Linköping Studies in Science and Technology

Thesis No. 790

Use-Oriented Documentation in Software Development

by

Erik Berglund

Submitted to the School of Engineering at Linköping University in partial fulfillment of the requirements for the degree of Licentiate of Technology

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

Linköping 1999

Use-Oriented Documentation in Software Development

by

Erik Berglund

Nov. 1999
ISBN 91-7219-615-7
Linköping Studies in Science and Technology
Thesis No. 790
ISSN 0280-7971
LiU-Tek-Lic-1999:48

ABSTRACT

*S*oftware documentation is an important tool in modern component-based programming. Building software applications requires detailed knowledge about a vast number of components and the structures they form. This knowledge is often acquired by reading reference documentation of application-programming interfaces (APIs). Thus, the design of the API reference documentation and its reading support affect the cost and quality of software development.

We examine how efficiency and quality in software development can be increased through the design of software documentation and reading support for software documentation. The thesis reports on the DJavadoc project and the reading support for online Java API reference documentation that it provides. The Java API reference documentation can be viewed as a collection of documentation designed for multiple needs. As a consequence, excessive information is present in most situations. In DJavadoc we have extended the official Java API reference documentation to achieve control over the visibility of information types. DJavadoc adds client-side, real-time redesign to the documentation to support the design of multiple views. As a result, the reader may further design views of the information that are more in line with the reader's personal and changing needs. In the thesis we also discuss online API reference documentation and its role in programming.

Our preliminary studies support the design strategy taken in DJavadoc. The DJavadoc architecture has also proven suitable for continuos redesign of online documentation. Furthermore, our work provides several future research directions for software documentation and communication of functionality. The Javadoc approach can be developed to achieve more use-oriented documentation. However, the need of use-oriented documentation may also have impact on the Java programming language and ultimately object orientation.

This work has primarily been supported by SSF (Swedish Foundation for Strategic Research) through ECSEL (Excellence Center in Computer Science and Systems Engineering in Linköping). Furthermore, our 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.

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

Abstract

Software documentation is an important tool in modern component-based programming. Building software applications requires detailed knowledge about a vast number of components and the structures they form. This knowledge is often acquired by reading reference documentation of application-programming interfaces (APIs). Thus, the design of the API reference documentation and its reading support affect the cost and quality of software development.

We examine how efficiency and quality in software development can be increased through the design of software documentation and reading support for software documentation. The thesis reports on the DJavadoc project and the reading support for online Java API reference documentation that it provides. The Java API reference documentation can be viewed as a collection of documentation designed for multiple needs. As a consequence, excessive information is present in most situations. In DJavadoc we have extended the official Java API reference documentation to achieve control over the visibility of information types. DJavadoc adds client-side, real-time redesign to the documentation to support the design of multiple views. As a result, the reader may further design views of the information that are more in line with the reader's personal and changing needs. In the thesis we also discuss online API reference documentation and its role in programming.

Our preliminary studies support the design strategy taken in DJavadoc. The DJavadoc architecture has also proven suitable for continuos redesign of online documentation. Furthermore, our work provides several future research directions for software documentation and communication of functionality. The Javadoc approach can be developed to achieve more use-oriented documentation. However, the need of use-oriented documentation may also have impact on the Java programming language and ultimately object orientation.

Acknowledgement

First and foremost the I would like to acknowledge the importance of my supervisor Henrik Eriksson for his dedication and valuable support. I am grateful for his constant availability, detailed supervision, and tactical support. Furthermore, I am also grateful for his keen interest in my project and the fascination of technology that we share.

Second to none is Magnus Bång, fellow Ph.D. student and close friend. Magnus had become a comrade in arms in the struggle for academic success. Our constant discussions and the valuable views he brings to them are much appreciated. Also, his philosophical skill has contributed much to my thinking. Coming from an engineering background, I have been fortunate to team up with Magnus.

I would like to thank my secondary supervisors Kjell Olhsson and Sture Hägglund for being part of the research escapade.

Continuing, I would like to thank my colleges at Linköping University (ASLAB, HCS, IDA). In particular I would like to mention Rego Granlund, Johan Jenvald, Johan Lübcke, Kristian Sandahl, and Eva Ragnemalm.

Thank you, Ivan Rankin, for improving my English. Hopefully, assuming that the English language has not been mistreated, I do not owe it all to you.

Ulf Magnusson at Ericsson, Mike Shrag at Experient Technologies, and Douglas Kramer at the Javadoc Team all deserve thanks for comments and enthusiasm.

I would like to thank SSF (Swedish Foundation for Strategic Research) for being our primary financial supporter through ECSEL (Excellence Center in Computer Science and Systems Engineering in Linköping). Furthermore, our 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, thank you family (Båge, Margareta, and so on) and close friends (the boys and last but definitively not least Charlotte Immerstrand). A particularly warm thought of gratitude goes to Hoffman, probably the best dog in the world.

Contents

Chapter 1

Introduction

Programmers read *application programming interface (API) reference documentation* as part of the programming task because they commonly use reusable components that they must have detailed knowledge about. The API reference documentation may not support programmers well because it is designed for multiple information needs, for instance both to describe these reusable components in general and to provide syntactical details. As a result, the API reference documentation contains *excessive information* in most situations. Naturally, sometimes there is too little or irrelevant information as well. Programmers must manually locate the relevant parts of the information or even read unnecessary large portions of text to extract the relevant information.

This thesis reports on the *dynamic Javadoc (DJavadoc)* project and the online API reference documentation for *Java* programmers it delivers. DJavadoc is available on the Internet for public use at `www.ida.liu.se/~eribe/djavadoc` [44]. The aim of the DJavadoc project is to support programming by further developing API reference documentation as a programming tool. More specifically, we provide user-controlled adaptation of the Java API reference documentation by taking advantage of the dynamic properties of the computer as a reading environment (in comparison with print) and the homogenous nature of API reference documentation. From a software-engineering perspective, the work focuses on the API reference documentation as a *programming tool.* From a human-machine-interaction perspective, the work is oriented towards features of the *computer-reading environment.*

1.1 Research Problem

Modern software development is based on the reuse of predefined software components, for instance, in the form of objects or functions. This approach is particularly central to object-oriented programming since the paradigm is based on the reuse and further specializing of predefined classes. Applications are generally built on top of APIs. These APIs can be in the form of function libraries, class libraries, or other sets of programming-language constructs. As a result, the composition of programs requires detailed knowledge on a vast number of components, as well as the context in which these components fit (e.g., the runtime environment), and sets of alternative components. APIs tend to grow large, a tendency of which the *Java Development Kit (JDK)* is an excellent example. The number of components is high in JDK, especially if the methods of classes are regarded as components in their own right. In JDK 1.2 there are about 1,800 classes and 15,000 methods and the API reference documentation alone is 80 Mbytes in size. These numbers can be compared with the 600 classes and 4,000 methods and 8 Mbytes worth of API reference documentation of JDK 1.1. In a programming task, only a few components may be of interest and some less often than others. Knowing which components to use and how to use them is a central part of the Java-programming task.

Thus, programmers spend much time informing themselves about components, mainly by reading API reference documentation. The API reference documentation is used for multiple purposes. Readers may remind themselves about coding syntax or read about new components. There are also relations among components that readers must understand to use the components properly. Compared to other work-categories, programming is a knowledge-intensive activity in regards to the need of constant acquisition of detailed knowledge since it is important to produce syntactically as well as logically correct code. Programmers must continuously double-check their information to avoid simple but time-consuming errors.

One problem with the official Java API reference documentation is that it contains excessive information in all work-situations since it is designed for multiple needs. We can view the official Java API reference documentation as an information source designed to meet several needs, as several intertwined pieces of documentation. For instance, the documentation describes components in general and their relations to their surroundings and also provides syntactical details. However, *professionals* focus on the task at hand when they read *work-related texts* such as the Javadoc-generated API reference documentation and search only for relevant information [15]. In a study of consumer reading behavior for instruction manuals Schriver [18] showed that most informants tended to skim when reading. Ideally, the reader should be presented with only the desired information in the right format. Unfortunately, this is different in different situations and for different readers. In our view, the problem of finding the right information is particularly important in the type of text that the official Javadoc API reference documentation represents since, in our experience, manual search is performed continuously (reading involves the continuous loading of new documents which have to be searched for the right information).

The overhead of reading API reference documentation can become an impediment in software engineering. In a component-based-programming paradigm, the information overload of the API reference documentation can have a serious effect on the programming task and therefore ultimately on the resulting software. The API reference documentation's ability to present the right information can effect both cost and quality of software projects. For instance, the time it takes to find the right information and understand the content of that information is a direct addition to the total programming time. Also, if it is difficult to understand how and when to use components, programmers may reinvent components. Though necessary at times, reinvention of components takes time and can also have an impact on the quality of the application since the risk of errors is increased (granted that the amount of bugs is proportional to the size of the source code).

We believe that the automatically-generated Javadoc API reference documentation is a step in the right direction but also that improvements to it will have a significant impact on Java programming since programmers spend much of their time reading it. To our mind, the online Javadoc API reference documentation supports Java programming. However, as the size and complexity of JDK increases, it becomes more vital to make improvements to, for instance, the content, typography, and organization of the API reference documentation; simply because the number of components is rapidly increasing, the need for more use-oriented API reference documentation increases.

1.2 Research Area: API reference documentation

From a software-engineering perspective, we work with API reference documentation as a programming tool. Without tools such as compilers it is extremely difficult to develop programs. However, programmers probably spend less time using the compiler than API reference documentation. The API reference documentation perhaps has more impact on the working conditions of programmers than does the compiler. The content and typography of the API reference documentation become important to software development, as do other reading issues such as navigation. The API reference documentation is thus an important development tool that programmers spend much of their time using. Ways to achieve reading support for API reference documentation, particularly from a use-perspective, is an important software-engineering issue, as shown by the rapid growth of JDK. In our view, the API reference documentation is a tool that should promote the use of its components and not simply describe them (the best use, the most common use, the use which fits different application profiles, and so on).

In the literature on software engineering and programming tools it is not uncommon that API reference documentation is omitted, see for instance [26, 40, 11]. Unfortunately, textbooks on software engineering and programming tools do not always acknowledge the importance of API reference documentation in programming. If documentation is discussed, it is often viewed as something produced in the project rather than used in the programming task (see [12]). In the book Software Engineering with Java, Schachs only advice is that documentation should be online [25]. This omission of a more in depth discussion on API reference documentation is particularly unfortunate since the API reference documentation, in our experience, is an essential tool in Java programming. However, there are exceptions such as the book "Software Engineering A Programming Approach" [29] in which the need of a cross-reference listing is mentioned. An underlying reason for overlooking API reference documentation may be the fact that programming traditionally involved a limited set of programming-language constructs. Perhaps programming languages can ultimately become sophisticated yet simple tools. In the meantime, however, programming languages are turning into large collections of components that humans have to handle. Java is an example of this development. In its short history the class libraries have grown and changed rapidly and in all directions. The API reference documentation will have an impact on the resulting software since it is a major source of information that programmers use continuously as part of the programming task. The API reference documentation will affect the way programmers conceive components and use them. Much time is spent learning and checking syntax which affects the cost, quality, and time needed in software projects.

From the perspective of human-machine interaction, our research area is computer-reading environments, particularly for work-related texts. Computers hold unique features as a reading environments, even though the screens are still limited in size and resolution compared to print. Hypertext is one feature of computer-reading environments which the literature on electronic texts seems mainly focused on [1, 14, 28]. However, computer-reading environments do not end with hypertext. For instance, the computer provides additional typographical features not possible in hardcopy. The use of color is, for instance, economically feasible to a higher degree in computers than in print and it is possible to animate texts [36]. Color coding is often used in editors to make reading easier. Computer-reading environments can also take advantage of typographical change. The Web is full of examples of dynamic typography. Roll-over effects on links and collapsible lists are commonplace and they enhance the reading environment. We find dynamic dimensions of typography intriguing because they go beyond printed text and can further enhance the computer-reading environment. We want to look beyond hypertext in the search for reading support (for an in-depth discussion on dynamic typography see Section 2.5).

1.3 Research Focus: DJavadoc

We focus on computer-reading environments for Java API reference documentation as an example of work-related texts. Specifically we consider the Java API reference documentation an example of a generated, homogenous, and structured information repository that is continuously read as part of the work-task. Our goal is to find new ways of presenting and organizing the API 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 APIs, knowledge about the distinctions among components with regard to their possible and recommended use, and so on. We are particularly interested in support that can be automated in some way, not because well-formulated tutorials can be replaced, but because automated reading support scale better for rapidly evolving information sources. Also, automatically generated documentation will not deviate from the source code.

More specifically, we have examined ways to enable different views of the same information by taking advantage of an explicit underlying information structure. The official Javadoc output (see Section 2.1.5) delivers class documents which contains an underlying, implicit *information model*. The information model is illustrated by the static-typography of the documents in which, for instance, bold style is used to emphesize method names. In a computer-reading environment, excessive information can be made less visible by changing the typography. For instance, we can turn the color of uninteresting texts into something not quite distinguishable from the background and thereby gray-out parts of the document. Similarly, we can remove excessive information from the reading surface by revoking the rendering of uninteresting text parts. As a result the visibility of the remaining information increases.

We have created a new version of Javadoc, named DJavadoc, that enables control over the visibility of information types in the API reference documentation. The official Java API reference documentation is generated by the Javadoc program designed by the *Javadoc Team* at *Sun Microsystems* [57]. In the DJavadoc project we extend Javadoc to augment its output with dynamic typographical functionality using *dynamic HTML (DHTML)* (see Section 2.2.2). DJavadoc is available on the Internet for public use on `www.ida.liu.se/~eribe/djavadoc` [44]. We are in the process of acquiring evaluation opportunities in industry to achieve real-world testing of DJavadoc. So far we have achieved some preliminary results.

Using DJavadoc and its following versions as a research vehicle, we hope to discover requirements that should be put on API reference documentation. DJavadoc is a product, an alternative API reference documentation that provides other types of support than Javadoc. However, it is also a tool for studying the requirements that should be put on API reference documentation. For instance, evaluation of DJavadoc will support the implemented requirements.

(Also, DJavadoc may function as a door opener providing access to the real world where professional Java developers work.)

1.4 Research Method

Applied task-driven technology-focused research will lead to greater understanding of both the technology and the domain. A research approach of applying technology to support a task in a domain will deliver a product. However, the process of development and evaluation of the product will result in more general knowledge, though the knowledge may be harder to generalize because of the applied, product-based approach. Also, the knowledge is probably less valid because it was formulated and tested in a more specific setting. Still, applied research represents a type of research that delivers general knowledge. We find applied computer science intriguing, in particular since the development of new technology is rapid in the area.

Our goal regarding research methodology is iterative application building based on user studies performed on prior iterations. Our research method is to first develop real applications that can be put to concrete use. We then perform user studies by evaluating the systems. In the first iterations we have based our design choices on our background as Java programmers. As the work progressed, we gained insights from development while designing prototypes that were evaluated informally. In the first iteration of the DJavadoc project we have had three generations: two prototypes and one final version. User studies are then performed on the final version in the iteration of which the result will be incorporated in future iterations. Our approach can so far be viewed as a form of *in situ development*.

The combination of qualitative and quantitative evaluation is preferred in our project. We chose to interest ourselves in qualitative methods because our research goal is to improve programming by humans though the API reference documentation. However, we are not advocates of qualitative studies per se but rather research pluralism in which qualitative and quantitative studies complement each other. Quantitative studies have their place in our research, for instance by studying how readers use DJavadoc.

1.5 Contributions

This work contributes foremost to the understanding of the online API reference documentation and the requirements that should be put on the API reference documentation as a programming tool. We also present the reference-documentation architecture used in the DJavadoc project. Furthermore, we analyze and discuss the continued development of the current Javadoc API reference documentation to achieve more use-oriented designs. How documentation can be used to analyze the Java language and ultimately object orientation is also discussed. Moreover, the thesis sheds some light on the computer-reading environment and its qualities beyond hypertext, particularly though the discussion of the concept of *dynamic typography*. The project also has additional practical contributions in the DJavadoc API reference documentation, which is a practically usable programming tool available on the Internet at `www.ida.liu.se/~eribe/djavadoc`.

1.6 Thesis Outline

In chapter 2 we provide a background discussion of technologies and concepts used in the thesis. The descriptions are given in the form of basic explanations and reflections.

In chapter 3 we discuss the methods used to design and to evaluate DJavadoc. The production of concrete systems as a means to perform research is also discussed and, furthermore, how evaluation should be performed and what types of knowledge the research method may deliver.

In chapter 4 we present DJavadoc in detail. DJavadoc is presented in contrast to the *Standard Doclet*, the specification of the official Java API reference documentation. The dynamic typography used in DJavadoc is described and illustrated by a series of screen-shots.

In chapter 5 we present preliminary studies that have been performed and report these results as experience. In addition, we outline the continuation of the DJavadoc-project studies.

In chapter 6 documentation or systems related to DJavadoc are presented. In our search for related systems we have taken the view point of programming-language environments and their reference material. We view such systems from a general acquisition-of-knowledge perspective and discuss their relations to the DJavadoc project.

In chapter 7 we provide a discussion based on our experience from the DJavadoc project. The DJavadoc project is connected to many aspects of the computer-reading environment and software engineering. We discuss API reference documentation, Javadoc and our future research directions.

In chapter 8 we summarize the thesis and provide conclusions from the DJavadoc project.

Chapter 2

Background

In this chapter we discuss different technologies and concepts related to the DJavadoc project. Sections 2.1 and 2.2 concern technologies whereas the remaining focuses on concepts and design issues. For a detailed description of DJavadoc see chapter 4.

2.1 Java

2.1.1 The Java Programming Language

The Java programming language was developed by Sun Microsystems [32, 52, 55]. Java has only been publicly available since May 1995 [74]. Originally, Java received much attention as a Web animation language made famous by its *Applet* concept, a small program executable in Web browsers. Today, Java is not primarily a Web-language, both because the language has developed in other directions and because other alternative Web animation languages have appeared, see Section 2.2.2.

Java is an object-oriented programming language, much like *C++* in syntax. In principle, all programming elements are considered to be *classes*, which contain fields, constructors, methods and inner classes (even though basic primitives, like integer and boolean, are still used). Classes are also grouped into packages. All classes extend the `java.lang.Object` class, which is the superclass of all classes. Class names are constructed using a package name and a class name, in order to achieve unique names. For the `Object` class, the package name is `java.lang` and the short name is `Object`. However, the package name is seldom spelled out when the class is referenced in the source code. Instead the package is *imported* so that the class can be referenced by its short name.

Java is an interpreted language which is compiled to an abstract, platform independent Java machine-code. The *Java virtual machine (JVM)* interprets the so called *byte-code* and maps this byte-code to machine code. By interpreting the byte-code, the JVM makes Java programs platform independent even though the JVMs are dependent on the platform. The byte-code step also makes Java more secure, in the sense that the program can be analyzed for prohibited behavior. The interpretation step, however, leads to slower performance.

The official Java core class library, the Java development kit (JDK), is tightly coupled to the language. Java is synonymous with JDK, even though the language and the standard class library are two different things. For instance, the `Object` class that all classes extend is part of JDK. In our opinion, Java is a language based on reuse which makes JDK and the JDK API reference documentation vital parts of the Java environment. Application builders deal with issues such as deciding which class to use and the need to understand how these classes are combined into applications.

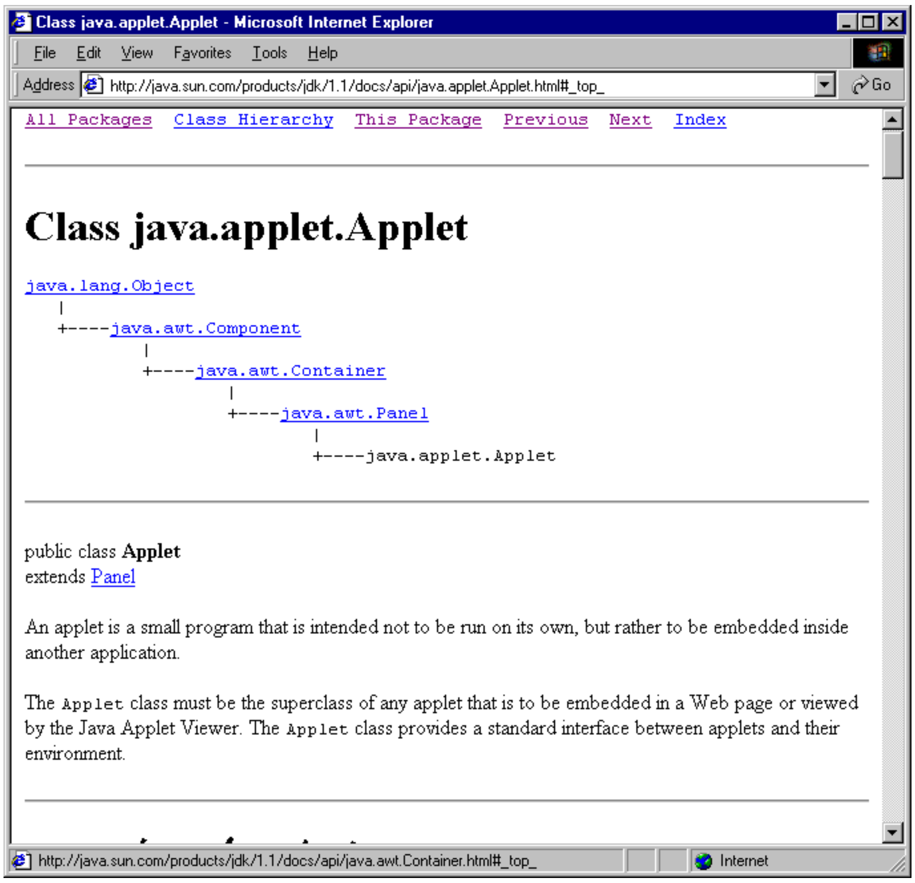
Java has established itself as one of the most used languages in spite of performance problems. The interpreted nature of the language leads to slower performance. Even though the performance on high-end PCs and workstations is acceptable, speed is definitively Java's Achille's heel. Much work is being put into the performance issue [53] but results have been somewhat discouraging until now. In the future Java performance issues will probably be resolved (the next Java release focuses on this issue). Also, judging by the recent PC evolution, the problem might just disappear on its own.

2.1.2 Java, JDK, SDK and so on

In this thesis we use the name Java to denote the Java programming language and the name JDK to denote the core class libraries. Java names are currently undergoing some change. For the next release (some would call JDK 1.3) the name will be Java 2 SDK 1.3 where SDK stands for standard development kit. JDK 1.2 was released before the name change to Java 2 SDK 1.2 was made and, of course, there has been much confusion. However, we use the names Java and JDK in this thesis because these are the currently the best known names.

2.1.3 Javadoc

Javadoc is a Java program that generates documentation from *Java source code* [9, 19]. The program runs the source though the first steps in the *Java compiler*, which delivers information about inheritance, class and method names, parameters, return values, exceptions and so on. Programmers write *tagged* comments in the source code, which are also extracted by the Javadoc program, see Figure 2.3. The Javadoc Team has defined a set of tags that will be recognized. Javadoc can generate API reference documentation for any combination of Java classes. Figure 2.1 show Javadoc class documents from JDK1.1


Class java.applet.Applet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://java.sun.com/products/jdk/1.1/docs/api/java.applet.Applet.html#_top_
Go
All Packages Class Hierarchy This Package Previous Next Index

Class java.applet.Applet

```
java.lang.Object
   |
   +----java.awt.Component
           |
           +----java.awt.Container
                   |
                   +----java.awt.Panel
                           |
                           +----java.applet.Applet
```

public class **Applet**
extends Panel

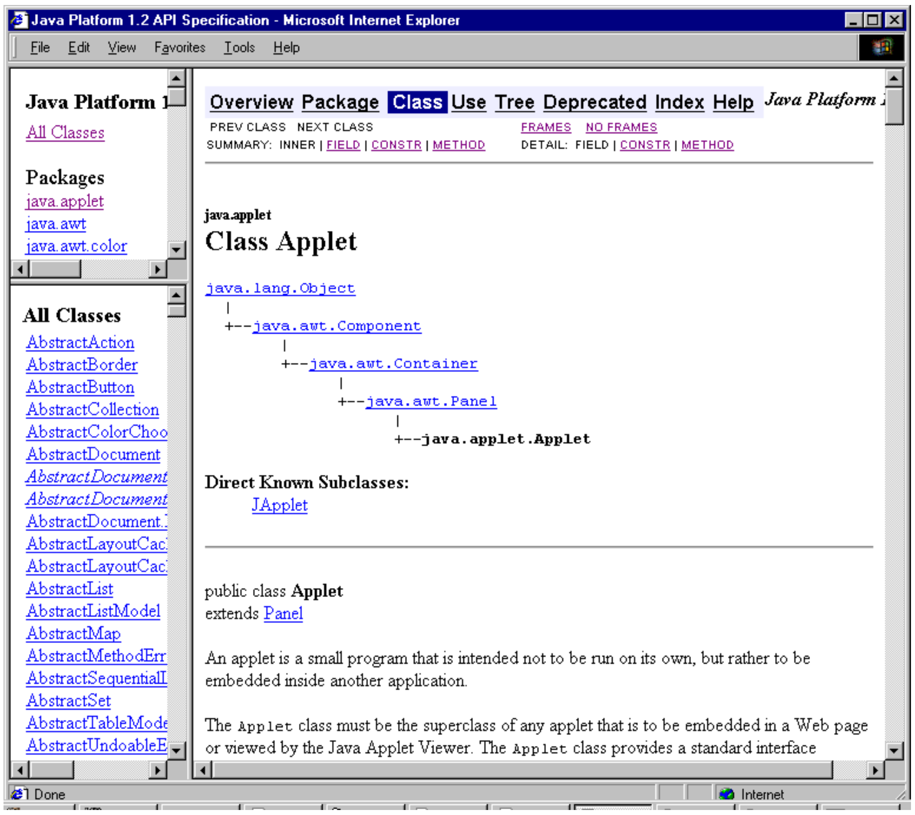
An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.

