source products, in fact it specifically mentions sales. It is the control over the source code that is key to the open-source certification that OSI provides (OSI2).

The OSI certificate protects the source code's ability to move freely though different development projects. This gives the potential for a critical-mass effect, in which the efforts of many globally distributed independent groups with different goals jointly develop software that is more powerful than anything they could have developed individually. In a sense, the software becomes a completely independent entity, which can grow and evolve in directions its original developers never envisioned. According to Bruce Perens, who wrote the original draft of the OSI definition for the Debian open-source project (Debian), the definition is a bill of rights for the computer user. Certain rights are required in software licenses for that software to be certified as Open Source (Perence 1999). Essentially the right to:

- Make copies of the program, and distribute those copies
- Have access to the software's source code, a necessary preliminary before you can change it
- Make improvements to the program

While this bill of rights adequately defines when software is open source (and amenable to open-source development), it does not really describe the nature of open source development. For instance, the famous open-source projects such as Linux, Mozilla and Apache have had large and organizationally independent groups contributing to the same development. How such groups can cooperate, and how a community with a range of involvement from individuals to companies can organize itself, are aspects that are not covered by the OSI definition.

Open source is often described as massive parallel development (Feller and Fitzgerald 2000, Raymond 1999a, Raymond 1999b, Sanders 1998). Furthermore, open source is often connected with individuals working together in a highly decentralized organization. The primary technological drivers for open source software include the need for more robust code, faster development cycles, higher standards of quality, reliability and stability, and more open standards/platforms (Feller and Fitzgerald 2000). Robustness is also one of the established benefits of open source (Willson 1999, Perkins 1999). Perkins writes that it is, in fact, the decentralized organization that helps the open-source community to consistently produce powerful, robust, useful software solutions (Perkins 1999). From a research perspective, open-source is a new but relevant area of investigation. The 1:a workshop on open-source-software engineering was held at the international conference on software engineering (ICSE) 2001, which hopefully will result in more research on the subject (Feller et al. 2001). One of the few in-depth analysis of open–source can be found in Feller and Fitzgerald's framework analysis of open source software development (Feller and Fitzgerald 2000). Furthermore, the book *Open Sources: Voices from the Open-Source Revolution* provide articles written by key figures in the early days of open source (DiBona et al. 1999).

The nature of open-source development still remains somewhat uncharted territory but is typically (among other characteristics) robust, public, just-in-time, user-driven, global, community-oriented, critical-mass dependent, non-directional in its growth, developed from the bottom up, and change-prone. We will elaborate on two aspects of open-source development: *user-driven* and *just-in-time*. The strength of these aspects is the focus they naturally put on relevance and priority. What gets built is what the users want when they want it bad enough.

III.3.1 User-driven

In many cases, open-source development is driven by demand for the product in the programming community itself (Vixie 1999). Users develop the systems they need or want themselves. As such, open-source development can be viewed as an ecological process, in which independent users jointly grow their desired systems. In this its purest form, open-source users are open-source developers. This approach makes the most sense for projects that are relevant to large groups of people, because small groups cannot generate the hours to develop a major system. The basis for open-source development is massive parallel development. [27, 5] Also, open-source projects can be utterly decentralized where no authority dictates what who shall work on and how. Still tremendous organization and cooperation emerges. [Perkins 1999]

Of course, to grow substantially from the efforts of a user-community an open-source project must generate a critical mass of developers that contribute. This is what successful projects, such as Linux and Apache, have done. Also, the critical mass

of users must be competent enough to understand and contribute on a highly detailed level – for instance system administrators – and as a result their needs will shine through in the software they produces. Which explains why, in the past, opens source projects mostly have been focused on operating and networking software, utilities, development tools, and infrastructure components (Feller and Fitzgerald 2000).

Of course, for products such as Linux the majority of users will, if the projects is successful, eventually be users in the traditional sense that do not add to the functionality of the code or even have the ability or intent to contribute. However, the open communication channels used in open-source communication (mailing list and web sites) still broadcast information and discussions to the world. Development is open also to those not directly involved and they may participate to lobby for functionality they need.

III.3.2 Just-In-Time

Open-source development can be considered just-in-time development because the users develop what they want when they want it bad enough. Of course, skeptics may argue that open-source development is mostly technically driven (and support technical desires rather than user needs) because people with technical skill define requirements by implementing them. However, in many open-source projects where the users are in fact technical people (for example, Perl and Apache) these distinctions become meaningless: technical desires are, in fact, the user needs.

Open-source projects are defined by very short release cycles (Feller and Fitzgerald 2000). According to Eric S. Raymond, one of the smart things Linus Torvalds did was to create an extremely short release cycle. Linus succeeded in getting solid feedback and responding to it in only 24 hours, something thought utterly bizarre at the time (Raymond 1999c). In this sense, Linus was also sensitive to requests in a just-in-time fashion and provided his community with rapid responses to their interest in the Linux project. So even when users are not implementing features themselves, the short cycle times and community involvement that typify open-source projects still provide just-in-time development.

III.4 An Open-Source Documentation Framework

Just as open-source development requires a framework through which a community can cooperatively develop code, open-source documentation requires a framework that captures the relevant qualities of open-source development (just-in-time and user-driven development) while accommodating the special requirements of documentation development.

The first step is simply to allow people to contribute, as Jones pointed out in a short article on open source and digital libraries (Jones 2001). Writing cannot be restricted to a privileged few: people outside the organization must be allowed to contribute. This is actually easier to consider for documentation, given that documentation is less dangerous in its possible effects (a badly written document won't erase your hard-drive - at least not directly - in the way software can).

The goal of the framework is to turn technical debate, currently taking place in mailing lists and discussion forums, into formal support for software products. In this section we define an open-source framework which is in subsequent sections matched with contemporary forums for technical debate and current technical writing techniques.

Open-source documentation should perhaps not be seen as text created through an open-source development model but rather as drawing from an accumulated pool of resources, which include both captured competence (text, multimedia) and living (persons) competence. An open-source framework can encourage the creation of these resources, from which a *documentation build* (by analogy to code builds) can create tutorials, standard documents, books, online reference manuals, and so forth as necessary for a particular project or delivery context.

III.4.1 Premises

There are a number of premises that must be met to even start considering open source documentation:

Electronic Documentation

An absolute requirement for open-source documentation is the electronic format. Open source projects must be editable on a global scale and it therefore becomes practically impossible to use print. However, this does not mean that the layout should exclude printable versions of the documentation because users will still want to print documentation. Hard-copy versions may, of course, be constructed from documentation builds.

Web-Site Driven

Since documentation source needs to be accessible to a global community of users, web sites are the logical organization and access mechanism. The easiest way to get started is to run your web site on a SourceForge server (either on the international SourceForge server at `www.sourceforge.com` or on your own downloaded copy of it[SF]). This provides a good starting point for managing your source via the web.

Open-source documentation License

Letting go of control requires the definition of a license over ownership of the open-source documentation and the ability to freely use the documentation source in documentation builds. The documentation source must be free to become part of many different projects. This includes allowing others to make documentation builds from the documentation source and even create new products from that pool. Without an open-source documentation license, there is less incentive for diverse groups to contribute to the effort, and little chance of achieving the necessary critical mass of contributors. Explicit open-source documentation licenses are also needed because the copyright applies to work regardless of medium and without copyright notice. For a discussion on copyright see the Stanford Copyright and Fair Use web site (SCFU).

Open-source documentation license do exist today, among which GNU Free Documentation License (GNUFDL) and the Open Content License (OCL) are the most commonly used for open source projects. These licenses allow distributions of verbatim copies and derived work under certain conditions. For instance, the GNU Free Documentation License allows distribution as long as the distributed copy also use the same license.

> We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does.
>
> – GNU Free Documentation License (GNUFDL)

Documentation Splits

Along with the open-source documentation license comes an acceptance of the possibility of branching projects. This allows fundamental disagreements in a community to be resolved through splitting the community and creating a new version that is maintained in parallel with the original.

First Prototype

The open-source documentation project must start with a small first prototype that jump-starts the process and makes it believable that the project will result in something valuable and worthwhile. This first prototype, and the first documentation build, are, in practice, the sales pitch for the project. The prototype must not be complete but rather make it believable that a relevant result can be produced. Subsequent documentation builds do not have to match the vision of the first prototype: its purpose is to provide a departure point, not an end-point for development.

III.4.2 On-Going Support

Once the project gets started there are a number of aspects that need special attention to keep the project running smoothly:

User Control

As in all projects, the quality of the content needs to be regulated. Control is a social issue in open source development, in which the community regulates itself (Jones 2001). Typically the community grants certain serious and dedicated users special rights that allow them to review contributions and disallow illegal or inappropriate submissions. Naturally the writing staff will be among such power users but people outside the organization must also be allowed to regulate content.

Social Structure

User control requires the construction of a social structure for the members of the community. Assignment of power-user status can be based on engagement, seniority and peer ranking. Many open source systems, such as Source Forge, use peer ranking. The Source Forge ranking system measures teamwork/attitude, coding ability, design/architectural ability, follow-through/reliability, and leadership/management. Social ranking has other advantages as well. For instance, social ranking acts as recognition of contribution and as rewards. Furthermore, social ranking organizes users in relation to their capacity and therefore also organizes users into resources. Social structures also support the feeling of a community.

In a documentation project, coding could simply be replaced with writing as a ranked competency. However, in a mixed project (where the documentation is being developed alongside a particular piece of software), it would be worthwhile to define separate measurements for writing and for information design/architecture, to allow meaningful rankings of good developers who are poor writers and vice versa.

III.4.3 Goals

Building Documentation

The focus of the open-source documentation project should be to build documentation of more traditional style, such as user guides and how- to documents. The documentation source should not be regarded as documentation in itself. There is a risk in open-source documentation that web-based information repositories similar to article collections replace documentation. Such repositories are likely to spread information around and make reading difficult by requiring the reader to perform extensive search and content extraction.

Short Release Cycles

Documentation should have short release cycles to accommodate the flow of requirements and implementations, such as questions and answers. Short cycles are not just good service, it is a necessity for the continuous accumulation of content. Short release cycles is another requirement for user-driven process because large development resources are required. Such a design will require the constant build of documentation from the documentation source perhaps even every 24 hours. In this sense, letting go of control is essential because the task of gate keeping a large documentation source within 24 hours requires manpower and trust. A power-user social structure helps appoint trustworthy users that can change with little or no intervention.

Live Communication Forums

An aspect of documentation creation that differs from code creation is that live, people-to-people communication can become an integral part of the process. Chats with power-users, people who are particularly knowledgeable, can be held and recorded as part of the actual documentation-source. Web-cams can also be utilized to provide live feedback that can also be collected and stored. Such live content transmissions also help build the sense of a community.

Automatic Correctness Verification

In open source software projects, a compiler is often used to verify that only syntactically correct programming is added to the common resource pool. Code that does not pass compilation is not accepted. Beyond compilation, verification is provided through the massive parallel development inherent in open source. Similarly, open source documentation projects could have a number of automatic checks on content, including DTD validation for XML or SGML source, HTMLTidy reports for HTML and XHTML, spellchecks, linkchecks, and so forth.

Writing by Moderating

The technical writing staff responsible for the open source project should take care of moving content around, improving language, correcting errors, identifying gaps, and so forth rather than concentrating solely on writing the content. This staff must also write the first documentation prototype.

Discussion through Annotation

Discussion forums and mailing lists are typically organized chronologically and by subject ("threads"). Documentation, however, needs to be based on topics or tasks, organized into FAQ documents. The transformation from chronological and thread-based organization to more architected FAQs, and the rechunking from threads to topics and tasks, is a core concern of the documentation project.

Traditionally this has been done by hand, through either cut and paste or more complete rewriting. A more dynamic solution might be add metadata to the threads, allowing for more intelligent searching of archived discussions. However, this approach only allows for search-based exploration of relevant topics, and requires constant updating of the metadata. A more integrated solution would be to directly annotate the text in each message, calling out explicitly what part of it is a query and what part is an answer. Query lists can, naturally, be generated from the source for users talented enough to answer them for the writing staff. Answers that have already been provided by the community can be assessed according to the ranked skill of the author, and edited if necessary by posting the edited answer to the end of the thread.

Discussion through annotation naturally adds user comments and discussion to a topic framework, unlike thread-based discussion, which require transformation. In this sense, annotation speeds up content-extraction process and thereby shortens the release cycles for documentation builds.

Multiple Views

A big part of documentation is navigation and as the documentation source grows the navigation problem grows. Navigation is also personal or task dependent and it is therefore difficult to generate a general but effective index. Multiple indices, however, can exist and this may well be one of the larger sections of an open source documentation project. By allowing the construction of navigational links across documentation based on user design the navigation infrastructure can evolve and grow with time.

III.4.4 Technical Questions

There are a number of technical issues that need to be addressed by the open source documentation framework:

Documentation Format

The web infrastructure and the openness make the technical issue difficult. The need for a web-site driven project, the formats usable become somewhat limited. For annotation systems (i.e., direct additions to the documentation source) the system must work directly in the browser. This requirement makes XML a highly relevant documentation format because the basic web infrastructure supports XML. However, automatic spell correction needs to be present as well which may make things a bit more difficult today. For longer comments, individuals can be free to use whatever word processor they like to construct their answers as long as they can convert to the project format.

Documentation Layout and Author Reliability

The layout of a documentation system that includes questions and answers from the user community needs to show the reliability of content. At least, the system should clearly indicate that the source is open for contribution from a worldwide community allowing participation from, in principle, anyone with web access. The annotated manual for the PHP open source project does this in two ways: by calling the manual annotated and by displaying annotations from users in differently styled sections of the text (PHPAM). Readers need to be made aware of who the writer is and their degree of competence.

III.4.5 Lifecycle

Initially, a documentation prototype provides the starting point for contributions from an open-source documentation community. As the project progresses, more and more of the content may be derived directly from the community, following a process of content creation and documentation builds can be summarized by the following lifecycle:

1. A user asks a question, either about existing content or by requesting information. The question is added to the source as a comment or as a new question.
2. Another user (may be a member of the writing staff) finds the question in some build from the source, perhaps a query listing or as part of a documentation build. The user answers the question and the answer is added to the source.
3. Other users provide answers, confirms answers or, adds comments and reposts to the source as an annotation.
4. Another user with editorial skills reworks the answer to and reposts.
5. The answer is automatically picked up in the next FAQ build, although ranked fairly low since it has only been asked once. The build may also validate that the FAQ has been correctly authored as a task, has no spelling errors, etc.

6. Another user with information architecture skills adds a reference to the task to pull it into the appropriate place in the overall task flow, and to include it in the appropriate indexes and tables of contents for whichever delivery contexts are appropriate.
7. Someone reads the documentation, has a problem with it, and asks a new question.
8. Repeat until software and documentation are perfect or obsolete, whichever comes first.

Alongside this process, documentation builds are continuously created from the source with layout visualizing the credibility of the different pieces. As a question-answer cycle matures the content become more and more integrated in the documentation by shifting style and location in the builds. Peers rate contributors that increase the status of such users. Automatic rating systems can be built in to the discussion format by measuring the addition of agreement, refinement, or disagreement to answers. For highly rated users, the technical staff investigates whether or not to grant user more privileges to cut corners in the gate keeping process.

III.4.6 Summary of Framework

The open-source documentation presented in this section focus on the creation of a user community that builds documentation by debating topics in a documentation source. From the source, documentation is built by extraction (automated if possible). The layout visualizes the credibility of content in style and position. As content mature through the community process, its visibility in subsequent documentation build releases increase.

Compared to traditional writing, open-source documentation focus on the user-driven, just-in-time aspects of content creation and the natural focus they put on relevance and priority.

III.5 Open Writing techniques

Open-source documentation also requires writing techniques that support the process of user-driven, just-in-time construction of documentation through an open-source model. In this section we discuss what current writing techniques offer in this respect.

III.5.1 Writing Reusable Units

Many online documentation projects currently use topic-oriented writing and information typing as ways to produce disciplined reusable information. Combined with task-oriented minimalism (Carroll 1998), these techniques can result in highly focussed, reusable, and user- focussed documentation. The question is how much of these techniques can be made accessible to a wide community, and can how consistency and accuracy be maintained, outside of the standard edit-publish-review cycle?

While various architectures define a variety of sizes and types of information, at minimum an information-typing architecture defines the size of a topic (a single

reusable "chunk" that describes a single idea, task, or thing) and three information types: concept, task, and reference. Multiple topics can be combined into task flows, organized by index or table of contents, and aggregated into books or websites (Priestley 2001).

Topic-oriented writing can seem quite alien to an accomplished technical writer more familiar with books, and there is often a significant learning curve associated with the change in writing goals and style. However, different as they are from a manual, they are in fact quite a natural fit for derivation from FAQs. Different types of question conform quite naturally to information types: how-do-I questions have tasks as answers, what-is-a or how-does-it-work questions have concepts or reference topics as answers. In addition, with the exception of extraordinarily long or vague questions, most FAQs are going to be naturally chunked at about the right size for a topic.

So is the fit between newsgroup source and topic-oriented, reusable content as easy as the normal gathering process that gives us FAQs? Nearly. Typing and chunking are the two main goals of an information typing architecture, but a website or book constructed out of topics needs coherence in its style and structure to look more than merely accidental, and to be predictable enough to be useful and usable.

III.5.2 Editors and Architects

The task of enforcing structural and stylistic guidelines can be in part taken up by the social structure: appointed or elected editors (users or contributors with highly rated writing and information architecture skills) can be reviewers and approvers of candidate topics. For example, in the case of topics harvested directly from marked-up newsgroup posts (as described in section 4.3.5), an editor could be required to forward the (edited, annotated) answer back to the group before it was considered a candidate for harvesting. Otherwise, contributors with editorial approval could perform the harvesting themselves, and impose a certain level of consistency as they went.

The two proposed skill measurements - writing and information architecture - point to two separate roles: the topic-level editor, who pays more attention to style and low-level content issues, and the collection-level editor, who defines the task flows and tables of contents that organize the topics into useful collections.

These two roles, and their responsibilities in a more structured development process, have been described in detail in (Priestley and Utt 2000).

III.5.3 Enforcing Structure with Markup

Structural guidelines can also be enforced by the use of a specialized markup language, whose DTDs or schemas prescribe particular structures for particular kinds of information. There are several possibilities for enforcing such structures:

HTML or XHTML

HTML is a very general standard, and as a result it does not usefully constrain the information you write in it: two equally valid topics (according to the HTML standard) can be as different as any two pages on the web. This is still better than

complete chaos, however, and tools such as HTMLTidy make it easy to eliminate tagging errors. XHTML is somewhat better, and has the two advantages of being customizable (you can choose which modules you require) and, as part of the XML universe, addressable with XSLT and XPath, which makes it easy to transform and reuse.

DocBook

DocBook is a more specific standard than HTML, and out of the box it is focussed on book authoring. While DocBook provides better validation than HTML or XHTML, and has a good set of output transforms and tools, it is not particular topic-oriented. However, parts of it are highly structured, and could be used for specific domains (such as messages) as-is.

Customized DocBook

Generally speaking, if you want to use DocBook, you will need to customize it. This is a well-documented process, with the warning that if you want to add your own tags (not just choose a subset of the DocBook ones) you'll need to write your own transforms and tools.

DITA

The Darwin Information Typing Architecture is a topic-oriented information typing architecture for writing and publishing technical documentation. Out of the box, it is oriented towards creating information-typed topics (concepts, tasks, and reference), and is quite restrictive in its structures (especially for tasks). However, it is a new and still-evolving architecture, and there are a limited number of transforms available (PDF via FO and HTML are available outputs at the time this paper was written).

Specialized DITA

The good news is that you can create specialized topic types (such as EJB API descriptions, configuration file formats, cooperative tasks, etc.) quickly and easily. Generally speaking, the more closely you tailor your topic's structures and tags to your domain (the particular kind of software you are documenting, for example) the easier it will be to learn (because it matches what the writers are trying to create) and the more it can enforce structural consistency. The more tightly you scope your domain, the more exactly you can define your content rules, and the more precisely you can control consistency, before an editor even gets involved.

III.5.4 Massive Parallel Writing

Topic-oriented chunks written by users, refined by editors and architects, and confined by markup languages can help get contributions right from the start. Users can acquire the writing skills to a certain degree and the ones that learn the most also get the highest ranking and the social structure thereby help produce quality documentation. At some point, however, technique, editors, architects, and markup may not be enough. This is where one of the fundamental points of open-source development

kicks in – massive parallel writing. When writers can read, redistribute, and modify the documentation source, the documentation evolves and become robust. People improve it, people adapt it, people fix bugs (see Section 3). If writing technique fails, open-source documentation will rely on the sheer size of a committed user community.