The `Applet` class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The `Applet` class provides a standard interface between applets and their environment.

http://java.sun.com/products/jdk/1.1/docs/api/java.awt.Container.html#_top_
Internet

Figure 2.1: Screen shot of the Javadoc 1.1 style class document.

and Figure 2.2 show Javadoc class documents from JDK 1.2. On the Java home page [52] Javadoc-generated documentation can be found for each of the different versions of JDK (see Section 2.1.1).



Java Platform 1.2 API Specification - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Java Platform 1

All Classes

Packages

java.applet

java.awt

java.awt.color

All Classes

AbstractAction

AbstractBorder

AbstractButton

AbstractCollection

AbstractColorChoo

AbstractDocument

AbstractDocument

AbstractDocument

AbstractDocument.

AbstractLayoutCac

AbstractLayoutCac

AbstractList

AbstractListModel

AbstractMap

AbstractMethodErr

AbstractSequentialL

AbstractSet

AbstractTableMode

AbstractUndoableE

Overview Package **Class** Use Tree Deprecated Index Help *Java Platform 1*

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.applet

Class Applet

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
```

Direct Known Subclasses:

JApplet

public class **Applet**

extends Panel

An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.

The `Applet` class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The `Applet` class provides a standard interface

Done Internet

Figure 2.2: Screen shot of the Javadoc 1.2 style class document.

```
/**
 * 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 2.3: An example of how comments are written into the source code using @-tags (for instance, @param) to structure the tags.

The purpose of Javadoc and the API reference documentation it delivers is to support programming by providing an interface to the API source code. Programmers use API reference documentation to learn and use API source code. They could read the source code directly but it is time-consuming and may require complex analysis. The aim of the API reference documentation is to portray the functionality of the API, correctly and efficiently. The API reference documentation presents information deemed particularly important to programming.

Principally all Java code (provided by Sun Microsystems or third-party providers) is 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 API reference documentation may be the only available documentation.

In our opinion, the Javadoc-generated API reference documentation is well-known in the Java community and easily read by knowledgeable Java programmers. Achieving a form of automatic documentation is the purpose of Javadoc. Automated visualization of functionality naturally has many practical advantages, such as reduced cost and a stronger coupling with the source code (the tagged descriptions may not be updated but are at least stored in the same files as the source). Of course, from a learning perspective an automatic listing of functionality cannot compete with well-written tutorials. However, the Javadoc-generated API reference documentation becomes an important, standardized learning environment for Java programmers.

We believe that Javadoc has played an important role in the rise of the Java language during the 1990s. Because it provided readily available JDK documentation, the rapid development of the Java language (see Section 2.1.1) has benefited from Javadoc. The API reference documentation has been available on Internet in principle since Java was introduced. Unlike tutorials the Javadoc-generated API reference documentation is easily updated and therefore never out of date, which has been a problem for books due to the speed with which JDK changes.

Javadoc is actually not one program but a complex structure of systems, illustrated in Figure 2.4, which are described in the following sections. Javadoc is the control program of a complex structure of systems used to generate API reference documentation from source code. In essence, Javadoc is a framework for Javadoc applications. The role of the Javadoc program is to build an information structure and then to hand over the process of generating output to a *Doclet*, see Section 2.1.5.

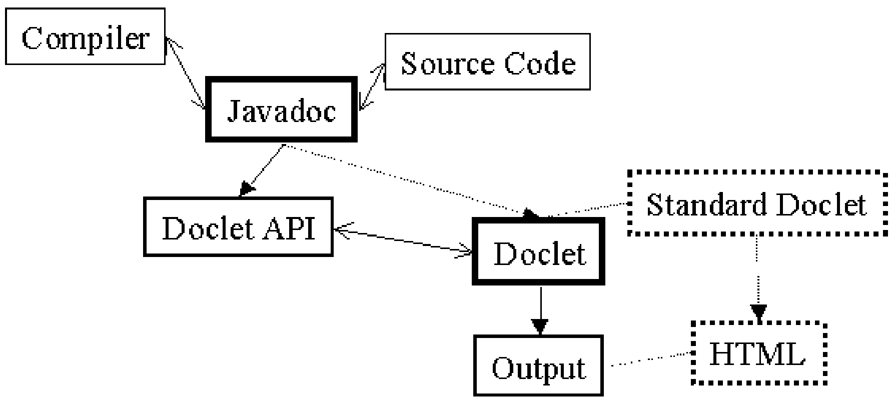


Figure 2.4: The structure of systems that are used in a Javadoc session to deliver API reference documentation.

2.1.4 Doclet Application Programming Interface

The JDK 1.2 release contains a *Doclet API* (see Section 2.1.1). Originally Javadoc was a Java program that could not be altered or changed in any way. However, with the release of the Doclet API, Java exposes the documentation classes used by Javadoc. The Doclet API represents a development platform for designing Javadoc documentation from Java source code. Printing is not contained in the Doclet API, only the means to access information from the compiler. In Figure 2.4 we illustrate how Javadoc generates the information structure accessible through the Doclet API. The Doclet API can be seen as an abstract data type for Javadoc(data storage and methods to access the data).

2.1.5 Standard Doclet

The Javadoc program uses a Doclet to define the Java API reference documentation content and typography. By providing a Doclet class as an argument to the Javadoc program, the printing of the API reference documentation can be changed. For instance, a new Doclet could print only the names of classes in an index. In Figure 2.4 the Doclet is given control by the Javadoc program and then uses the Doclet API to access the information structure. The Doclet is intended to deliver an output of some sort [9] but could 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).

The Standard Doclet is the Doclet used to generate the official hypertext-based Javadoc API reference documentation developed by the Javadoc Team at Sun Microsystems. In combination with the Doclet API, the Standard Doclet is delivered as the default Doclet class used by Javadoc to control printing. The produced API 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 (see Section 2.2.1) are used to create relations (i.e., relations that directly lead to related documents) among class documents. Return-value classes, parameter classes, and inheritance classes are linked from the class document. Section 4.2 describes in detail the output of the Standard Doclet. In Figure 2.4 we illustrate how the Standard Doclet is used to deliver the official Java API reference documentation in HTML.

DJavadoc is a Doclet implementation that extends the Standard Doclet (see Section 2.1.5). As an extension, DJavadoc introduces dynamic features in the Standard Doclet by adding DHTML (see Section 2.2.2) to the existing hypertext API reference documentation. It is important that the Standard Doclet is easily extendable, at least if different versions of the Javadoc generated API reference documentation are desirable. The development of new Doclets benefits greatly from the Standard Doclet in the sense that the basic structure have already been developed. Having a structure to extend increases the ease with which new versions are created. Also, the Standard Doclet sets a typographical standard that new versions can conform to.

2.2 Web Technology

2.2.1 HTML

Hypertext markup language (HTML) is a human readable and machine readable format for defining graphical typography of media (e.g., text and images) and connections between media documents [51, 50]. HTML is the most commonly used hypermedia language today. As a simple Web language it has gained its popularity. HTML is the most used and most well-known instance of the *standardlized generalized markup language (SGML)* [73]. XML is another SGML derivative which is currently receiving much interest [82, 81]. Unlike HTML but like SGML, XML is a meta-language used for defining mark-up languages. A style sheet language is required to define the typography of XML or to convert to another language. XML is currently viewed as a future alternative to HTML but that will still take some time.

The bulk of HTML is related to typography, even though HTML is considered a hypertext or hypermedia language. Only a small fraction of HTML represents hyperlinks. Also, up to now, the development of HTML has not concerned hypermedia but rather the construction of more advanced and controlled typographical techniques.

2.2.2 Dynamic HTML (DHTML)

HTML (see Section 2.2.1) is a static language that in principle contains no means of expressing dynamic behavior of the typography or of document relations. The popular Web-page language, HTML, contains elements for expressing headings, tables, forms, hyperlinks and so on. The static typography of these elements is specified by *browsers*, such as *Microsoft Interent Explorer* and *Netscape Navigator* but can also be defined using style-sheet languages (see [43]). However, the ability to define dynamic changes in the typography of the HTML elements is very limited. One of the few dynamic features in typography in HTML is the color difference between visited and unvisited hyperlinks. Another example is the tool-tip or balloon help that pops up over images and other HTML elements.

Dynamic HTML (DHTML) is a term grouping all (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 it is possible. Instead DHTML is used to 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 reader is about to activate. The collapsible list is another frequent example.

The core DHTML technology is the *scripting language* used to introduce algorithmic behavior in the HTML-page.*JavaScript* was the first and is probably still the most used scripting language [4, 60]. The proposed international *ECMAScript* standard is currently an accepted standard of the major browsers *Netscape Navigator and Microsoft Internet Explorer* [2, 46, 47].

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)* [45] defined by the *world wide web consortium (W3C)* [41, 79] 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 in 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 markup language.

2.3 Multiple Views

A view of an information source distinguishes among parts of the information. Views are constructed by making parts of the information more or less visible to the reader. In principle every presentation of an information source represents a view of that source.

Multiple views of API reference documentation are required because readers have multiple purposes. Since the API reference documentation contains information for more than one purpose and on more than one level of detail, the multiple views are needed to create descriptions that are in line with the needs of an individual reader. Readers have a particular purpose at hand when reading the API reference documentation and therefore want the information matching this purpose to be presented. As purposes change, the presented information should change.

Multiple views are also required because readers are use-oriented. Readers have particular tasks in mind as they read the API reference documentation. They are looking for specific information or specific types of information. Their interest is to acquire knowledge to perform tasks and not to be amused or entertained. Furthermore, because their interests change, one design of the API reference documentation cannot be fully adapted to readers' needs.

In addition, multiple views are required because individual readers want the same information displayed in different ways. Different programmers like different types of presentation of the same information. In our experience of working with computer-science students we conclude that readers want the source code described in different forms.

2.4 User Control

Flexible and configurable API reference documentation requires choices, performed either by the reader or for the reader. What pieces of information are uninteresting must be decided by someone. The reader is a natural candidate for making the choices and manipulating the settings of the computer-reading environment. However, information models that describe the alternatives may be complex, in which case the reader might find it difficult and (or) time consuming to control the reading environment.

In the intelligent-user-interface community much work is beeing put into the design of systems that can make choices on user's behalf [38]. An intelligent API reference documentation would create a representation of the reader's intentions and map them to actions in the computer-reading environment. Intelligent documentation could alternatively provide suggestions instead of making choices. Furthermore, intelligent documentation could work with the information model to present a less complex interpreted model.

For the Javadoc API reference documentation, an intelligent interface is probably not required and definitely not the first thing to introduce. The information is homogenous and the information model is in our experience well known, at least by knowledgeable readers. Introducing intelligent support may be a second step, but to introduce user-controlled support is the first. We also strongly believe that readers of API reference documentation want to make their own choices, especially knowledgeable readers. Professionals want to control their environment, to decide when and how things happen.

Furthermore, programmers are well suited for high-level tasks in computer-reading environments. Programmers are professionals both in the domain and the technology. As programmers they are likely to have much practical and theoretical knowledge about computer science. They are also likely to be well

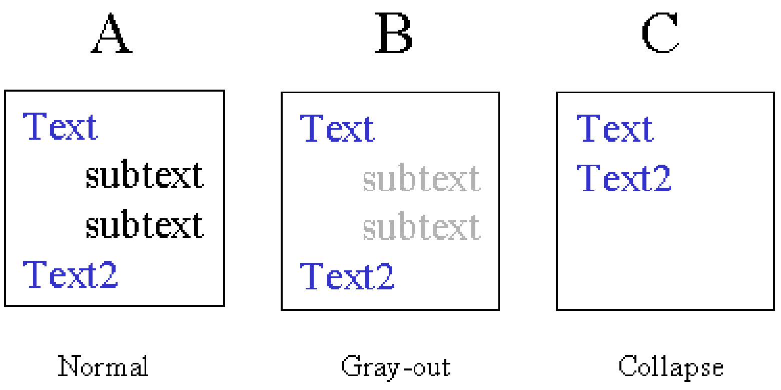


Figure 2.5: An example of dynamic typography in which A, B, and C represent different typographical states of the same information. A is the fully expanded view. In B the gray-out method has been applied to decrease the visibility of the sub-levels. In C the sub-levels have been collapsed.

acquainted with Web technology. In general, there is reason to believe that programmers are sufficiently competent to control the computer-reading environment, perhaps more so than other professionals.

2.5 Dynamic Typography

2.5.1 Defining Dynamic Typography

Dynamic typography is the description of the change in appearance of information in a reading environment, illustrated in Figure 2.5. Typography is the user interface of the document paradigm. Static typography involves the setting of a unchangeable appearance using terms such as font, font-size, margin, line-space, white-space. Movement of information entities over the reading surface can also be characterized as static or directed typography (typography does not generally include a language to define movement but it is touched upon in [36]). It is characteristic for static typography is that the setting is predefined and controlled by the typographer. Once the typography is set, it is, in essence, etched into the reading surface. A dynamic dimension of typography would extend this realm with change due to interaction, time, as a relation among different typographical entities, and so on. Defining the behavior of the typography in a changeable media such as the computer-reading environment becomes relevant when neither the reading surface nor the content is clearly defined. An example of dynamic typography is to define a roll-over effect that will highlight a piece of information when the pointer is located over it by changing the color.

Specifying dynamic typography may be a matter of defining what types of change we agree to and what methods of change we prefer. Also, new typographical concepts may be required. An example of dynamic typography is the definition of a row-length range, specifying the acceptable change in row length. Ranges, which define reasonable typography for different forms of media, are known within typography. For instance, the line length should be 35-65 symbols [3]. Another example is to define collapsibility as a typographical concept, specifying that for all lists we agree on collapsibility and 'gray-out' as our change method. On a more general level we could define red as the general roll-over color and let the system apply that change in color whenever it encounters *rollable* items. For rollable images we would supply two image sources to switch between. The list of concepts, change specifications and change methods is probably quite extensive. However, we believe it is important to further investigate dynamic typography as a concept to better support reading in computer environments. Also, by investigating dynamic typography we may discover typographical knowledge that can be applied in Web applications.

2.5.2 Dynamic Typography versus Hypermedia

Features of the computer-reading environment extends beyond hypertext. Hypertext and hypermedia have received much attention as a result of the popularity of the Internet, even though the work on hypertext started long before [39]. The ability to download any referenced material in a matter of seconds as opposed to days or weeks through ordinary library sources is, of course, a major breakthrough in reading. However, the computer-reading environment holds more than hypertext. Alternative approaches to computer-based reading are interesting simply because of the strong focus on hypertext.

The development of Web technology also speaks for a shift in interest from hypertext to dynamic typography. The development of mainstream browsers, such as Internet Explorer, and HTML (i.e., the de-facto standard hypermedia language) stands in contrast to the focus on hypertext and hypermedia. While much has been written on the topic of hypertext and the need for more sophisticated hyperlink-functionality (see for instance [31, 20]), the development of mainstream Web technology has surrounded typography and dynamic presentation. The bulk of HTML concerns typography (only a few attributes in HTML 4.0 are hyperlinks) [50] and the development of HTML has also focused on the creation more complex typography components such as tables. Recently DHTML has appeared as a term grouping dynamic presentation techniques on the Web (see Section 2.2.2). It is the cinematic look and feel of Web pages that drives the DHTML development forward, that is typography. In the future the development of more advanced hypertext may well appear but there is still a mismatch between the interest in hypertext and development of mainstream Web technology.

Furthermore, dynamic typography is an important area of investigation because it will have profound implications on static-typography. The long tradition of typography dating back to Gutenberg and the fourteenth century is concerned with printed text, that is a static surface. Even today, typography is mainly concerned with the static medium be it on paper or screen (see for instance [3, 36, 18]). In [36], the dynamic features of typography is touched upon as an extension into the dimension of time in which type in motion is discussed (speed, direction, duration and variation in size as typographical characteristics of type in motion). The time-dimension is only discussed in relation to a cinematic, choreographed, and controlled movement and not in relation to interaction or change in the reading surface. Typography is still discussed as a predefined and, in essence, static art. At the same time the Web is full of dynamic-typography examples. Roll-over effects on links and collapsible lists are commonplace. If a dynamic dimension is introduced into typography it becomes less obvious how to express the type-setting of information. The typography language may have to change. For instance, the font-size of a text can still be defined but additionally we may have to describe its response to changes in the environment. This lack of dynamic dimensions of typography is also an argument for looking at dynamic typography as a means to support the reading task.

2.5.3 Dynamic Typography in DJavadoc

In DJavadoc we introduce a simple *collapse and expand* functionality, much like collapsible lists but for entire texts. We chose to regard this functionality as dynamic typography, even though it is in essence the same as *stretch-text* (which is a form of hypertext in which information nodes are expanded inside the document). The reason for not using the term 'stretch-text' is to underline the connection to typography as opposed to information composition. Collapsible text is only one example of dynamic typography that creates new views on the same material. Another example is the graying out of information by changing the color to something not quite distinguishable from the background. Graying out to enhance visibility was used in the conceptual-filtering generation of the DJavadoc project (see Section 4.8.3). Compared to collapsible text, graying out gives a graphical description of the relation between the grayed information and the remaining information but does not free screen space.

By introducing dynamic typography, it becomes possible to collapse and expand information types as well as specific pieces of information from the API reference documentation (see Section 2.5). Such collapse and expand functionality enable the reader to make visible interesting pieces of information by removing less interesting pieces.

It is important that readers are aware of the collapsed material [8]. The non-rendered parts of the API reference documentation may not be unknown to the reader. The system should take responsibility for presenting the invisible parts.

API reference documentation, which is homogenous in nature, may have a strict information model that is known by the reader. Compared to heterogeneous information, the invisibility will be less dangerous. Knowing what parts of the model are collapsed may suffice as visual cues for the knowledgeable reader.

2.6 Other Design consideration

We believe that work-related texts, such as API reference documentation, should be designed for professionals. Novices need tutorials to learn the language, culture, and basic facts of the subject area. The need for pedagogical, well-formulated explanations decreases as the novice progresses. Reading between the lines and making sense of incomplete information is something the professional handles well (see [15]). API reference documentation, unlike tutorials, should be viewed as work-related texts and not educational material (although learning does continue in the work-situation). As such, the documentation should be focused and use-oriented. Professionals want to find the core of the information and then move on. They also know what amount of information they want and when. Therefore user controlled adaptation of information becomes an attractive means to improve work-related texts.

For evaluation purposes, changes were not introduced to the *static typography* and the *information organization* of the Standard Doclet Section 2.1.5. The Standard Doclet defines the standard look for Javadoc generated API reference documentation and serves as the platform on which DJavadoc is built. Changes to the static typography or the information organization could have enhanced our possibility to leverage dynamic typography. It is also possible that such changes would enhance the reading environment regardless of the dynamic typography. However, testing the effects of dynamic typography in DJavadoc would have become more difficult and therefore we restrained from any such changes.

Chapter 3

Method

In this chapter we discuss methods of development and evaluation that are related to the DJavadoc project. We describe research methods used to discover requirements and architectures for API reference documentation in our work as well as other's. Javadoc is discussed in Section 2.1.3. The actual development and the early, informal studies are discussed in chapters 4 and 5.

3.1 Different Research Methods

Requirements on API reference documentation can be discovered by performing user studies. There are a series of methods to study users and determine their requirements on, for instance, API reference documentation. Common to all these methods is that the user is the primary source of input and that the observed difficulties the user has should guide the development. To some degree research based on this approach assumes that users have the ability to express the needed changes on their own or that it can be observed. A potential problem with this approach is that the users current work situation might be conserved rather than changed (which is both good and bad).

An alternative approach is to test concepts by implementing them in a series of situations. An example is the action-oriented approach to instructional material Carroll introduced with the minimalist concept [16]. Minimalist instructional material should inspire action, support and encourage exploration, be brief, provide error information, and so on. The minimalist design has been applied in a series of instructional texts and tested [17]. There has, of course, also been a continuing refinement of the concept along the way. We are inspired by Carroll's approach.

By designing working systems, new architectures and idees are presented. The value of the architectures and ideas are arguments for and, to some degree, determined by the widespread use of the system. The WEB system is one example (this is not the World Wide Web). The system was the first embodiment of Knuth's literate-programming vision, in which programs were written as novels containing both comments and code [37, 7], WEB is also discussed in Section 6.5. Building a series of systems provides the researcher with design experience and empirical evidence of the effect of design choices.

3.2 Our Research Process

3.2.1 Iterative System Design as Research Method

We apply iterative system building as a research method. Our research takes its stance in a task, that is the task of acquiring knowledge from API reference documentation. This task is a subtask of the programming task and has subtasks in itself. We chose to conduct research by developing systems (somewhere in between prototypes and well-supported systems) to support this task and to evaluate the system. The goal of introducing new support is to both test the requirements implemented and to elicit new requirements for the next iteration. To conduct an iterative development in which systems are created in a design-evaluate-redesign loop is currently part of many development methods, see for instance the Unified Development Process by Rational [35, 71].

Currently we have performed in situ development. Based on our own experience of using API reference documentation in small to mid-sized one-man, non-commercial projects we have developed some requirements and implemented them. The next step is to incorporate other developers in the loop.

We decided to address only parts of the functionality in the current version of the API reference documentation. By completely redesigning the API reference documentation we might have developed a better version. However, we chose to work with the existing system by enhancing it with new abilities. One major reason is the ability to create a new, fully functional documentation and still focus only on certain aspects.

We want to test our systems on professional practitioners. The type of informants we are focusing on is the professional Java developer who uses API reference documentation on a day-to-day basis. Professionals are considered to be task-oriented in the sense that they strive to optimize their work and reflect upon deficiencies in current support. Also, they may devise ways of overcoming deficiencies in the current support that are of interest to our research.

3.2.2 The Advantages of our Approach

By building systems to support a task we achieve a concrete, practical research approach. Tasks are closer to our everyday life and therefore perhaps easier to relate to than are, for instance, technologies. Supporting tasks with systems potentially leads to relevant research of a general nature that can be tested in the task environment. Concrete applications may also inspire genuine evaluation interest from task performers because in their eyes the system may be valuable. Furthermore, the introduction of new technology may generate new lines of thought in users. For users it may be easier to react to a system than to come up with ideas of their own.

Another advantage is the system-building perspective that may lead to the discovery of another range of knowledge, compared to, for instance, user studies. In itself, development is a creative process that requires the concrete application of technology to the ideas of a researcher. The process of converting the ideas into something that works, in our experience, brings new ideas to the surface.

Systems open doors. They provide a bridge between the research laboratory and the real world. As researchers we achieve a starting point for discussions with practitioners. Getting practitioners to pay attention to the research we are performing may not be easy simply because they do not have the time. However, if we deliver something they can benefit from directly, we might be more successful. The system-focused approach represents one way to generate reactions from a body of users that may have general consequences. Once the connection is established, it might be possible not only to perform evaluation of the system itself but also more general studies.

It is important to note that the final goal is not to produce systems in themselves but the knowledge that the development and use of systems provide. Development is conducted as a way of testing and experimenting with ideas and to further extend them. The development and use of systems then provide insights on a more abstract level, for instance, requirements and architectures for a type of system. It is these more abstract findings that we consider research contributions.

3.2.3 The Drawbacks of our Approach

One major disadvantage of a task-driven approach is the great distance between task-based evaluation and general evaluation. Results and contributions may not result in major change in a greater context but will still be real in the task-context. It is not possible to arrive at general conclusions if the research is performed in the task-environment, only to provide evidence of general principles from that environment. The system-building approach leads to an indirect testing of design principles. Conclusions drawn from an evaluation are perhaps not representative for the design in general. Issues such as system performance and smooth interaction will also have impact on the result (which may be positive as well). Also, change in the performance of a task or the task itself may occur simply on the basis of the introduction of new and appealing technology.

Real-world testing may bring a set of restrictions to the research. From our experience in acquiring real-world contacts it is also obvious that there has to be some form of mutual exchange involved. Also, the effort required from the industrial partner must be minimized. Furthermore, professionals may be hard to find, they may have little time over for interaction with a researcher. Since Java is only 4 years old (May 1999) it can be difficult to locate such professionals. It is less likely that corporations will have large divisions of experienced Java programmers, as is the case for older programming languages.

3.3 Evaluation

The DJavadoc project deals with human performance and therefore qualitative methods are close at hand. Qualitative methods, such as interviews, are natural in the study of humans and organizations. There is a continuing debate on the value of qualitative methods, addressed for instance in [10, 24]. We acknowledge this debate but do not consider it further in this thesis. In our view, both qualitative and quantitative methods are valuable but imperfect tools that both have roles in our research.

So far, we have performed unstructured, informal discussions with potential DJavadoc users that provide comments on the value of the system. In the process of acquiring industrial contacts we have demonstrated and discussed the system with 15 members of software-development projects. Furthermore, 7 technical writers have viewed the system. These discussions have been informal and some performed by telephone or by email.

3.4 End Product

The type of contribution that can be reached using our research method is knowledge about the domain, the community, and technologies in the form of research experiences. In principle, we may only provide evidence of the usefulness of different technological solutions in a particular domain. Our experience in working in this research environment, both partaking in a creative task-oriented activity and by performing evaluation, may only point to possible knowledge. The findings may nevertheless be relevant. Our work will be of a more descriptive character than of proven hypotheses. We may point to and argue for different interpretations based on our evaluation and our own experience. Developing systems as a research method leads to indirect testing of theories (particularly non-formal development).

In comparison to an older system, we may show that new technology adds substantial value. However, we cannot show that the same value cannot be achieved by extending the technology already in use. Also, it is not possible to show the inapplicability of technology since it is dependent on our ability to write code. However, we may have developed arguments for why the applied technology will not serve the task well.

Chapter 4

DJavadoc

In this chapter we describe DJavadoc in detail. DJavadoc is publically aviable on the Interente via `www.ida.liu.se/~eribe/djavadoc` [44]. Since DJavadoc is an extension to the Standard Doclet we describe DJavadoc by first describing the Standard Doclet and then explaining the extensions. We also examine prototype generations that preceded the final DJavadoc version. Basic understanding of the technology is assumed and we do not go into detail on, for instance, Java, Javadoc, and the Standard Doclet. For explanations of such terms see chapter 2. The difference between hypertext and dynamic typography can be found in Section 2.5.

4.1 DJavadoc Overview

4.1.1 The Official Java API reference documentation

The official Java API reference documentation is basically a component catalogue that describes a set of classes. The 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. For a more detailed description of the documentation generated by the Standard Doclet, see Section 4.2.

The API 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 form 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 API reference documentation create a view of the source code. The design of the API 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.

4.1.2 What DJavadoc Adds

Dynamic Javadoc (DJavadoc) provides programmers with means to further specialize their API reference documentation during work with the aim of constructing more use-oriented designs. The Standard Doclet provides a view of Java source code. In DJavadoc we augment the Standard Doclet with the ability to dynamically remove more information, thus further specializing the view of the source code. This extended functionality is applied to reduce time-consuming, repetitive manual searching for well-defined information types. DJavadoc does not assume that one information type is more vital than another but rather that they are relevant in different works situations.

In DJavadoc the reader controls the visibility of information types. We can view the API reference documentation generated from the Standard Doclet as documentation designed for several information needs, for instance the need of understanding a class or the need to look up names for coding purposes. In DJavadoc we provide control over the visibility of information types (on group level and individual level) in the computer-reading environment. The reader can collapse and expand information, thus increasing the degree of visibility of relevant information. By removing certain parts the reader moves other parts up into the visible space of the browser. Also, by removing surrounding texts the reader makes elements of greater importance more visible.

The information model used in DJavadoc is the explicit version of the implicit information model in the Standard Doclet. The Standard Doclet prints an implicit model in the API reference documentation based on the *Doclet API*. The static typography exposes the model and we use it to define the information model that conceptually groups pieces of information into information types. The model is fairly straightforward, making it easier to identify a plausible information model without exhaustive studies. However, more sophisticated models could, of course, be devised.

The organization, content or static typography of official Java API reference documentation is not addressed in the DJavadoc project. In one of the prototype generations, described in Section 4.8.3, we experimented with information filtering based on conceptual types of class members (e.g., basic methods, and event-related methods), but in the final DJavadoc version we focus on user-controlled views of information based on an underlying model. We refrained from addressing the organization, content, and static typography because we wanted to keep DJavadoc similar to the Standard Doclet. The assumption is that programmers will accept DJavadoc more readily simply because it is an extended version of the Standard Doclet and not a new one. However, the DJavadoc research project does not exclude these issues.

4.1.3 Arguments for the DJavadoc Extensions

DJavadoc supports experts' reading behavior. Experts want to search for relevant information when they read; they skim, browse, and skip ahead to find relevant parts [15]. By excluding information on the basis of type it is possible that experts may find an information source closer to their needs (particularly for homogenous sources). Searching for the relevant type of information can in some cases be reduced to manipulating the visibility of information types.

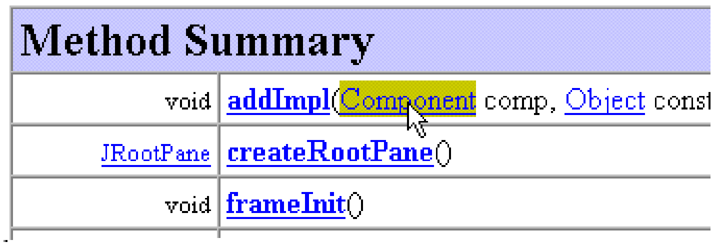
By reducing the manual search of information types reading may become less time consuming. Repetitive search for information types can be removed. By collapsing information other parts are pulled up into the visible space of the browser. The amount of scrolling needed to locate the relevant information is reduced. The amount of information in different views is also reduced, thus requiring less skimming. If the excess information is intertwined with the relevant information, collapsing will produce cleaner sets of information.

There is a possibility that programmers will work more efficiently if they can alter their API reference documentation to more focused, action-oriented views. In our experience, Java programmers spend much time reading the API reference documentation. Ways of reducing this time may have a direct effect on the cost of software development.

Reducing manual search for information is an important problem, particularly for the Java API reference documentation. Reading Java API reference documentation is performed by browsing a large set of documents rather than reading one document. This may be common for object-oriented programming languages or component-based programming languages in which applications are built by combining several components. The time scrolling down to the relevant section of each new document can become a large portion of the total reading time.

4.1.4 Interaction Principles of DJavadoc

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 from different perspectives. 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 concerns on interaction principles in the DJavadoc project are that efficient interaction should be achieved and that Web tradition should be maintained. Also, we have aimed to be consistent and simple. For instance, we 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).



Method Summary	
void	addImpl(Component comp, Object const
JRootPane	createRootPane()
void	frameInit()

Figure 4.1: 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.

Blue, underlined text can be clicked, either to follow a hyperlink or to perform a DJavadoc-specific action. Web tradition states that blue, underlined texts are 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. We 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 API reference documentation. The full class name is hidden in tool-tips that can be exposed by setting the pointer over class names. Also, classes can be saved to a bookmark list (see Section 4.3.5) by pressing the alt-key and clicking the class name in class documents. Removing classes or packages is achieved by the same alt-key interaction. The fact that the class is being saved or removed is illustrated by a short-time change in background (see Figure 4.1). The alt-key interaction is a simple and efficient way of achieving fast interaction.

4.1.5 Dynamic Typography in DJavadoc

For the reader of DJavadoc API reference documentation the distinction between dynamic typography and hypertext might seem academic and not really related to the issue of reading API reference documentation. From a software-engineering perspective the key issue is what DJavadoc delivers in terms of productivity and the effects on programmer's working environment. Whether the DJavadoc project is based on typography or hypertext is of little or no concern to the programmer. Such an attitude is understandable and well motivated. It may also be a reason for the confusion over the term 'hypertext'. In its original definition hypertext represents non-linear text [27]. Current literature in the area still focuses on the original definition (see for instance [13]). However, in practice, hypertext and hypermedia are often used to denote practically everything on the Web.

Distinguishing between hypermedia and other dimensions of computer-reading environment are part of the clear definition of the DJavadoc project. In our view, it is important to discuss why the DJavadoc extensions are not of a hypertext nature since hypertext, in practice, is not clearly defined. Other concepts are needed to fill out the area of computer-reading environments.

We regard the extensions made in DJavadoc as dynamic typography and not as hypertext. In our view hypertext is related to information access and information composition. Hypertext is defined as non-linear text [27], networks of text that can be read in principle in any sequence. However, the graphical presentation of text concerns typography [3, 21]. A change in the typography of information should therefore be attributed to a dynamic realm of typography and not hypertext. Collapsing and expanding are examples of the type of manipulations we regard as dynamic typography. Other examples include graying out, roll-over effects, and resized fonts.

4.1.6 Web Applications

DJavadoc is a Web application. We can view Web browsers as runtime environments. Web browsers, such as Microsoft Internet Explorer and Netscape Navigator, provide advanced support for HTML-based applications. In the DJavadoc project we make use of this highly specialized technology to deliver what is basically an information system.

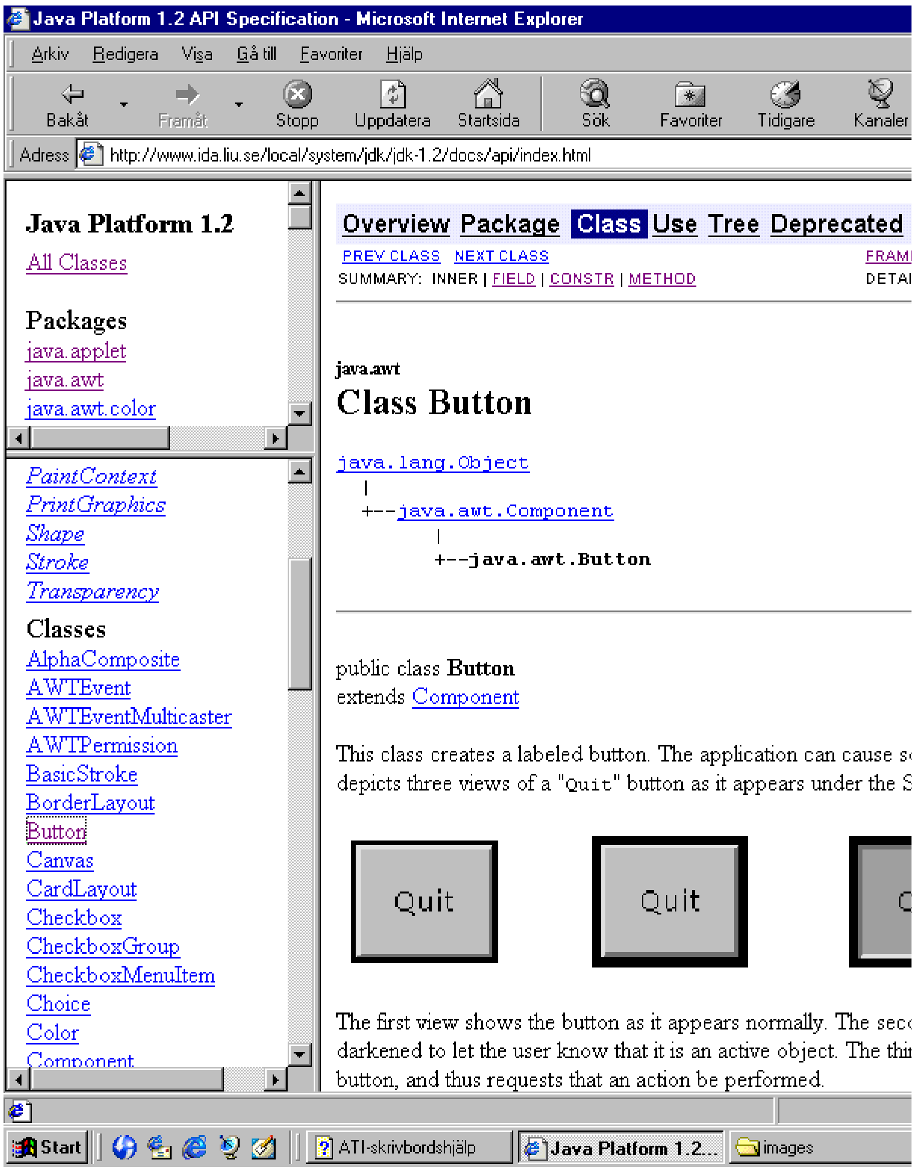
In certain areas the Web is a well developed platform for applications. Web browsers provide much support for information systems that consist mainly of text and interaction with text. Internet Explorer in particular provides high-level support for interactivity and manipulation of Web pages on the client machine. For these types of applications the Web is a suitable platform. We find Web browsers that are well-developed for the type of presentation and manipulations we want to achieve in DJavadoc.

Using Web browsers as application platforms also has consequences. The limitations of the Web browser are imposed on browser applications. For instance, security restrictions normally prohibit Web pages to access the client machines to save data or to interact with other programs. The restriction is advisable but still makes Web-application development more difficult. In the DJavadoc project, we have used *cookies* [65, 66] to save data on the client machine. Cookies are a relatively poor form of data storage. In Internet Explorer 5 the possibilities to save is extended somewhat [64].

4.2 The Official Java API reference documentation

4.2.1 Class Documents

The class documents are the core of the API reference documentation that the Standard Doclet generates. Figure 4.2 shows a sample class document from the Standard Doclet of JDK 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 surface of the classes (e.g., method source code is not exposed). Hypertext is used to link from the class documents either to related class documents or to other parts of the document.



Adress http://www.ida.liu.se/local/system/jdk/jdk-1.2/docs/api/index.html

Java Platform 1.2
All Classes

Packages
java.applet
java.awt
java.awt.color

PaintContext
PrintGraphics
Shape
Stroke
Transparency

Classes
AlphaComposite
AWTEvent
AWTEventMulticaster
AWTPermission
BasicStroke
BorderLayout
Button
Canvas
CardLayout
Checkbox
CheckboxGroup
CheckboxMenuItem
Choice
Color
Component

Overview Package **Class** Use Tree Deprecated
PREV CLASS NEXT CLASS FRAM
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAI

java.awt
Class Button

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Button
```

public class **Button**
extends Component

This class creates a labeled button. The application can cause s
depicts three views of a "`Quit`" button as it appears under the S

Quit Quit

The first view shows the button as it appears normally. The sec
darkened to let the user know that it is an active object. The thi
button, and thus requests that an action be performed.

Figure 4.2: An example of the Standard Doclet presenting a class document and the table of contents. The class documents describe each class that the reference documentaion was generated for (e.g., entered into the Javadoc program). The table of contents is a source index, listing classes in packages. Classes are grouped into packages by the developers.

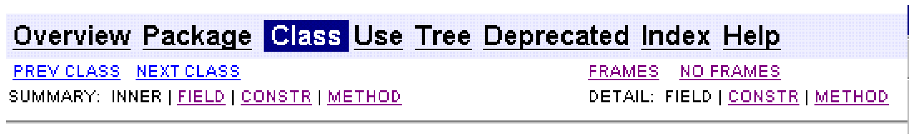
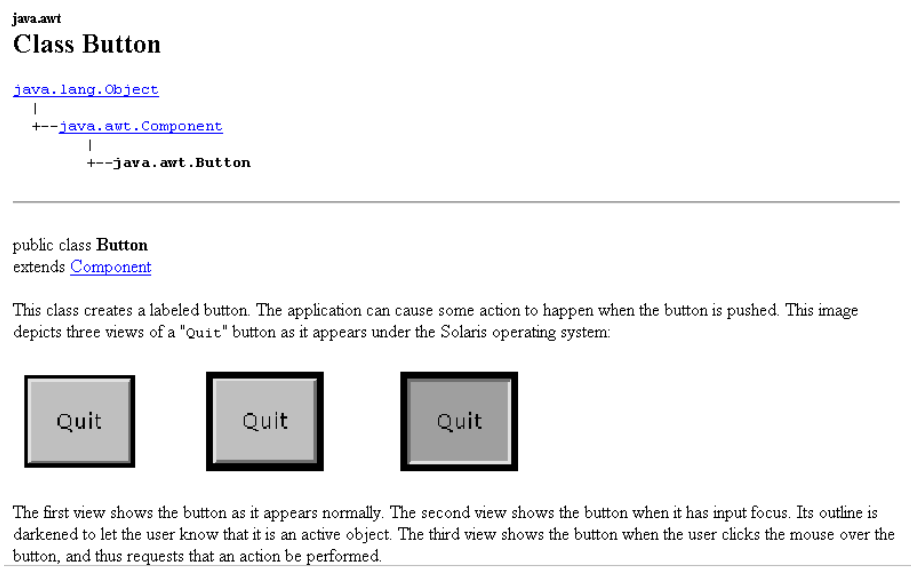

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Figure 4.3: 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.

There is a general hyperlink section at the top (and also at the bottom) of the class document (see Figure 4.3). Class documents have a header and a footer that contain hyperlinks to other documents. The hyperlinks point to various documents in the reference document, to strategic points in the class document, and also to a non-frame version of the reference document (which does not show the table of contents to the left of the class document [see Figure 4.2]). The same header and footer are available in all documents but links are not always active.



java.awt

Class Button

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Button
```

public class **Button**
extends Component

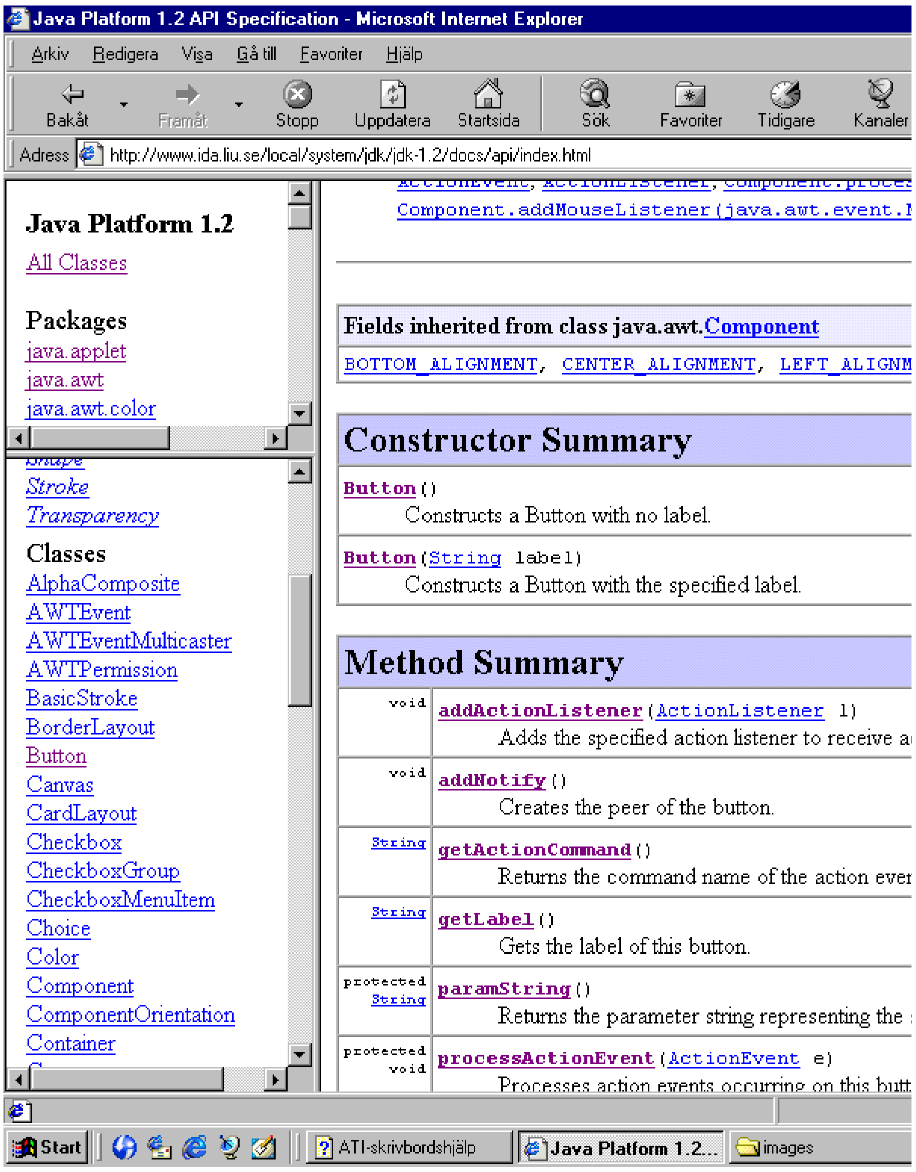
This class creates a labeled button. The application can cause some action to happen when the button is pushed. This image depicts three views of a "Quit" button as it appears under the Solaris operating system:

The first view shows the button as it appears normally. The second view shows the button when it has input focus. Its outline is darkened to let the user know that it is an active object. The third view shows the button when the user clicks the mouse over the button, and thus requests that an action be performed.

Figure 4.4: 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.

After the header comes a general description of the class containing both technical information and a general description of the class (see Figure 4.4). The class description starts with the class name. Furthermore, the class description contains an inheritance tree, information about known sub classes and so on. Finally the description ends with a text written into the source code by the programmer. For JDK it is common that the written comments also include hyperlinks to related classes that the programmer regards as being of interest.

The class document continues with summaries of the class members (see Figure 4.5). Inner classes, fields, constructors, and methods are summarized separately and ordered alphabetically. The summary contains information about the member's type, parameters, return values, and name (if they apply for the particular member type).


Java Platform 1.2 API Specification - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress http://www.ida.liu.se/local/system/jdk/jdk-1.2/docs/api/index.html
Start ATI-skrivbordshjälp Java Platform 1.2... images

Java Platform 1.2

All Classes

Packages
java.applet
java.awt
java.awt.color

Shape
Stroke
Transparency

Classes
AlphaComposite
AWTEvent
AWTEventMulticaster
AWTPermission
BasicStroke
BorderLayout
Button
Canvas
CardLayout
Checkbox
CheckboxGroup
CheckboxMenuItem
Choice
Color
Component
ComponentOrientation
Container

`ActionEvent, ActionListener, Component.proce`
`Component.addMouseListener(java.awt.event.M`

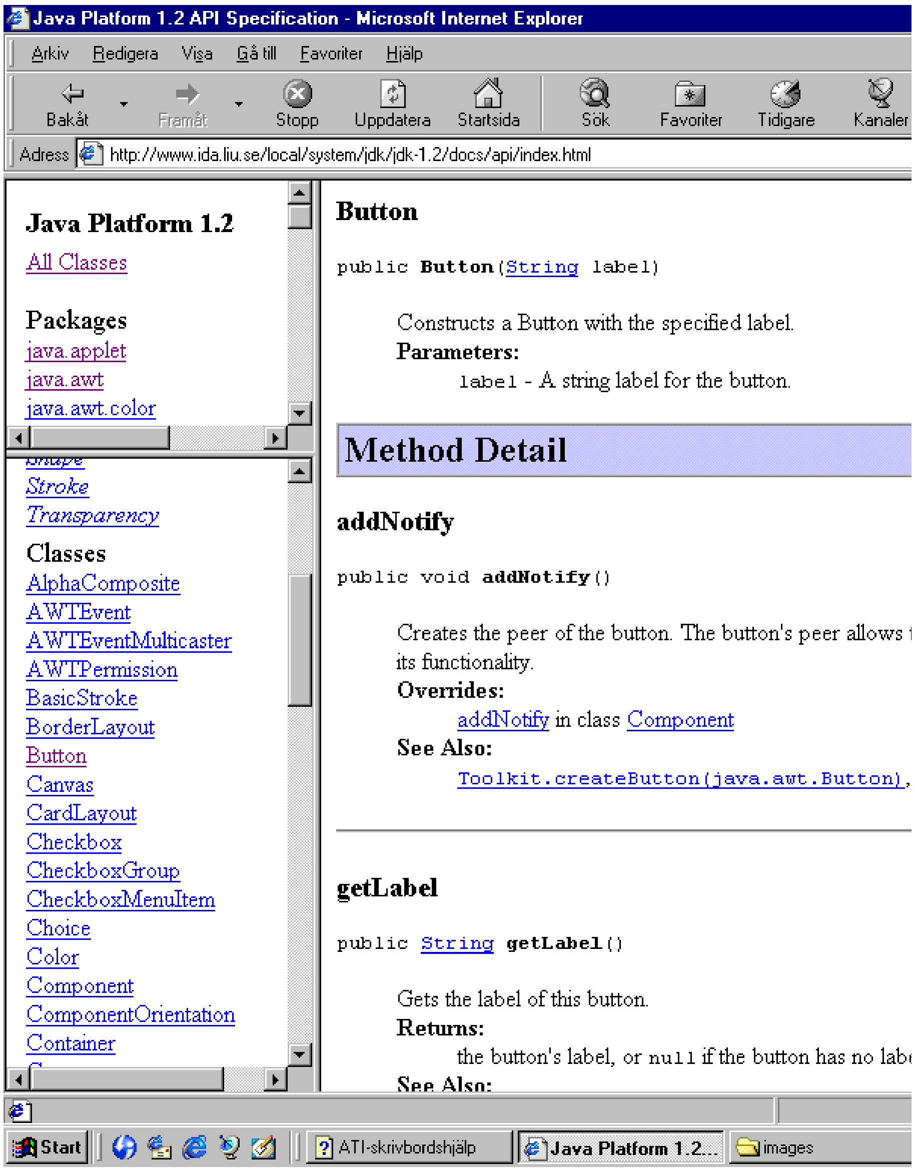
Fields inherited from class java.awt.Component
`BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNM`

Constructor Summary
`Button``()`   Constructs a Button with no label.
`Button``(String label)`   Constructs a Button with the specified label.

Method Summary	
`void`	**`addActionListener`**`(ActionListener l)` Adds the specified action listener to receive a
`void`	**`addNotify`**`()` Creates the peer of the button.
`String`	**`getActionCommand`**`()` Returns the command name of the action ever
`String`	**`getLabel`**`()` Gets the label of this button.
`protected String`	**`paramString`**`()` Returns the parameter string representing the
`protected void`	**`processActionEvent`**`(ActionEvent e)` Processes action events occurring on this butt

Figure 4.5: 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.

Finally, the detailed descriptions of the class members are provided (see Figure 4.6). In the member description, the typography of the comments follow the tagging convention of Javadoc. Generally a description of what the member is, details on what parameters stand for, and what may be returned are provided. The quality of the description is dependent upon comments written by programmers, even though some comments are inferred from the source code.


Java Platform 1.2 API Specification - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress http://www.ida.liu.se/local/system/jdk/jdk-1.2/docs/api/index.html
Start ATI-skrivbordshjälp Java Platform 1.2... images

Java Platform 1.2

All Classes

Packages

java.applet
java.awt
java.awt.color

Shape
Stroke
Transparency

Classes

AlphaComposite
AWTEvent
AWTEventMulticaster
AWTPermission
BasicStroke
BorderLayout
Button
Canvas
CardLayout
Checkbox
CheckboxGroup
CheckboxMenuItem
Choice
Color
Component
ComponentOrientation
Container

Button

```
public Button(String label)
```

Constructs a Button with the specified label.
Parameters:
`label` - A string label for the button.