III.6 Contemporary Open-Source Documentation

Though genuinely open-source documentation cannot always be found even among open-source software projects, there are some documentation projects and communication media that contain the user-driven, just-in-time production aspects we are searching for. Discussion forums, mailing lists, online annotated manuals, online editable manuals, and open-source documentation projects can be considered user-driven and just-in-time, but they do not necessarily conform to other aspects of our framework.

For instance, even when documentation uses an electronic format and is web accessible, it is rarely accompanied by an open-source documentation license. Documentation for open-source software projects often remains proprietary, and resistant to external contributions.

The Linux Documentation Project, as an example, explicitly prohibits open use of the documentation source without written permission:

> Any translation or derivative work of Linux Installation and Getting Started must be approved by the author in writing before distribution. ...
>
> These restrictions are here to protect us as authors, not to restrict you as learners and educators.
>
> – Linux Documentation Project Copying License (LinuxDPDPCL)

While many open-source projects do have a more relaxed approach to copyright and some use clearly open licenses, in reality few members of open-source software projects participate in the development of documentation and the writing staff is a relatively limited group of people. The most open-source documentation projects can be found in the online annotated and editable manuals, for instance the PHP annotated manual (PHPAM), the MySQL commented manual (MySQL), and the Squeek editable manual (SM). These systems allow users to comment on, or in the case of Squeek, directly edit, the documentation. The licensing policy is, however, unclear or closed in these examples, and there is no explicit social structure to aid in assessing contributors' credibility.

Discussion forums and mailing lists provide a high degree of user control, flexibility, and openness to contributions. The members of the community easily participate. User control over content is built in to the submission structure. Release cycles can be very short as answers to questions are posted often within hours. Splits are not uncommon into different strands of continued discussion. Unfortunately, discussion

forums and mailing are lists have difficulty supporting the task of building documentation. Extraction of material into documentation is seldom performed, making discussions concerning topics difficult to track. Search engines do exist for such purposes, but require a common terminology across submissions and support only active search (not passive browsing).

To find really good examples of open-source documentation we have to look at more general projects. A well known example from the software world is Slashdot (`www.slashdot.org`), which has been around since 1997 and where the majority of the work is done by the people who e-mail stories to the site (Slashdot). Slashdot puts a strong focus on documentation development through moderated discussion, but an explicit open-source documentation policy is still lacking and there is little focus on building documentation.

Even more developed open-source documentation projects can be found outside the software world. The Nupedia (Nupedia) and Wikipedia (Wikipedia), globally written encyclopaedias, are examples of projects that develop information using the GNU Free Documentation License and that provide a social structure for writers and editors. In many ways these projects can be viewed as being open-source documentation projects.

In conclusion, many open-source documentation projects today are not really open, even in open source software projects. What is lacking is largely an open-source documentation license policy, explicit social structures and documentation builds. To a certain degree human resources are also lacking: for instance, open source software projects have not really focused their resources on documentation. The strongest existing examples are general in nature and are not concerned with producing documentation for specific software systems or development projects.

III.7 Would It Work

In this paper we have discussed open-source development as a production model that results in user-driven, just-in-time content. We have provided a framework for open-source documentation projects that illustrates what aspects of development need to be taken into account. Furthermore, we have examined open writing techniques and the current state of the profession in real open-source documentation projects.

Open-source documentation may well be an attractive method for user-driven, just-in-time production of documentation, in particular seeing as much of the production is performed free of charge. However, that does not mean it will work. The fact that most of the software needed for handling open-source documentation projects already exists for open-source software development is advantageous. However, documentation has its own problems that do not exist in the software realm. For instance, changing the documentation does not change the functionality of the software, and incorrect content is not as easily caught by compilers and test cases. Greater care and more review may be required for open- source documentation compared to open-source software.

It is also important to remember that the completeness of the open-source documentation project may not be the ultimate goal. Documentation should provide answers to user questions and does not need to totally describe the system. Let's put it another way: the absence of description in an open-source documentation project

may in itself be a source of knowledge. If users do not request documentation for a particular feature, it may be because the answer is made obvious by the design of the interface, or the feature may simply not be used (assuming that users faithfully report their needs). In the latter case, the hole in the documentation may soon have a matching hole in the software! Using an open-source documentation process provides a way to measure areas of use and kinds of interaction, and may therefore be valuable to the development process. As much as users are involved in the documentation process by providing discussion content, asking questions and answering them, they are also providing requirements for tomorrow. What users question and provide answers for can demonstrate what parts of the software they use.

What will become of the writing staff in an open-source documentation project? The writing staff should be dedicated members of the open- source projects. Given that a large enough user community exists, the writing staff would service the writing community with their expert knowledge about the system and help developers articulate themselves. Gate-keeping the production of content becomes a vital task. Furthermore, the writing staff should create documentation by extracting content that passes though mailing lists and discussion forums: FAQs, development documentation and technical manuals. Such content extraction would serve both the documentation and the development process. If fewer users contributed, the writing staff would need to increase their original content production.

Success ultimately depends on the open-source documentation project's ability to accumulate enough users that can and will contribute to the process. Open-source software has shown that it is possible to generate even large applications from the efforts of users. Projects such as Nupedia have also shown that this fact translates to open source documentation. However, smaller projects may have difficulties producing enough user contribution. On the other hand, let us not forget that users definitely can provide questions even when they can't provide answers. In this sense, open-source documentation provide much needed relevance and priority assessments to the documentation process.

References

Apache (open-source web server) `http://www.apache.org`

Carroll J.M. Ed. (1998) *Minimalism Beyond the Nurnberg Funnel.* MIT Press.

Debian (open-source Linux) `http://www.debian.org`

DiBona C., Ockman S., and Stone M. Eds. (1999) *Open Sources: Voices from the Open Source Revolution.* O'Reilly.

Feller J., and Fitzgerald B. (2000) *A Framework Analysis of the Open Source Software Development Paradigm.* In Proceedings of the 21st International Conference on Information Systems 2000, Brisbane pp. 10–13

Feller J., Fitzgerald B., and van der Hoek, A. (2001) *(W18) 1:st Workshop on Open Source Software Engineering, position paper for the workshop.* In Proceedings of the 23rd International Conference on Software Engineering, 2001 pp. 780–781

GNU Emacs (open-source extensible editor) `http://www.gnu.org/software/emacs/`

GNUFDL, GNU Free Documentation License http://www.gnu.org/copyleft/fdl.html

GNU Software (original free software initiative, origin of open-source) http://www.gnu.org

Jones P. (2001) *Open(sourcing) the Doors: for Contributor-Run Digital Libraries* Communications of the ACM vol. 44. no. 5 pp. 45–46.

LinuxDPCL, *Linux Documentation Project copying license*, viewed August 2001. http://www.linuxdoc.org/LDP-COPYRIGHT.html

Linux.org (central source of Linux information) http://www.linux.org

Mozilla *open-source web browser* (development project for Netscape 6, based on the original Netscape source code) http://www.mozilla.org

MySQL *annotated manual* (online annotated manual) http://www.mysql.com/doc/

Netcraft *Web Server Surveys*, viewed June 2001 http://www.netcraft.com/survey/

Nupedia *open-source encyclopedia* http://www.nupedia.com/

OCL, *Open Content License* http://www.opencontent.org

OSI, *Open Source Initiative* http://www.opensource.org

OSI2, *Open Source Initiative definition of open-source* http://www.opensource.org/docs/definition.html

Perence B. (1999) *The Open Source Definition.* In Open Sources: Voices from the Open Source Revolution. Eds. DiBona C., Ockman S., and Stone M. O'Reilly.

Perkins (1999) *Culture Clash and the Road to World Domination.* IEEE Software January/February 1999 pp. 80–84.

Perl *open source programming language* http://www.perl.org

PHPAM, *PHP online annotated manual.* http://www.php.net/manual/en/

Priestley, M. (2001) *DITA XML: A Reuse by Reference Architecture for Technical Documentation.* In Proceedingso of ACM SIGDOC 2001

Priestley, M., and Utt, M. H. (2000) A unified process for software and documentation development Conference Proceedings, IEEE/ACM IPCC/SIGDOC 2000

Python (open source programming language) http://www.python.org

Raymond E. S. (1999a) *The Cathedral & the Bazaar.* O'Reilly. Sebastapol CA, USA.

Raymond E. S. (1999b) *A Brief History of Hackerdom.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

Raymond E. S. (1999c) *Linux and Open-Source Success* (interview) IEEE Software. January/February. pp. 85–89.

Sanders J. (1998) *Linux, Open Source, and Software's Future.* IEEE Software September/October. pp 88–91.

SCFU, Stanford Copyright and Fair Use web site http://fairuse.stanford.edu

Slashdot (open software e-zine) viewed August 2001 `http://slashdot.org/about.shtml`

SF, *SourceForge,* online open-source project web site. `http://sourceforge.net`

Stallman R (1999) *The GNU Operating System and the Free Software Movement.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

SM, Squeek manual *online editable manual* `http://squeak.cs.uiuc.edu/documentation/index.html`

Utt, M.H., and Mathews, R. (1999) *Developing a User Information Architecture for Rational's ClearCase Product Family Documentation Set.* Conference Proceedings, ACM SIGDOC 1999, pages 86-92.

Vixie P. (1999) *Software Engineering.* In Open Sources: Voices from the Open Source Revolution, eds. DiBona C., Ockman S., and Stone M. O'Reilly.

Wikipedia *open-source encyclopedia* `http://www.wikipedia.com`

Willson (1999) *Is the Open-source Community setting a Bad Example?* IEEE Software January/February 1999 pp. 23– 25.

Paper IV.
Writing for Adaptable Documentation

Published in the proceedings of IPCC/SIGDOC 2000, September 24–27, Cambridge, Massachusetts

Abstract

The rapid development of reusable software components results in an *information-overload problem* in the development process. Software developers must read large amounts of documentation. *Adaptive documentation* is one way to address this problem and support efficient reading. However, in our view, adaptive documentation requires a writing process that delivers the pedagogical strategies for adaptivity. In this paper, we take stance in a project on adaptive software reference documentation and discuss the requirements on writing. We also discuss writing trends and Web languages in relation to adaptivity. We conclude that describing change in documentation is not supported on an authoring level but rather on a programming level.

IV.1 Introduction

Reuse-driven software development results in large amounts of software components. An example is the *Java Standard Development Kit* (Java SDK). In the current version of this core Java class library there are about 2,100 classes and the reference documentation is 97 Mbytes in size. It can be argued that the Web has contributed to such growth by facilitating worldwide collaboration and sharing of program components.

Although growing worldwide libraries of reusable software components support the development of software, they can also create new problems. One such problem is the *information-overload problem*, that is the difficulty of keeping up-to-date in a fast-moving area. Learning which components to use and how to use them becomes a central part of software development. Consequently, software documentation becomes an important tool in software development, comparable with source-code editors. The

ability with which the documentation provides efficiency and quality in reading affects the cost and quality of software.

One way to address the information-overload problem is to provide *adaptable documentation*, documentation that changes its content and presentation to better suit the reader. Adaptivity in text has been addressed in the research community, see for instance (Brusilovsky and Vassileva 1996, Beaumont and Brusilovsky 1995, Bra and Calvi 1998, AH&H). Research on adaptive documentation has generally focused on the modeling of the reader and tracking of reader behavior.

The topic of this paper is the implications of adaptable documentation on the writing process. Baker (1997) argues that interactive information products require new authoring and design techniques. Hackos (1997) points out that information written for paper cannot simply be transferred into online form. We take stance in the Dynamic Javadoc (DJavadoc) research project (Berglund 1999, DJavadoc) and discuss requirements on the writing process to achieve adaptivity in documentation. DJavadoc is an adaptable version of the Java class library reference documentation (Kramer 1999). In DJavadoc readers can constantly tailor the documentation to better suit their changing needs. During the development of DJavadoc some forms of adaptation were omitted because the information needed was neither present in the Javadoc documentation nor supported by the Java documentation structure.

The first two sections, provide background on adaptivity and software reference documentation. Continuing, Section IV.4 describes the adaptable documentation of the DJavadoc research project. The design for adaptivity in DJavadoc is described. In Section IV.6, the writing process in relation to adaptivity is discussed in relation to the DJavadoc project. Finally, in Section IV.7, we elaborate on current writing trends and Web authoring languages in relation to adaptivity and in Section IV.8 the paper is concluded.

IV.2 Adaptable Documentation

According to Brusilovsky (1996), adaptive hypermedia systems model their users and adapt various visible aspects of the system according to this model. Kantorowitz and Sudarsky (1989) defined an adaptable user interface as an interface supporting a number of different dialog modes that can be switched among at any time (they also required smooth and natural switching and easy to learn dialogues). Here adaptable documentation is defined as documentation that has the ability to *change* its content and presentation *to better suited forms with regard to internal or external variables*. Examples of such variables include models of the reader, models of groups of readers, models of the information, and browsing history.

One of the advantages of electronic text is the possibility to change it dynamically. The computer provides a general mechanism for change that seamlessly integrates interaction, internal state, time, and so on into the documentation. The computer as a presentation system provides a technological foundation for adaptivity. However, adaptivity is not necessarily a quality of the electronic presentation system. Adaptivity is change in accordance with pedagogical strategies and ultimately a question of information design (Jacobson 1999)

From a writing perspective, adaptive documentation can be viewed as consisting of *change strategies*, *change mechanisms*, and *change constraints* (in addition to the

body of text the documentation contains). Change strategies capture the pedagogical strategies used to guide the change into suitable forms. It is relevant to speak both of evolution and temporal change in this context. Change mechanisms describe how the change should take form, for instance to alternate between more detailed and briefer versions of the same text (see also Figure IV.1). Change constraints regulate the change process and describe in what situations change may not be applied. An example may be to disallow temporal removal of information types.

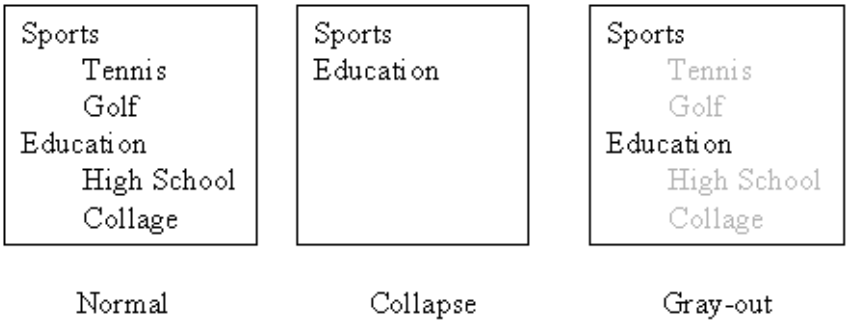

Figure IV.1: Different change mechanisms have different properties. Collapse, for instance, uses less screen space where as gray-out provide information about the size of the missing content.

IV.3 Current State of Software Reference Documentation

IV.3.1 In General

Software reference documentation is still most commonly provided as system-oriented books, online books, or hypertext systems that do not change or evolve. Online reference documentation is available for most languages (see for instance AR, PR, JR). Software documentation has also been addressed in the research community (Knuth 1992, de Olivera Braga et al. 1998, Soloway et al. 1988). The focus has been on the generation of software documentation and the understanding of programs. Our work focuses on software documentation as part of the programming environment and the adaptation of documentation to increase efficiency in reading.

IV.3.2 Javadoc

A widely-used item of software reference documentation is the online Java reference documentation, generated from the source code by the Javadoc tool (Kramer 1999,

JD, Javadoc). In this paper, Javadoc is discussed in more detail because DJavadoc is an extension to Javadoc.

The Javadoc documentation is a system-oriented hypertext documentation listing available software components at class level along with a series of navigational indices. Hyperlinks are used to cross-reference among class documents via parameters, types, return values, written comments, and so on. In our view, Javadoc has become somewhat of a model for online software reference documentation. Javadoc, however, delivers static text that does not change or evolve.

Javadoc can also be viewed as a source-code browser, providing the reader with what is assumed to be a correctly typeset and relevant extract of Java source-code files (including written comments). In this sense, Javadoc is perhaps more of a programming tool than a strict text. Actually it is, in our view, one of the most commonly used Java programming tools.

IV.3.3 Adaptable Software Reference Documentation

There are examples of adaptable software reference documentation. Two such examples are the Microsoft Developers Network Online Web Workshop (MSDN) and the Mathemathica Help Browser (Wolfram 1996). Both these documentation systems have sections that can be collapsed or expanded in the documentation. Both systems also remember whether or not a section was collapsed or expanded last time the reader visited a document. The change strategy seems to be that old changes also reflect future needs. Our personal experience as frequent users of the MSDN Online Web Workshop is the opposite, but this is an empirical question that we have not investigated further.

Development environments, such as Visual Café (VC), may also provide adaptation. In Visual Cafe it is possible to browse the documentation directly from the source code. The source code becomes a use-oriented and evolving index to the documentation (see the Section IV.6.5).

IV.4 DJavadoc: an Example of Adaptive Documentation

DJavadoc is a product of a research project focused on adaptivity as a means of providing use-oriented documentation in the Java domain. This section provides a description of the DJavadoc documentation. However, since hard copy does not adequately reflect change, the interested reader should visit the DJavadoc web site at `http://www.ida.liu.se/~eribe/djavadoc/` (only for Microsoft Internet Explorer in the current version).

IV.4.1 What is DJavadoc

Dynamic Javadoc (DJavadoc) (Berglund 1999, DJavadoc) is an alternative to the official reference documentation of software components developed in the Java programming language, known as Javadoc (Kramer 1999, also described in Section IV.3). The DJavadoc documentation provides the reader with means to create more focused

views by locally and temporarily removing more information from Javadoc documents. The reader can collapse and expand sections and thereby increase the degree of visibility of relevant information. The system is implemented using dynamic HTML (DHTML); that is, script-based manipulation of HTML pages in web-browsers.

Generalizing, DJavadoc is an adaptable documentation based on structural and pedagogical knowledge and change mechanisms. The system allows for local redesign of style and structure. DJavadoc makes Javadoc documentation adaptable by providing a prepared flexibility. The reader is allowed to redefine assumptions made about the appropriate level of detail and organization of the information found in DJavadoc.

IV.4.2 Change Strategy, Mechanisms, and Constraints

In DJavadoc, the goal is to reduce time-consuming, repetitive manual searching for well-defined information types. Javadoc documentation can be viewed as documentation supporting multiple information needs, for instance understanding a component in general or looking up syntactical specifications. Depending on the purpose of the reader different parts of the documentation are relevant. To a certain degree, DJavadoc allows the reader to match style and content with reading purpose and thereby supports experts' reading behavior (Hackos 1998).

The change strategy in DJavadoc is focused on moving relevant information up into the visible space of the browser by temporarily removing excessive information. Readers should also be in control of the change process. For heterogeneous information sources and for inexperienced readers such a strategy may not be appropriate.

The documentation also evolves by allowing readers to create and edit a bookmark index (see Section IV.5). The index represents an extract of the navigational indices in the documentation. In this way the documentation assimilates knowledge over time. Other sources of evolution could be *project indices* created by groups of people, *code indices* based on readers source code, and *history indices* built from tracked reading behavior.

A collapse and expand change mechanism is used in DJavadoc, in principle stretchtext functionality. Color coding and more specifically *graying-out* have also been considered in the DJavadoc project (Berglund 1999). The documentation does not pose any particular constraints other than what is implicit in the information model and the change mechanism.

IV.5 Detailed Description of DJavadoc

In DJavadoc, the reader interacts with the information model, thereby developing views of the documentation. Figures IV.2A and IV.2B show how the reader controls the visibility of different information types by checking elements on or off. As the reader browses the documentation, documents are redesigned in accordance with the view. By direct manipulation the reader may also collapse or expand individual sections but these changes are not stored.

DJavadoc also allows the reader to keep a bookmark list of relevant documents, shown in Figure IV.3. This adapted index is an extract of the entire alphabetic index that evolves during reading (as the reader saves or removes entries). Since there are 2,100 documents in the current version of the Java Standard Development Kit (core

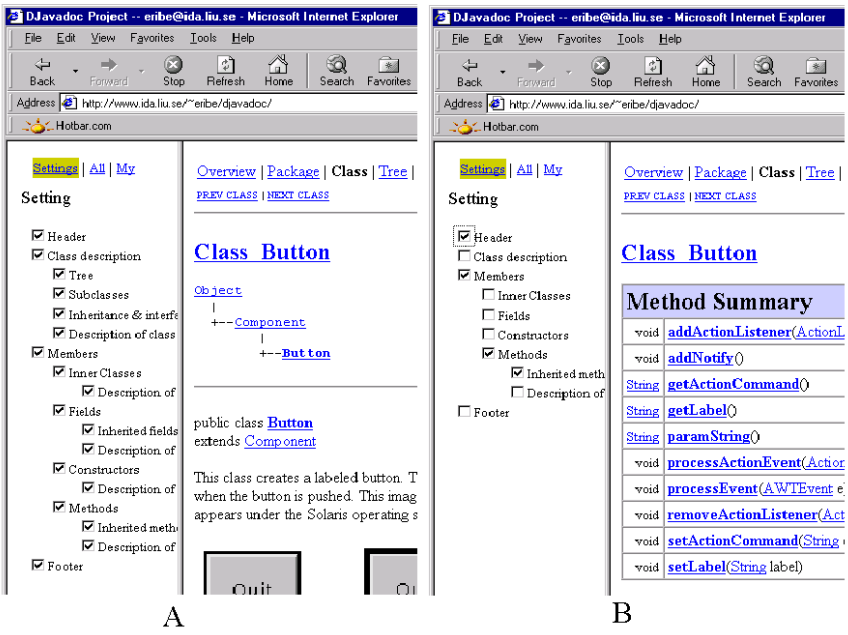

Figure IV.2: DJavadoc allows the reader to redesign the view of the class documents. Elements that are checked off in the Settings model (the left pane) will be collapsed in the class document (the right pane) as the reader browses the documentation.

class library for Java application developers), more focused indices are needed. The bookmark index provided in DJavadoc exemplifies the type of focused indices we envision.