Method Detail

addNotify

```
public void addNotify()
```

Creates the peer of the button. The button's peer allows
its functionality.
Overrides:
addNotify in class Component
See Also:
`Toolkit.createButton(java.awt.Button),`

getLabel

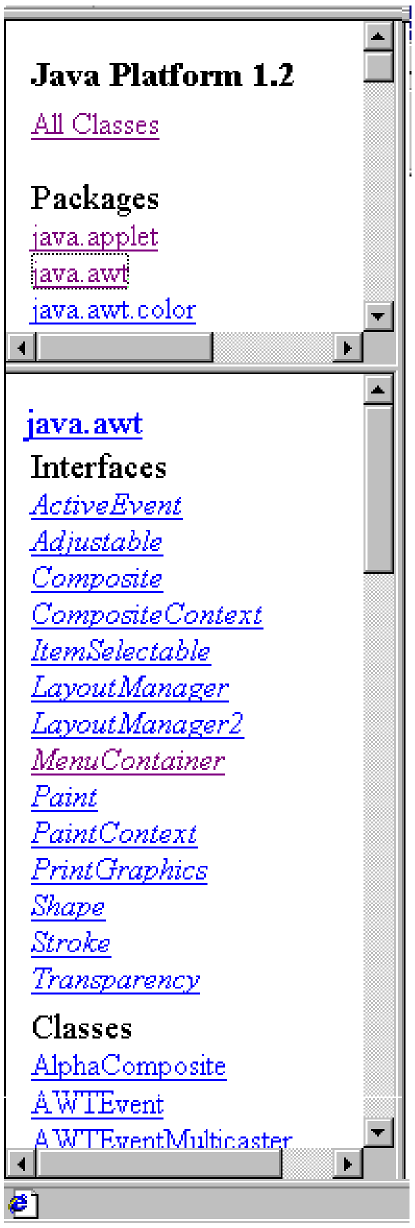
```
public String getLabel()
```

Gets the label of this button.
Returns:
the button's label, or `null` if the button has no lab
See Also:

Figure 4.6: The description of class members is provided separately from the summary. It contains comments on the source code written by developers. The member descriptions can be reached via hyperlinks in the summary.

4.2.2 Table of Contents

The table of contents is the navigational device for the Standard Doclet API reference documentation (see Figure 4.7). From the table of contents readers 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.



Java Platform 1.2

All Classes

Packages

java.applet
java.awt
java.awt.color

java.awt

Interfaces

ActiveEvent
Adjustable
Composite
CompositeContext
ItemSelectable
LayoutManager
LayoutManager2
MenuContainer
Paint
PaintContext
PrintGraphics
Shape
Stroke
Transparency

Classes

AlphaComposite
AWTEvent
AWTEventMulticaster

Figure 4.7: As a navigational tool, the table of contents is placed to the left in the browser. It is an index of all classes ordered alphabetically in package groups. The index represents a source development organization.

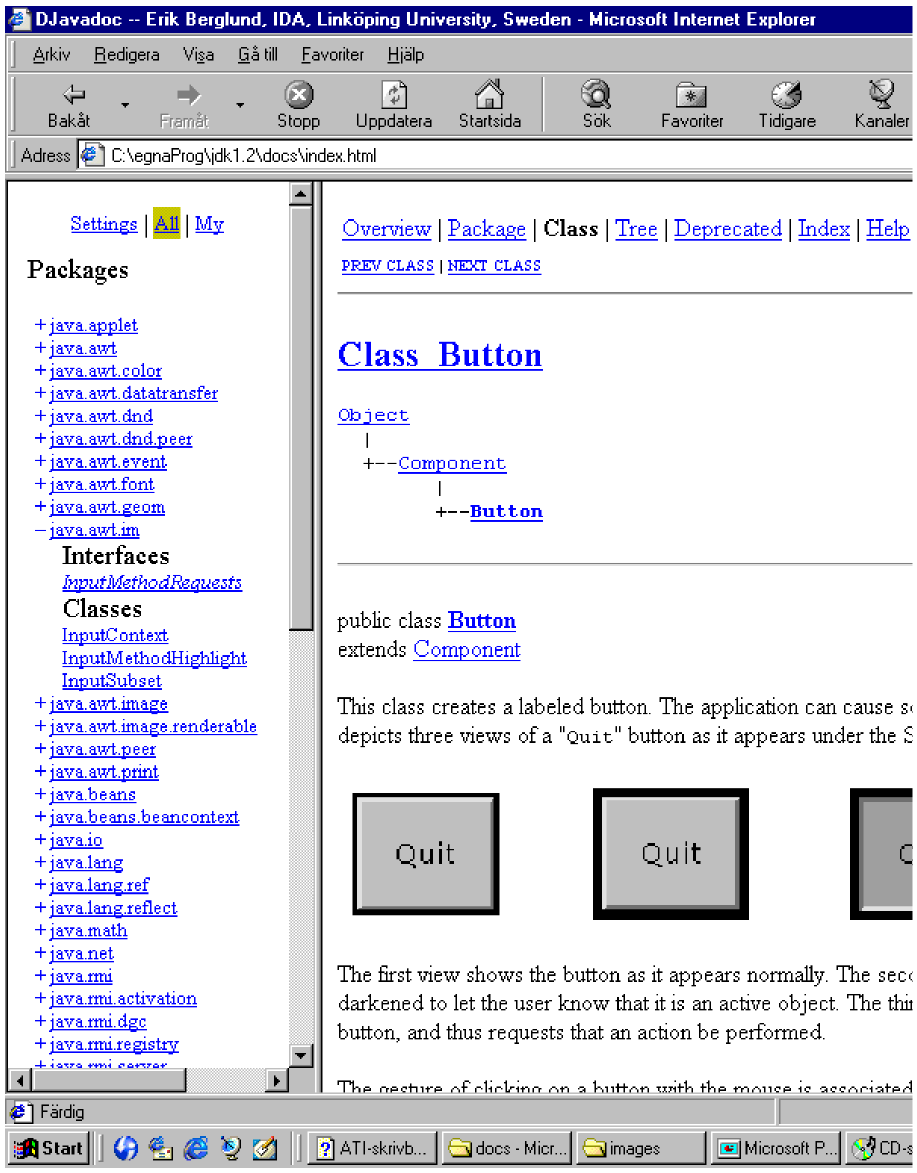
4.2.3 Other Documents

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. We do not describe them in detail because they have not been part of the DJavadoc project. These documents mainly provide additional navigational views on the API reference documentation.

4.3 Extensions Introduced in DJavadoc

4.3.1 Basic Comparison

The extensions made to DJavadoc all concern dynamic typography, not the content of the API reference documentation nor the static typography as shown in Figure 4.8. The official JDK API reference documentation for Java programmers is defined by the Standard Doclet. As an example of a computer-based API reference documentation several aspects of computer-based reading (e.g., hyperlinks, search engines, and typography) are related to the Standard Doclet. In the DJavadoc project we examine only one such aspect, namely the introduction of dynamic typography functionality. From a perspective of computer-based API reference documentation, the issue of dynamic typography is of major interest because it represents a dimension of reading not available in print. However, from a software-engineering perspective, dynamic typography is one way of achieving multiple views of information. To help Java programmers other aspect may be equally important or more important to address (e.g., introducing code examples into the Javadoc-generated documentation).


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Färdig
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

Packages

+java.applet
+java.awt
+java.awt.color
+java.awt.datatransfer
+java.awt.dnd
+java.awt.dnd.peer
+java.awt.event
+java.awt.font
+java.awt.geom
−java.awt.im
Interfaces
InputMethodRequests
Classes
InputContext
InputMethodHighlight
InputSubset
+java.awt.image
+java.awt.image.renderable
+java.awt.peer
+java.awt.print
+java.beans
+java.beans.beancontext
+java.io
+java.lang
+java.lang.ref
+java.lang.reflect
+java.math
+java.net
+java.rmi
+java.rmi.activation
+java.rmi.dgc
+java.rmi.registry
+java.rmi.server

Overview | Package | **Class** | Tree | Deprecated | Index | Help

PREV CLASS | NEXT CLASS

Class Button

```
Object
  |
  +--Component
        |
        +--Button
```

public class **Button**
extends Component

This class creates a labeled button. The application can cause s
depicts three views of a "`Quit`" button as it appears under the S

Quit Quit

The first view shows the button as it appears normally. The seco
darkened to let the user know that it is an active object. The thi
button, and thus requests that an action be performed.

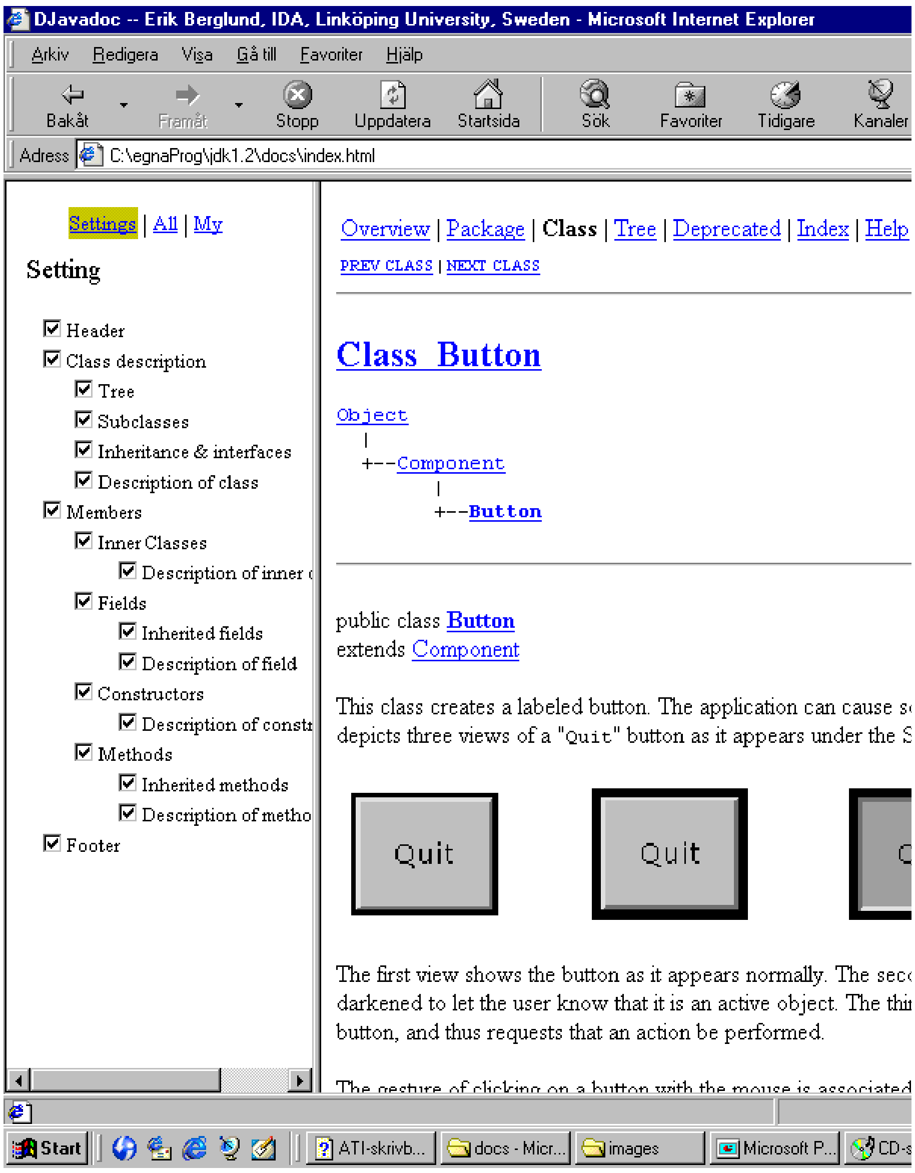
The gesture of clicking on a button with the mouse is associated

Figure 4.8: The fully expanded DJavadoc version of the Standard Doclet is equal both in content and class organization. On the surface the table of contents appears to be the most different.

4.3.2 Settings

In DJavadoc it is possible to define a default setting for the visibility of *document elements*. The Doclet API delivers an information model for Javadoc-generated documentation. In the Standard Doclet the information model has been transformed into an information model by the Javadoc Team. Static typography is used to draw attention to important pieces of information. However, in DJavadoc we propose another method for making important information more visible and excessive information less visible (in fact invisible). Using DHTML-technology available in Microsoft Internet Explorer 4 (or later versions) we have added dynamic typography to the API reference documentation. Pieces of information can be removed from the reading surface of the browser, thereby both removing excessive information and increasing the visibility of the remaining information.

In principle we have taken the existing information model, made it explicit, and created an interaction device with which the reader can control the visibility of information types. Based on our own programming experience, we have marked up the class documents into conceptual groups of relevant chunks of information. The model is represented in the interaction device contained in the `Settings-map` in the table of contents (see Figure 4.9). The document is divided into two parts: a class description part and a member part. The class description contains a full inheritance tree, known sub-classes, 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.



C:\egnaProg\jdk1.2\docs\index.html

Settings | All | My

Setting

- ☑ Header
- ☑ Class description
  - ☑ Tree
  - ☑ Subclasses
  - ☑ Inheritance & interfaces
  - ☑ Description of class
- ☑ Members
  - ☑ Inner Classes
    - ☑ Description of inner c
  - ☑ Fields
    - ☑ Inherited fields
    - ☑ Description of field
  - ☑ Constructors
    - ☑ Description of constr
  - ☑ Methods
    - ☑ Inherited methods
    - ☑ Description of metho
- ☑ Footer

Overview | Package | **Class** | Tree | Deprecated | Index | Help

PREV CLASS | NEXT CLASS

Class Button

```
Object
  |
  +--Component
        |
        +--Button
```

public class **Button**
extends Component

This class creates a labeled button. The application can cause s
depicts three views of a "`Quit`" button as it appears under the S

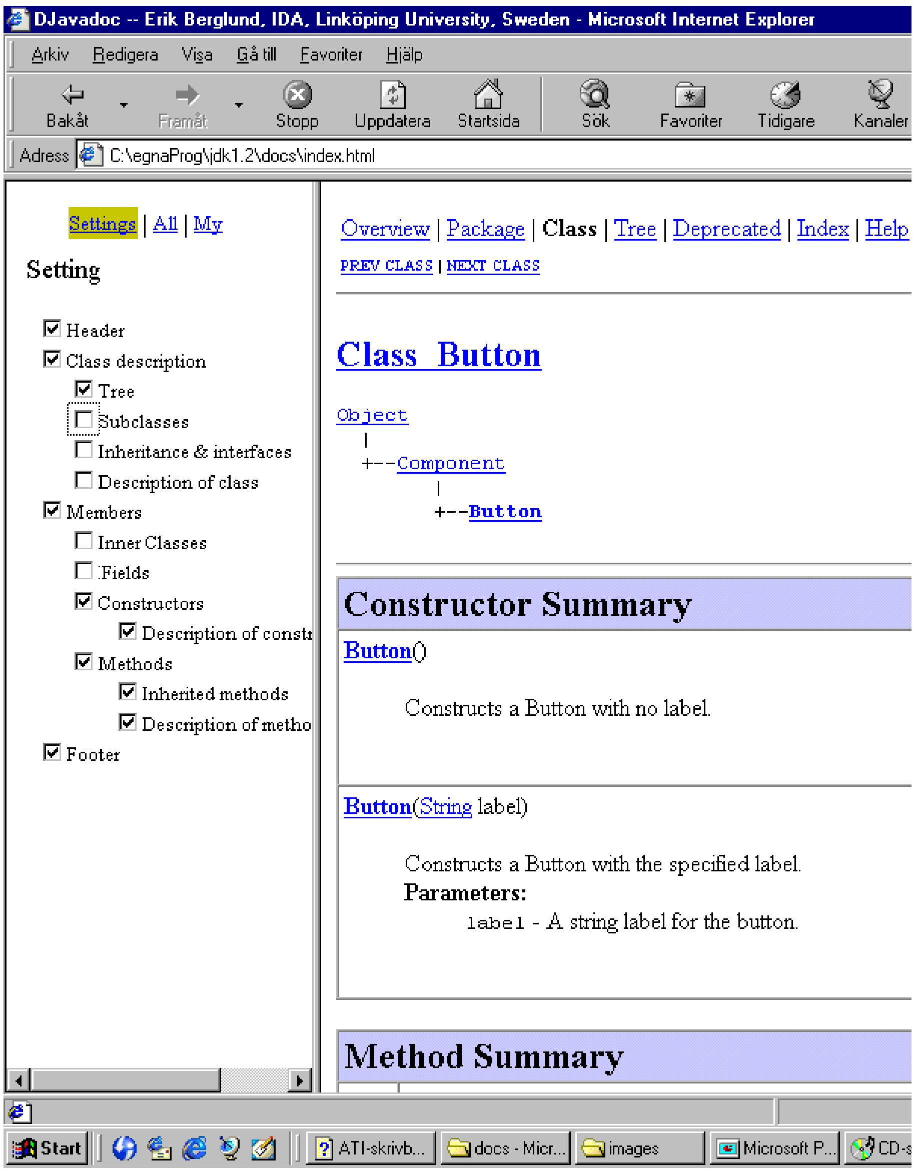
Quit Quit Q

The first view shows the button as it appears normally. The seco
darkened to let the user know that it is an active object. The thi
button, and thus requests that an action be performed.

The gesture of clicking on a button with the mouse is associated

Figure 4.9: 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.

Using the `Settings-map` the reader may define a default for the visibility of information types. By checking off different parts in the `Settings-map` the reader may collapse parts of the document. Figure 4.10 shows the difference between the fully expanded document and a class document in which the class description is collapsed. Notice that the `Settings-map` also collapses itself. Checking off an element in the `Settings-map` will lead to the collapse of all instances of that element in the document. The default 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 interesting setting. Figure 4.11 shows the consequences of this setting.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

Setting

- [✓] Header
- [✓] Class description
  - [✓] Tree
  - [ ] Subclasses
  - [ ] Inheritance & interfaces
  - [ ] Description of class
- [✓] Members
  - [ ] Inner Classes
  - [ ] Fields
  - [✓] Constructors
    - [✓] Description of constr
  - [✓] Methods
    - [✓] Inherited methods
    - [✓] Description of metho
- [✓] Footer

Overview | Package | **Class** | Tree | Deprecated | Index | Help

PREV CLASS | NEXT CLASS

Class Button

```
Object
  |
  +--Component
        |
        +--Button
```

Constructor Summary

Button()

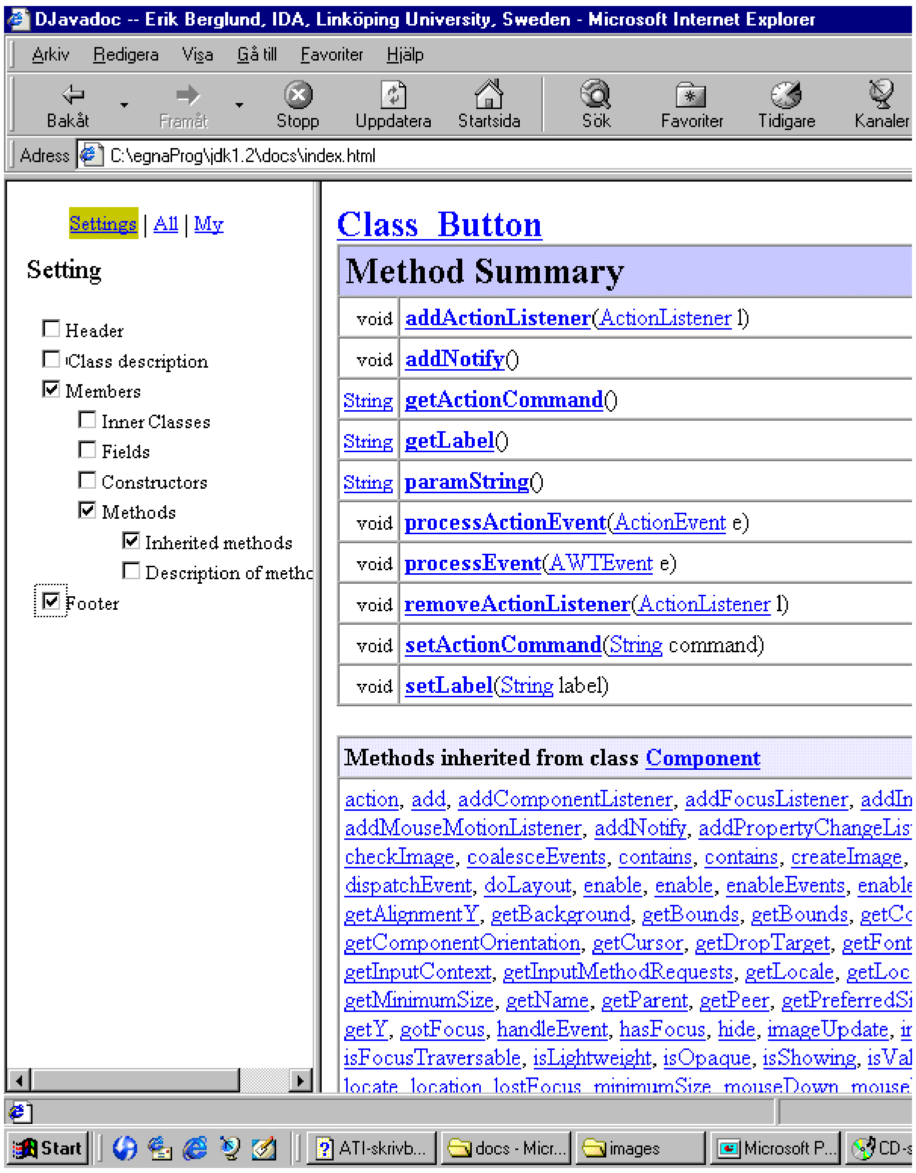
Constructs a Button with no label.

Button(String label)

Constructs a Button with the specified label.
Parameters:
`label` - A string label for the button.

Method Summary

Figure 4.10: 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.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

Setting

☐ Header
☐ Class description
☑ Members
☐ Inner Classes
☐ Fields
☐ Constructors
☑ Methods
☑ Inherited methods
☐ Description of methc
☑ Footer

Class Button

Method Summary	
void	**addActionListener**(ActionListener l)
void	**addNotify**()
String	**getActionCommand**()
String	**getLabel**()
String	**paramString**()
void	**processActionEvent**(ActionEvent e)
void	**processEvent**(AWTEvent e)
void	**removeActionListener**(ActionListener l)
void	**setActionCommand**(String command)
void	**setLabel**(String label)

Methods inherited from class Component

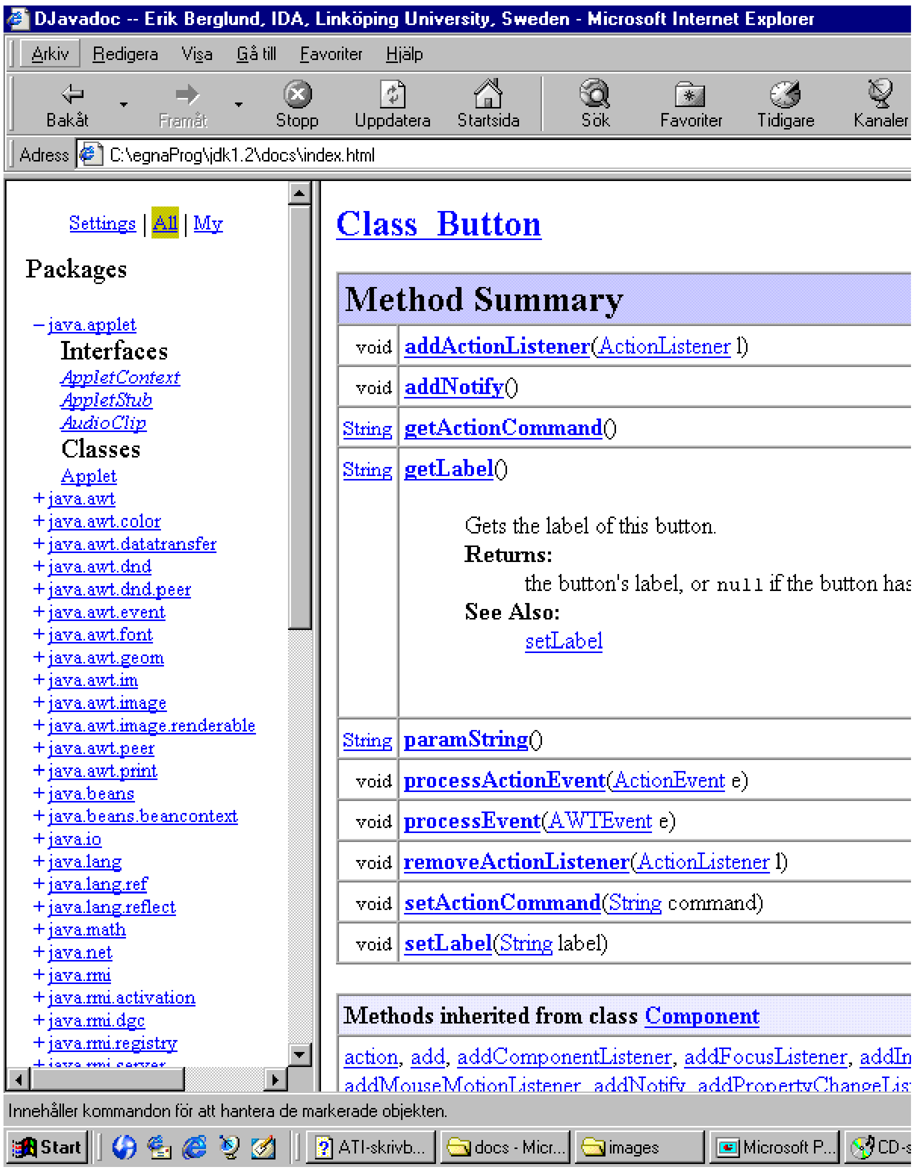
action, add, addComponentListener, addFocusListener, addIn
addMouseMotionListener, addNotify, addPropertyChangeLis
checkImage, coalesceEvents, contains, contains, createImage,
dispatchEvent, doLayout, enable, enable, enableEvents, enable
getAlignmentY, getBackground, getBounds, getBounds, getCo
getComponentOrientation, getCursor, getDropTarget, getFont
getInputContext, getInputMethodRequests, getLocale, getLoc
getMinimumSize, getName, getParent, getPeer, getPreferredSi
getY, gotFocus, handleEvent, hasFocus, hide, imageUpdate, in
isFocusTraversable, isLightweight, isOpaque, isShowing, isVal
locate, location, lostFocus, minimumSize, mouseDown, mouse

Figure 4.11: Readers 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 readers who are familiar with the particular classes.