IV.6 Requirements on Writing

In this section we take stance in the DJavadoc project and discuss the writing requirements of adaptable software reference documentation.

IV.6.1 Explicit Information Model

The predominant requirement DJavadoc places on the documentation is an explicit information model as a basis for change. The text is marked with information types used to distinguish units in the text, regardless of the static typography of the text. This approach is similar to SGML and XML languages where DSSSL (DSSSL) and XSL (XSL) transformation specifications are used to transform text into different views. The reader can be seen to manipulate the transformation specifications as part of reading.

However, in DJavadoc the information model is also used as a basis for interaction. The terms used in the model are presented as labels for interaction devices. The usability of the adaptable documentation is therefore dependent on how well the information model corresponds to readers' mental models of the documentation (Norman 1988) and also on the complexity. Since the reader interacts directly with the transformation process, the model must be carefully constructed to suit the reader's needs and terminology. Task analysis is required to arrive at an appropriate model.

From our ongoing user-evaluation of DJavadoc, we have indications that a much more restricted model than the current model would suffice for many purposes. DJavadoc testers most commonly express that they like the settings in Figure 2B (left pane) from which they may expand the remaining sections.

IV.6.2 Connections Across Documents

A form of adaptation we would have liked to work with in DJavadoc is application-type views of the entire documentation, such as database-application views. Explicit connections cutting across documents are present in Javadoc as hyperlinks. These connections are based on the input and output parameters of methods and reflect one step of the dependencies that exist among software components. However, to determine which of these connections are relevant for different application profiles is currently very difficult. This would have required a grouping that cut across the basic organization of the documentation (see Figure IV.4). Essentially, multiple categorization of components on a detailed level is required.

IV.6.3 Categories

In an early prototype version of DJavadoc we experimented with category-based filtering for methods. (Java software components are classes that have methods. It is not uncommon for a class to contain 40-100 methods.) Specifically we wanted

Figure IV.3: DJavadoc allows the reader redesign the view of the class documents. Elements that are checked off in the Settings model (the left pane) will be collapsed in the class document (the right pane) as the reader browses the documentation.

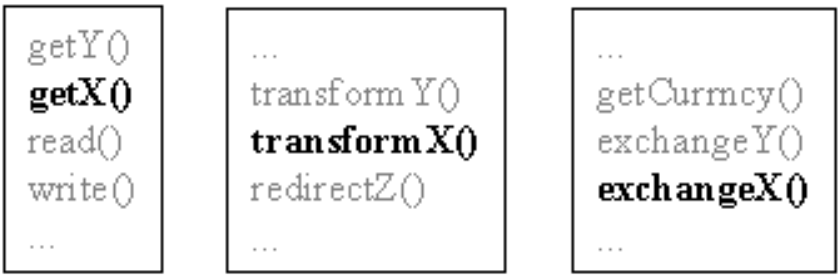

Figure IV.4: For a particular situation, the relevant view of the information is a subset of several Javadoc documents. To present such views in an adaptable documentation, multiple categorizations of components is needed.

to filter on terms such as basic methods, advanced methods, event-related methods, runt-time methods, and so on. It seems likely that readers would often want to find the basic methods, similar to the illustration in Figure IV.4. The programming language provides very limited support for automated categorization of this kind. The writing process would have to complement the language with these categories, which to some degree is already being done in tutorials (see for instance the Java Tutorial, [Campione and Walrath 1998]).

IV.6.4 Change Mechanism

The ways in which it is illustrative to manipulate documentation depend on many factors. The *purpose of the change* is one such factor. In DJavadoc, the purpose is to move relevant information up into the visible space of the browser. This effect was achieved by collapsing excessive information. An alternative mechanism could be to reorganize the information. Collapsing the document provides a shorter document, which is more easily scanned. However, information is removed from the page and therefore made invisible to the reader. This brings us to a second factor, i.e. the *effect of the change mechanism* (illustrated in Figure IV.1). In our experience working with DJavadoc, collapse and expand are suitable for smaller changes but become somewhat disorienting when large parts of the visible space of the browser are affected. DJavadoc would benefit from a combination of change mechanisms. Furthermore, removing information may not always be acceptable, for instance in medical records. The *tradition of the change mechanism* in the reader's environment also affects the choice. Color-coding, for instance, is often used in program editors and the DJavadoc reader may therefore respond well to such a change mechanism.

In our view, change mechanisms should be selected with care because they will affect the reading process. Determining what mechanisms are useful requires typographical knowledge, domain knowledge, and pedagogical knowledge.

IV.6.5 Evolution

The DJavadoc Bookmarks (see Section IV.4 and Figure IV.3) represent an evolving index of relevant classes to and from which the reader can continuously save or remove entries. DJavadoc is currently a one-user system but it is relevant to consider group-based bookmarks lists for project groups or even for communities of users. Evolving group indices could support dissemination of knowledge among developers. However, the quality of the index will depend on the evolution strategy. One solution is to treat all readers equally and base evolution on statistics by tracking reading behavior. However, different roles in development teams may indicate that the reading behavior of different readers should be treated differently. Another solution is to recommend that one person is responsible for the evolution. Evolution should, then, perhaps not be derived from reading behavior but rather set by user input. In the end, these questions come down to domain knowledge, pedagogical knowledge, and knowledge about the individuals in a group. It seems unlikely that the computer will be able to determine these issues. Therefore it is important that writers can easily express, for instance, which hyperlinks to include in an evolving index.

IV.7 Discussion

IV.7.1 Writing Trends

Currently, hypertext and task-oriented writing are two central concepts in technical writing. Hypertext provides a basis for structuring adaptable documentation into a web of possible pathways. Task-related writing also provides a structural basis in the connection between task and information. However, these concepts do not directly address change and allow the author to express change in a straightforward manner. Change implies time, interaction, multiple forms, alternative forms, and so on, which are not directly part of a task-based or hypertext-based perspective on writing.

IV.7.2 Expressing Change

Standard generalized markup language (SGML), extensible markup language (XML), and hypertext markup language (HTML) and related languages, in our view, represent an advanced text technology base that allow developers to express text-related structure and style on a high level (XMLCP, W3C). These world-wide-web consortium (W3C) languages are also particularly relevant at present since they form the backbone of authoring for the Web. Dynamics of web pages are currently implemented by connecting event handlers on document objects to functions expressed in scripting languages such as JavaScript. Transformation from document structure to multiple forms is possible using the extensible stylesheet language (XSL) for XML and the document style semantics and specification language (DSSSL) for SGML. The XML and XSL combination is particularly relevant because XML-related languages are becoming part of the general Web technology infrastructure. However XSL and DSSSL are fundamentally programming languages that programmers can use to define mechanisms for conditional transformation.

Currently the only language related to change in the W3C proposals is the synchronized multimedia integration language (SMIL). SMIL allows an author to express

temporal behavior of a presentation, the layout of the presentation on the screen, and to associate hyperlinks with media objects (W3C). The beginning, duration, and position of media elements can be expressed. Change strategies, mechanisms, and constraints can therefore be implicitly expressed in time sequences and hyperlinks.

In the research community frameworks for adaptive documentation have been addressed. De Bra and Calvi (1998) proposed a generic adaptive hypermedia system that allows you to enter conditional statements for the presentation of the text. Rutledge et al (1997) proposed another framework for generating adaptable hypermedia based on DSSSL. The *Exemplar* system provides a rule-based object-oriented framework for dynamic hypertext generation (White 1998). Common to these systems is that they use programming mechanisms to express change and addition.

To our mind, these technologies are suitable for programming but not for authoring. Authoring requires concepts that incorporate change semantics and enable writers to specify change without having to construct the change procedures. Let us use the *collapsible list* as an example of an authoring concept with change semantics (a list with collapsible subsections). Using DHTML, a programmer can create a collapsible list. An event handler of a list-entry must be connected to a function, which conditionally manipulates certain style properties. However, authors should not have to work with event handlers, function calls, and object access. Instead the collapsible list should be a list type just like an intext list and a displayed list (terminology from Dupré 1995). The collapsible list should incorporate collapse and expand functionality on a semantic level.

IV.8 Conclusion

From the DJavadoc project, it has become clear that adaptivity requires a writing process preparing for adaptively. The documentation must contain information about how to adapt. Information designers therefore need tools that encapsulate change semantics and provide clear concepts for adaptivity.

In our view, Web technology (as most text technologies) is not yet advanced enough to express change strategies, mechanisms, and constraints at an appropriate level. Too much programming is required. Authors of dynamic content must handle control statements, conditional statements, function calls, event mechanisms, and so on. Essentially the Web currently delivers a general-purpose programming machinery for change but no change concepts. Functionality is spread in the community as cut-and-paste scripts. We envision a development Web language that allows authors to describe change on a higher level. The semantics of change could be included in the technological Web infrastructure.

References

AH&H, *Adaptive Hypertext & Hypermedia*, web Site – `http://wwwis.win.tue.nl/ah/`

AR, *Ada 95 Reference Manual* – `http://www.adahome.com/rm95/`

Beaumont I. and Brusilovsky P. (1995) *Adaptive educational hypermedia: From ideas to real systems.* In H. Maurer (Eds.), Proceedings of ED-MEDIA'95 - World

conference on educational multimedia and hypermedia, Graz, Austria, June 17–21. Charlottesville, AACE. pp. 93–98.

Barker M. (1997) *From Document Design to Information Design.* Proceedings of the 15th annual international conference on Computer documentation October 19 - 22, Snowbird, UT USA.

Berglund E. (1999) *Use-Oriented Documentation in Software Development.* Linköping Studies in Science and Technology, Licentiate Thesis no. 790, School of Engineering at Linköping University ISBN: 91-7219-615-7. PDF version online: `http://www.ida.liu.se/~eribe/lic/berglund.pdf`

Brusilovsky P. and Vassileva J. (Eds.) (1996) *Special Issue on: Adaptive Hypertext and Hypermedia.* User Modeling and User-Adapted Interaction No. 6.

Brusilovsky P. (1996) *Methods and Techniques of Adaptive Hypermedia.* User Modeling and User-Adapted Interaction No. 6 pp. 87-129.

Carroll J.M. (Ed.) (1998) *Minimalism Beyond the Nurnberg Funnel* MIT Press.

Campione M. and Walrath K. (1998). *The Java Tutorial: Object-oriented programming for the Internet.* Addison-Wesley. (Also available as `http://java.sun.com/docs/books/tutorial/`)

De Bra P. and Calvi L. (1998) *Proceedings of the 2nd Workshop on Adaptive Hypertext and Hypermedia HYPERTEXT'98* `http://wwwis.win.tue.nl/ah98/`. Pittsburgh, USA, June 20–24.

de Olivera Braga C., von Staa A., and do Prado Leite J.C.S. (1998) *Documentu: A Flexible Architecture for Documentation Production Based on a Reverse-engineering Strategy.* Journal of Software Maintenance: Research and Practice, vol. 10 279(303).

DJavadoc, *DJavadoc Home Page* – `http://www.ida.liu.se/~eribe/djavadoc`

Dypré L. (1995) *BUGS in Writing: A guide To Debugging Your Prose.* Addison-Wesley.

Hackos J.T. (1997) *Online Documentation: The Next Generation.* Proceedings of the 15th annual international conference on Computer documentation October 19 - 22, Snowbird, UT USA.

Jacobson R. (Ed.) (1999) *Information Design.* MIT Press.

Javadoc, *Javadoc Home Page* – `http://java.sun.com/products/jdk/javadoc/`

JD, *Java documentation* – `http://java.sun.com/docs/`

JR, *Java Reference* – `http://java.sun.com/products/jdk/1.2/docs/api/index.html`

Kantorowitz E. and Sudarsky O. (1989) *The Adaptable User Interface.* Communications of the ACM, no. 31 vol. 11.

Knuth D.E. (1992) *Literate Programming.* Center for the Study of Language and Information, Leland Stanford Junior University.

Kramer D. (1999) *API Documentation for Source Code Comments: A Case Study of Javadoc.* In Proceedings of the Seventeenth Anual International Conference of Computer Documentation (SIGDOC'99), New Orleans, September 12-14.

Lewis J.E. and Weyers A. (1999) *ActiveText: a Method for Creating Dynamic and Interactive Texts*. Proceedings of the 12th annual ACM symposium on User interface software and technology November 7–10, Asheville United States.

MSDN, *Microsoft Developers Network Online Web Workshop* –`http://msdn.microsoft.com/workshop/`

Norman D.A. (1988) *The Design of Everyday Things*. Basic Books.

PR, *Python Library Reference* – `http://www.python.org/doc/current/lib/lib.html`

Rutledge L., van Ossenbruger J., Hardman L., and Bulterman D. (1997) *A Framework for Generating Adaptable Hypermedia Documents*. Proceedings of the Conference on Multimedia '97 November 9–13, Seattle, WA USA.

Soloway E., Pinto J., Letovsky S., Littman D., and Lampert R. (1988) *Designing Documentation to Compensate for Delocalized Plans*. Communications of the ACM vol. 31, no. 11, 1259(1267).

VC, *Visual Café* – `http://www.symantec.com/domain/cafe/vc4java.html`

W3C, *World-Wide-Web Consortium*, web site – `http://www.w3.org`

White M. (1998) *Designing Dynamic Hypertext*. Proceedings of the 2nd Workshop on Adaptive Hypertext and Hypermedia HYPERTEXT'98 (`http://wwwis.win.tue.nl/ah98/`), Pittsburgh, USA, June 20–24.

Wolfram S. (1996) *The Mathematica Book*. Wolfram Media/Cambridge University Press.

XMLCP, *The XML Cover Pages* – `http://www.oasis-open.org/cover/sgml-xml.html`

Paper V.
Dynamic Software Component Documentation

Co-authored by Henrik Eriksson, Department of Computer and Information Science, Linköping University email: her@ida.liu.se

Published in the proceedings of the Second Workshop on Learning Software Organizations, June 20 2000, Oulu, Finland

Abstract

Software development is based on the reuse therefore requires detailed knowledge of a vast number of software components and their context. Typically, developers acquire this knowledge by reading reference documentation. However, software documentation is generally provided as static text and does not facilitate project-related evolution. Thus, documentation cannot be an active tool in the knowledge management of software development projects. In this paper we discuss the role of software documentation in learning software organizations by presenting and discussing two dynamic software-documentation projects aimed at use-oriented change, evolution, and adaptation of documentation. The systems enable redesign of document layout and content to a varying degree of generality. The first system, DJavadoc, is based on Dynamic HTML (DHTML) and the second system, which is currently under development, uses the knowledge-acquisition tool Protégé as a platform.

V.1 Introduction

The Web is an important tool for software developers. In fact, the Web has already changed the way software developers work. Today, it is possible to collaborate and to share program components with other software developers worldwide. Communication technologies accelerate the rate of progress and enable software developers to make new products available immediately.

Although there are many advantages of the improved communication technologies, they can create new problems for programmers. One such problem is the difficulty of keeping up-to-date in fast-moving areas. The current rate of technical progress in the area of software development tools such as programming languages, application-programmer interfaces (API), and component libraries, makes it difficult for professional programmers to keep up.

Therefore, information search and information retrieval are becoming increasingly important for programmers. To avoid redundant work, it is important to be up-to-date with the dynamic software libraries available both within the organization and globally. The increased availability of these resources has changed significantly the way programmers develop software. However, software development methods and techniques have not yet fully incorporated these new practices. This *indexing* problem is a major obstacle to the reuse of library components.

Reference documentation is becoming a backbone of the software-development process. There is a strong dependency between the knowledge found in documentation and the programs that developers write. By tailoring the documentation, the software projects can take advantage of focused views of the vast reusable libraries that their products rely upon (thereby facilitate organizational memory and learning). Today, this knowledge is not captured. Instead, it remains in the heads of the developers. Consequently, there is a need to further develop documentation systems into tools that collect and distribute such knowledge.

Furthermore, it is sometimes too restrictive to consider knowledge and learning in a single organization. Today, corporations and other organizations develop a significant amount of software in *open projects* (i.e., the status and the intermediate results of an ongoing project are available publicly). Such projects often involve a *community* of users, third-party developers, and international standardization organizations. Furthermore, the projects typically rely on common tools and widely-accepted frameworks. In this environment, software documentation plays an even more important role as the communication medium.

V.2 Background

V.2.1 Technology Changes

The current rate of technical progress in the area of software-development tools such as programming languages, APIs, and component libraries, makes it difficult for professional programmers to keep up. Communication technology, such as the Internet and the Web, has accelerated the rate of progress, and has enabled software developers to make new products available immediately.

Web technology has already improved collaboration and spurred an increased sharing of program components. Even though developers reused code before the Web, the amount of available components and the rate with which new components are introduced has increased. We believe that this component sharing is only the first step in a series of changes. The increased collaboration provided by the Web will change the way programmers work. Two major factors that contribute to this change:

1. The architecture of the target systems is changing in that the systems become

more and more distributed. Application programmers add functions to on-line services rather than create complete traditional programs. Furthermore, developers migrate conventional applications to Web services.

2. The Web will serve as a platform and programming environment for software engineers. On-line information and discussion groups for programmers have existed for some time. Future programmers will take advantage of more advanced on-line services for supporting their work, for example on-line documentation with advanced navigation support and on-line development tools.

The improved communication and the emerging program-component sharing taking place today are certainly promising. However, this process has also created information problems for software developers that must be addressed before it is possible to use the full potential of the Web in software engineering.

V.2.2 Knowledge and Learning Changes

Traditionally, programmers have developed software by applying the knowledge and skills they acquired on different types of courses (or, sometimes, learned by themselves). Today, professional software developers already get a large proportion of their information through the Web. In the future, the amount of information developers get from the Web will increase. For the advanced Java programmer, for instance, it is necessary to use the Web as an encyclopedia and to get information about technical news and changes (Campione and Walrath 1998) (because the Web is the primary media for distributing Java information). This approach is a new way of developing software.

Previously, programmers worked with a relatively well-defined body of knowledge found in textbooks. Today's programmers often have to acquire even the basic information from multiple sources (e.g., about APIs, and standards). Furthermore, for fast moving technology areas such as the Web, textbooks are often, in our experience, incomplete and out of date. Although the information resources available on line have the potential of improving the programming efficiency (e.g., because reuse eliminates redundant work), there are also limitations of this approach. For instance, it becoming increasingly important to search for information about predefined components in preexisting program libraries.

V.3 Documentation of Software Components

There are important questions that emerge from the problem of communication among programmers. One such question is how to approach sharing and reuse of program components, such as classes in object-oriented programming. Creating libraries of components and reusing them in other development projects have proven difficult. There are two major obstacles to reuse of such library components:

1. The *indexing* problem. That is, the problem of structuring a component library so that it is possible to find and retrieve components. In large class libraries, the task of finding the right class is a serious practical problem.

2. The *adaptation* problem. That is, the problem of understanding and configuring a component so that it is possible to use it in a given context. As the reusable components grow larger and more complex, it is difficult to understand the functionality of them.

As the size and complexity of the component libraries increase, the indexing problem becomes significant. A major challenge for Java programmers is to deal effectively with the libraries available and to take full advantage of them. For example, the Java Development Kit (JDK) version 1.3 contains over 2,100 classes in its class library. The size of the documentation in HTML format is about 97MB. It is almost impossible for an individual programmer to have detailed knowledge about every class. However, such detailed knowledge not required normally. What is important is to be able to find your way around the body of classes.

V.4 The DJavadoc Approach

In general, online reference documentation cannot be adapted to particular situations. Knowledge about the readers context (e.g., interests and current tasks) cannot be inserted into the documentation to improve the match between a programming situation and the documentation. The documentation has no ability to assimilate project-related knowledge to serve as a memory that relates a project and its source code to the documentation.