4.3.3 Individual Description Openers

In addition to the default setting, the reader 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 readers consider important. However, readers 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.

The general class description and the description of individual class members can be collapsed and expanded by direct interaction. 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 4.12 shows how an individual element is expanded.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer

Arkiv Redigera Visa Gå till Favoriter Hjälp

Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler

Adress C:\egnaProg\jdk1.2\docs\index.html

Settings | All | My

Packages

− java.applet
Interfaces
AppletContext
AppletStub
AudioClip
Classes
Applet
+ java.awt
+ java.awt.color
+ java.awt.datatransfer
+ java.awt.dnd
+ java.awt.dnd.peer
+ java.awt.event
+ java.awt.font
+ java.awt.geom
+ java.awt.im
+ java.awt.image
+ java.awt.image.renderable
+ java.awt.peer
+ java.awt.print
+ java.beans
+ java.beans.beancontext
+ java.io
+ java.lang
+ java.lang.ref
+ java.lang.reflect
+ java.math
+ java.net
+ java.rmi
+ java.rmi.activation
+ java.rmi.dgc
+ java.rmi.registry

Class Button

Method Summary	
void	addActionListener(ActionListener l)
void	addNotify()
String	getActionCommand()
String	getLabel() Gets the label of this button. **Returns:** the button's label, or `null` if the button has **See Also:** setLabel
String	paramString()
void	processActionEvent(ActionEvent e)
void	processEvent(AWTEvent e)
void	removeActionListener(ActionListener l)
void	setActionCommand(String command)
void	setLabel(String label)

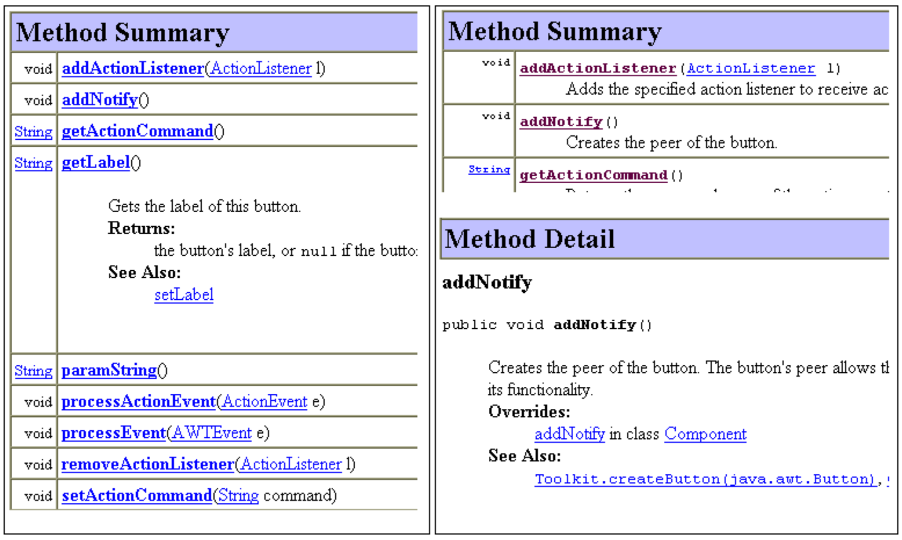
Methods inherited from class Component

action, add, addComponentListener, addFocusListener, addIn
addMouseMotionListener, addNotify, addPropertyChangeLis

Innehåller kommandon för att hantera de markerade objekten.

Start ATI-skrivb... docs - Micr... images Microsoft P... CD-

Figure 4.12: While reading the class document the reader can use the dynamic typography features to expand or collapse individual elements regardless of the default setting. In this example, the reader has expanded a particular method description. The reader actively chooses which information should be visible in the browser.



Method Summary

void **addActionListener**(ActionListener l)

void **addNotify**()

String **getActionCommand**()

String **getLabel**()

Gets the label of this button.

Returns:

the button's label, or `null` if the butto

See Also:

setLabel

String **paramString**()

void **processActionEvent**(ActionEvent e)

void **processEvent**(AWTEvent e)

void **removeActionListener**(ActionListener l)

void **setActionCommand**(String command)

Method Summary

void **`addActionListener`**`(ActionListener l)`

Adds the specified action listener to receive ac

void **`addNotify`**`()`

Creates the peer of the button.

String **`getActionCommand`**`()`

Method Detail

addNotify

`public void addNotify()`

Creates the peer of the button. The button's peer allows t
its functionality.

Overrides:

addNotify in class Component

See Also:

`Toolkit.createButton(java.awt.Button),`

Figure 4.13: The dynamic typography has enabled a more natural placing of the descriptions in DJavadoc compared to the Standard Doclet. The reader is relieved of a hyperlink-jump to the description.

4.3.4 Moving Text Parts

Texts have been moved in DJavadoc but only as a transition from static typography to dynamic typography. We have moved two text types in DJavadoc compared to the Standard Doclet. However, these relocations are the result of the change from static typography to dynamic typography and are not fundamental changes. In our opinion, the texts have achieved their natural placing, which was prohibited by the static-typography.

The descriptions of *class members* have been included in the summary list because dynamic typography allows a combination of summary and detailed form. In Section 4.2.1 we showed that class members were presented both in a summary and in a detailed list the Standard Doclet. Figure 4.13 shows the difference between DJavadoc and the Standard Doclet. By placing the description in the original summary we believe we have found a more natural place. The placing of description text in the Standard Doclet seems to us to be forced by the static typography. The summary can sill be achieved because DJavadoc can collapse the descriptions. In any case, readers are relieved of the hyperlink-jump (from the summary to the description) which might be disorienting. Also, a minor effect is that the size of the documents has decreased, mainly the screen size but also memory size.

Similarly, the class list in the table of contents has been relocated to the package list. In Section 4.2.2 we described how the package and class lists were put into two different frames. In DJavadoc we place the class lists as nested lists in the package list and add collapse and expand functionality. Figure 4.14 shows the difference between the Standard Doclet solution and the DJavadoc solution. As with the description texts, we think we achieved a more natural placing, which is less confusing from an orientation point of view. The primary advantage is, however, the ability to expand several packages at the same time. A somewhat negative effect is that the document grows very large as the packages are opened. Packages are pushed down, outside the visible field of the browser. Also the document can become a large structure of HTML elements that the browser must handle and changes to the document may therefore become slow. In retrospect, moving the class lists is not as clear a design choice as moving the description texts. However, from the perspective of the DJavadoc project the move illustrates the use of dynamic typography.

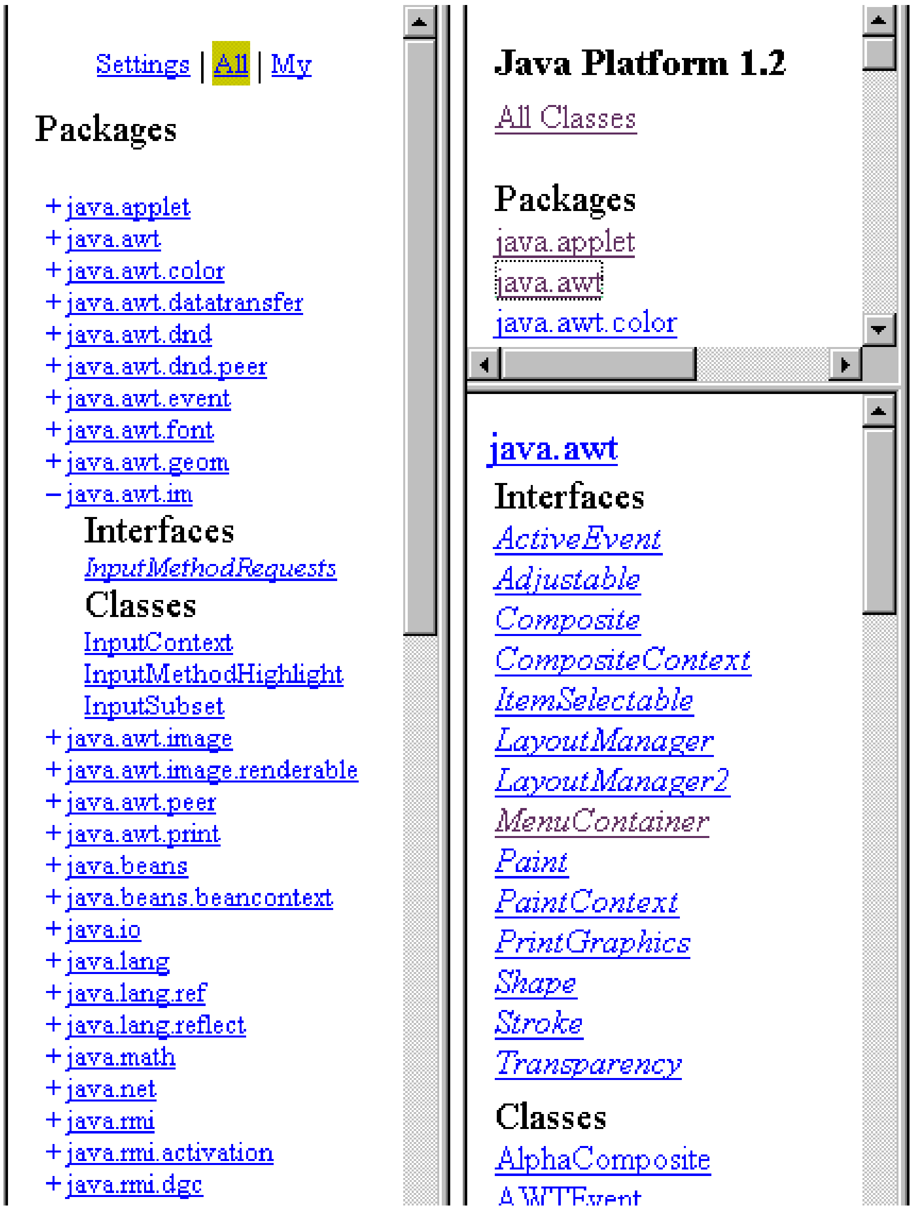
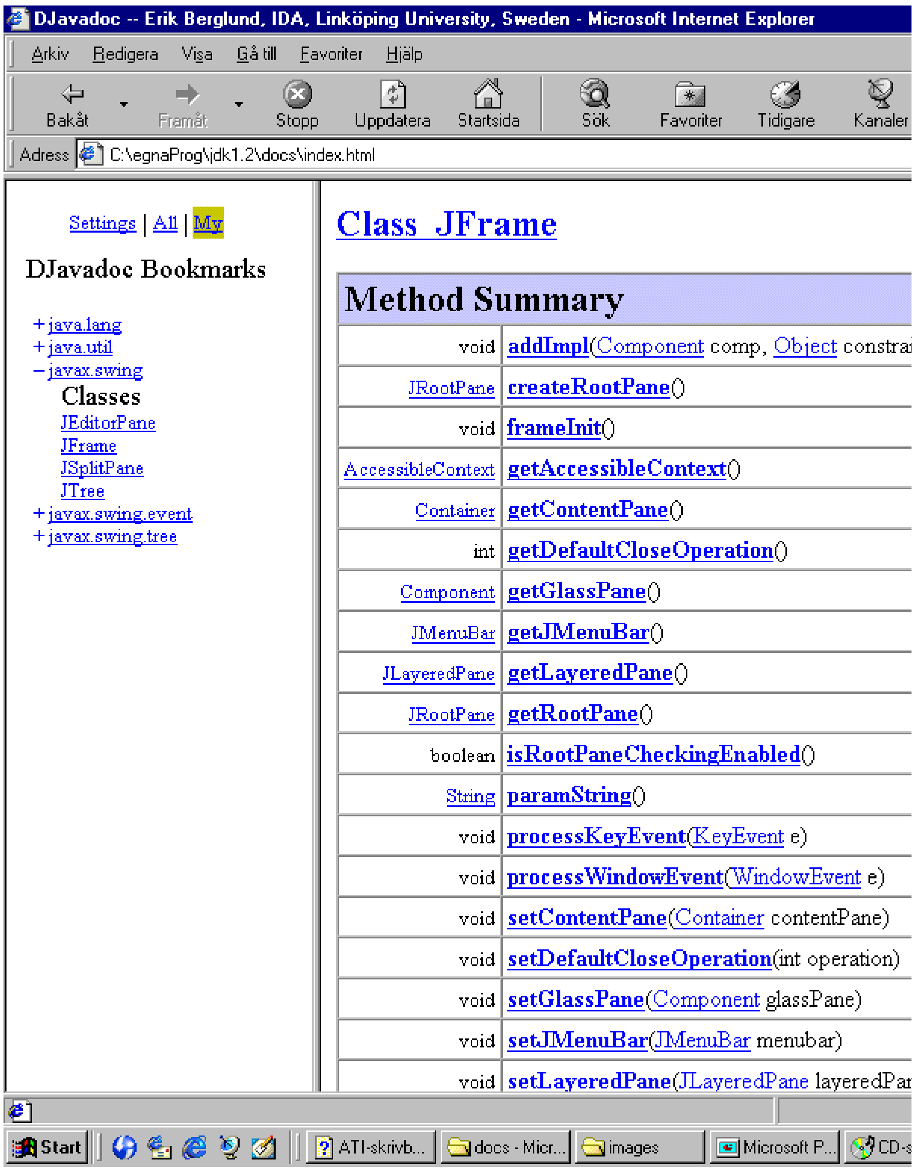


Figure 4.14: Compared to the Standard Doclet, DJavadoc offers an alternative table of contents in which the class lists are nested in the packages list. Class lists can be expanded to reveal the contents of the packages and remain expanded as new packages are opened. However, the class list can become very long and will, in such a case, push the remaining content out of the visible space when expanded.

4.3.5 DJavadoc Bookmarks

The table of contents can also be collapsed into a smaller subset of interesting hyperlinks in a so-called DJavadoc Bookmarks list. In Section 4.3.4 we discussed the moving of class lists into a collapsible package list. With over 1,800 classes in JDK 1.2 it becomes obvious that the reader needs more focused class lists to easily locate relevant classes. In DJavadoc we achieve this by constructing a separate table of contents (contained in the `My-map`) in which readers save classes in a bookmark fashion. Figure 4.15 shows the `My-map` filled with classes a reader might find interesting. Classes and even whole packages can also be easily removed from the `My-map`.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

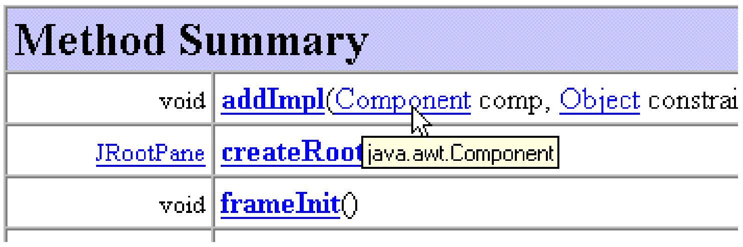
DJavadoc Bookmarks

+java.lang
+java.util
−javax.swing
  Classes
  JEditorPane
  JFrame
  JSplitPane
  JTree
+javax.swing.event
+javax.swing.tree

Class JFrame

Method Summary	
void	addImpl(Component comp, Object constrai
JRootPane	createRootPane()
void	frameInit()
AccessibleContext	getAccessibleContext()
Container	getContentPane()
int	getDefaultCloseOperation()
Component	getGlassPane()
JMenuBar	getJMenuBar()
JLayeredPane	getLayeredPane()
JRootPane	getRootPane()
boolean	isRootPaneCheckingEnabled()
String	paramString()
void	processKeyEvent(KeyEvent e)
void	processWindowEvent(WindowEvent e)
void	setContentPane(Container contentPane)
void	setDefaultCloseOperation(int operation)
void	setGlassPane(Component glassPane)
void	setJMenuBar(JMenuBar menubar)
void	setLayeredPane(JLayeredPane layeredPar

Figure 4.15: 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 reader. To keep the content relevant the reader should update the `My-map`.



Method Summary	
void	addImpl(Component comp, Object constrai
JRootPane	createRoot java.awt.Component
void	frameInit()

Figure 4.16: The full name of the class is hidden in a tool-tip box. Full names of classes are important because they define the unique name of the class. However, the package part of the name is seldom spelled out in code. Also, reading may be obstructed when several full names follow one another.

The idea behind the DJavadoc bookmarks is that the readers should actively update it to keep the content relevant. The reader's interest in classes changes, both during a project and between projects. Filling the `My-map` with classes of interest is a good initial way of collecting potential classes. However, if the `My-map` grows large, it becomes less useful. The reader should continually revise the class list.

4.3.6 ToolTip for Full Class Name

The full class name of classes in the class documents have been hidden in tool-tip text. The full name of Java classes contains a package part and a class part. The full class name is important because it represent a unique identifier (which the short class name does not necessarily do). However, in source code the full name is seldom spelled out. Instead the package name is imported and the class is referenced by its short name. Also, in the parameter list the text becomes clouded by all the extra text that may not be needed by the reader to tell classes apart. In the Standard Doclet the full class name is not spelled out. In DJavadoc we have placed the full name of the class in a tool-tip text, so that the information is available to the reader but does not obstruct reading. Figure 4.16 shows how the full class name is exposed by placing the mouse over the class name. Once again, this illustrates the use of dynamic typography in the API reference documentation.

4.3.7 Other Extensions

A few insignificant changes were also made to the Standard Doclet. To accompany the bigger changes in DJavadoc we had to make a number of smaller changes. The header and footer hyperlinks were changed to add and remove hyperlinks. The non-frame option was removed for technical reasons (the DJavadoc status is saved inside the table of contents document). A DHelp page was introduced to explain how the DJavadoc version works and its intended use. These changes are of a technical nature and does not concern the reading environment of the API reference documentation.

4.4 DJavadoc Performance

In most cases the performance of DJavadoc is instantaneous. 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 JDK classes, we experienced some delay. However, by redesigning the documentation for performance we can probably achieve instantaneous performance in several of these cases. As an example we solved the performance problem of the table of contents, see Section 4.8.4.

4.5 Using DHTML

In the DJavadoc project we used DHTML in the Microsoft Internet Explorer browser. DHTML technology cannot yet be viewed as being independent of the browser in which the system was developed, even though the browsers usually accept and implement W3C standards, see [79].

Using DHTML technology in Microsoft Internet Explorer, in our experience, is a straightforward development effort that benefits from a highly-developed application platform. The browser can be seen as an API and as such we found it most adequate for our needs. In our experience, the DJavadoc development has benefited from the relatively high level of support browsers deliver by implementing W3C standards. The browser provides support for interactivity in text based on HTML.

4.6 Technichal Data on DJavadoc

DJavadoc does not add to the Javadoc generation time and memory requirements. Javadoc 1.2 requires 120 MB of memory to generate the JDK 1.2 documentation (55 packages), which takes 8 minutes on an Ultra Enterprise with 512 MB of memory [58]. Javadoc keeps the Doclet API information structure in memory and the memory requirements are therefore very large. Time is a factor of the sheer size of the JDK 1.2. The extensions made in DJavadoc have a minimal effect on the time or memory requirements of Javadoc.

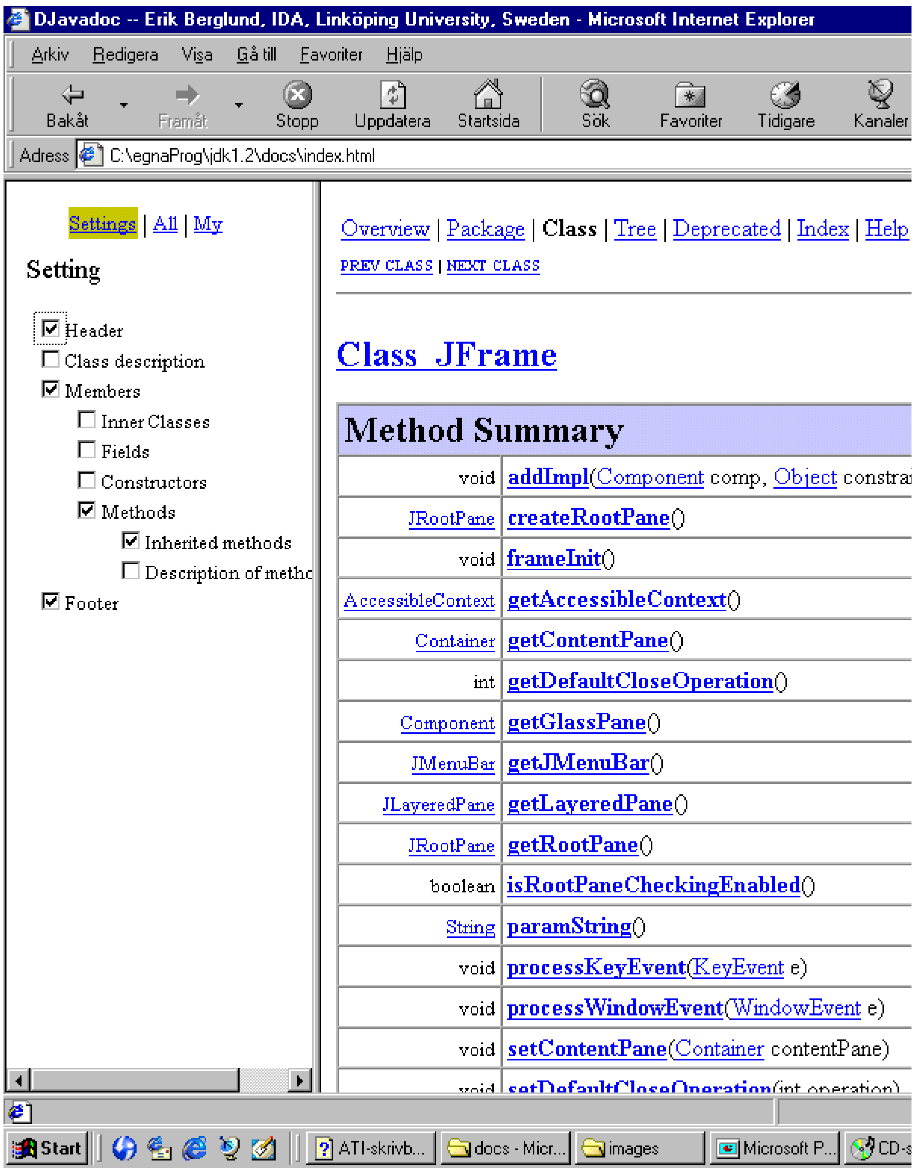
The extensions made in DJavadoc are principally rearrangements and additions to the Standard Doclet and the inclusion of DHTML scripting. The total amount of work that has gone into the DJavadoc project's implementation phase is about 5 man-months (spent on understanding, adding changes and rearranging 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.

Changing or extending DJavadoc may require only a small effort. Much of the work involved in designing DJavadoc is the definition of an information structure. Extensions and changes to DJavadoc that are based on this structure may require only small amounts of DHTML scripting. However, changes to the content, static typography, or the organization of the Standard Doclet's output will require major changes.

4.7 Example Working Scenario for DJavadoc

Let us consider a sample scenario describing how a reader might use the dynamic functionality of DJavadoc. A working example can put the usefulness of DJavadoc into perspective. DJavadoc may be interesting but we also hope to show its practical importance. In our example the reader is using the API reference documentation as a syntactical index that provides coding specification. The reader knows which classes to use and what those classes are designed for.

The reader sets the default setting in the `Settings-map`. Either from a previous session or as the reader begins to read, the reader defines the setting shown in Figure 4.17. The methods are kept expanded but with collapsed descriptions. The reader 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.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

Setting

- ☑ Header
- ☐ Class description
- ☑ Members
  - ☐ Inner Classes
  - ☐ Fields
  - ☐ Constructors
  - ☑ Methods
    - ☑ Inherited methods
    - ☐ Description of methc
- ☑ Footer

Overview | Package | **Class** | Tree | Deprecated | Index | Help

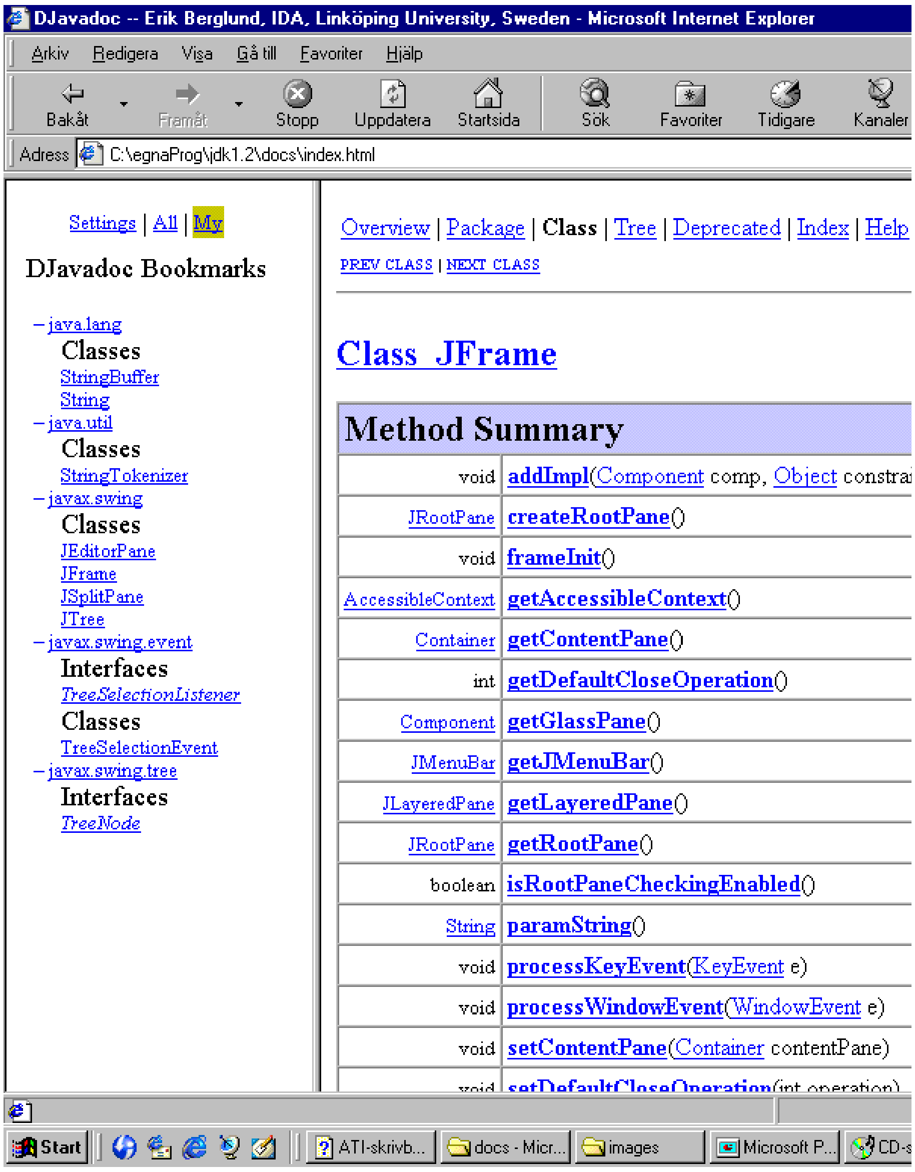
PREV CLASS | NEXT CLASS

Class JFrame

Method Summary	
void	**addImpl**(Component comp, Object constrai
JRootPane	**createRootPane**()
void	**frameInit**()
AccessibleContext	**getAccessibleContext**()
Container	**getContentPane**()
int	**getDefaultCloseOperation**()
Component	**getGlassPane**()
JMenuBar	**getJMenuBar**()
JLayeredPane	**getLayeredPane**()
JRootPane	**getRootPane**()
boolean	**isRootPaneCheckingEnabled**()
String	**paramString**()
void	**processKeyEvent**(KeyEvent e)
void	**processWindowEvent**(WindowEvent e)
void	**setContentPane**(Container contentPane)
void	**setDefaultCloseOperation**(int operation)

Figure 4.17: The reader chooses a default setting which only presents methods without description but with the inherited methods. The document is altered acordingly, as is each new class document the reader loads into the browser.

The reader has an ongoing project and therefore starts browsing from the `My-map`, containing the personal DJavadoc Bookmarks. During previous browsing the reader has collected a list of class documents of interest (see Figure 4.18). DJavadoc bookmarks represent personal views of the table of contents. Our reader 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.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

DJavadoc Bookmarks

- java.lang
  - Classes
    - StringBuffer
    - String
- java.util
  - Classes
    - StringTokenizer
- javax.swing
  - Classes
    - JEditorPane
    - JFrame
    - JSplitPane
    - JTree
- javax.swing.event
  - Interfaces
    - *TreeSelectionListener*
  - Classes
    - TreeSelectionEvent
- javax.swing.tree
  - Interfaces
    - *TreeNode*

Overview | Package | **Class** | Tree | Deprecated | Index | Help

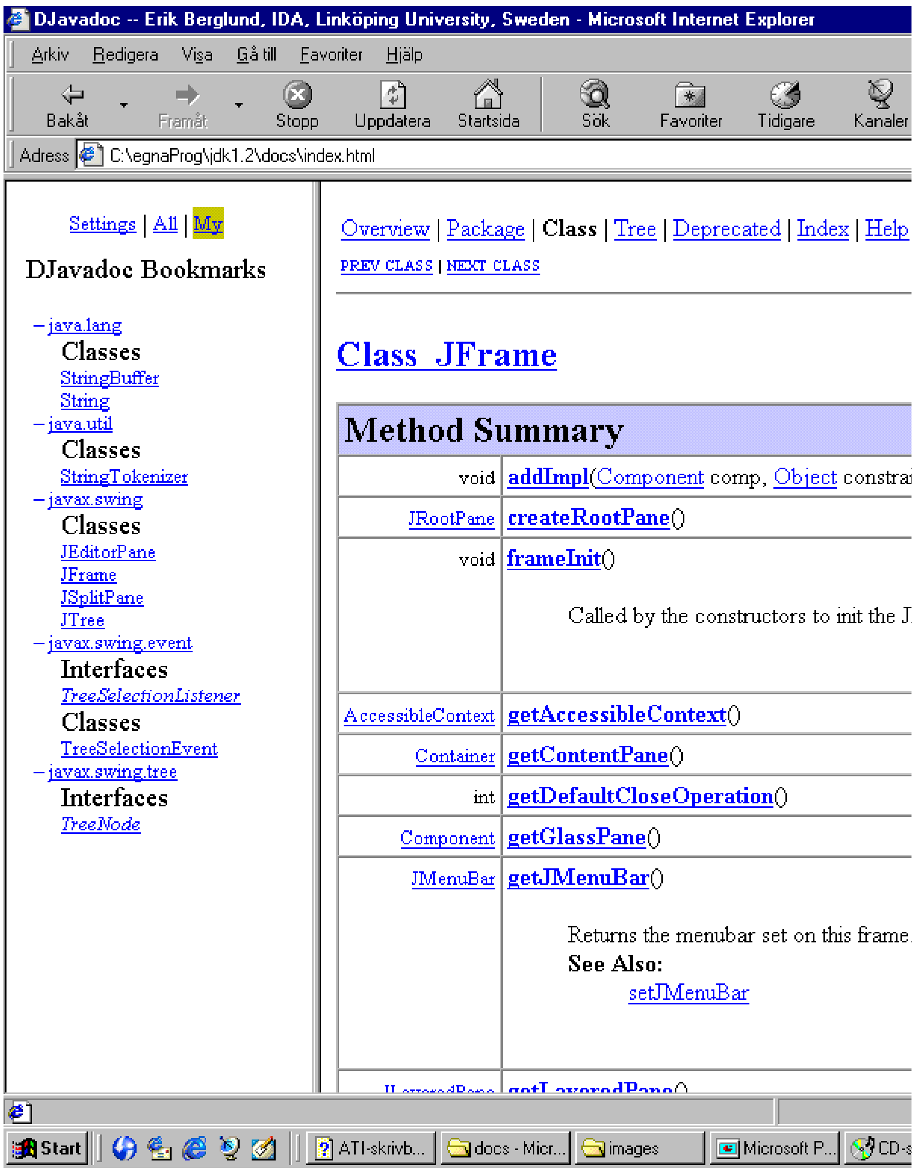
PREV CLASS | NEXT CLASS

Class JFrame

Method Summary	
void	**addImpl**(Component comp, Object constrai
JRootPane	**createRootPane**()
void	**frameInit**()
AccessibleContext	**getAccessibleContext**()
Container	**getContentPane**()
int	**getDefaultCloseOperation**()
Component	**getGlassPane**()
JMenuBar	**getJMenuBar**()
JLayeredPane	**getLayeredPane**()
JRootPane	**getRootPane**()
boolean	**isRootPaneCheckingEnabled**()
String	**paramString**()
void	**processKeyEvent**(KeyEvent e)
void	**processWindowEvent**(WindowEvent e)
void	**setContentPane**(Container contentPane)
void	**setDefaultCloseOperation**(int operation)

Figure 4.18: Previously the reader has created a `My-map` containing classes of interest. For instance, the reader might use the classes in a programming project. Another plausible reason is that reader often uses the classes.

While reading, the visibility is manipulated to display the inner workings of elements of interest (see Figure 4.19). In the default setting the reader may only define the visibility of information types, not instances. The collapsed descriptions represent a type that is not of interest to our reader. However, specific individual descriptions are relevant during coding and therefore our reader expands a few descriptions while reading (by clicking on the method names). For instance, the descriptions may reveal how parameters are interpreted.


DJavadoc -- Erik Berglund, IDA, Linköping University, Sweden - Microsoft Internet Explorer
Arkiv Redigera Visa Gå till Favoriter Hjälp
Bakåt Framåt Stopp Uppdatera Startsida Sök Favoriter Tidigare Kanaler
Adress C:\egnaProg\jdk1.2\docs\index.html
Start ATI-skrivb... docs - Micr... images Microsoft P... CD-s

Settings | All | My

DJavadoc Bookmarks

- java.lang
  - Classes
    - StringBuffer
    - String
- java.util
  - Classes
    - StringTokenizer
- javax.swing
  - Classes
    - JEditorPane
    - JFrame
    - JSplitPane
    - JTree
- javax.swing.event
  - Interfaces
    - *TreeSelectionListener*
  - Classes
    - TreeSelectionEvent
- javax.swing.tree
  - Interfaces
    - *TreeNode*

Overview | Package | **Class** | Tree | Deprecated | Index | Help

PREV CLASS | NEXT CLASS

Class JFrame

Method Summary

void	**addImpl**(Component comp, Object constrai
JRootPane	**createRootPane**()
void	**frameInit**() Called by the constructors to init the J
AccessibleContext	**getAccessibleContext**()
Container	**getContentPane**()
int	**getDefaultCloseOperation**()
Component	**getGlassPane**()
JMenuBar	**getJMenuBar**() Returns the menubar set on this frame. **See Also:** setJMenuBar

Figure 4.19: Whilste reading, the reader finds it relevant to open up the description of certain methods. Perhaps the reader needs to be reminded about the meaning of a return value.

The reader saves new classes to the bookmark list. During browsing the reader comes upon new, potentially useful classes. By alt-clicking (see Section 4.1.4) the reader saves new bookmarks which are added to the list according to the package structure. For now, our reader saves without giving it much consideration because later the reader removes classes that in retrospect were not so relevant.

After a while, the reader changes the default setting to include constructors. The reader is well aware of the missing components in the documentation. At some point the reader realizes that constructors are lacking from the documentation and changes the default. The constructor description is left collapsed just like the description of methods.

4.8 Different Prototype Generations

4.8.1 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 [30]. In this project we used Protégé, a tool for generation of knowledge-acquisition tools [34, 69] 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.

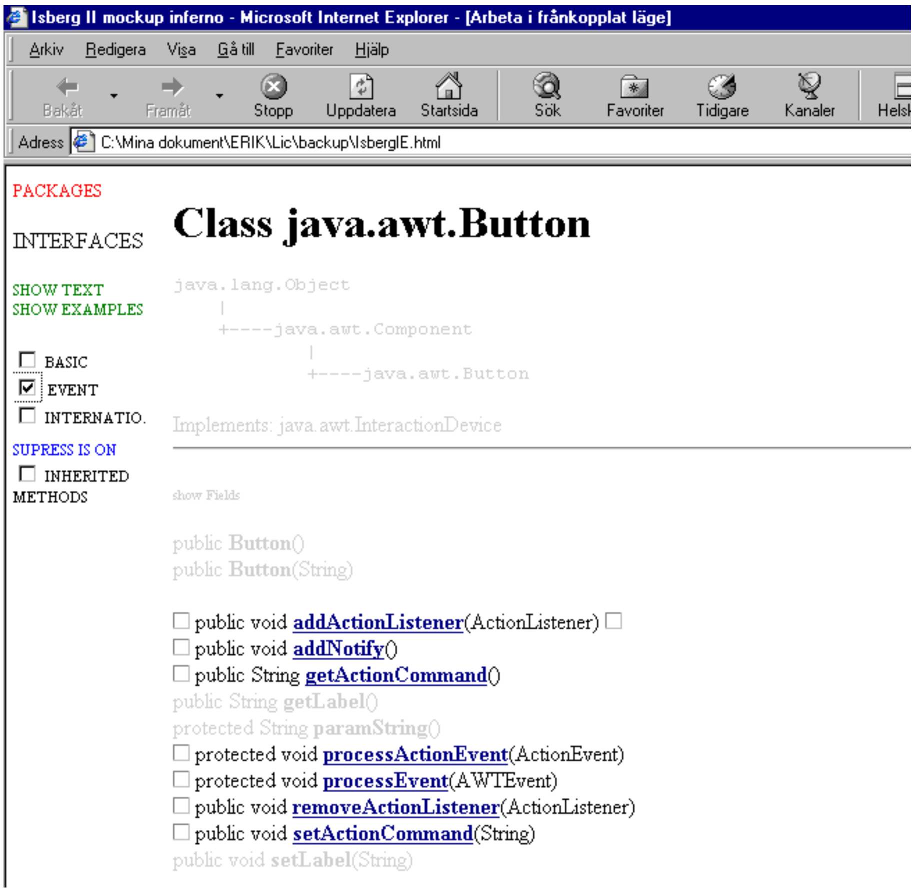
4.8.2 Pop-up Information Hiding (first DHTML)

Pop-up frames were used in the first Javadoc-related DHTML generation that started the DJavadoc project. In the first generation we 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 we 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.

4.8.3 Conceptual Filtering

Information filtering based on conceptual grouping of class members was the topic for the second DHTML generation. We 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 we 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). We enabled this both by graying out methods (i.e., changing their 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 4.20 shows the resulting mockup from this generation.



Class java.awt.Button

```
java.lang.Object
   |
   +----java.awt.Component
           |
           +----java.awt.Button
```

Implements: java.awt.InteractionDevice

show Fields

public Button()
public Button(String)

☐ public void addActionListener(ActionListener) ☐
☐ public void addNotify()
☐ public String getActionCommand()
public String getLabel()
protected String paramString()
☐ protected void processActionEvent(ActionEvent)
☐ protected void processEvent(AWTEvent)
☐ public void removeActionListener(ActionListener)
☐ public void setActionCommand(String)
public void setLabel(String)

Figure 4.20: 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 as seen by programmers, which we 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 JDK. 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, we still believe that conceptual filtering of class members is a good way to reduce the overhead of reading API reference documentation.

4.8.4 Scaling the Final Implementation

When we scaled the final implementation to JDK 1.2 size we ran into performance problems with the table of contents. The DJavadoc table of contents is not a very large HTML-file. For JDK 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 we 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, we felt that the performance had to be improved. Also, for larger class libraries, for instance future versions of JDK, the problem would perhaps grow.

This problem was solved by storing chunks of HTML in comments as unparsed text and extracting them on demand. 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 we were able to reduce each class list to one HTML element in the parser's perspective. On demand we 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.

4.9 Base Technology

So far we have used a mixture of Java and DHMTL as the base technologies in the DJavadoc project. The DJavadoc Doclet is a Java program that defines the output of a Javadoc session. The program delivers API reference documentation in DHTML. The dynamic typography and content are defined in the Java program, but only as strings (of HTML and JavaScript) and not in Java.

For the continuation of the project we are contemplating a transition from

Java to XML. XML is a meta-language originating from the SGML meta-language (in which HTML is defined). Currently there is a strong push for XML in the Web community and for applications in general. Internet Explorer 5 has highly developed support for XML [64]. A transition to XML would make our support more general. We would not leave the Java domain and could come back to a Java platform when Java develops its support for XML to a greater extent (see [59]).

For the Doclet editor discussed in Section 7.6 both Java and XML are plausible base technologies. A DJavadoc scripting language can perhaps be devised by analyzing the model of XML documents and by extracting information from the XSL stylesheet specifications ([81]). A negative point for the XML-Web solution is the separation of browsers and the local computer (a well-motivated Web-security decision). A Java solution is also plausible. It would be possible to work directly from the Doclet API by saving the Doclet API-object tree (even more so if the Java compiler was extended to include Javadoc text in compiled class files). Java scripting is possible, for instance trough the use of JPython [63]. Generalizing outside the Java domain for Java meta-solutions requires the construction of APIs for different information sources (such as the Doclet API) which can then be used as the underlying information structure for the construction of content scripting. What is negative for the Java solution is the lack of a highly developed document surface such as the browser. Also, compared to Web-technologies Java lacks high-level textual widgets.

4.10 DJavadoc Improvements

4.10.1 Using Dynamic typography

In addition to collapse and expand functionality there are many examples of how a dynamic typography can be used to change the presentation and achieve multiple views of the same information. Currently class documents describe the individual components and their links to related components. We introduced user-controlled views of information based on the underlying information model. In essence, we used dynamic typography to control the visibility of information. Besides the collapse and expand functionality it could be of interest to gray-out, to move around, to enter parts into layers, and so on. Furthermore, typographical measures such as font size and color could be changed dynamically to increase the visibility of different information types. For instance, when member summaries are collapsed, the font size of the member names could be reduced to lift more of the information into the visible space of the browser.

Using dynamic typography, we could also enable brief glances at class documents without leaving the current context. By moving the descriptions into the class-member-summary list (and hiding them) we were able to present descriptions without performing hyper-jumps (unlike the official Javadoc generated documentation). Similarly we can reduce the number of hyper-jumps among class documents by downloading short versions of class documents and displaying them on command in a layered text (much like tool-tips text). The layered text would enable quick glances at the core of class documents without actually jumping to other class documents. The concentrated views do not require additional, redundant files but could be extracted from the original class document.

4.10.2 Class Documents

Changing the static typography of the class document is one way of experimenting with more efficient class documents. To achieve similarity with the Standard Doclet we restrained the DJavadoc project from changes to the static typography. However, changes to the static typography can have impact on the class document. In our view, more use-oriented typography could be achieved. For instance, we would place the method summary on top (it is currently found 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 sub-list methods with the same name but different parameters under the one of these method that has the smallest number of parameters.

Altering the contents of the class document is another way of developing effective API 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 API reference documentation.

The conceptual-filtering direction, see Section 4.8.3, should be further developed as a way of reducing the information. During the conceptual-filtering generation of DJavadoc, we 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 alreade declared a number of new tags that we find highly relevant [68]. In our experience, in several cases 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.

4.10.3 Indices

We view the DJavadoc Bookmark described in Section 4.3.5 as one example of several applications of interest for information filtering on a navigational level. The DJavadoc bookmarks are one way of filtering parts of the table of content into an index. However, it requires much activity on the part of the reader. We use it to illustrate the dynamic-layout dimension on an index level or even on a Web-site level. There might be a need for several indices such as the bookmark index found in `My-map`.

Application indices that point to a group of classes that are useful for particular applications profiles could be useful. For different types of applications (e.g, client-server applications and database applications) different parts of, for instance, JDK 1.2 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). Application indices could be designed by proficient Java programmers.

A history index is another useful navigation device. By tracking the reader'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 developed to balance frequency and degree of recentness.

A context index that draws its entries from parsed source files is a third relevant example of how a filtered index could be achieved. The context in which the programmer is currently working provides relevant information for a filtration effort. Ultimately, by coupling the editor and the API 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. Preferably, we would like to see a strict client solution.

A use-based index could describe how the JDK classes are used. How classes are used in Java programming is another source of information for filtered indices. 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 [52]. A third statistical source could be online-tutorials such as the Java Online Tutorial [55] 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 in his or her own right).

Project indices could also prove valuable. According to some heuristic, the joint use of Java classes in a project group working on a common task could be assembled into a project index (one-man projects as well). 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 within that group. 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. The indices discussed here aim to perform filtering to such a degree that the bulk of classes are removed (at least for libraries similar in size to JDK 1.2). 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. For inherited methods, in some cases, it could be relevant to order the methods under the class which they were referenced from (not always the implementing class).

4.10.4 Using layered Text

In DJavadoc we have used tool-tips to display the full name of classes. The full class names (including the package prefix) was hidden in tool-tips to achieve cleaner text. In our experience, class documents become hard to read if full names are presented, particularly parameter lists. The tool-tip conveniently provides access to information that may be of interest but at the reader's discretion. The use of layered text, such as tool-tips, could be much more diverse in the DJavadoc-generated documentation.

Tool-tip texts, or layered text, is becoming a common form of dynamic-layout on the Web. Farkas [5] discusses the use of layers as safety nets in minimalist documents and provides several examples of such use. Supplementary texts are hidden in layers that can be accessed by the reader. The tool-tip does not change the underlying document, but places a frame of text above the current document. In online help layered text is commonly used. On the Web, the tool-tip has mainly been used to present a text on top of images. In Internet Explorer 4 most HTML elements can have tool-tip texts presented over them and the use of layering may therefore increase. Also, both in Internet Explorer 4 and Netscape 4 (and later versions) it is possible to design layered text components.

In DJavadoc layers could be used to a much higher degree. In our opinion, layered text is useful in the design of concentrated yet diverse and information-rich documents. By designing a number of layers for different information types we can perhaps achieve the same type of views as we did using collapse and expand functionality. Without permanently changing the page we could glance at the collapsed description text. In addition, layers could be used as windows to other documents allowing us to peek at them without leaving our current context. For instance, when a class-document hyperlink was pointed to, we could display a method list in a layer.

If layers were considered as a whole in the DJavadoc project it would perhaps be best to completely rethink the typography in terms of a dynamic, 3D compilation of texts. The use of layers in the API reference documentation will completely redefine the premises for typography. A more use-centered typography could be devised and should preferably be the result of user studies. However, simply designing a layered version could be a starting point for such a study in order to provoke reposes from the Java community.

In the development towards a layered DJavadoc it is important to consider that users, in the long run, want simple but efficient solutions rather than appealing wonders of technology. Even though layers hold fascinating typographical possibilities (along with other features of the computer-reading environment), they should be handled with care. To achieve relevant alternatives to printed media the wonders of technology must be put to proper use. As a result it is important to perform user studies, both to evaluate and to form a basis for development.

4.11 Summary

DJavadoc adds dynamic typography to Javadoc to enable user-controlled views of the Java API reference documentation In DJavadoc the reader can redesign in real-time the API reference documentation. As a result, views of the API source code that are more in line with different readers' needs can be created (and recreated as the needs change).

Chapter 5

Preliminary Studies

A small study of the DJavadoc project is discussed in this chapter. We elaborate on the goal of the studies. We also present preliminary results that have been reached so far. The continuation of the study is outlined. The pros and cons of these study methods are not discussed here but in chapter 3.

5.1 Study Goal

Our study goal is to contribute to the understanding of how computer-reading environments should be designed to support reading of API reference documentation. Exemplified by Java programmers and their Javadoc-generated API reference documentation, our underlying goal is to understand how to support the task of computer-based reading of work-related, homogenous texts. In part we want to evaluate the technological support that we have developed. Furthermore, we want to perform more general inquiries to understand Java programmers' needs to form a basis for further development. Particularly, we want to learn how to support the task by adding new technology in useful ways.

To reach innovative and relevant support at the end of the DJavadoc project we will continue the development guided by real-user input. The evaluation of the implementation should be viewed as a component in the development and in the process of acquiring real-user views on the project. Minutely determining the value of the dynamic-typography features implemented in DJavadoc is beyond the scope of this work. It would be a time-consuming activity with relatively limited benefit for our research goal. We also believe that the preliminary results we have achieved support our ideas.

5.2 Preliminary Results

So far we have achieved preliminary results as part of the task of trying to achieve study opportunities. Acquiring access to the real world is a time-consuming activity. According to Gummesson [10], access is one of the major issues in qualitative research. As part of this process we have had informants viewing the DJavadoc implementation. Their comments, based on a demonstration of functionality, form what we call preliminary studies.

At a glance, there has been a positive response to the ideas implemented in DJavadoc by informants that we have had contact with. As part of the contact-making process 15 programmers and program managers, and 7 technical writers have looked at the DJavadoc-generated documentation and commented positively. However, the comments have been shallow and perhaps also overly positive as part of their social interaction with us. Two contacts have been made on the informants' initiative: one from England and one from the USA. What was most appreciated was the possibility of collapsing and expanding the method description. One informant expressed the desire to continue expansion down to the source-level; thus exposing the inner workings of the methods. One informant commented on the table-of-contents as being of little importance because the informant used a development environment that enabled navigation directly from the code (see Section 6.3).

Other than our personal use of the DJavadoc-generated documentation, no real experience has been gained. Personally, we have experienced the DJavadoc-generated documentation in action. In programming we have found it valuable to collapse and expand method descriptions and also to change the default setting as our interests change. In our experience, several default settings are relevant, which support the idea of a user-controlled dynamic typography as opposed to a new static typography. Also, for a programming project the ability to create bookmarks of relevant classes was beneficial. However, these findings are personal and cannot be considered valid results.

From these preliminary results we can only conclude that there has been some interest in the user-controlled views of information implemented in DJavadoc. Our initial contacts have pointed to the possible usefulness of DJavadoc in relation to the Standard Doclet (see Section 2.1.5). We may conclude that there is a range of dynamic-typography functionality that may be useful for programmers reading API reference documentation. However, we cannot conclude that the functionality is genuinely useful and not just appealing.

In the process of acquiring real-world contacts we have learned lessons that other may find relevant. Setting up real-world study opportunities is a time-consuming activity. It is desirable that studies are as non-intrusive as possible because commercial partners can be pressed for time and have limited resources for external activity. It is also important to be precise about goals, costs, and benefits in the early stages of cooperation. Here, DJavadoc has served as a concrete demonstration of design principles. However, using a particular platform, such as Microsoft Internet Explorer, has also been a hindrance in some cases since companies can have policies about the software they use.

5.3 Future Studies

5.3.1 Setting Up Real-World Studies

It is our desire to perform studies in real-world settings. We have established contacts with commercial organizations that can provide us with DJavadoc testing for real use in real-world situations. These study opportunities are still being developed.

It is possible for us to gain access to informants all over the world through the World Wide Web but currently we see it as a supplementary source. We have already made contact through the WWW with a USA cooperation which has started experimenting and using our API reference documentation. However, we acknowledge the difficulties of handling this relation and consider studies with them as supplementary.

5.3.2 Character of the Informants

The desired type of informants are professional developers because we assume they are interested in use-oriented support. Ideally, we want to collaborate with experienced professionals who have used Java for some years. However, finding several experienced professionals may require a distribution over several work places since Java has only been around since 1995 [74]. Generally, our desire to work with professional developers is a straightforward consequence of the fact that we are focusing on support for them. Our hope is to gain access to a number of commercial groups of programmers and to gather data from interaction with them, preferably to have deeper relations to one or two groups.