The Dynamic Javadoc (DJavadoc) project [1] is a research agenda in this direction (Berglund 1999). DJavadoc adds client-side real-time redesign to the standard Java reference documentation[2] and is currently undergoing user evaluation. The Java reference documentation is essentially a source-code browser, providing navigational indices and typeset views of the source-code files. Assumptions about the degree of relevance of different parts of the information and about the organization of the information are represented in the documentation. DJavadoc enables the reader to alter the views the browser provides as part of the reading process, thus making the documentation dynamic. The system is implemented using dynamic HTML (DHTML), which basically is script-based manipulation of HTML pages in web-browsers.

From a knowledge management perspective, the ability to create indices views is of particular interest in DJavadoc. The Javadoc table of contents can be collapsed into a smaller subset of relevant hyperlinks. With the current number of classes in JDK, it becomes obvious that the readers need more focused class lists to easily locate relevant classes. By constructing a separate table of contents, readers save references to classes in a bookmark fashion (see Figure V.1).

The DJavadoc bookmark is an example of information filtering on a navigational level. However, it requires much activity on the part of the reader. Examples of other possible (but not currently implemented) indices of the same type include:

- *Project index.* The joint use of Java classes in a project group working on a common task could be assembled into a project index. (This approach is useful for single-person projects as well.)

[1] DJavadoc is available on-line at `http://www.ida.liu.se/~eribe/djavadoc/`

[2] Commonly known as the Javadoc, from the program Javadoc that generates the HTML reference documentation from the source code files.

Figure V.1: In a bookmark fashion, classes are saved in and removed from a bookmark index by the reader.

- *Use-based index.* Describes how the JDK classes are used on a more general level. For example, such an index can be constructed by statistically analyzing a large number of Java source files. Another alternative is to analyze all Java source code on the Sun's Java Web-site, perhaps providing a use-based index of sample code. A third statistical source could be online-tutorials such as the Java Online Tutorial[3].
- *History index.* By tracking the readers' browsing behavior, a history index could be designed to present the most frequently and most recently accessed classes.
- *Code index.* This index draws its entries from parsed source files. The context in which the programmer is currently working provides a relevant filtering basis.
- *Application index.* This index points to groups of components that are useful for particular applications profiles (e.g., client-server applications and database applications).

Figure V.2 illustrates how the reader can redesign Java class documents in DJavadoc by checking on or off sections in a document model (representing the information structure in the Java class documents), thus creating a default view of the documents. This default view is then used by the system to collapse or expand different sections in the documents as the reader browses the documentation. Without changing the default settings, the reader may also locally collapse and expand individual sections.

DJavadoc makes Javadoc documentation adaptable and sensitive to knowledge about the users needs. In essence, the reader is allowed to redefine assumptions made about the appropriate level of detail and organization of the information found in DJavadoc. In a sense, the DJavadoc reader is populating a project ontology by saving bookmarks and by checking on or off sections of the default view. However, the redesign is restricted to a predefined dynamics in the system. A more general redesign mechanism is required to fully support knowledge management and organizational learning across projects and among readers. For this purpose, the DJavadoc project leads on to the Protégé Javadoc project which focuses on project ontologies for documentation and knowledge-engineering solutions to evolving and adaptable documentation.

V.5 The Protégé Javadoc Approach

Knowledge-acquisition tools, especially tools for ontology management, can be useful for creating and browsing documentation (Eriksson 1992). We have experimented with the use of the knowledge-acquisition tool Protégé[4] (Grosso et al. 1999) for managing Javadoc-generated documentation. Our goal was to study the feasibility of using this type of tool for software documentation. We used Protégé to define an ontology for the Java object system. This ontology contains classes for Java concepts, such as class, constructor, method, and so on. We then developed a doclet that transforms the structure of the doclet API to a representation format that Protégé

[3]The Java Tutorial `http://java.sun.com/docs/books/tutorial/`

[4]Protégé is available at `http://smi.stanford.edu/projects/protege/`.

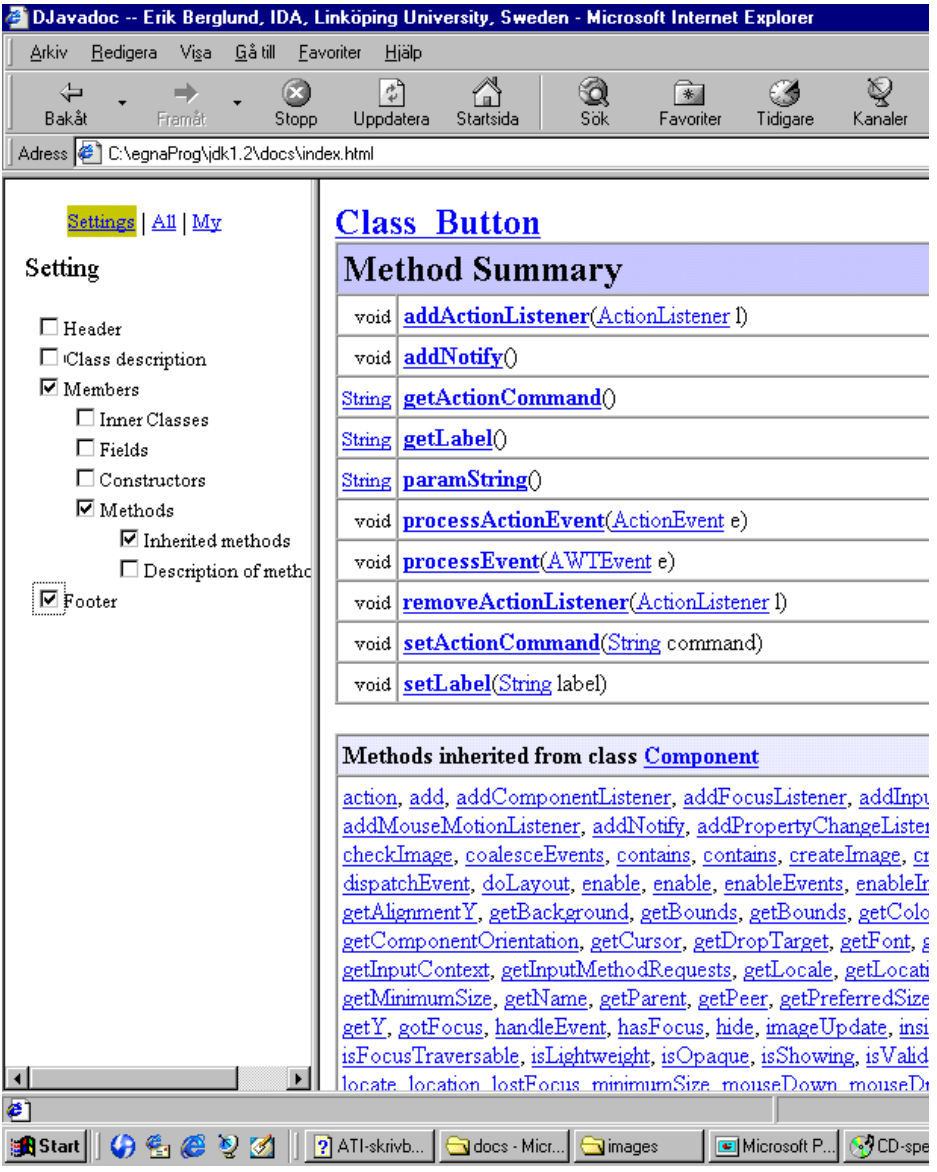

Figure V.2: DJavadoc allows the reader redesign the view of the class documents.

can read. It is possible to use this doclet together with the Javadoc program to produce instances of the Java ontology in Protégé.

Because Protégé is a meta-level tool, we can use it to create custom-tailored layouts for class documentation. Figure V.3 shows the custom-tailored layout we use for the class documentation. The left-hand side of the window contains a class browser for the Java classes. It is possible to use this tree browser to navigate among the classes. The right-hand side of the window shows the documentation for the selected class (in this case JButton). The class from contains fields for the class name, class modifiers, fields, methods, and textual documentation.

One of the major advantages of using the Protégé approach over standard Javadoc and DJavadoc is the flexibility that the meta-tool provides as a rapid application development environment. In Protégé, it is possible to modify the layout of the class form by changing the form definition. Furthermore, it is possible to add new types of information to the documentation, such as new items in the class form. In essence, the entire documentation content and layout can be redefined. Examples of items that could be interesting from an organizational perspective are author, manager, project, and version. Because addition of information fields is straightforward, software developers can create project-specific versions of the documentation tool (Gennari and Ackerman 1999).

The use of Protégé for Java documentation illustrates that it is possible to take advantage of knowledge-acquisition tools for software documentation purposes (Eriksson 1992). This prototype is an early proof of concept, but a thorough implementation is required to perform user studies. In particular, the addition of custom-tailored user-interface components (plug-ins) to Protégé could improve significantly the usability of the tool. Although the current prototype is rather crude, it was possible to develop it with relatively little effort.

V.6 Related Work

Reference documentation for programming languages and software component libraries are most commonly presented in the form of online component catalogues. Online component catalogues are available for languages such as Ada[5], Python[6], Java[7] A widely-used component catalogue is the online Java API reference documentation, generated from the source code by the Javadoc tool[8](Kramer 1999). The Javadoc documentation lists available components at class level and provides a series of navigational indices. Hyperlinks are used to cross-reference among class documents via parameters, types, return values, and so on. In our view, the Java API reference documentation has become somewhat of a model for online reference documentation. Javadoc, however, still delivers static text that does not change or evolve.

The Microsoft Developers Network (MSDN) Online Web Workshop[9] and the Mathemathica Help Browser (Wolfram 1996) are examples of reference documentation that allow some form of change are. Both these documentation systems remember

[5]Ada 95 Reference Manual `http://www.adahome.com/rm95/`.
[6]Python Library Reference `http://www.python.org/doc/current/lib/lib.html`.
[7]Java Reference `http://java.sun.com/products/jdk/1.2/docs/api/index.html`.
[8]Java documentation `http://java.sun.com/docs/`.
[9]MSDN Online Web Workshop `http://msdn.microsoft.com/workshop/`.

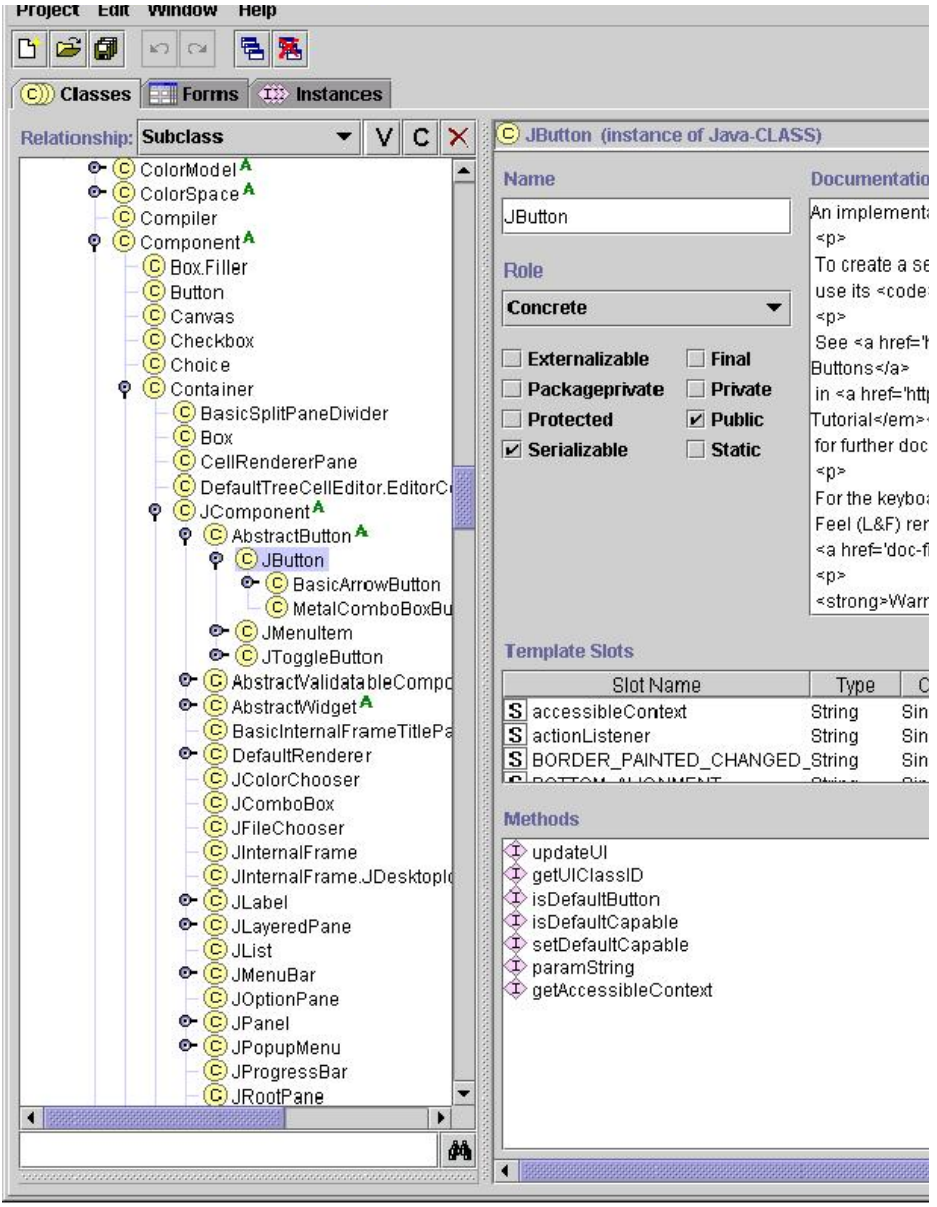

Figure V.3: Protégé-generated tool for Java documentation. This prototype shows the Java class hierarchy (left) and the documentation for the JButton class (right)

whether or not a section was collapsed or expanded last time the reader visited a document. DJavadoc intentionally does not remember which individual sections were collapsed or expanded last time a document was visited because we do not believe that the interests a reader had previously represent their current interest (instead a default setting is used). On the contrary, it is our experience as frequent users of the MSDN Online Web Workshop, that the memory more often requires the reader to change settings. However, this is an empirical question that we have not investigated further.

Development environments, such as Visual Age[10], Visual Cafe[11], and JBuilder[12], often provide project management of source code. In some cases, for instance in Visual Cafe, the reference documentation is also included and it is possible to browse the documentation directly from the source code. This is definitively a step in the right direction, in which the source code becomes a use-oriented and evolving index to the documentation (an example of a *code index*, see section 4).

Software documentation has been addressed in the research community (Knuth 1984, Kramer 1999, de Olivera Braga et al. 1998, Soloway et al. 1998). The focus has been on the generation of software documentation whereas our work focuses on management of software documentation as part of the programming environment.

V.7 Conclusion

Documentation of program components, such as classes, is an important aspect of the memory of a software organization. Dynamic reading environments for documentation allow software developers to access and understand the documentation rapidly (as illustrated by the DJavadoc project). Furthermore, knowledge-acquisition tools can assist in the authoring of the documentation and in documentation browsing and understanding. We believe that the use of Protégé for software documentation is a viable approach because it provides flexibility and because it works well together with the Javadoc approach. In our continued efforts, we are developing a more sophisticated Protégé implementation of Javadoc than the current, rather limited, prototype.

An interesting extension to documentation systems such as DJavadoc is to support communication among developers (i.e., the documentation readers). For example, each class-documentation page could have a common annotation area where developers could enter notes about their use of this component, including problems and solutions found. Today, this type of information is difficult to browse and find because it is spread over numerous message forums and discussion groups. Because of the trend towards software development in open projects, we believe that it is important to support communication within communities in addition to intra-organization communication.

[10] Visual Age `http://www-4.ibm.com/software/ad/vajava`.
[11] Visual Cafe `http://www.symantec.com/domain/cafe/vc4java.html`.
[12] JBuilder `http://www.borland.com/jbuilder/`.

References

Berglund. E. (1999) *Use-Oriented Documentation in Software Development*. Licencate thesis no. 790, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden. (DJavadoc is available on-line at `http://www.ida.liu.se/~eribe/djavadoc/`)

Campione M. and Walrath K. (1998) *The Java Tutorial: Object-oriented programming for the Internet*. Addison-Wesley.

de Olivera Braga C., von Staa A., and do Prado Leite J.C.S. (1998) *Documentu: A Felxible Architecture for Documenetation Production Based on a Reverse-engineering Strategy*. Journal of Software Maintenance: Research and Practice, vol. 10 279(303).

Eriksson H. (1992) *A survey of knowledge acquisition techniques and tools and their relationship to software engineering*. Journal of Systems and Software. 19(1):97–107.

Gennari J.H. and Ackerman M. (1999) *Extra-Technical Information for Method Libraries*. In Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling, and Management, Banff, Canada, October 16-21.

Grosso W.E., Eriksson H., Fergerson R.W., Gennari J.H., Tu S.W., and Musen M.A. (1999) *Knowledge modelling at the millennium (The design and evolution of Protégé-2000)*. In Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling, and Management, Banff, Canada, October 16-21.

Knuth D.E. (1984) *Literate Programming*. Computer Journal. vol. 27. May. pp. 97–111.

Kramer D. (1999) *API Documentation for Source Code Comments: A Case Study of Javadoc*. In Proceedings of the Seventeenth Anual International Conference of Computer Documentation (SIGDOC'99), New Orleans, September 12-14.

Soloway E., Pinto J., Letovsky S., Littman D., and Lampert R. (1988) *Designing Documentation to Compensate for Delocalized Plans*. Communications of the ACM vol. 31, nr. 11, 1259(1267).

Wolfram S. (1996) *The Mathematica Book*. Wolfram Media/Cambridge University Press.

Paper VI. Intermediate Knowledge trough Conceptual Source-Code Organization

Co-authored by Henrik Eriksson, Department of Computer and Information Science, Linköping University email: her@ida.liu.se

Published in the proceedings of the 10:th International Conference on Software Engineering & Knowledge Engineering, June 18-20 1998, San Francisco Bay CA USA, pp 112–115

Abstract

Program understanding is difficult. When performing maintenance programmers must understand relations between conceptual models (for instance requirement specifications) and source code. They have to acquire *intermediate knowledge.* However, intermediate knowledge cannot always be intuitively acquired by comparing conceptual models and source code.

This paper discusses a new tool-supported approach to documenting intermediate knowledge in terms of concepts regardless of the source- code structure, *conceptual source-code organization* (CSCO). Programmers can use CSCO to navigate the source code in search of relevant source-code segments, and to acquire a source-code overview.

VI.1 Introduction

Design should preferably result in a natural mapping between conceptual models of programs and the source code. We characterize the knowledge about this mapping as intermediate knowledge. By reviewing the source code it should, for instance,

be apparent what source-code segments implement which parts of the requirement specification. Since program understanding is difficult (Woods and Yang 1996, von Mayrhauser and Vans 1994), a natural mapping would help programmers with the task of changing or adding to exsisting source code (i.e., performing maintenance).

However, software-engineering issues other than design affect source- code structure, for instance reuse. As a result the source code deviates from the conceptual models. Consequently intermediate knowledge cannot be intuitively acquired by comparing conceptual models and source code.

To address this problem, we have experimented with intermediate knowledge though *conceptual source-code organization* (CSCO); a new tool-supported approach for documenting intermediate knowledge. CSCO provides intermediate knowledge by organizing source code in terms of concepts regardless of source-code structure. Our experiments resulted in a prototype environment, named PROTERCIS, developed using meta-tools originating from the knowledge-engineering community (Eriksson 1992, Eriksson and Musen 1993).

Although we have not yet evaluated the prototype formally, our experiments using PROTERCIS to organize a medium-sized program give some indications of the value of CSCO. PROTERCIS can provide source-code navigation and overview though a set of conceptual paths into the source code, enable rapid identification of relevant source-code segments in maintenance situations, visualize groups of segments working together to implement concepts regardless of source-code structure, and preserve intermediate knowledge over time as information understandable by computers.

VI.2 Intermediate Knowledge

In the context of this paper, intermediate knowledge represents the mapping between conceptual models of programs and the programs themselves. The intermediate knowledge identifies the connection between concepts and source-code segments, and the interaction schemata among segments. Let us consider an example: In a distributed game program of Football, Player objects, a Coach object, a Bank object and an Owner object implement the notion of a *team*.

Intermediate knowledge is not always intuitive, which is illustrated in Figure VI.1. A concept (FigureVI.1A) can be implemented by a group of source-code segments interacting with one another (FigureVI.1B). In practice there are several reasons why conceptual models and source code deviate which are not related to poor design. For instance, when building from a software-reuse platform syntactical formalisms are inherited, such as event structures and interaction schemata. In our example, we want to reuse AI- planning objects for our Players. The AI-Planning objects coordinate themselves via a central AI-Management object, which in turn formulates strategies from a Knowledge-Base object. A Knowledge- Composer object collecting information from Sensor objects updates the Knowledge-Base object.

Apparently, our reuse platform does not fit our application. We use it only because it implements AI-techniques we want to avoid learning in detail. Furthermore, programming languages are syntactically limited in their ability to express abstract structures such as objects residing on more than one computer. In our Football example, our team might be a distributed system with Player objects residing on different computers. To interact we have to introduce streams, remote procedure

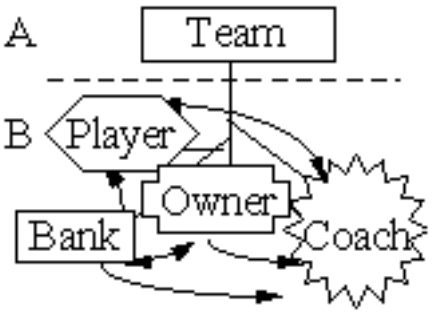

Figure VI.1: A concept and its group of implementing source-code segments, with a flow of interaction.

calls, and so on. Moreover, engineering constructs enforce structural elements for engineering reasons only. Engineering constructs can for instance be design patterns (Gamma et al. 1995).In our example, we want to be able to exchange the underlying motivations of a Player object without reimplementing the Player. Therefore we encapsulate the notion of motivation in a Motivation object, which in turn is produced by a Motivation-Factory object. Thus, changing the Motivation-Factory object is equivalent to changing the motivation of the player. However, the Motivation- Factory object does not represent something in the conceptual model; it exists only to introduce source-code flexibility.

VI.3 Conceptual Source-Code Organization

Our approach to overcoming non-intuitive source code is to provide intermediate knowledge using CSCO. CSCO has two components, conceptual models and source code segments. We view conceptual models as general, imprecise descriptions of programs, see FigureVI.2. Conceptual models represent different views of the program: one view could be the requirements of the program, another all segments accessing synchronized memory.

By attaching groups of source-code segments to our conceptual models we formulate a CSCO, see FigureVI.3. The segments can be viewed as indexed pieces of source code, intended to be small enough for straightforward analysis by programmers.

The intermediate knowledge is the structure, represented by the lines in FigureVI.3. This structure forms paths into the source code based on concepts. Applying conceptual organization, developers can organize the source-code segments according to concepts regardless of the source-code structure.

How does the architecture address the problems discussed in Section VI.2? Well,

CSCO enables mapping between source code and concepts regardless of code structure. In FigureVI.3 we see that Team and Referee in part use the same source-code segments, for example a segment used to save data in a Result object. Furthermore, it is possible to design several conceptual models, representing different views of the source code. In FigureVI.4 we see a technical view of our Result object, which is a synchronized data object. We model the access to the Result object for later use because we may have to change the synchronization schema.

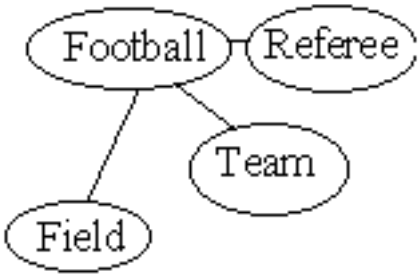

Figure VI.2: A conceptual model, representing a program.

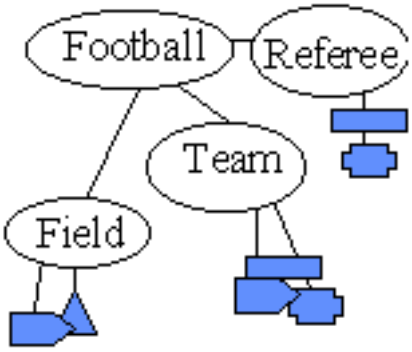

Figure VI.3: The conceptual source-code organization based on concepts and segments.

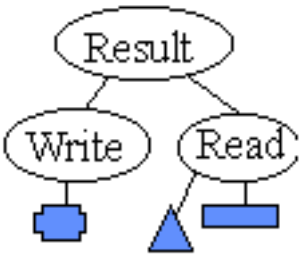

Figure VI.4: Another conceptual mode representing another view of the source code, binding the same segments as the model in Figure VI.3.

VI.4 Prototype

Our experiments resulted in a prototype development environment supporting CSCO. The prototype, named PROTERCIS, was designed using Protégé, which is a suite of meta tools developed at the Section on Medical Informatics at Stanford University (Musen 1989, 1997). We used Protégé to generate PROTERCIS automatically from declarative specifications (Eriksson et al. 1994, Puerta et al. 1992).

In PROTERCIS, developers act as experts and register their intermediate knowledge. They formulate CSCO using hierarchical conceptual models and express source-code segments at method level.

To examine PROTERCIS informally we used the source code of ERICS (Berglund and Eriksson 1998), a system implemented in Java which took four person-months to implement. ERCIS is a distributed system that inherits the syntactical formalism from the Java core platform.

Figure VI.5 shows a concept in the conceptual model. The front panel is a concept which has a group of concept children represented by the diamonds in the graph. The parent of this concept is seen in the background.

Figure VI.6 shows a segment. The front panel is a segment editor showing a segment and its code editor. The second panel is a concept that contains a list of implementing segments.

On the basis of a general, imprecise understanding of ERCIS, we use PROTERCIS to locate groups of source-code segments working together. PROTERCIS promotes source-code overview by visualizing groups of segments working together, regardless of their location in the source code. PROTERCIS also preserves intermediate knowledge over time (unlike in our heads) and stores the intermediate knowledge together with the source code.

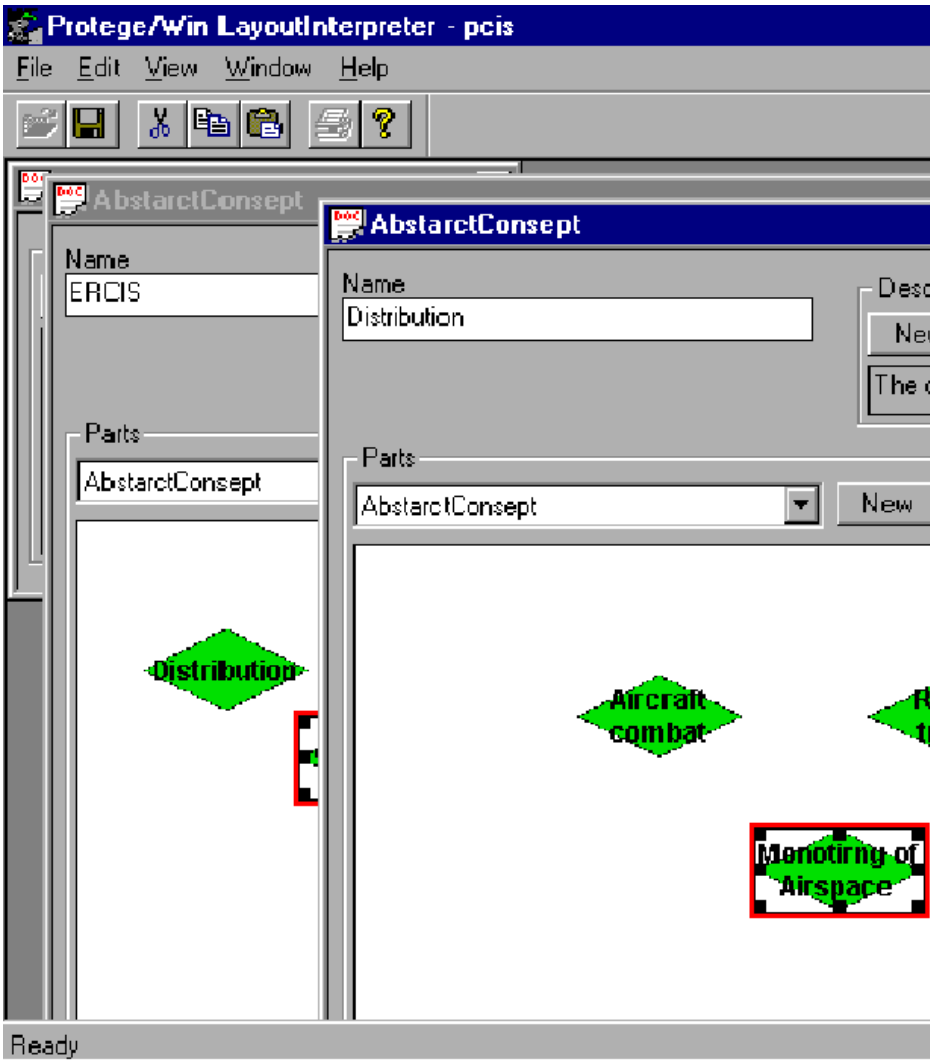

Figure VI.5: A hierarchical conceptual model in PROTERCIS. The panel at the front contains concepts of ERCIS.

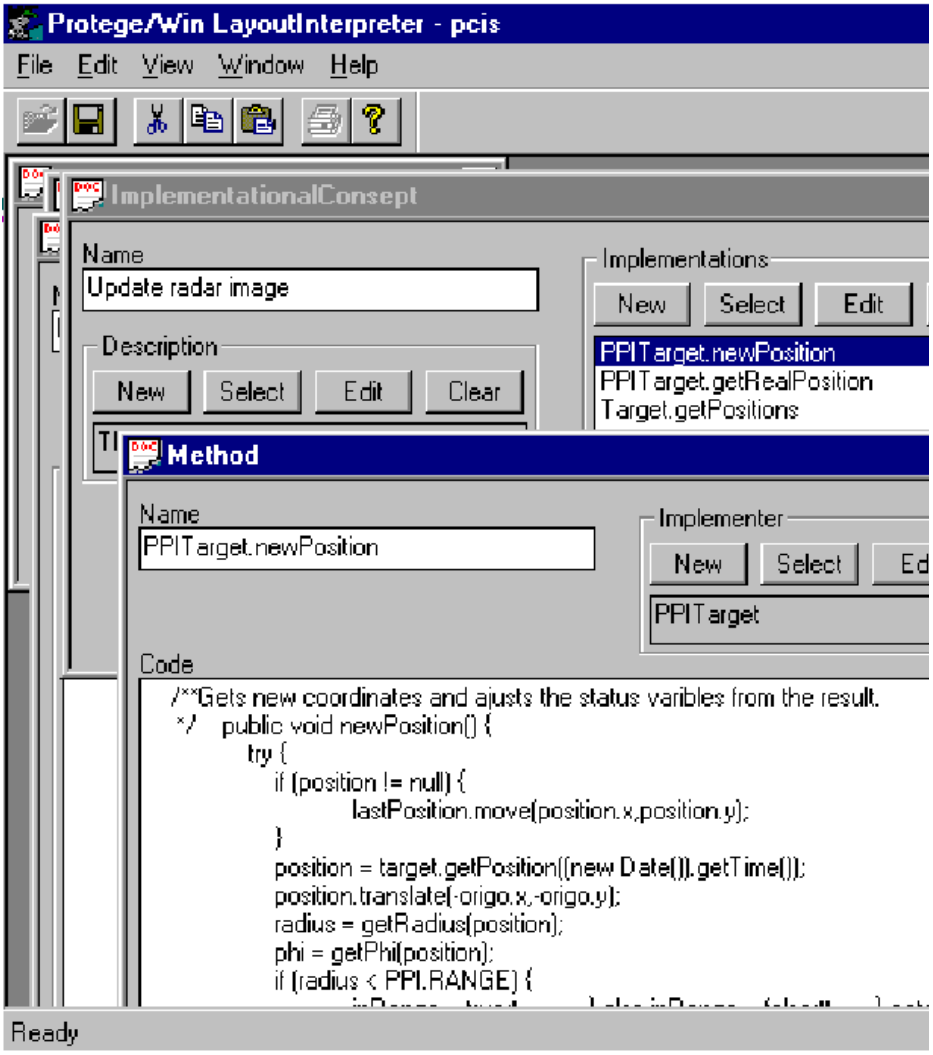

Figure VI.6: A source-code segment in PROTERCIS. The panel at the front contains the code of a method, which is part of the information about a concept in ERCIS.

VI.5 Related Work

Related to our work there are other efforts to overcome non- intuitively in source code. One example is discussed by Soloway et al. (Soloway et al. 1988) who address delocalized programming plans (plans executed by source code segments scattered across the source code) by designing documentation to make them explicit. Another example is LaSSIE, a knowledge-based software-information system that contains architectural and conceptual information (Devanbu et al. 1991). SeeSys is another system that visualizes statistics associated with code (Baker and Eick 1994). One of the most used development environments for large software projects is Rational Rose, which is based on the 4+1 views of the system (Kruchten 1995).

Unlike these systems, CSCO separates the conceptual models and the program model. We view the source code as an incomplete and non- intuitive basis for defining conceptual models.

VI.6 Discussion

PROTERCIS can be used to express intermediate knowledge, describing relations between conceptual models and source code. An interesting extension to PROTERCIS would be to organize document segments in the same manner. In addition to addressing documentation overview and navigation, this approach would link source-code segments to segments of documents. Such cross-reference is rare (Musen 1989). Another extension of PROTERCIS would be to express graphically the interaction schemata and dependencies among source-code segments.

Protégé, a suite of meta-tools, automatically generates PROTERCIS. Development environments such as PROTERCIS pose special requirements on meta tools. For instance to enable compilation, meta tools must either produce source code from their database or store intermediate knowledge using the commenting features of most programming languages. Meta tools must also be able to communicate with other programs, such as compilers, and editors. Moreover, meta tools must enable cooperative work for large projects, which introduces new demands on security and restrictions.

PROTERCIS exemplifies how knowledge acquisition can be useful in software engineering. Another interesting challenge of using knowledge acquisition in software engineering is the design of tools where the knowledge of domain experts and programming-language experts become the components we use to build software. Yet another challenge is to monitor project progress. If we can acquire knowledge on design and implementation progress, we can use it to modify software-project plans during execution.

VI.7 Summary and Conclusion

In this paper, we have discussed why intermediate-knowledge is not always intuitively expressed in source code. We present a new approch to addressing this problem, CSCO, which is the primary contribution of this work. It is our belief that CSCO can overcome non-intuitivity. We also present a prototype environment, PROTER-

CIS, which can be used to organize the source code according to conceptual models. Our experiments using PROTERCIS to conceptually organize a middle-sized program indicate that CSCO can increase source-code overview and enable source code navigation, store intermediate knowledge together with the source code as structural information understandable by computers, and preserve intermediate knowledge over time by recording the programmers' expertise.

References

Baker M. J. and Eick S. G. (1994) *Visualizing Software Systems.* In Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy. pp. 59–67.

Berglund E. and Eriksson H., (1998) *Distributed Interactive Simulation for Group-Distance Exercises on the Web.* In Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation, San Diego USA, January 11–14. pp. 91–95.

Devanbu P., Brachman R. J., Selfridge P. G., and Ballard B. W. (1991) *LaSSIE: A Knowledge Based Software Information System.* Communications of the ACM. vol. 34. no. 5. pp. 34–49.

Eriksson H. (1992) *Meta-tool Support for Custom-Tailored Domain- Oriented Knowledge Acquisition.* Knowledge Acquisition. vol. 4. no. 4. pp. 445–476.

Eriksson H. and Musen A. M. (1993) *Conceptual models for automatic generation of knowledge-acquisition tools.* The Knowledge Engineering Review. vol. 8 no. 1. pp. 27–47.

Eriksson H., Puerta A. R., and Musen A. M. (1994) *Generation of Knowledge-Acquisition Tools from Domain Ontologies.* International Journal of Human Computer Studies. vol. 41 pp. 425–453.

Gamma E., Richard H., Johnson R., and Vlissides J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Welsley.

Kruchten P. (1995) *The "4+1" View Model of Software Architecture.* IEEE Software, vol. 12 no. 6. pp. 42–50.

Musen A. M. (1989) *Automated Generation of Model-Based Knowledge-Acquisition Tools.* Morgan-Kaufman. San Mateo, CA, USA.

Musen A. M. (1997) *Domain Ontologies in Software Engineering: Use of Protégé with the EON Architecture.* Jacksonville, Florida.

Puerta A. R., Egar J., Tu S., and Musen A. M. (1992) *A Multiple Method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-Acquisition Tools.* Knowledge Acquisition. vol. 4. no. 2. pp. 171–196.

Soloway E., Piont J., Letovsky S., Litman D., and Lampert R. (1988) *Designing Documentation to Compensate for Delocalized Plans.* Communications of the ACM. vol. 31. no. 11. pp. 1259–1267.

von Mayrhauser A. and Vans A. M. (1994) *Comprehension Processes During Large Scale Maintenance.* In Proceedings of the 16th International Conference on Software Engineering. pp. 39–48.

Woods S. and Yang Q. (1996) *The program Understanding Problem: Analysis and a Heuristic Approach.* In Proceedings of the 18th International Conference on Software Engineering. pp. 6–15. Berlin, Germany.

Appendix A

Javadoc

In this appendix, I present the Javadoc system and Dynamic HTML, which have been the underlying technologies in the DJavdoc system, presented in Appendix B. (The appendix is mainly constructed from extracts of my Licentiate Thesis no. 790 entitled "Use-Oriented Documentation in Software Development" published in November 1999 by the School of Engineering at Linköping University ISBN: 91-7219-615-7)

For more detailed descriptions of Javadoc, Doclets and the Standard Doclet see (Kramer 1999, Friendly 1995, Campione and Walrath 1998, JAVA). For more detailed description of DHTML see (W3C, Deitel et al. 2000).

A.1 Javadoc

Javadoc is a Java program that generates documentation from Java source code which includes *tagged* comments extracted by the Javadoc program, see Figure A.1. The Javadoc Team has defined a set of tags that will be recognized and also a list of reserved tag-names that may become used in future implementations. Javadoc can generate library reference documentation for any combination of Java classes.

Figure A.2 provides a screenshot of a Javadoc class document from Java SDK 1.2. For more detailed examples, visit Sun's Javadoc web site (JAVADOC).

The purpose of Javadoc and the library reference documentation is to support programming by providing an interface to the library source code. Programmers use library reference documentation to learn and use library source code. They could read the source code directly but it is time-consuming and may require complex analysis. The aim of the library reference documentation is to portray the functionality of the library, correctly and efficiently. The library reference documentation presents information deemed particularly important to programming.

Principally all Java code (provided by Sun Microsystems or third-party providers) is today presented in the form of Javadoc-generated documentation. Alongside more descriptive texts, Javadoc-generated documentation is the standard way of illustrating the available functionality on code level. For early releases of class libraries, Javadoc library reference documentation may be the only available documentation.

```
/**
 * Draws as much of the specified image as is currently available
 * with its northwest corner at the specified coordinate (x, y).
 * This method will return immediately in all cases, even if the
 * entire image has not yet been scaled, dithered and converted
 * for the current output device.
 * If the current output representation is not yet complete then
 * the method will return false and the indicated {@link ImageObserver}
 * object will be notified as the conversion process progresses.
 *
 * @param img       the image to be drawn
 * @param x         the x-coordinate of the northwest corner of the
 *                  destination rectangle in pixels
 * @param y         the y-coordinate of the northwest corner of the
 *                  destination rectangle in pixels
 * @param observer  the image observer to be notified as more of the
 *                  image is converted.  May be <code>null</code>
 * @return          <code>true</code> if the image is completely
 *                  loaded and was painted successfully;
 *                  <code>false</code> otherwise.
 * @see             Image
 * @see             ImageObserver
 * @since           JDK1.0
 */
public abstract boolean drawImage(Image img, int x, int y,
  ImageObserver observer) {
   ...
   ...
   ...
}
```

Figure A.1: An example of how comments are written into the source code using @-tags (for instance, @param) to structure the tags.

Figure A.2: Screen shot of the Javadoc 1.2 style class document.

The library reference documentation is a typeset view of the source code. Some parts have been removed, others have been relocated, and the text is typeset differently from the source code. Comments are generally added to the source code and these are presented in the documentation. In a sense, these comments describe parts of the source code that are of particular relevance.

The removals, relocations, and typesetting in the library reference documentation create a view of the source code. The design of the library reference documentation reflects assumptions about what programmers need to know. For instance, the choice is made only to present the signature of class members (name, parameters, and so on) but not the entire source. Also, members are summarized alphabetically, though inherited members are summarized separately in much shorter format.

A.2 Doclets

Javadoc is actually not one program but a structure of systems, illustrated by Figure A.3. The Java SDK 1.2 release introduced the *Doclet library* that exposes the documentation classes used by Javadoc. The Doclet library represents a development platform for Javadoc documentation. Printing is not part of the Doclet library, only the means to access information from the Javadoc program and the Doclet library can therefore be viewed as an abstract data type for Javadoc.