5.3.3 Study Procedure

We believe it is possible to determine whether DJavadoc genuinely improves the design found in the Standard Doclet (see, Section 2.1.5). By studying DJavadoc in use we can determine if the implemented extensions in DJavadoc provide genuine value. Our plan is to perform such studies in real-world settings and for this purpose we have started setting up study opportunities. It is from these activities our preliminary studies are drawn.

For the study of DJavadoc it would be possible to determine value by logging system data. A relevant measure is the use of the default settings. If users continuously change their default setting, it may indicate that the type of flexibility that DJavadoc introduces has genuine value and that it is not feasible to aim for one form of content and typography of the API reference documentation. In particular, if different readers use the default settings differently the need for flexibility is supported. Another appropriate measurement is how much scrolling DJavadoc removes, if any. Such measurements would require a control group for the scrolling behavior of the official Java API reference documentation. The hypothesis underlying the scrolling measurement is that DJavadoc moves relevant information up into the visible space of the browser. Thus, DJavadoc users should scroll less than Javadoc users.

Such quantitative measurements should also be complemented with small qualitative studies. By asking the users of DJavadoc if they find the support it provides rewarding and why, we can supplement the quantitative measurements. The studies can be performed either by interview or by having informants fill out forms.

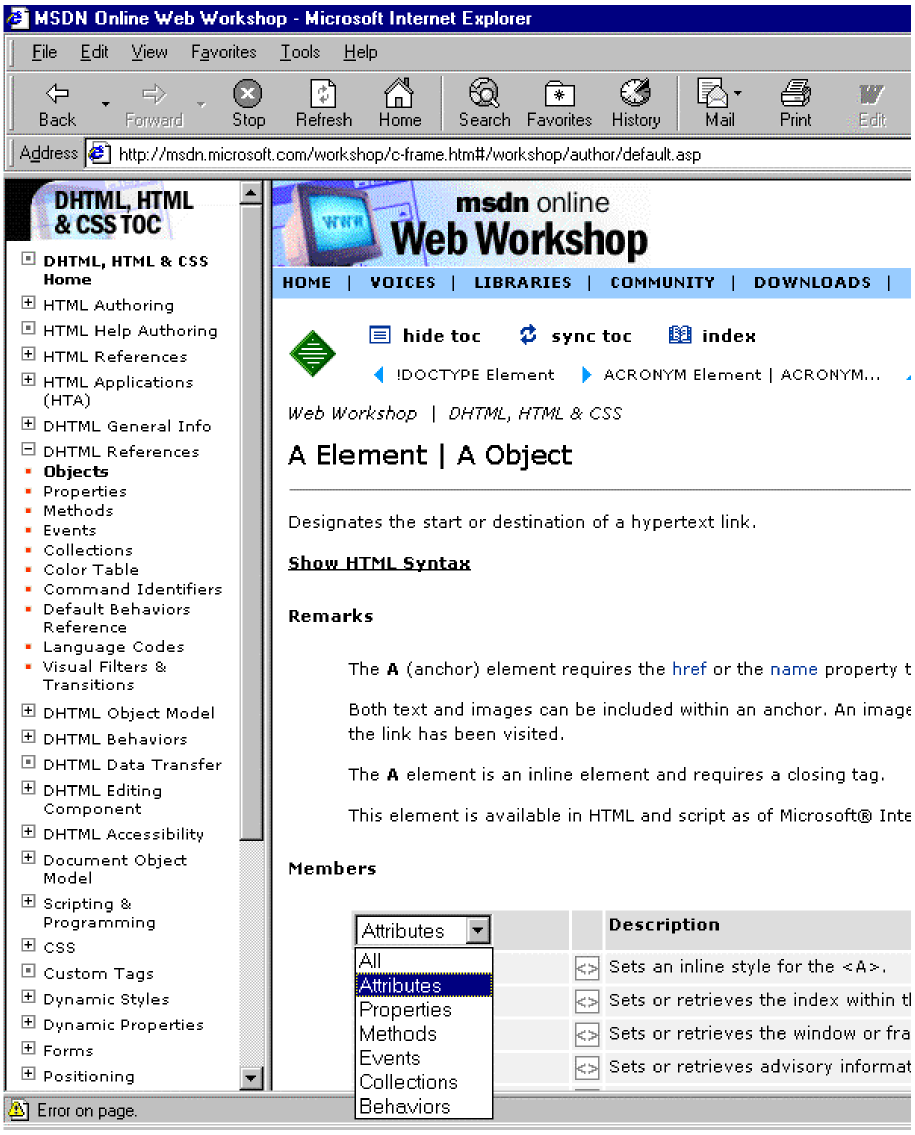
Chapter 6

Related Work

In this chapter we present systems and documentation related to the DJavadoc project. In our studies we have focused on computer-reading environments for API reference documentation of programming languages and application-programming interfaces (APIs). Javadoc is, of course, related to the DJavadoc project but since it is discussed throughout the thesis we do not address it in this chapter. The related work is not described in full but rather contrasted with Djavadoc. For explanations of terms see chapter 2.

6.1 MSDN Online Workshop

The *MSDN Online Workshop* is the Microsoft Developers Network workshop for Web-developers available on Internet which we consider to represent the state of the art in relation to DJavadoc [64]. At the MSDN Online Workshop readers are presented with a mixture of general articles and API reference documentation about software components available in applications such as the Internet Explorer browser. A concrete example of such API reference documentation is the section on DHTML from which the screen-shot presented in Figure 6.1 is taken.


MSDN Online Web Workshop - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address
DHTML, HTML & CSS TOC
msdn online Web Workshop
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS |
hide toc sync toc index
!DOCTYPE Element ACRONYM Element | ACRONYM...
Attributes
All
Attributes
Properties
Methods
Events
Collections
Behaviors
Error on page.

http://msdn.microsoft.com/workshop/c-frame.htm#/workshop/author/default.asp

DHTML, HTML & CSS Home
HTML Authoring
HTML Help Authoring
HTML References
HTML Applications (HTA)
DHTML General Info
DHTML References
- Objects
- Properties
- Methods
- Events
- Collections
- Color Table
- Command Identifiers
- Default Behaviors Reference
- Language Codes
- Visual Filters & Transitions

DHTML Object Model
DHTML Behaviors
DHTML Data Transfer
DHTML Editing Component
DHTML Accessibility
Document Object Model
Scripting & Programming
CSS
Custom Tags
Dynamic Styles
Dynamic Properties
Forms
Positioning

Web Workshop | *DHTML, HTML & CSS*

A Element | A Object

Designates the start or destination of a hypertext link.

Show HTML Syntax

Remarks

The **A** (anchor) element requires the href or the name property t

Both text and images can be included within an anchor. An image
the link has been visited.

The **A** element is an inline element and requires a closing tag.

This element is available in HTML and script as of Microsoft® Inte

Members

Attributes		Description
	<>	Sets an inline style for the <A>.
	<>	Sets or retrieves the index within tl
	<>	Sets or retrieves the window or fra
	<>	Sets or retrieves advisory informat

Figure 6.1: In the Microsoft Developers Online Workshop, API reference documentation for Microsoft Web-technologies is presented using the same DHTML technologies as in DJavadoc. However, the collapse and expand functionality is predefined and readers therefore must continuously reorganize the pages as they are loaded.

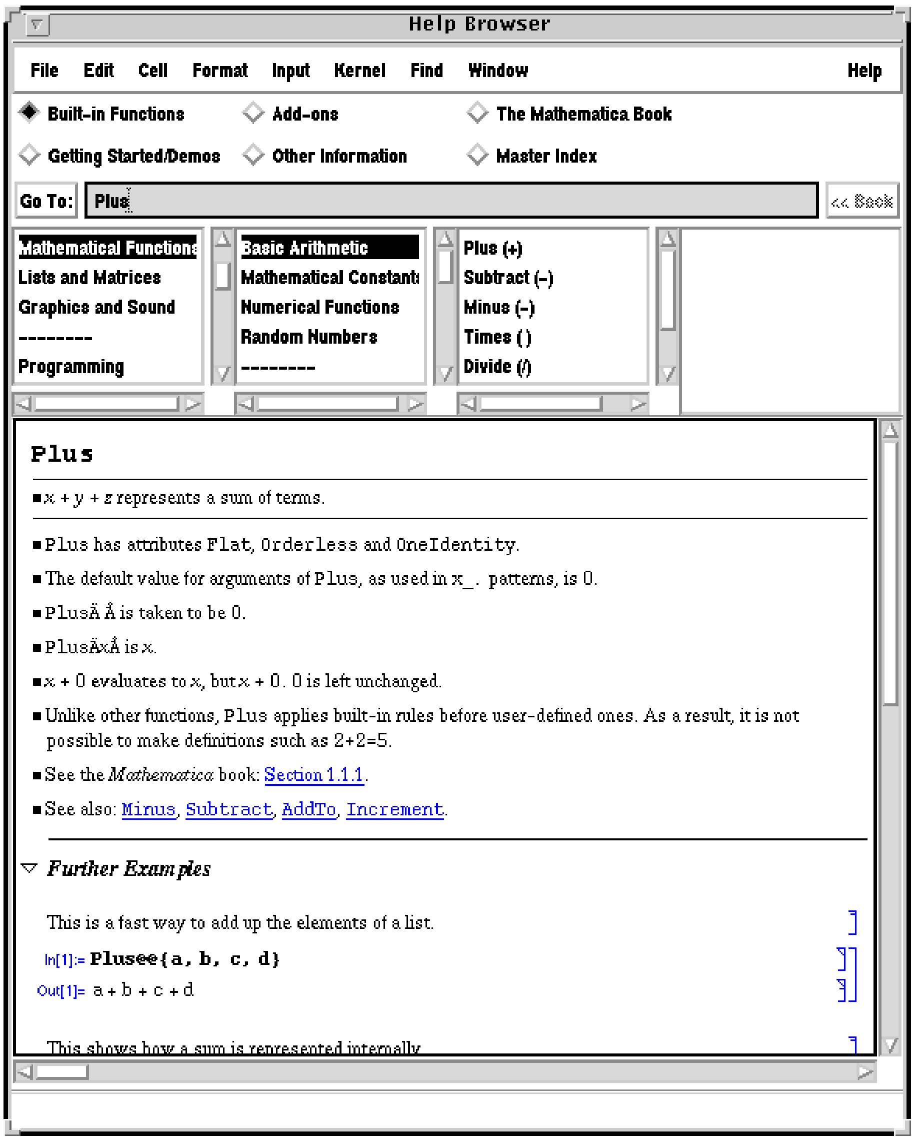
MSDN Online Workshop API reference documentation uses dynamic typography in similar ways to DJavadoc. In the MSDN API reference documentation components are presented using collapse and expand functionality. The table of contents is presented as a collapsible list. Individual component documents also contain collapsed and expanded information. Figure 6.1 describes an HTML element. The syntax of the HTML element is hidden and can be expanded. The HTML element also has methods, properties and so on (being a DHTML object in the Internet Explorer browser). Different lists of these members (of the HTML element) can be presented to provide multiple views of the functionality. The style properties of the HTML element are also collapsed in a separate list (not visible in Figure 6.1).

However, unlike DJavadoc, MSDN does not allow users to control the visibility of information types by setting default views for the information. The dynamics of MSDN are set to a static default. As a result, readers continuously have to reorganize homogenous documents to access information as they browse the documentation. Sometimes the right information may even be hidden from the readers. This is an important impediment since the MSDN API reference documentation is read by continuously jumping among different components. Furthermore, MSDN does not expand member descriptions but instead loads a new page, which from our stand-point is a negative feature in MSDN but which also may be a domain-specific design choice.

6.2 Mathematica Help Browser

In the *Mathematica Help Browser* the Mathematica language is described. The Mathematica environment contains a Mathematica kernel (a runtime environment) and Mathematica front-ends. The *Notebook* is a graphical front-end to Mathematica which combines Mathematica input and output with texts, graphics, and so on. Notebooks are interactive documents that have access to the full capacity of the Mathematica language. The Mathematica Help Browser is an example of the type of interactive documents that can be built using the Mathematica Notebook. The Help Browser is hypertext-based API reference documentation of the Mathematica language which describes all Mathematica functions (see [22]).

The Help Browser collapses and expands example sections, see Figure 6.2. In the Help Browser, functions are presented individually with a description and a section of usage examples. The usage examples are collapsed by default and can be expanded by the reader.



Plus

■ x + y + z represents a sum of terms.

■ Plus has attributes Flat, Orderless and OneIdentity.

■ The default value for arguments of Plus, as used in x_. patterns, is 0.

■ PlusÄ Å is taken to be 0.

■ PlusÄxÅ is x.

■ x + 0 evaluates to x, but x + 0. 0 is left unchanged.

■ Unlike other functions, Plus applies built-in rules before user-defined ones. As a result, it is not possible to make definitions such as 2+2=5.

■ See the *Mathematica* book: Section 1.1.1.

■ See also: Minus, Subtract, AddTo, Increment.

Further Examples

This is a fast way to add up the elements of a list.

```
In[1]:= Plusee{a, b, c, d}
Out[1]= a + b + c + d
```

This shows how a sum is represented internally

Figure 6.2: The Mathematica Help Browser expands and collapses example sections. Furthermore, the examples presented can be executed and the result is presented.

The examples can also be executed in the Help Browser. The result is presented in the browser and the reader can thereby test the components described in the API reference documentation.

In DJavadoc the same type of component testing could be achieved, in particular since Java is a Web-enabled language in browsers such as Netscape and Internet Explorer. The methods of a class could be listed and executed on an object of that class. The result would project itself on the object and in some cases it would have a graphical result.

Though intriguing, this kind of support can be difficult to automate for all JDK classes and in some cases it would require visualization. Graphical widgets could easily be displayed and prepared for manipulation. The reader could test the effects of method calls on the objects and see the result. However, for a large part of the method additional objects are required as parameters, which would increase the complexity. Sometimes return values may also be relatively abstract objects that in turn require visualization. Furthermore, classes may not produce easily presentable results (for instance streams, files, URLs, Data Models) and therefore require visualization. Another complication for execution support in DJavadoc is the fact that several classes are often needed to produce a relevant composite component.

Automatizing the creation of test execution is possible but the Doclet API would have to supply more information and therefore the Javadoc tagging mechanism should be extended. It is possible in DJavadoc to automate test execution for part of JDK 1.2 but it requires manual tagging of how classes can be tested. Such tagging should be added Javadoc tagging convention and be delivered by the Doclet API. The original developers have the expert knowledge needed to decide how classes should be presented and which methods are relevant. The original developer may also have to supply test programs that set up graphical simulations that can be manipulated.

6.3 Development Environments

Visual Age is an example of a development environment that also represents a computer-reading environment, see [76]. IBM delivers a development environment for programming directed at supporting the programming task. In the Java version JDK source code is included and the environment therefore becomes a computer-reading environment for components used in programming. JDK source is treated as code in general and presented in the same way as code under development. Source code is reorganized into projects, classes, and interfaces that in turn contain methods.

Compared to the Standard Doclet and DJavadoc, Visual Age presents the source as raw code and not as documentation of that code. Visual Age does not generate documentation of the source. It is less similar to the book format and does not take advantage of the range of typography features that can help present information.

We find some features of Visual Age relevant for computer-reading environments of API reference documentation. In Visual Age it is easier to copy and paste because the editor is also a part of the reading environment. Furthermore, Visual Age enables the reader to filter methods based on their public modifier (public methods are the methods most developers are intended to use). Methods are made more visible than other members of a class which in our opinion represents a use-oriented design choice.

Another example of a development environment acting as a reading environment is *Visual Cafe*, see [77]. Visual Cafe is a development environment developed by Symantec. The environment incorporates Javadoc documentation and is therefore also a computer-reading environment. Similarly to Visual Age, Visual Cafe organizes code into projects and classes.

Visual Cafe delivers more use-oriented navigation than does Javadoc and DJavadoc. In Visual Cafe it is possible to access Javadoc documentation from the code. By selecting a class name in the code the reader can access the documentation. Navigation, to some extent, originates from the code and not from within the hypertext documentation. By integrating navigation and the code a use-oriented navigation mechanism is achieved. The classes used in the code are perhaps likely to be more frequently accessed. Achieving use-oriented navigation is particularly interesting for large information repositories such as Javadoc-generated documentation. We find the coupling of the source code and the documentation particularly interesting, both for inspection of the programmer's own code and that of others.

Doc++ is a Javadoc authoring and editing tool that is also integrated in JBuilder. Doc++ is a tool delivered by WoodenChair Software as part of their Utility+ product line of which part is prepared for integration with the JBuilder development environment from Borland [61], and also Visual Cafe, and Visual Age [80]. Doc++ is a Javadoc application that enables graphical handling of tags and also spell checking. We find the Doc++ application relevant to the DJavadoc project because it is an advanced Javadoc application. However, to our knowledge it does not include any features of dynamic typography.

Other development environments also contain the features we have described in this section. We choose to point to specific development environments as representatives of a group of systems that have relevant functionality for the DJavadoc project. Other development environments also include these features but we will not discuss them individually. Examples of other development environments for Java include Microsoft's Visual J++ [78], and Java's experimental Java Workshop [56].

6.4 Hypertext Reference Manuals

Several programming languages have hypertext-based reference manuals available online. Most programming languages have online API reference documentation available on Internet. Java [62], Ada [42], Perl [67], Python [70], and so on. The use of hyperlinks is, to our knowledge, seldom exploited to a higher degree than in the Standard Doclet.

6.5 WEB

Literate programming, of which Javadoc is an example, was introduced by Knuth and implemented in the WEB-system for Pascal [6] and also for C and C++ in the CWEB version [37]. The idea of literary programming was to write code and documentation simultaneously in one format and then separate the two into source code and documentation. A vision of writing programs as essays was presented. The full vision of literate programming is perhaps more romantic that useful. In relation to WEB, we believe that Javadoc requires less work, handles change in source-code better, and is less dependent upon the efforts of the individual programmer because of the higher degree of automation and standardization. In our view, programming should contain a minimum of documentation to reduce the risk of faulty descriptions and to reduce maintenance work. Documentation should preferably not describe the code but rather make the code more readable. In this view documentation takes the role of an outline and is written to support a code-reading task. Javadoc defines a limited form of literate programming.

In WEB programmers have full control over both content and typesetting of the API reference documentation. Programmers write WEB in TEX-format and therefore have complete control over the generated documentation, both content and typesetting of the documentation (even though the TEX-system separates the typesetting and content). In Javadoc the programmer must write a new Doclet to change the typesetting but with the same control, in principle. However, developers do not generally design new Doclets because it is a major programming effort. To allow for greater control of content and typesetting more tags should be introduced into Javadoc or a meta-tag language should be developed to allow for dynamic-incorporation of tags.

6.6 Emacs Info System

An interesting feature in the info reader of Emacs [48] is the use of both command-line interface and direct-manipulation interface. Texinfo is a hypertext system commonly read in Emacs and often used for technical documentation [49]. The use of both command-line interface and direct-manipulation interface is relevant in relation to the DJavadoc project. In many situations, programmers know which class they are looking for and a command-line navigation tool could therefore increase the speed with which they navigate. Also, programmers are perhaps tolerant towards command-line interfaces because they often have previous experience of using such systems.

6.7 Documentation Function in LISP

In Common LISP the documentation of a function can be accessed online [23]. In Common LISP there is a convention to include a documentation string in function specifications to describe the function. This function also exists in other programming languages and systems. In the interactive LISP environment the documentation function can be used to produce the documentation string online.

Similarly, a connection between the API reference documentation and Java editors could be achieved to provide explanations of classes, methods, and so on interactively. In the DJavadoc project we have looked at the API reference documentation and the editor as separate entities. However, whether a separation is the best solution is left undecided. In any case, the editor (or rather the Java files under development) contains information about the reader's context that is of importance to the reading environment. Getting closer to the reader's needs is a key goal for the DJavadoc project. The editor represents a much more use-oriented browser than do the Standard Doclet and DJavadoc. The editor can become a portal into the API reference documentation, given that the editor and the reading environment are integrated. In a sense, the documentation function in Common LISP can be viewed as a means of navigation (in the context of a computer-reading environment).

6.8 Summary of Related Work

DJavadoc provides control over the presented material on the basis of the underlying information model unlike the systems presented here. Though MSDN Online Workshop and the Mathematica Help Browser enable collapse and expand functionality to some extent, the reader is not in control of the visibility of the entire information set. The MSDN Online Workshop API reference documentation is the system most silimar to DJavadoc API reference documentation. In our view, MSDN Online Workshop also represents the state of the art in computer-reading environments for API reference documentation. However, unlike DJavadoc, the MSDN API reference documentation does not allow for views controlled by the user and therefore often presents excessive information and pushes relevant information out of the visible space of the browser. Hence we believe that for DJavadoc-type API reference documentation DJavadoc control adds substantial value.

The systems presented in relation to DJavadoc can be divided into computer-reading environments, systems with relevant navigational features, and systems for combining programming and documentation. The Mathematica Help Browser, the MSDN Online Workshop, the hypertext reference manuals, and DJavadoc are all examples of computer-reading environments used by programmers in the development process. Development environments, to some degree, also fit into this category. The systems presented in this chapter based on their navigational features are the LISP documentation function and the Emacs Info System. Development environments also fit into this category, for instance Visual Cafe. WEB and also Javadoc are systems for combining programming and documentation.

Chapter 7

Discussion

In this chapter we discuss the results of the DJavadoc project. We discuss the Javadoc approach, how implications for object orientation can be discovered, and how we would like Javadoc to evolve. Further we identify and discuss requirements and generalizations that can be made. We discuss dynamic typography as a concept. Finally we summarize with a view of the future. The terms and technologies we refer to are discussed in chapter 2.

7.1 The Javadoc Approach

Depending upon perspective Javadoc and Java API reference documentation can be seen as different things. Though Javadoc delivers documentation in various forms, Javadoc can be viewed as several different types of systems, as discussed in the following.

The most common view is probably that Javadoc is a documentation system. Javadoc is an automated tool for Java source code documentation. As such it can be compared to Rational's SoDA system that generates reports from Rational's tools [72]. However, Javadoc describes only the details of the source code since the tagging structure does not currently support higher-level comments.

Javadoc can also be seen as a specification system, a principle discussed by Douglas Kramer from the Javadoc Team in [9]. At Sun the decision was made to put the specification of Java APIs in the source code as tagged comments. Since Sun was not in control of all specifications, some had to be referenced via hyperlinks in the comments.

An alternative view of the output of Javadoc is a computer-reading environment for source code. The official API reference documentation is, to a great extent, a typeset view of the source code. Thus, Javadoc delivers increased visibility of relevant parts of the source code. The API reference documentation is the graphical user interface of the API. (Note that in computer science a product may be equivalent to its documentation, unlike other engineering areas.) In this view, the role of Javadoc is to enable more efficient acquisition of knowledge from the source code.

Javadoc can also be seen as a programming tool used by developers to produce applications. Just like compilers, editors, debuggers and other programming tools the Javadoc API reference documentation is a programming tool. Actually it is, in our view, one of the most commonly used Java programming tools. Javadoc should not be considered as delivering documentation but rather information systems (i.e., applications). As such it should perhaps not only passively describe components but actively advocate the use of particular components for certain application profiles.

Furthermore, Javadoc can be seen as an application platform. The Javadoc structure, illustrated in Figure 2.4, is an application platform for the use of the information structure that Javadoc delivers. This structure was designed to allow for different forms of documentation and also provide the basis for a wider range of applications. For instance, editors could use the Javadoc structure to access information about classes and class members. For such purposes Java Reflection is often used, see [54]. However, via Reflection developer comments cannot be accessed.

Moreover, Javadoc-generated API reference documentation can be viewed as one of the major Java learning environments. Developers learn to program from tutorials, books, code examples, and so on. They also learn from the API reference documentation. How well the API reference documentation visualizes structures, underlying assumptions, design patterns, and so on have, in our view, an impact on the software programmers deliver. From this perspective, by making design choices in API reference documentation, we are in fact making choices about how programming should be performed. For instance, if a set of methods in a class is intended for external use and others for use within an internal structure, the API reference documentation should present these methods differently to support the original design. (This situation exists in the Swing packages in JDK, see [75], which provide functionality related to graphical widgets.)

7.2 Implications for Object Orientation

The Javadoc API reference documentation can unveil limitations in the Java programming language and ultimately object orientation as a programming paradigm. An important advantage of automatically generated documentation is that it maps directly to the source code (unlike manual descriptions which often deviate from the source code). However, automatically generated documentation can only reflect what the programming language can express. By examining what is difficult to understand in Java API reference documentation we may discover deficiencies in the Java programming language. Another source of limitations is the difference in content between manually written documentation (such as the tagged source-code comments) and automatically generated documentation. Key concepts of object orientation are challenged by the effects they have on the communication of functionality in the API reference documentation.

As an example, inheritance has led to a separation of class functionality into declared and inherited functionality which represents a systems-oriented organization but not a use-oriented organization. Inheritance leads to classes that, for instance, accumulate large sets of methods. For instance, the `java.awt.Button` class has 178 methods in JDK 1.2. It has 10 declared methods, 157 methods inherited from `java.awt.Component` and 11 from `java.lang.Object`. For absolute basic use there are 1 or 2 methods at `Button`-level that are relevant. However, beyond the absolute basics, such as changing the background, a few of the 157 methods at `Component`-level must be used. Most methods declared in `Button` are still not relevant to the user. (Another extreme example is the `javax.swing.JMenu` class that has 381 methods declared in the class or inherited from its 6 ancestors.) In the official Java API reference documentation, the declared and inherited methods are presented separately for practical reasons. However, as our `Button` example illustrates, the set of methods most commonly used is not likely to be the declared methods but rather a subset of methods from all levels in the hierarchy. In effect, relevant methods are presented as less important than other methods in the API reference documentation simply because these methods are inherited. Thus, we conclude that inheritance does not provide an efficient mechanism for separating functionality into use-related categories.