The Javadoc program uses a Doclet program to define the Java library reference documentation content and typography. By providing a Doclet class as an argument to the Javadoc program, the printing of the library reference documentation can be changed. For instance, a new Doclet could print only the names of classes in an index. In Figure A.3, the Doclet is given control by the Javadoc program and uses the Doclet library to access the information structure. The Doclet is intended to deliver an output of some sort (Kramer 1999) but can do anything within range of the Java programming language (e.g., surf the Web for Java classes that extend the classes in the Javadoc session).

A.3 Standard Doclet

The Standard Doclet is the Doclet used to generate the official hypertext-based Javadoc library reference documentation developed by the Javadoc Team at Sun Microsystems. In combination with the Doclet library, the Standard Doclet is delivered as the default Doclet class used by Javadoc to control output of the Javadoc batch programs. The produced library reference documentation focuses on class documents that contain listings and descriptions of the inheritance, fields, methods, constructors, and inner classes; basically a signature of the class. HTML links are used to represent relations among class documents. Return-value classes, parameter classes, and inheritance classes are linked from the class document.

A.3.1 Class Documents

The class documents are the core of the library reference documentation that the Standard Doclet generates. Figure A.2 shows a sample class document from the

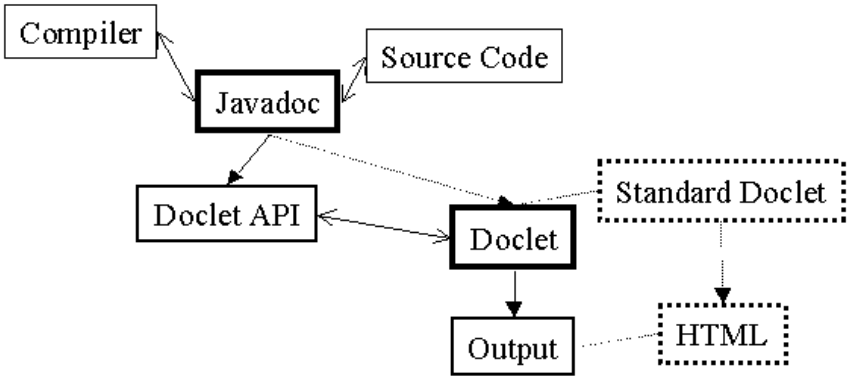

Figure A.3: The structure of systems that are used in a Javadoc session to deliver library reference documentation.

Standard Doclet of Java SDK 1.2. The purpose of the class document is to describe the class and its relations to other classes. As the primary knowledge source, the class document maps directly to the source code and presents the external signature of the classes but not the underlying implementation (i.e., the body of class methods). Hypertext is used to link from the class documents either to related class documents or to other parts of the document. There is a general hyperlink section at the top (and also at the bottom) of the class document (see Figure A.4). After the header, a general description of the particular class is provided, containing both technical information, for instance inheritance, and a general written description of the class (see Figure A.5). The next item in the class document is summaries of the class members. Inner classes, fields, constructors, and methods are summarized separately and ordered alphabetically (see Figure A.6). The summary contains information about the member's type, parameters, return values, and name (if they apply for the particular member type). Finally, the detailed descriptions of the class members are provided (see Figure A.7).

A.3.2 Table of Contents

The table of contents is the navigational device for the Standard Doclet library reference documentation (see the two right frames in Figure A.2). From the table of contents programmers find their way among the classes using hyperlinks. The underlying package structure of Java classes is the basic indexing model for the Standard Doclet. The packages and classes are listed alphabetically; packages in the upper frame and classes in the lower frame. However, classes are grouped into types (i.e., interface, class, exception, and error) before being ordered. The class list also contains information about the package the classes belongs to. Clicking on a package brings a new class list to the class frame. Clicking on a class loads the referenced class document into the class-document frame. Finally, there is also a document containing all classes ordered alphabetically.

A.3.3 Other Documents

The Standard Doclet also generates a few supplementary documents. Besides the class documents and the table of contents the Standard Doclet generates some supplementary documents that can be reached from the header and footer hyperlinks. I do not describe them in detail because they have not been part of the DJavadoc project. These documents mainly provide additional navigational views of the library reference documentation.

A.3.4 DHTML

DJavadoc is implemented using a mixture of Java and Dynamic HMTL (DHTML). DHTML is a term grouping client-side technologies used to create dynamics in HTML. DHTML is used to create Web pages that react to interaction and display different material in different contexts. Being more of a dynamic content and typography technology, DHTML cannot be viewed primarily as dynamic hypermedia. The general purpose of DHTML is not to define hyperlinks that can relate to different sources depending upon the context, even though this is possible. Instead DHTML is used to

Figure A.4: The header (and also footer) in the Standard Doclet contains additional hyperlinks. Some documents can only be reached via this navigation-bar. However, the header also pushes contents of the class documents down below the visible space in the browser.

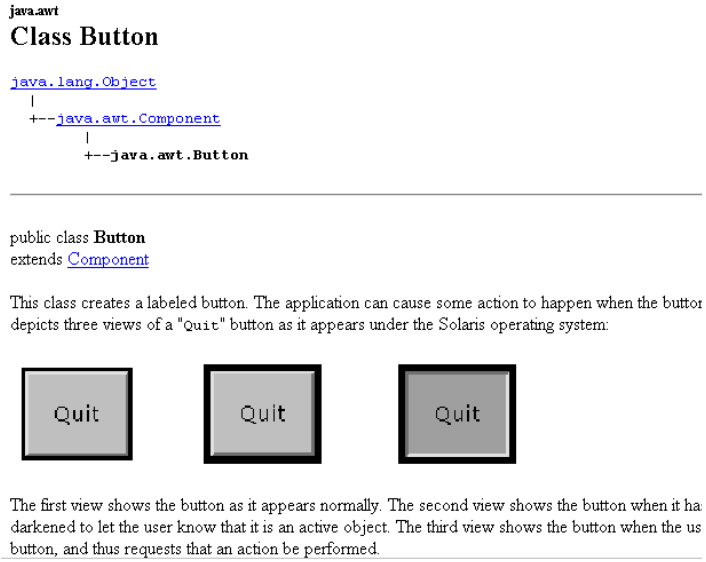

Figure A.5: The general class description is provided after the header in the class document. Both a description written by the developer and information derived from the source code (e.g., super classes and sub classes) are presented.

Figure A.6: The class members are presented in a summary, which may contain name, types, parameters, return-values and a brief description. If a more detailed description was provided with the source code, it is presented further down in the class document.

Figure A.7: The description of class members is provided separately from the summary description, see Figure A.6. It contains comments on the source code written by developers. The member descriptions can be reached via hyperlinks in the summary.

create graphically appealing and living pages that look good and feel professional. A popular example of DHTML is the *roll-over effect* that illustrates what hyperlink the programmer is about to activate. The collapsible list is another frequent example. The core DHTML technology is the *scripting language* used to introduce algorithmic behavior into the HTML-page. JavaScript was the first and is probably still the most used scripting language (JAVASCRIPT). The proposed international ECMAScript standard is currently an accepted standard of the major browsers, Netscape Navigator and Microsoft Internet Explorer (ECMA, ECMASTANDARD).

For DHTML purposes, the scripting language is used to manipulate functionality available in the browser, rather than to serve as a separate programming language. The more advanced browsers have an elaborate list of events that are fired when the user interacts or when the browser has performed certain steps. Scripts written in the Web page can be set as *handlers* of these events which will then be activated if the event is triggered. Scripting languages generally access functionality available in the browsers to manipulate or change the appearance of the Web page. The *document object model* (DOM) defined by the *world wide web consortium* (W3C) opens up the object structure of the Web page so that individual elements can be accessed and manipulated. DOM is a central component in DHTML that the major browsers do or will implement.

DHTML has not followed the tradition of HTML development, in which new and more complex tags have been developed as part of the markup language. During the evolution of HTML several new tags have appeared. The transition from a simple markup language to a typography-centered language has been driven by the introduction of new tags defining more complex typographical functionality. However, even though some DHTML applications have the potential of becoming tags (e.g., collapsible lists and roll-over images), no new tags have been introduced. Instead Web design is becoming more complex, involving several different technologies and taking the form of a programming language rather than a single markup language.

Appendix B

Dynamic Javadoc (DJavadoc)

DJavadoc has been a research vehicle in this research but is also a practical result of this work and is publicly available at `http://www.ida.liu.se/~eribe/djavadoc`. DJavadoc is also a system developed from a tradition of electronic reference documentation systems that provide different valuable features for contemporary reference documentation. This history is described in section rel.history in the thesis. (The appendix is mainly constructed from extracts of my Licentiate Thesis no. 790 entitled "Use-Oriented Documentation in Software Development" published in November 1999 by the School of Engineering at Linköping University ISBN: 91-7219-615-7)

The goal of the Dynamic Javadoc (DJavadoc) project is to find new ways of presenting and organizing the library reference documentation that will facilitate use-oriented reading: a task that includes navigation, information access, acquisition of detail syntax and semantic knowledge, knowledge of structures enforced on programmers by libraries, knowledge about the distinctions among components with regard to their possible and recommended use, and so on. Automated support is particularly interesting because automated reading tools scale better for rapidly evolving information sources. In addition, automatically generated documentation will not deviate from the source code.

More specifically, the DJavadoc project examines individual adaptation though real-time redesign (view creation) by taking advantage of an explicit underlying information structure. The official Javadoc output (see section A) delivers class documents that contain an underlying, implicit information model. The information model is illustrated by the typography of the documents in which, for instance, bold style is used to emphasize method names. In an adaptive environment, excessive information can be made less visible by changing the typography, for instance, by turning the color of uninteresting texts into something not quite distinguishable from the background and thereby graying-out parts of the document. Similarly, excessive information can be removed. As a result the visibility of the remaining information increases.

The purpose of the DJavadoc system is to provide a research vehicle through

which I can discover requirements of electronic library reference documentation. It is also a product, an alternative Java reference documentation that provides another type of support. In this appendix, I describe the DJavadoc system.

DJavadoc provides programmers with the means to individually adapt Java library reference documentation during work with and thereby construct more use-oriented designs. The Standard Doclet provides a view of Java source code. In DJavadoc I augment the Standard Doclet with the ability to temporarily remove information and thereby further specialize the view of the source code. This extended functionality is applied to reduce time-consuming, repetitive manual searching for well-defined information types.

In DJavadoc, the programmer controls the visibility of information types. The Standard Doclet, in essence, provides information for multiple needs, for instance the need of understanding a class or the need to look up names for coding purposes. DJavadoc, however, provides control over the visibility of information types (both on a group level and an individual level). The programmer can collapse and expand information, thus increasing the degree of visibility of relevant information. By removing certain parts of the information the programmer moves other parts up into the visible space of the browser. Also, by removing surrounding texts the programmer makes elements of greater importance more visible.

The information model used in DJavadoc is the explicit version of the implicit information model existing in the Standard Doclet. The static typography of the Standard Doclet exposes the model and I use it to define the information model that conceptually groups pieces of information into information types. The model is fairly straightforward, mapping directly to the structure of Java source code.

B.1 DJavadoc Interaction Principles

The main ideas behind the interaction principles in DJavadoc are efficiency and to follow Web conventions. Interaction is a complex domain with several important criteria. The work presented here does not concern interaction in particular, but it is important since the DJavadoc reference document is intended for real use. The concern for interaction principles in the DJavadoc project is that efficient interaction should be achieved and that Web tradition should be maintained. I have also aimed to be consistent and simple. For instance, I use Java naming conventions as visual queues for signals of particular interaction to avoid clouding the documentation with more labels or icons. It is important to bear in mind that the documentation is intended to be used by professionals, both in programming and Web conventions (the Standard Doclet is Web-based).

- *Blue, underlined* – represents text that can be clicked, either to follow a hyperlink or to perform a DJavadoc-specific action. Web tradition states that blue, underlined text is active, even though the Web is full of exceptions. Browsers generally display hyperlinks as blue, underlined text if the particular Web page does not state otherwise. It is also common that blue, underlined texts represent calls to scripts. I have chosen to follow this tradition for interaction through text: blue, underlined texts have functionality. For instance, class members can be collapsed and expanded by clicking the blue, underlined name of the class member. If no description is available, the name is colored black.

- *Class names* – signal additional interaction (but only in the class documents and the table of contents). Java programmers recognize class names by convention and on their placing in the library reference documentation. Classes can be saved to a bookmark list by clicking the left-hand mouse button or pressing the alt-key and clicking. Removing classes or packages from the bookmark list is achieved in the same way. The fact that the class is being saved or removed is illustrated by a short-time change in background (see Figure B.1).

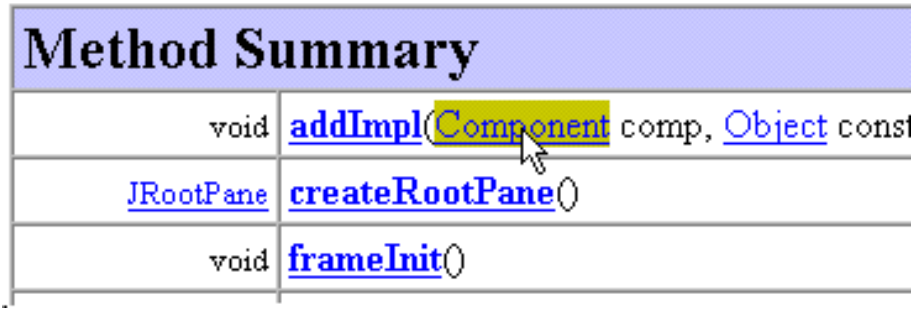

Figure B.1: When saving classes to the DJavadoc Bookarks the background of the class name changes for a short while to signal that the action was registered.

B.2 DJavadoc Features

B.2.1 General Redesign

In DJavadoc it is possible to perform real-time redesign of class documents by defining a default *setting* for the visibility of document elements. Pieces of information can be temporarily removed from documents, thereby both removing excessive information and increasing the visibility of the remaining information. In principle I have taken the existing information model, made it explicit, and created an interaction device with which the programmer can control the visibility of information types. The model is represented in the interaction device contained in the table of contents (see Figure B.2.) The class document is divided into two parts: a class description part and a member part. The class description contains a full inheritance tree, known subclasses, the declaration of the class (its name, super class and implemented interfaces), and a written description. The member types are inner classes, fields, methods, and constructors. The document provides a description part for each member and a section for inherited members of that type. Both the description and inherited members can be collapsed (however only fields and methods are inherited). The header and footer may also be collapsed, primarily to lift more of the class document into the visible space of the browser.

Figure B.3 shows the difference between the fully expanded document and a class document redesigned to exclude the class description. Notice that the Settings model

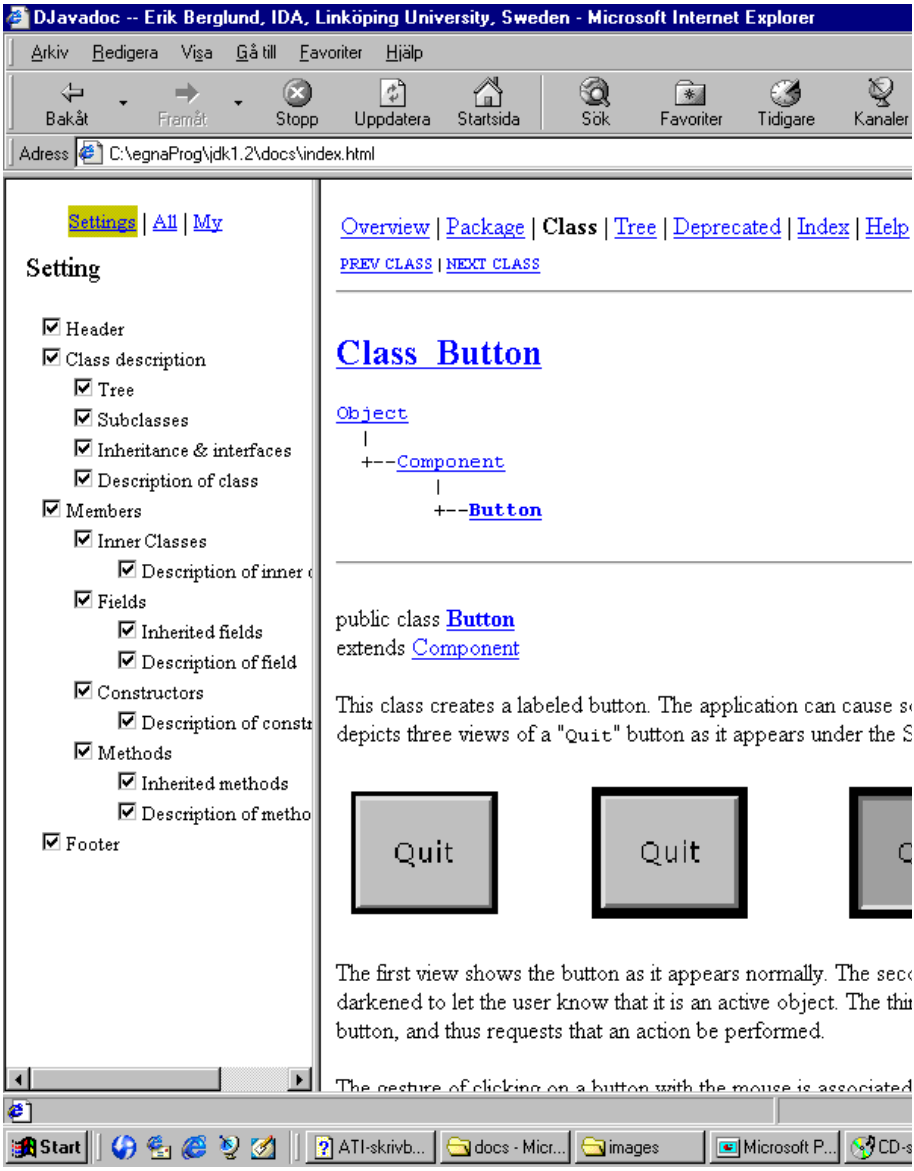

Figure B.2: In DJavadoc the table of contents has a `Settings-map` with which the default expand and collapse behavior of the class document can be defined. The reader checks off information types deemed uninteresting.

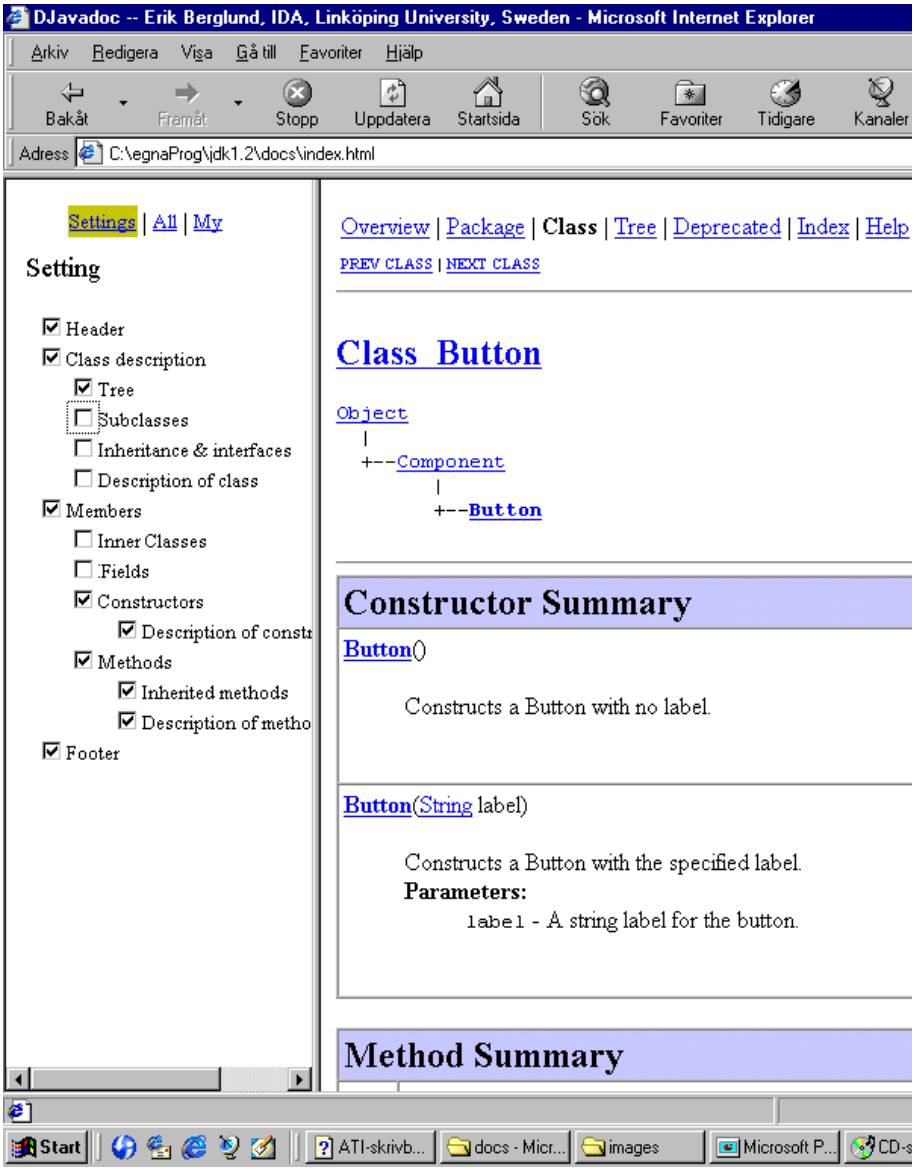

Figure B.3: As the settings are changed the class document alters the visibility of its elements. By checking off uninteresting information types the reader both makes important information more visible and remove excessive information.

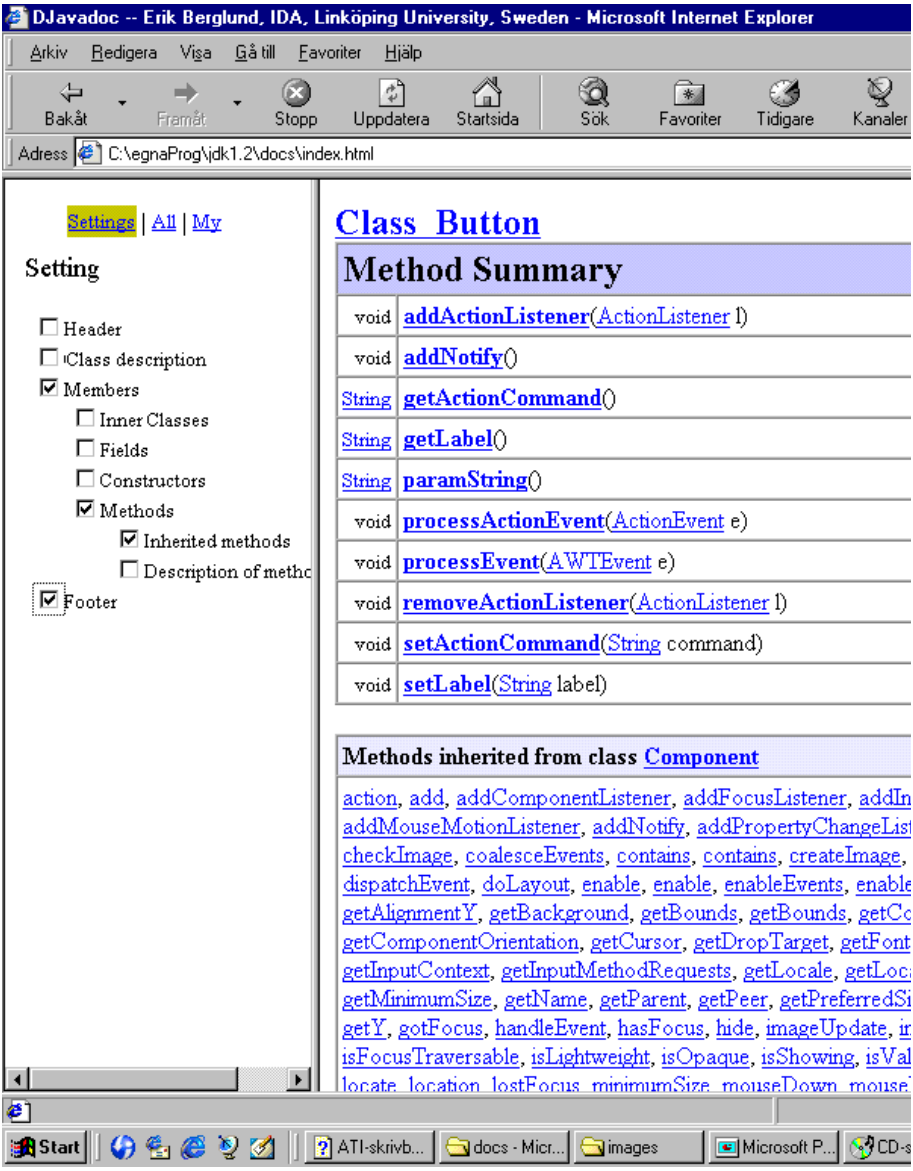

Figure B.4: Programmers may find a setting interesting in which only the methods are presented without displaying their descriptions. This example represents a compact, efficient typography of the document for programmers who are familiar with the particular classes.

also collapses itself. Checking off an element in the settings will lead to the collapse of all instances of that element in the document and the setting will be enforced on new documents loaded into the browser. As an example, checking off everything but the methods and the inherited methods is an relevant efficient setting, see Figure B.4.

B.2.2 Temporary Manipulation

In addition to the default setting, the programmer may also change the visibility of key elements by direct manipulation to expose individual parts of particular interest. The default setting defines which types of document elements programmers consider important. However, programmers might still want to explore the underlying information without having to change the default. It might also be the case that only individual document elements might be of interest. Therefore it is possible in DJavadoc to open up certain key-elements in the documentation by direct manipulation.

In DJavadoc it is possible to collapse and expand the description of class members by clicking on the name of the member (if such a description exists). It is also possible to collapse and expand the whole class description by clicking on the class name. (However, the default setting of the elements inside the description will not be changed.) Figure B.5 shows how an individual element is expanded.

B.2.3 Bookmarks

The table of contents can also be collapsed into a smaller subset of interesting hyperlinks in a so-called DJavadoc bookmarks list. With over 1,800 classes in Java SDK 1.2 it becomes obvious that the programmer needs more focused class lists to easily locate relevant classes. In DJavadoc this is achieved by constructing a separate table of contents in which programmers save classes in a bookmark fashion.

Figure B.6. shows an example DJavadoc bookmark. Classes and even whole packages can also be easily removed. The idea behind the DJavadoc bookmarks list is that the programmers should actively update it to keep the content relevant. The programmer's interest in classes changes both during a project and between projects. Filling the bookmarks classes of relevance is a good initial way of collecting potential classes for future programming. However, if the bookmark list grows large, it becomes less useful. Therefore programmers should continually revise their bookmark list.

B.2.4 Copy/Paste Support

As a result of the study presented in Paper I the DJavadoc project includes a copy/paste support design. Programmers need to transfer code from documentation to source code and explicitly the documentation therefore should support this task. In DJavadoc, two copy/paste strings are provided that can be copied manually or by a single mouse click. The first string represents the minimal syntax that can be reused by programmers. The second string also contains information about what programmers must add. Figure B.7. shows this design.

The idea of the copy/paste strings is to provide efficient support for copying class member syntax. The use of hyperlinks unfortunately makes copying more difficult since the mouse pointer keeps turning into the hand symbol from which it is not

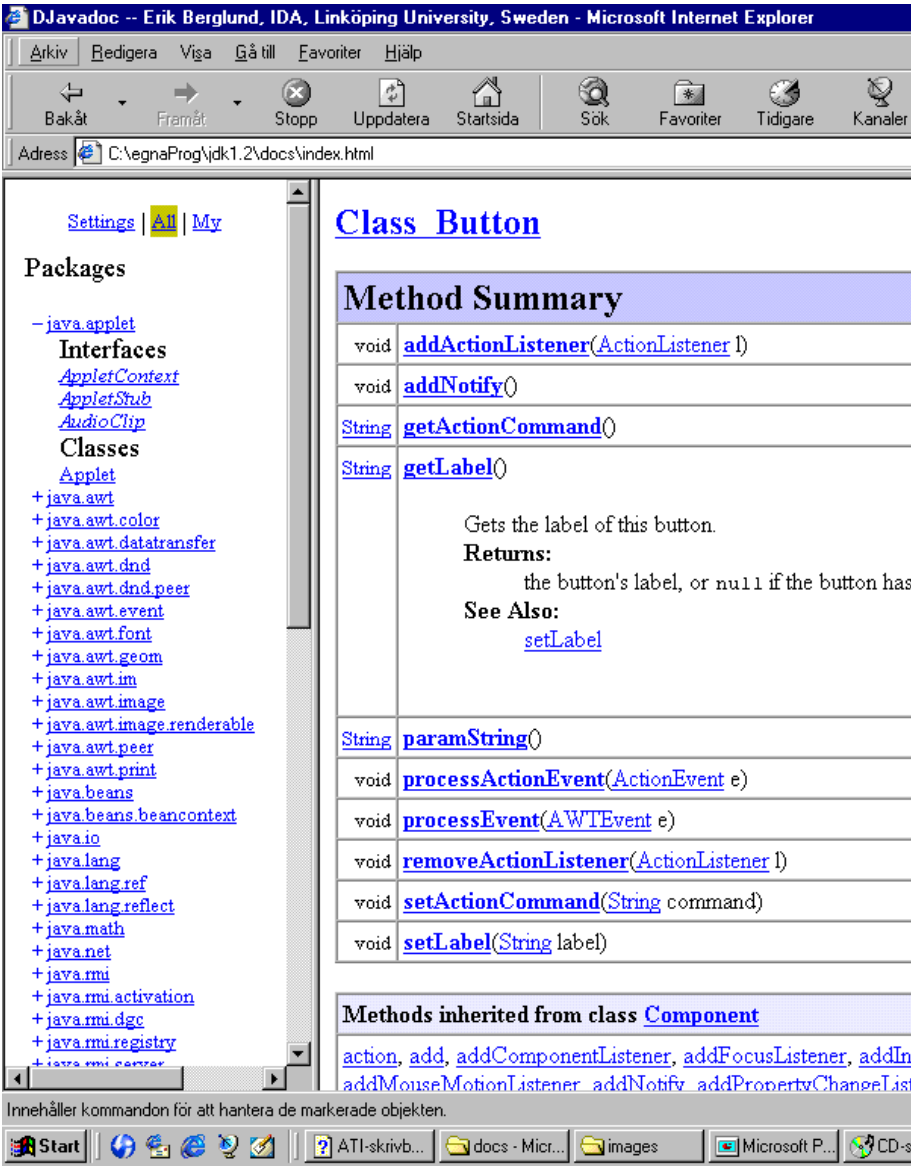

Figure B.5: While reading the class document, the programmer can use the dynamic typography features to expand or collapse individual elements regardless of the default setting. In this example, the programmer has expanded a particular method description. The programmer actively chooses which information should be visible in the browser.

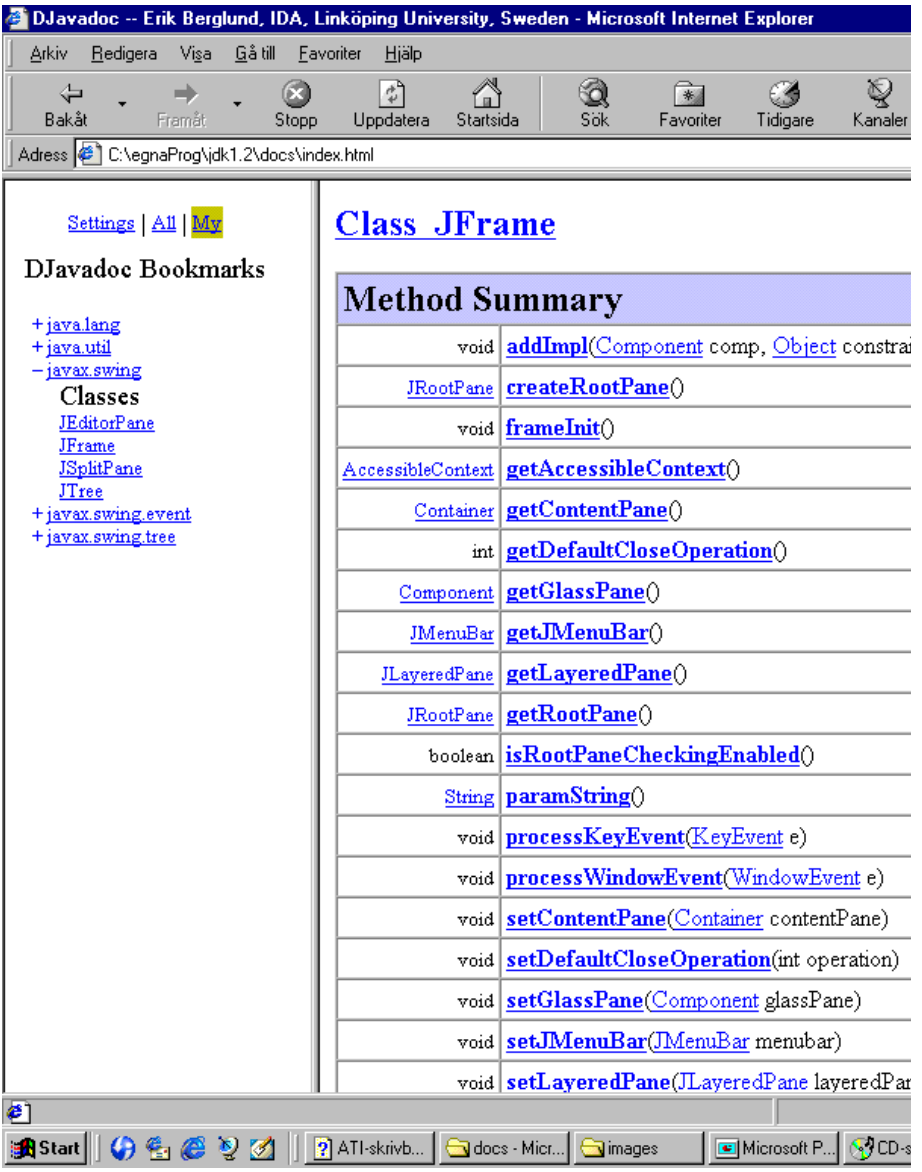

Figure B.6: DJavadoc has a My-map that represents a personal view of the entire table of contents. In a bookmark fashion, classes are saved in and removed from the My-map by the programmer. To keep the content relevant, the programmer should update the My-map.

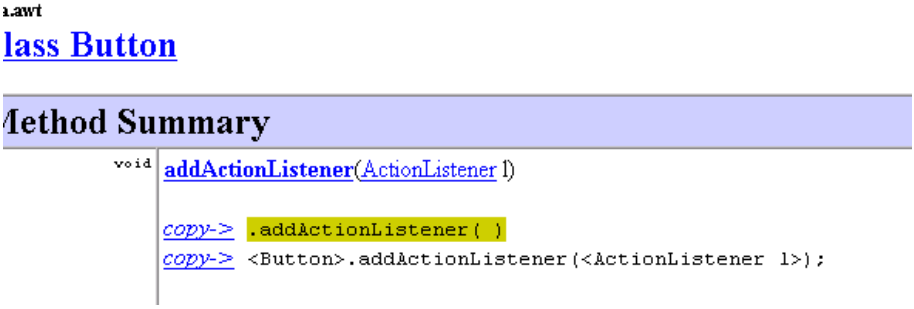

Figure B.7: By providing multiple strings designed for copy/pasting by direct manipulation DJavadoc address the need to transfer code from documentation to source files.

possible to copy text. Furthermore, the signatures used in the Javadoc documentation cannot be directly used in code and therefore I provide additional copy strings. What constitutes suitable copy/paste string design is an empirical question that needs to be evaluated.

B.2.5 Gray-out of Deprecated Methods

As another result of the study presented in Paper jss the DJavadoc project includes a graying-out of deprecated methods. Deprecated methods are methods that no longer are recommended and that may not be supported in future Java releases. In order to filter out deprecated methods, I have added a filter to the settings described in section B.2.1. If a programmer decides to filter out deprecated methods they are displayed in a color not quite distinguishable from the background and in this sense disappear. Figure B.8 shows the use of gray-out filtering of deprecated methods.

Applying gray-out filtering is an alternative approach to the collapse and expands mechanism used previously in DJavadoc. It provides knowledge about the material which has been removed in terms of size and location but still removes the content of the text from the documentation.

B.3 DJavadoc Performance and Technical Data

DJavadoc was developed and tested on a 200 MHz PC. In most cases the redesign of the documents did not result in any notable time delay. However, for larger rearrangements, such as collapsing all the class-member descriptions in one of the larger Java SDK classes, some delay is experienced. However, this is without performance optimizations. To give an example, I solved the performance problem of the table of contents, see section B.5.4.

DJavadoc does not add to the Javadoc generation time and memory requirements. Javadoc 1.2 requires 120 MB of memory to generate the Java SDK 1.2 documentation

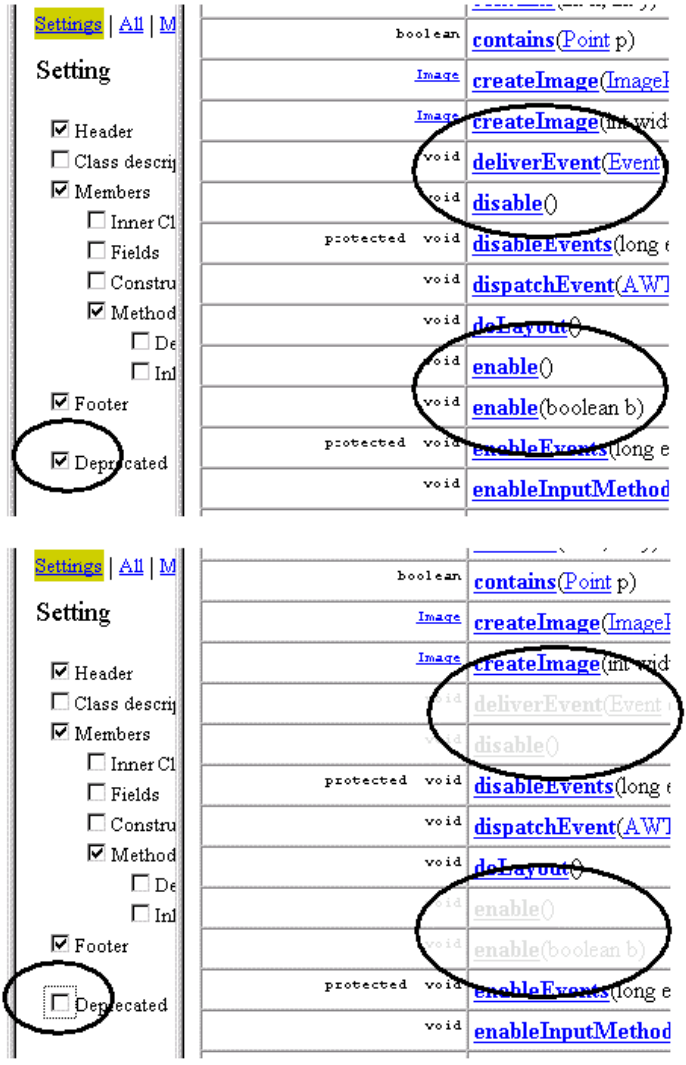

Figure B.8: Deprecated methods, methods that are may not be supported by future releases, can be grayed-out in DJavadoc and thereby disappear visually while still providing information about the size and location of missing content.

(55 packages), which takes 8 minutes on an Ultra Enterprise with 512 MB of memory (JAVADOCMEM). Javadoc keeps the Doclet library information structure in memory and the memory requirements are therefore very large. Time is a factor of the sheer size of the Java SDK 1.2. The extensions made in DJavadoc have a minimal effect on the time or memory requirements of Javadoc.

The extensions made in DJavadoc are principally rearrangements and additions to the Standard Doclet and the inclusion of DHTML scripting. The total amount of work that has gone into the DJavadoc project's implementation phase is about 7 man-months (spent understanding, adding changes and rearranging the Standard Doclet, and designing DHTML scripts). The amount of code that has been added totals some 2,500 lines of code in Java and in Javascript. Of course, to a great extent, the implementation work has been a matter of understanding the Standard Doclet, which is a complex program of over 11,000 lines of code.

B.4 Example Working Scenario for DJavadoc

Let us consider a sample scenario describing how a programmer might use the dynamic functionality of DJavadoc. A working example can put the usefulness of DJavadoc into perspective. DJavadoc may be interesting but I also hope to show its practical importance. In our example the programmer is using the library reference documentation as a syntactical index that provides coding specification. The programmer knows which classes to use and what those classes are designed for.

The programmer has defined the setting in the Settings-map shown in Figure B.9. The methods are kept expanded but with collapsed descriptions. The programmer uses this setting to learn method names, return values, and parameters both as a means of discovering methods of interest and to remember the exact syntax. During browsing among class documents, the documentation is transformed from the fully expanded view to the chosen default setting.

The programmer has an ongoing project and therefore starts browsing from the My-map, containing the personal DJavadoc Bookmarks. During previous browsing the programmer has collected a list of class documents of interest (see Figure B.10). DJavadoc bookmarks represent personal views of the table of contents. Our programmer uses the bookmarks to access certain documents that are used in the current programming project. The bookmarks consist of small fragments of several packages since application programming generally involves classes of different characters.