What is difficult to understand from documentation or difficult to document suffers from poor design (which is the point being made in this section). Being the graphical user interface of the API, reference documentation is the publication of Java classes and thus advocates the Java language and object orientation. In a sense, functionality in Java and object orientation that is not communicated well is irrelevant. Of course, deficiencies in the API reference documentation cannot automatically be contributed to the design of the programming language or the programming paradigm. For instance, problems with the Java API reference documentation may follow from insufficient analysis of the source code. However, if complex analysis must be performed to discover important information, this also reflects negatively on the programming language.

7.3 Potential Improvements to Javadoc

On a number of issues we have design requests for the continued development of Javadoc-generated API reference documentation. The DJavadoc project only focuses on a small aspect of the computer-reading environment and the official Javadoc API reference documentation. However, our work with DJavadoc and our experience as Java programmers has led us to certain conclusions.

Javadoc should use the source code directly instead of generating intermediate HTML format (i.e., Javadoc should provide a source-code browser). Javadoc generates documentation automatically to counteract the possible deviation of documentation and source code. However, the documentation is generated in HTML. To achieve a tighter coupling between the source code and the API reference documentation we suggest that a source code browser is developed that parses source code and presents Standard Doclet-type documents. However, such a browser would not remove the need to generate documentation in other formats. For instance, the current HTML format protects the integrity of the source code by only containing the signature of APIs.

The analysis that is performed by Javadoc on the source code should be developed with the aim of finding structural information such as key classes, key class members, and particular chains of methods. Some classes or class members play key roles in program structures such as the Java runtime. For example, `java.awt` graphical widgets must be added to a `java.awt.Container` object using one of the `java.awt.Container.add()` methods or else the widget will not appear on the screen. These methods are thus central to the graphical user interface structure. Also, there are chains of methods cutting across several classes to deliver a relevant result. For instance, the method chain `DriverManager.getConnection().createStatement().executeQuery()` delivers an object representing the answer to an SQL query, unlike the method chain `DriverManager.getConnetion().createStatement().close()` which is pointless. Analysis of the source code can perhaps uncover structural information of these types, representing knowledge difficult to acquire from the current API reference documentation.

The layout of the class documents should be designed for use to a higher degree. Heuristics about use could be applied to the layout of the API reference documentation to achieve a more use-oriented design. For instance, methods are central to the use of Java class and should therefore be placed as high up in the browser as possible. Another example is to group 'get' and 'set' methods together (get and set are a general naming convention for methods that manipulate data). Furthermore, methods with the same name but different parameters represent the same functionality and the space they take should be minimized.

The tags used to write source code comments should be further developed. Developers document their classes using tags. There are a series of tags that may become part of future Javadoc implementations [68]. The tags play an important role in the documentation process by advocating documentation practice. Ideally, these tags communicate a disposition of what quality documentation contains. We would like to see a continued development of the tags to further specify the desired content. One example is the description which could be modularized into a set of tags that separate general description, descriptions about the context of class or a class member, descriptions of version changes, and so on.

Furthermore, we would like to see a more dynamic tagging mechanism that allows for new tags to appear in the Javadoc even though they were not conceived of from the beginning. Currently the tags that the Standard Doclet handles are statically defined. The opposite of the current solution is a separate tag programming language that allows for any type of tag constructions. Such full-fledged dynamics are principally what the WEB system delivered (see Section 6.5). We believe in a dynamic tagging mechanism within reason. Javadoc should be more flexible to enable additions and also provide the basic set of tags that define quality documentation. We envision tag types being used instead of specific tags. These tags types represent the kind of information documentation should consist of. As an example the `@decriptionItem` tag could be used to provide a general tag for class-member-description items. The convention for the tag would be `@decriptionItem name text` and it would be placed into the member-description with the `name` as a headline and `text` as a paragraph under that headline. Javadoc would not need to know what developers put under `name`. Similarly such a dynamic tagging mechanism could enable dynamic DJavadoc filtering (e.g., `@filter name`).

We would also like to see command-based navigation in the API reference documentation. Scrolling down to the method summary is a tedious task. Pressing `Ctrl-M` would speed up this process. Several navigational issues could be dealt with through a command-base approach. Even though such command-based navigation is domain-dependent and therefore requires training, it would provide experienced users with a powerful navigational tool. Loading particular classes by simply writing the name is another example of how command-based navigation can be helpful (e.g., by typing `Ctrl-L java.lang.String`), particularly if the API reference documentation completed the name when possible.

7.4 Requirements

From our work on DJavadoc we have found some requirements on API reference documentation concerning the type of support DJavadoc delivers. DJavadoc focuses on only a part of the entire architecture of API reference documentation. However, for this part and from our work with DJavadoc we list some requirements on the computer-reading environment for API reference documentation.

The dynamic changes in the documents (i.e., the collapse and expand functionality) must be instantaneous. The idea behind DJavadoc is that the reader continuously manipulates the documentation because what is relevant information changes as the reader proceeds. In view of this scenario, if performance is slow it severely handicaps the reader. For a while we had performance problems in the table of contents and waiting for the collapse and expand functionality to take a few seconds seemed very long. The need for seemingly instantaneous performance makes client-side solutions more feasible than server-side solutions (dynamic generation on the server side is often preferred to overcome the shortcomings of older web browsers).

Using collapse and expand functionality seems to work best for smaller sections that do not push all of the following text out of the visible space of the browser. As sections are expanded they push information down and ultimately out of the visible space of the browser. In our experience, it can be somewhat disorienting if the all the information below a section is pushed out of the visible space.

It is not feasible to envision one design for API reference documentation that will please everyone. From my contacts with programmers, both students and in industry, it is clear that people want different things from the API reference documentation. Different persons want different types of information, some written descriptions and others source code examples. The knowledge they are after is the same; they just want it presented in different ways. API reference documentation should therefore support multiple views, for instance in the way DJavadoc does.

Ultimately programmers must provide the source code with structural information alongside the source code descriptions to enable the type of API reference documentation we envision. The ability to collapse and expand parts of the documentation is useful. In particular, the ability to collapse or expand individual elements by direct manipulation is important. The functionality also scales well but is, of course, dependent on the quality of the underlying information structure. In some cases the structural information must be provided by developers because the programming language does not capture this knowledge. Thus, documenting becomes the task of both describing and providing filtering information used by the computer-reading environment. The proposed Javadoc tags include some tags related to this (for instance the `@category` tag) [68].

7.5 Task Generalization of DJavadoc

Java programmers read to do or to learn to do (see Schriver [18]), that is with the purpose of learning to perform or of executing some type of action. Software development requires detail knowledge about the components programmers reuse. Several other work categories require the same type of knowledge. For instance, engineering in general is often based on the reuse of predefined components. Physicians, too, are required to read patient records during work. There are numerous work categories that continuously read to do or to learn to do.

The text Java programmers work with represents a homogenous, structured, work-related text. The Java API reference documentation can be characterized as a component catalogue or as a set of reports containing the same type of information. A characteristic of this text is also its integral relation to work tasks and the need for efficient acquisition of knowledge through reading.

In principle our solution relies only on an underlying information model, even though we currently have a technical solution restricted to the Java-domain. The support we have built for Javadoc-generated API reference documentation is general in nature. In a sense, DJavadoc extends the computer-reading environment rather than the API reference documentation. In the current DJavadoc implementation this generality is hard coded to Java. However, a continued development could achieve support for structured information sources in general.

7.6 Doclet Editor

Another generalization step that we would find relevant makes DJavadoc a Doclet editor, providing the reader with the freedom to define the content and typography of Java API reference documentation. The official Javadoc-generated API reference documentation is currently set in its ways. Changes to the API reference documentation can be achieved by creating a new Doclet. However, developing Doclets such as the Standard Doclet is a major programming task (more than 11,000 lines of code in the 1.2.2 version). However, a Doclet could be both API reference documentation and a Doclet editor. Others have worked with tools of this kind, see for instance [33].

In short, a Doclet editor can simplify the design of higher specialized and more individual versions. The development of editor support can reduce the effort needed to define alternative Doclets. It can also increase the possibility of rapid reference-documentation prototyping. Furthermore, it can also increase the speed with which new version of the Standard Doclet can be developed (even though individual programmers never use the tool). The lack of a Doclet editor may be a reason for the fact that there are not many alternative Doclet implementations available of the size of the Standard Doclet.

Both a Doclet scripting language and a graphical Doclet editor are plausible. Since our readers have a background in computer science, they can handle programming-based development. The content could perhaps be defined as a mapping of the methods found in the Doclet API. However, for the typography it is not equally clear how to construct a simple but powerful scripting language. A graphical Doclet editor would handle the typography by providing a graphical-user-interface editor.

DJavadoc can be viewed as a client-side Doclet editor. In DJavadoc we enable client-side manipulation of the official Java API reference documentation. Both client-side and server-side editors could be useful, and perhaps also for different purposes. For a server-side editor we envision alternative views like the output of the Standard Doclet. However, dynamic functionality on the client-side may still be relevant.

7.7 Alternative Technical Solutions for DJavadoc

Let us discuss some alternative technical solutions to DJavadoc that we did not take but that could have been considered. In the DJavadoc project we choose a DHTML solution because we thought that the new technology would drive our research in a different direction. The use of DHTML has definitively produced such results, for instance the concept of dynamic typography that we would not have come across in many other technological environments. However, in order to support readers of Javadoc-generated API reference documentation the choice of technology is not as clear. Other technical solutions would have proven equally beneficial, perhaps even more so than the DHTML approach. Let us discuss some alternative approaches.

Building a Javadoc application in Java instead of the current Web-based application is a resonable alternative. Since the beginning Javadoc has generated HTML. However, a Java-based reading environment is an alternative, especially since most readers already have a Java environment installed. Javadoc applications could use the highly developed graphical components available in Java to create a multifaceted computer-reading environment. However, Java has a shortage of advanced text-widgets in comparison to Web-technologies. Of course, it is plausible that we would have been able to find a third-party library supporting our needs. We could also have developed them ourselves. Building an application, we would not have been equally restrained to store data as in the Web-solution. A Java based application would not have to transform the Doclet API to another format but could work directly from the stored objects. Coupling the API reference documentation to development environments would perhaps be easier from a Java application than from a Web-solution. A negative point about Java applications is the poor performance that the language is only starting to come to grips with.

Considering the 1,800 classes and 15,000 methods in JDK 1.2, a database solution is perhaps motivated. The need for multiple views of the same information also supports a database solution. Building a database from the Doclet API and delivering dynamic class documents by SQL-query composition is, of course, an alternative. The database solution works both on the Web and for applications. It would scale better than ordinary files since databases are designed to handle large amounts of data. Also, if the DJavadoc project included editing of the API reference documentation, a database solution would provide valuable content management. A negative point from our research perspective is that the database solution would probably reduce dynamic typography to dynamic composition thus reducing text from an animate entity to a dynamically composed static entity.

Building a client-server application (Web or application) is another technical alternative that would increase the possibilities of service integration. A client-server solution would increase the possibilities of handling external data. It can serve to address the limitations made in Web-browsers to store personal data, particularly for off-line versions in which the browser works with the file system. In this case the server would have to be downloaded and run locally. Also, a central server could be used to exchange data among Java programmers to disseminate browsing information. A knowledge-sharing server could be set up that tracks the browsing behavior of a group of readers. The individuals could then benefit by asking what other readers examine, perhaps in relation to a particular class. The negative side of several client-server approaches is the need to be online. However, for intranet solutions this may not be an issue.

7.8 Dynamic Typography

A result of the DJavadoc project is the concept of dynamic typography. We define dynamic typography as changeable appearance of information with the purpose of supporting knowledge acquisition through reading. The concept is relevant both for information-system research and typography research. Dynamic typography has the potential of making text more of an animate entity that reacts to changes in the surroundings and reconfigures itself to best support the reader in a new context.

Developing support for dynamic typography as well as guidelines is a relevant area for continued research that we feel should be explored but that would also take us in a drastically new direction. Currently Internet is filling up with active text and dynamic typography but the field of dynamic typography is not singled out and dealt with in a scientific manner. For the field of typography, dynamic typography should be relevant because it removes the fundamentals on which static typography rests (2D, defined measurements of the reading surface, content always being visible on the reading surface, and so on). For the electronic text community, dynamic typography complements hypertext as an attribute of electronic text. In our view, dynamic typography characterizes a large portion of the Web-development that is currently taking place better than hypertext does. The use of hypertext is often the result of a desire to present text in small chunks and not a desire to achieve non-linearity. This may well result in documents that are difficult to read, print, and maintain. In our view, the concept of non-linear text does not apply to all texts and should be used with care. It is also important to note that non-linear reading does not always require non-linear writing.

Dynamic typography is a genuine feature of the electronic text and by developing it the computer-reading environment will increase its distinctive features in relation to print. Reading from print and electronic text are often compared. A common comment is that electronic texts will never replace print until the screen resolution is at least comparable to print resolution. However, Web browsers did not need a resolution comparable to print to have impact on the reading behavior of a large community. Instead it provided functionality that did not exist in print. In our view, print and electronic texts are different altogether and should not be compared as equals. The computer-reading environment has the potential of being a completely different reading environment that supports another kind of reading than does print. By developing the concept of dynamic typography we could increase the set of distinctive features that the electronic text holds.

Web technologies can serve as a basis for experimentation with dynamic typography because of the highly developed text technology. The Web consists mostly of text (even though other media are also available) and the technology for describing typography of text is highly developed compared to other applications platforms. With the DHTML development the ability to define dynamic behavior in text has also evolved. Furthermore, with the coming of XML as a meta-language that Web browsers can interpret, it becomes possible to design new Web languages that have dynamic typography but resemble HTML in simplicity.

7.9 Summary and Continued Work

We are filled with anticipation for the future because we feel that the DJavadoc project provides the relevant groundwork required for our continued studies. There are several open questions and research issues that arise from our work with DJavadoc: how should API reference documentation evolve into a more use-oriented tool; how can more complex analysis of the source code deliver use-oriented information about software components and the structures of software components; what implications for object orientation can be found by analyzing documentation; how can quality documentation be ensured in the Javadoc approach; and how can we define measures for dynamic typography. A concrete example of how to continue the work is to analyze the design of the Java programming language from the perspective of communications of functionality. Differences in hand-written documentation and automatically-generated documentation can point to limitations in the language. However, our possibilities for continued work stretch across several general and relevant research areas such as API reference documentation, automatic documentation, programming tools, programming languages, typography, and Web technology. Also, in our opinion such work is needed because the growing size and complexity of APIs (or more generally work-related information repositories) present cost and quality problems in software development (or in knowledge-intensive work in general).

Chapter 8

Summary and Conclusions

8.1 Summary

API reference documentation is an important programming tool. Modern programming is often component-based in the sense that a vast number of components are used by programmers developing applications. Programmers have to acquire detailed knowledge to learn which components to use and how to handle these components. Java programmers perform this acquisition by reading the Javadoc-generated API reference documentation. The acquisition of knowledge is integral in the work-task and it is performed continuously.

DJavadoc strives to support programming by further developing the API reference documentation and thereby improve the process of the acquisition of detailed knowledge. The DJavadoc project supports computer-reading of API reference documentation for Java programmers. Generalizing, DJavadoc supports software development. Currently we are addressing the need for multiple views of the Java API reference documentation. The Javadoc-generated API reference documentation is written for multiple needs and therefore contains excessive information in all situations. The project strives to provide a more efficient reading environment that can be dynamically tailored by the reader. Currently we are applying DHTML technology available in Microsoft Internet Explorer 4 (and more recent versions) to achieve dynamic typography.

So far we have achieved preliminary results for the evaluation of DJavadoc. In the process of acquiring real-world study opportunities we have demonstrated DJavadoc to several members of software development teams (programmers, project managers, and technical writers). Their responses have been positive which indicates that DJavadoc may well add genuine value to the API reference documentation. However, these results are only preliminary and must be complemented with more thorough evaluation based on use experience. Currently we have a handful of contacts that are promising. Our first users have already started using DJavadoc.

8.2 Conclusion

From the DJavadoc project we can so far conclude that our preliminary studies support the usefulness of the DJavadoc design. However, more thorough studies must be performed. We have also found that DHTML provided adequate and straightforward technology for client-side real-time redesign of documentation such as the official Java API reference documentation.

DJavadoc adds an explicit information model to the official Java API reference documentation and provides mechanism (based on the underlying model) by which the reader controls the visibility of information. From our experience with DJavadoc, we conclude that instantaneous performance is required for dynamic typography such as collapse and expand functionality. Also, such dynamic typography works best for smaller changes. We believe these requirements favor the type of client-side architecture that DJavadoc contains.

We also conclude from our work in this field that API reference documentation is system oriented and should be designed for more use-orientation. The redesign concerns both reading issues, such as typography and navigation, and issues in automated generation of documentation.

Currently the process of designing alternative Java API reference documentation is cumbersome, judging from the size and complexity of the Standard Doclet. More rapid development tools should be designed, for instance in the form of Doclet editors.

Furthermore, we conclude that the documentation can play an important role in the analysis of the Java language and ultimately object orientation. The documentation is the graphical user interface to the source code and will therefore reflect limitations in the programming language. In our opinion, the ability with which a programming language facilitates automated documentation will have direct impact on software cost and quality, particularly for component-based programming.

Finally, we conclude that design applied in DJavadoc fundamentally represents an extension to the computer-reading environment and electronic text. Dynamic typography as a concept should be further developed to discover typographical knowledge to control and enhance computer-reading.

For the future we believe there is great potential in the further development of use-oriented API reference documentation as a communicator of functionality in software engineering.

Bibliography

[1] Andrew Dillon and. *Designing Usable Electronic Text : Ergonomic Aspects of Human Information Usage.* Taylor and Francis, 1994.

[2] Barry Phillips and. Designers: The browser war casualities. *IEEE Computer*, 31(10):14–16, 1998.

[3] Christer Hellmark and. *Typografisk handbok (The Typographers Guide).* Ordfront, 1998.

[4] David Flanagan and. *Javascript: The Definitive Guide.* O'Reily and Assosiates, 1998.

[5] David K. Farkas and. Layering as a safety net for minimalist documentation. In John M. Carroll and, editor, *Beyond the Nurnberg Funnel.* MIT Press, 1998.

[6] Donald E. Knuth and. *Literate Programming.* Center for the Study of Language and Information, Leland Stanford Junior University, 1991.

[7] Donald E. Knuth and. *The Stanford GraphBase: a platform of combinatorial computing.* ACM Press, 1993.

[8] Douglas A. Norman and. *The Design of Everyday Things.* Basic Books, 1988.

[9] Douglas Kramer and. Api documentation for source code comments: A case study of javadoc. In *the Seventeenth Anual International Conference of Computer Documentation*, September 12-14 1999.

[10] Evert Gummesson and. *Qualitative Methods in Management Research.* SAGE Publications, 1991.

[11] Ian Sommerville and. *Software Engineering.* The Bath Press, 1989.

[12] J. G. Brookshear and. *Computer Science An Overview.* Benjamnin/Cummings, 1994.

[13] J. Nielsen and. *Multimedia and Hypertext: the Internet and Beyond.* AP Professional, 1995.

[14] Jay David Bolter and. *Writing Spaces The Computer, Hypertext and the History of Writing.* Lawrence Erlbaum Associates, 1991.

[15] JoAnn T. Hackos and. Choosing a minimalist approach for expert users. In John M. Carroll and, editor, *Beyond the Nurnberg Funnel.* MIT Press, 1998.

[16] John M. Carroll and. *The Nurnberg Funnel.* MIT Press, 1990.

[17] John M. Carroll and, editor. *Minimalism Beyond the Nurnberg Funnel.* MIT Press, 1998.

[18] Karren A. Schriver and. *Dynamics in Document Design.* Wiley, 1997.

[19] Lisa Friendly and. The design of distributed hyperlinked programming documentation. In *1995 International Workshop on Hypermedia Design*, 1995.

[20] Norman Meyrowitz and. Hypertext - does it reduce cholesterol too? In James Nyce and Paul Kahn and, editors, *From Memex to Hypertext : Vannevar Bush and the Mind's Machine.* Academic Press, 1991.

[21] R. Bringhurst and. *The Elements of Typographic Style.* Hartley and Marks, 1996.

[22] S. Wolfram and. *The Mathematica Book.* Wolfram Media/Cambridge University Press, 1996.

[23] Steel JR. Gy L. and. *Common LISP.* Digital Press, 1990.

[24] Steinar Kvale and, editor. *Issues of Validation in Qualitative Research.* Studentlitteratur, 1989.

[25] Stephen R. Schach and. *Software Engineering with Java.* Irwin, 1997.

[26] Steven P. Reiss and. Software tools and environments. *ACM Computing Surveys*, 28(1):281–284, 1996.

[27] Theodor H. Nelson and. *Literary Machines.* The Distributors, 1987.

[28] William Horton and. *Designing and Writing Online Documentation Help Files to Hypertext.* John Wiley and Sons, 1990.

[29] Doug Bell, Ian Morrey, and John Pugh and. *Software Engineering A Programming Approach.* Prentice Hall, 1987.

[30] Erik Berglund and Henrik Eriksson and. Intermediate knowledge through conceptual source-code organization. In *Tenth International Conference on Software Engineering and Knowledge Engineering*, June 18-19 1998.

[31] M. Bieber, F. Vitali, H. Ashman, V. Balasubramanian, and H. Oinas-Kukkonen and. Fourth generation hypermedia: some missing links for the world wide web. *International Journal of Human-Computer Studies*, 47:31–65, 1997.

[32] Mary Campione and K. Walrath and. *The Java tutorial : object-oriented programming for the Internet.* Addison-Wesley, 1998.

[33] Henrik Eriksson and Mark A. Musen and. Metatools for knowledge acquisition. *IEEE Software*, 10(3):23–29, 1993.

[34] Henrik Eriksson, Angel R. Puerta, and Mark A. Musen and. Generation of knowledge-acquisition tools from domain ontologies. *International Journal of Human Computer Studies*, 41:425–453, 1994.

[35] Ivar Jacobson, Gary Booch, and James Rumbaugh and. *Unified Software Development Process.* Addison-Wesley, 1999.

[36] Paul Kahn and Krzysztof. Lenk and. Principles of typography for user interface design. *Interactions*, pages 15–29, 1998.

[37] Donald E. Knuth and L. Silvio and. *The CWeb System of Structured Documentation, version 3.0.* Addisson Wesley, 1994.

[38] M. T. Maybury and W. Wahlster and. *Readings in Intelligent User Interfaces.* Morgan Kaufmann, 1998.

[39] James Nyce and Paul Kahn and, editors. *From Memex to Hypertext Vannevar Bush and the Mind's Machine.* Academic Press, 1991.

[40] Hans van Vilet and. *Software Engineering Principles and Practice.* John Wiley and Sons, 1993.

[41] *About W3C.* www.w3.org/Consortium/.

[42] *Ada Online.* www.adahome.com/rm95/.

[43] *CSS at W3C.* www.w3.org/Style/CSS/.

[44] *DJavadoc Home Page.* www.ida.liu.se/ eribe/djavadoc.

[45] *DOM at W3C.* www.w3.org/DOM/.

[46] *ECMA Home Page.* www.ecma.ch.

[47] *ECMAScript Specification (PDF).* www.ecma.ch/e262-pdf.pdf.

[48] *Gnu Emacs.* www.gnu.org/software/emacs/emacs.html.

[49] *Gnu Texinfo.* www.delorie.com/gnu/docs/texinfo/texinfo_toc.html.

[50] *HTML 4.0 Specification.* www.w3.org/TR/REC-html40/.

[51] *HTML at W3C.* www.w3.org/MarkUp/.

[52] *Java.* java.sun.com.

[53] *Java Hotspot.* www.java.sun.com/products/hotspot/index.html.

[54] *Java Reflection.* java.sun.com/products/jdk/1.2/docs/guide/reflection/index.html.

[55] *Java Tutorial.* java.sun.com/docs/books/tutorial/.

[56] *Java Workshop.* www.sun.com/workshop/java/.

[57] *Javadoc.* java.sun.com/products/jdk/javadoc/.

[58] *Javadoc FAQ: memory and time requirements.* java.sun.com/products/jdk/javadoc/faq.html#memory.

[59] *Java's XML Home Page.* java.sun.com/xml.

[60] *JavaScript at Netscape.* developer.netscape.com/tech/javascript/index.html.

[61] *JBuilder.* www.borland.com/jbuilder.

[62] *JDK 1.2.* java.sun.com/products/jdk/1.2/.

[63] *JPython.* www.jpython.org/.

[64] *MSDN Online Workshop.* msdn.microsoft.com/workshop/.

[65] *Netscape Cookie Discussion.* help.netscape.com/kb/consumer/970226-2.html.

[66] *Netscape Cookie Discussion.* msdn.microsoft.com/workshop/networking/wininet/overview/http

[67] *Perl Online.* reference.perl.com/.

[68] *Proposed Additional Javadoc Tags.* java.sun.com/products/jdk/javadoc/proposed-tags.html.

[69] *Protege Home Page.* www.smi.stanford.edu/projects/protege/.

[70] *Python Online.* www.python.org/doc/current/lib/lib.html.

[71] *Rational Home Page.* www.rational.com.

[72] *Rational SoDA.* www.rational.com/products/soda/.

[73] *SGML Home Page.* www.oasis-open.org/cover/sgml-xml.html, www.sgml.org.

[74] *Some Java History.* java.sun.com/features/1998/05/birthday.html.

[75] *Swing.* java.sun.com/products/jfc/tsc/index.html.

[76] *Visual Age.* www-4.ibm.com/software/ad/vajava/.

[77] *Visual Cafe.* www.symantec.com/domain/cafe/vc4java.html.

[78] *Visual J++.* msdn.microsoft.com/visualj.

[79] *W3C.* www.w3.org.

[80] *WoddenChair Software.* www.woodenchair.com.

[81] *XML at W3C.* www.w3.org/XML/.

[82] *XML Specification.* www.w3.org/TR/1998/REC-xml-19980210/.