While reading, the programmer makes temporary manipulations to display the inner workings of elements of interest (see Figure B.11). In the default setting the programmer may only define the visibility of information types, not instances. The collapsed descriptions represent a type that is not of interest to our programmer. However, specific individual descriptions are relevant during coding and therefore our programmer expands a few descriptions while reading (by clicking on the method names). For instance, the descriptions may reveal how parameters are interpreted.

The programmer saves new classes to the bookmark list. During browsing the programmer comes upon new, potentially useful classes. By left-clicking the programmer saves new bookmarks that are added to the list according to the package structure. For now, our programmer saves without giving what he saves much consideration because later the programmer removes classes that in retrospect were not

Figure B.9: The programmer chooses a default setting which only presents methods without description but with the inherited methods. The document is altered according to the new setting, as is each new class document the programmer loads into the browser.

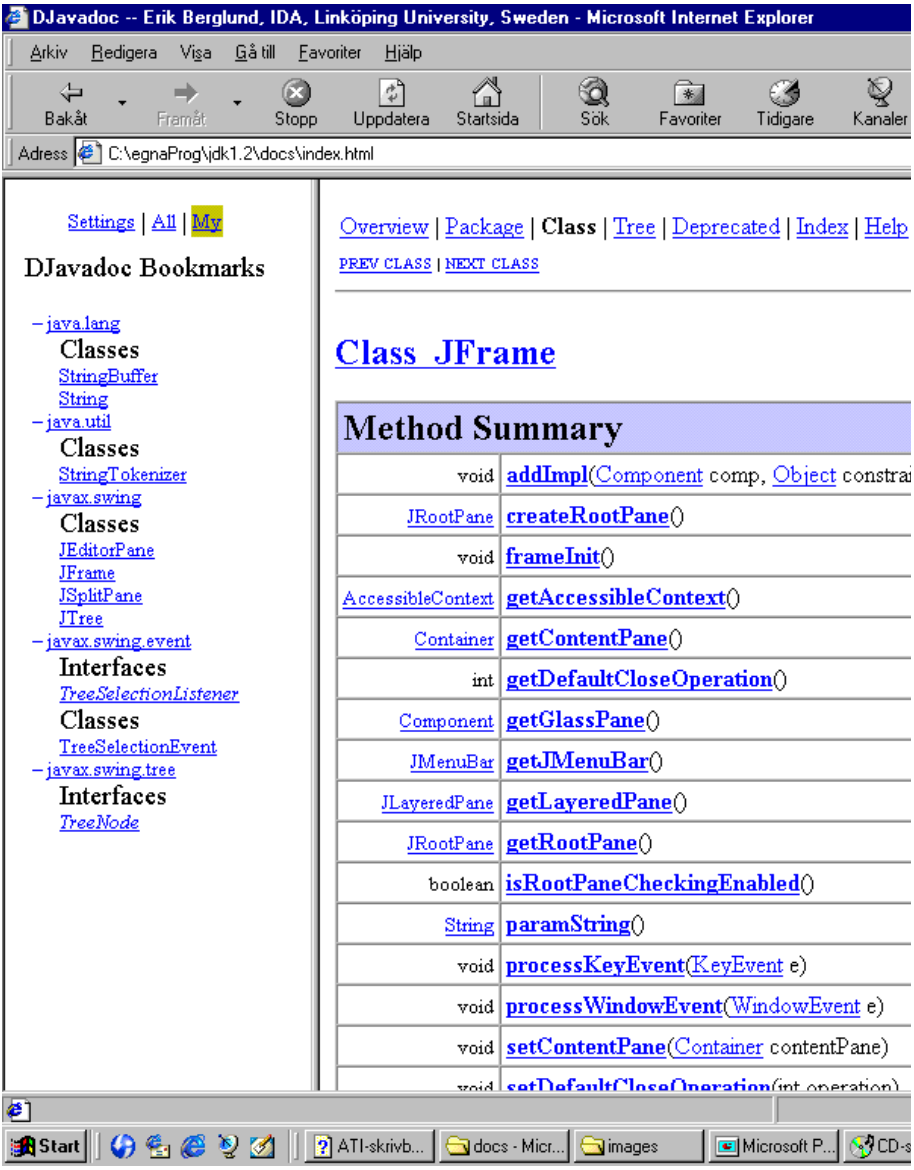

Figure B.10: Previously the programmer has created a My-map containing classes of interest. For instance, the programmer might use the classes in a programming project. Another plausible reason is that programmer often uses the classes.

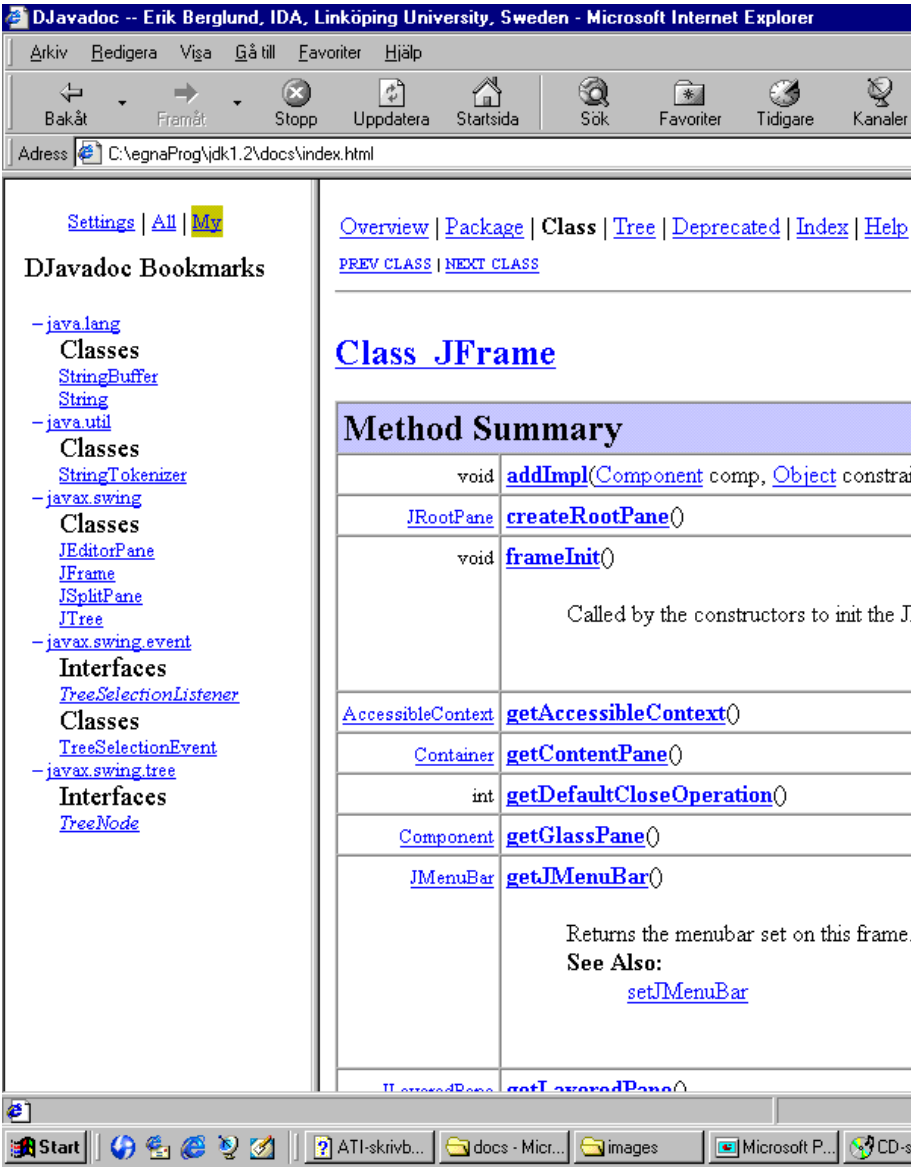

Figure B.11: Whilst reading, the programmer finds it relevant to open up the description of certain methods. Perhaps the programmer needs to be reminded about the meaning of a return value.

so relevant.

After a while, the programmer changes the default setting to include constructors. The programmer is well aware of the missing components in the documentation. At some point the programmer realizes that constructors are missing from the documentation and changes the default. The constructor description is left collapsed just like the description of methods.

B.5 Preceding Prototype Generations

Leading up to the DJavadoc project, time was spent designing mock-up prototypes that where informally tested. These are described here.

B.5.1 Conceptual Source-Code Organization

The DJavadoc project has its origin in a project on acquisition and visualization of intermediate knowledge in code level programming though conceptual source-code organization, see Paper VI. In this project I used Protégé, a tool for generation of knowledge-acquisition tools (Eriksson et al. 1994, PROTEGE) to develop a prototype tool for conceptual grouping of source elements of Java programs. The idea was to enable swift documentation of a program's conceptual relations as seen by programmers.

B.5.2 Pop-up Information Hiding (first DHTML)

Pop-up frames were used in the first Javadoc-related DHTML generation that started the DJavadoc project. In the first generation I experimented with Javadoc documents by applying DHTML technology. The goal was to determine what could be done using these new technologies for dynamic display and manipulation of HTML elements. In the pop-up generation I used Netscape 4 and the DHTML-support that was available for that browser. The basis of the pop-up was to use tool-tip-like pop-up sections to present information in the Javadoc class documents. The table of contents was also developed in much the same way as the final DJavadoc. During this generation it became apparent that Internet Explorer 4 provided more DHTML support.

B.5.3 Conceptual Filtering

Information filtering based on conceptual grouping of class members was the topic of the second DHTML generation. I continued experimenting with the Javadoc class documents, now with the full DHTML-capacity of Internet Explorer 4. The DHTML support enabled smooth manipulation of HTML elements in real time, which I experienced as an important step forward in technology for the DJavadoc ideas. The aim here was to differentiate class members using filters based on conceptual member types (e.g., basic methods and event-related methods). I enabled this both by graying out methods (i.e., changing the color to something not quite distinguishable from the background) or by collapsing them. Descriptions could also be collapsed or expanded. The table of contents was principally the same as in the final DJavadoc version. Figure B.12 shows the resulting mockup from this generation.

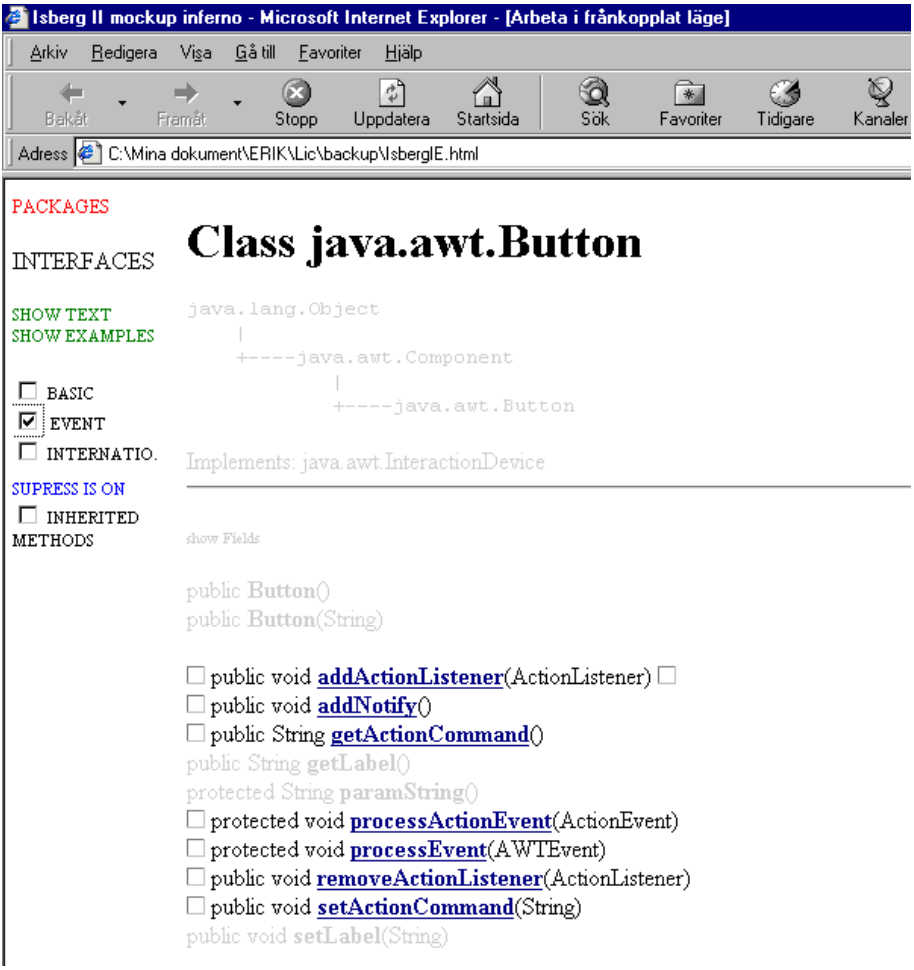

Figure B.12: The second DJavadoc generation used much the same dynamic typography as the final version. The main idea in this generation was to differentiate class members conceptually, which I still consider a rewarding approach.

Although a strong idea, conceptual grouping of class members scales less well than the final DJavadoc project since it requires some form of expertise. The conceptual filtering of class members requires conceptual knowledge. An expert would have to define what categories were relevant and record which members were contained in what category for the whole of Java SDK. Thus, conceptual filtering scales less well than the final DJavadoc version. Gathering the knowledge would require expertise, time, and knowledge-acquisitions tools. The final DJavadoc version illustrates the same dynamic typography features in the computer-reading environment with much less work. However, I still believe that conceptual filtering of class members is a good way to reduce the overhead of reading library reference documentation.

B.5.4 Scaling the Final Implementation

When I scaled the final implementation to Java SDK 1.2 size I ran into performance problems with the table of contents. The DJavadoc table of contents is not a very large HTML-file. For Java SDK 1.2 it requires 300 Kbytes. However, it is very compact in the sense that over 75 percent of the file consists of HTML tags. In effect, the table of contents represents a tree structure of more than 1,800 entries that Internet Explorer must traverse in most of the scripts I have designed. This, of course, slowed down DJavadoc, which would affect the project. Even though the lag time was in the vicinity of normal hyperlink-access time, I felt that the performance had to be improved. Also, for larger class libraries, for instance future versions of Java SDK, the problem would perhaps grow.

Storing chunks of HTML in comments as unparsed text and extracting them on demand solved this problem. The table of contents consists of a nested list of packages and classes. Only a few class lists would be open at the same time and still they were responsible for the bulk of the HTML elements. By placing the class lists in HTML COMMENT objects I were able to reduce each class list to one HTML element in the parser's perspective. On demand I could then lift the class list out of the COMMENT and paste it into the package list, which activates the rebuilding of the tree. Consequently, good performance was restored.

B.6 DJavadoc Future Work

B.6.1 DOM and XML Compliance

A natural step for DJavadoc, as well as for Javadoc, is to move towards an XML base format to avoid using a Javadoc specific data format and allow for more general redesign based on style sheet specification. There has been talk about an XML Doclet for a long time in the Javadoc Team at Sun Microsystems but so far nothing has happened. For DJavadoc, it is also equally important to redesign the Microsoft Internet Explorer specific DHTML support to the *document object model* (DOM) implementation to adhere to open standards. DOM compliance would make DJavadoc general to many web browsers. At the time DJavadoc was developed neither XML nor DOM was integrated into the standard Web infrastructure but today both are commonly accepted technologies in major web browsers such as Microsoft Internet Explorer and Netscape Navigator.

B.6.2 Server Javadoc (SJavadoc)

Javadoc is a batch program that generates documentation for distribution to programmers. The problem with this design is that programmers download documentation from many different source which all have been produced in separate batch runs. As a consequence, Java programmers have little ability to integrate their different library documentation sources, making it not only more difficult to work with the documentation but also to see relations among interrelated libraries. Instead Javadoc should be based on the distribution of Doclet files and local generation of documentation from a programmer's documentation sources. Each time a programmer downloads a new documentation source, SJavadoc would generate the necessary addition to the Javadoc documentation viewed locally with the web browser.

B.6.3 Improving the Class Documents

A simple but efficient means for improving Javadoc is to change the layout of the class documents. In the DJavadoc project, I did not address the layout to provide documentation similar to what the one the Standard Doclet delivers. In my view, however, more use-oriented layout can be achieved. For instance, by placing the method summary on top because this is the most commonly used information by programmers (currently found at the bottom below the description, fields, and constructors). Another example is to place get and set methods (a Java naming convention) together. Yet another example is to use nested methods list for methods with the same name but different parameters.

Altering the contents of the class document is another way of developing effective reference documentation. Currently the class document delivers only the signature of the class members (i.e., return value, name, parameters and so on). However, in many cases the body of the class members holds relevant information. Particularly if the written comments are sparse or if the class is complicated, programmers might need to analyze the source code. Altering the contents of the class documents will, of course, affect the amount of knowledge that can be derived from the library reference documentation.

The conceptual-filtering direction, see section B.5.3, should be further developed as a way of reducing the information. During the conceptual-filtering generation of DJavadoc, I examined the possibilities of grouping methods on the basis of their conceptual character (e.g., event-related methods, basic methods, methods used primarily by another class in a component structure). In the DJavadoc project the conceptual-filtering approach was abandoned because it could not be easily automated. In a sense it could be resolved with a new Javadoc tag requiring retagging of all Java classes. Javadoc has already declared a number of new tags that I find highly relevant (JAVADOCTAGS). In my experience, several method lists are filled with redundant methods of little or no relevance from a use perspective. These seldom-used methods are currently presented as equals to other methods. In fact, the methods are presented as more important than inherited methods that may well be more central to the use of the class.

B.6.4 Indices

The DJavadoc bookmark in section B.2.3 is one example of several applications of interest for information filtering on a navigational level. However, it requires much activity on the part of the programmer. There is a need for several indices such as the DJavadoc bookmark based on alternative sources.

Another useful type is *application indices* that point to a group of classes useful for particular applications profiles. For different types of applications (e.g, client-server applications and database applications) different parts of Java SDK, for instance, are relevant. The groups are perhaps primarily located in one or a few packages but the package structure does not cover all relevant applications. For instance, certain widgets are used more frequently in database applications than others and therefore some widgets should be part of the database application index but not all (compare tables and canvases). Proficient Java programmers could design application indices.

History indices is another example. By tracking the programmer's browsing behavior a history index could be designed to present the most frequently and most recently accessed classes. To avoid replication of the common browser backward and forward lists, the history index should perhaps not rely only on the most recently accessed class documents. A useful heuristic would have to be empirically discovered in relation to the Java domain to balance frequency and degree of recentness.

A *context index* that draws its entries from parsed source files is a third relevant example. The context in which the programmer is currently working provides relevant information for a filtration effort. Ultimately, by coupling the editor and the library reference documentation the context information could be exchanged automatically. As a first step, a context index could parse source files specified by the programmer (thus removing the tool-synchronization step). The context could be determined by detecting all classes and class members in use. However, the context index requires access to the files and therefore comes into conflict with general Web-security policy. This problem can be overcome both by server-client and client solutions. Ideally, I would like to see a strict client solution but a solution including a local server would also work.

A *use-based index* would describe how the Java SDK classes are used. By statistically analyzing large numbers of Java files a usage-based filter could be implemented. For instance all Java files available on the Internet could be analyzed, both Java and class files. Another alternative is to analyze all Java files on the Sun Java web site (JAVA). A third statistical source could be online-tutorials such as the Java Online Tutorial (Campione and Walrath 1998) to determine which classes and class members are relevant. A fourth example is an index based on votes from Java programmers all over the world. Furthermore, non-statistical methods could be used. Developers of the Java language could, for instance, design use-based tables-of-content (as could any Java programmer).

Project indices could also prove valuable. In fact, the process of defining a project index could be used as a project-standardization process. The classes that the group decides to use could be assembled into the index to provide active information for coding conventions. Furthermore, the joint browsing history of the projects members could be used as a basis for a project index, which could then be used to disseminate knowledge about classes of interest among group members.

For many of the indices discussed here, class members could also be included. If

a small but relevant set of classes can be realized, it could also be relevant to track class members, particularly methods, and enter them into the indices. In some cases it could be relevant to order methods under the class which they were referenced from (not always the implementing class, for instance for inherited methods).

B.7 DJavadoc Summary

DJavadoc adds individual adaptation and evolution to Javadoc to enable user-controlled views of the Java library reference documentation In DJavadoc the programmer can redesign in real-time the library reference documentation. As a result, views of the library source code that are more in line with different programmers' needs can be created (and recreated as the needs change). DJavadoc also support directly programmer need to transfer source code from documentation to source files.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson**: A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert**: Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund**: Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson**: The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh**: Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson**: Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell**: Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas**: Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson**: Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald**: The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin**: A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström**: A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov**: Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham**: Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson**: A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren**: Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson**: Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty**: NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck**: Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson**: Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist**: Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier**: A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea / ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin**: Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm**: Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald**: Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth**: Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson**: Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering - att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm**: Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic**: Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson**: Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind**: Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin**: Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.