# A New System for Automatic Creation of Hypertext API Documentation in Java[TM]

Michael Hilberg

4/18/2000

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

## TABLE OF CONTENTS

1 Introduction ... 3
1.1 Related Work ... 4
1.2 Organization of this Paper ... 4
2 The Javadoc tool ... 4
2.1 Version History ... 4
2.2 System Overview ... 4
2.3 Javadoc 1.1 ... 5
2.4 Improvements in Version 1.2 ... 6
2.4.1 Additional Features ... 6
2.4.2 Frame Layout ... 7
2.4.3 Doclet Architecture ... 7
3 Discussion of Javadoc's usability ... 8
3.1 Orientation ... 9
3.2 Interface/Presentation ... 9
3.3 Search ... 10
3.4 Navigation ... 10
3.5 Index ... 10
3.6 Links ... 11
3.7 Literary Paradigm ... 12
3.8 Electronic Augmentation ... 12
3.9 Customization/Personalization ... 12
4 Requirements for AN enhanced System ... 13
4.1 Improvements to the Issues Discussed in Chapter 3 ... 13
4.2 Backward Compatibility ... 13
4.3 Platform Independence ... 14
4.4 Efficienct Distribution ... 14
4.5 Reusability ... 14
5 Design of a Javadoc viewer ... 14
6 Prototype status ... 16
6.1 The Doclet ... 16
6.2 The Viewer ... 16
6.3 Case study ... 18
7 Conclusions ... 19

# 1 INTRODUCTION

Every software development process necessarily produces a number of documents describing the internals and externals of the product. Maintainability, repairability, evolvability, reusability, portability and interoperability are just a selection of software qualities that are highly impossible to achieve without appropriate documentation. As Fred Brooks puts it: "For even the most private of programs, some such communication is necessary; memory will fail the author-user, and he will require refreshing on the details of his handiwork." [4]

Usually a number of separate documents in either printed or electronic form exist, each describing a certain phase of the development process (e.g. requirements, design) or certain parts of the product (e.g. sub-systems, user interface). However, these documents are highly interrelated (e.g. the design has to address the requirements, sub-systems are communicating with each other). The paper medium is obviously not an adequate medium, as users tend not to read documents from beginning to end but frequently jump between related topics. In the electronic medium, hypertext has been recognized as a superb mechanism to be used with these documents [3]. It provides a standard way to structure and link the documents and offers tools to edit and navigate them.

One area to which the use of hypertext is especially applicable is code documentation. These documents come in small modules (e.g. one per function) and have natural relationships (e.g. usage, inheritance), that users often navigate over. By providing these links, hypertext can highly increase the efficiency of the person using the documentation. Another issue, that contributes to the applicability of hypertext, is the fact, that a fair part of the documentation can be extracted from the source or object code automatically. This frees programmers from the task of linking the documents in a sufficient manner, which can be intractable in large projects.

The Java™ programming language is just one language, that implements the idea of an automatically created hypertext code documentation. Its original specification contains a mechanism to automatically produce HTML from the program source code and from comments embedded in the source code [7]. In addition to specifying the mechanism and the format of the source code comments, the necessary conversion tool, Javadoc, is provided with every Java Software Development Kit, thus making it available to every programmer on any platform. The target audience for the created documentation can be the internal programming team only, developers using the implemented programming interface or even conformance testers, using the documents as a specification [9].

While the first version of Javadoc produced a collection of simple HTML pages allowing to browse the packages, classes and class members and their associated comments, the second version, among other things, enhanced the usability by using frames. The current version also adds an architecture that allows for extensions to the tool to create arbitrary outputs mainly targeting the generation of printed documentation.

As good as its concept may sound, the hypertext documentation generated by Javadoc suffers from the limited capablities of the static HTML format. This paper will discuss

these limitations, propose some improvements and present a prototype for an enhanced system.

## 1.1 Related Work

Other approaches to deliver an improved Java API documentation have been made: The JavaHelp Specification [2] introduces a standard Java extension that allows to flexibly create help systems for applications, components and web pages. It is mainly targeting end-user help systems, but as a proof of concept a prototype API viewer has been created. The system is implemented in Java and is using XML to store its information. It provides an index and full-text search funtionality.

Priestley [8] and Green [10] report on efforts made during the development of IBM's VisualAge IDE tool. Their system uses compressed HTML and a local lightweight web server to deliver documentation for both the Java API and the IDE functionality.

## 1.2 Organization of this Paper

Chapter 2 will introduce the details of the mechanism underlying Javadoc including the improvements made in recent versions and the Doclet architecture to create custom output. Chapter 3 will discuss the shortcomings of the produced documentation by evaluating it with respect to a set of usability criteria. This evaluation will lead to requirements for an enhanced system as presented in chapter 4. The design and current status of a prototype will be treated in chapter 5 and 6.

# 2 THE JAVADOC TOOL

The following chapter will give an overview of the Javadoc system and will briefly introduce its main features and its version history. Its a summary of the information available on the Javadoc Tool Home Page [1], and the interested reader is referred to it for further details.

## 2.1 Version History

Version 1.1 of Javadoc was included in the Java Development Kit 1.1, Javadoc 1.2 is part of Java 2 (version 1.2). Version 1.3 is currently being developed and will be released with the next version of the Java Development Kit. There are significant differences between version 1.1 and 1.2 in both architecture and style (see sections 2.3 and 2.4 for details) while 1.3 does not add any significant new features and mainly represents a bug fix release. This paper will refer to version 1.1 as "the first version" and to 1.2 and 1.3 as "the second version".

## 2.2 System Overview

The Javadoc tool takes package names or source file names as input, parses the source code and extracts the structure of packages, classes and interfaces and the signature of

```
/**
 * Compares two Objects for equality.
 * Returns a boolean that indicates whether this Object
 * is equivalent to the specified Object. This method is
 * used when an Object is stored in a hashtable.
 * @param      obj  the Object to compare with
 * @return          true if these Objects are equal;
 *                  false otherwise.
 * @see        java.util.Hashtable
 */
public boolean equals(Object obj) {
     return (this == obj);
}
```

**Figure 1 – Sample method comment**

members. These elements are annotated in the source code using a special format as specified in the Java Language Specification [7]. The comments consist of text of arbitrary length that can include HTML tags and broadly describe the element in question. This text is augmented by special tags that define certain characterisics of the member, for instance parameters, return values and thrown exceptions for a documented method. The author may also use the @see tag to reference other elements or embed HTML links to arbitrary sources. Figure 1 (method equals(Object) from class Object [1]) shows a sample comment.

From these comments, Javadoc produces a set of HTML pages. The output format varies significantly from different version and will be treated in the next chapters.

## 2.3 Javadoc 1.1

Version 1.1 creates one file for each package, class or interface. An overview page list all documented packages and links to the package pages listing all classes and interfaces in the respective package. An index of all members (remarkably not including classes and packages) and a document presenting the inheritance tree are also produced.

The class page starts with a graphical representation of the classes inheritance structure and with the class-level comments. This is followed by overviews (unfortunately called "index" in this version) of the constructors, fields and methods. The first sentence of the comment for each is presented in this section to give a quick introduction to the class members. All members are documented in more detail further down on the page. The index entries link to these detailed descriptions.

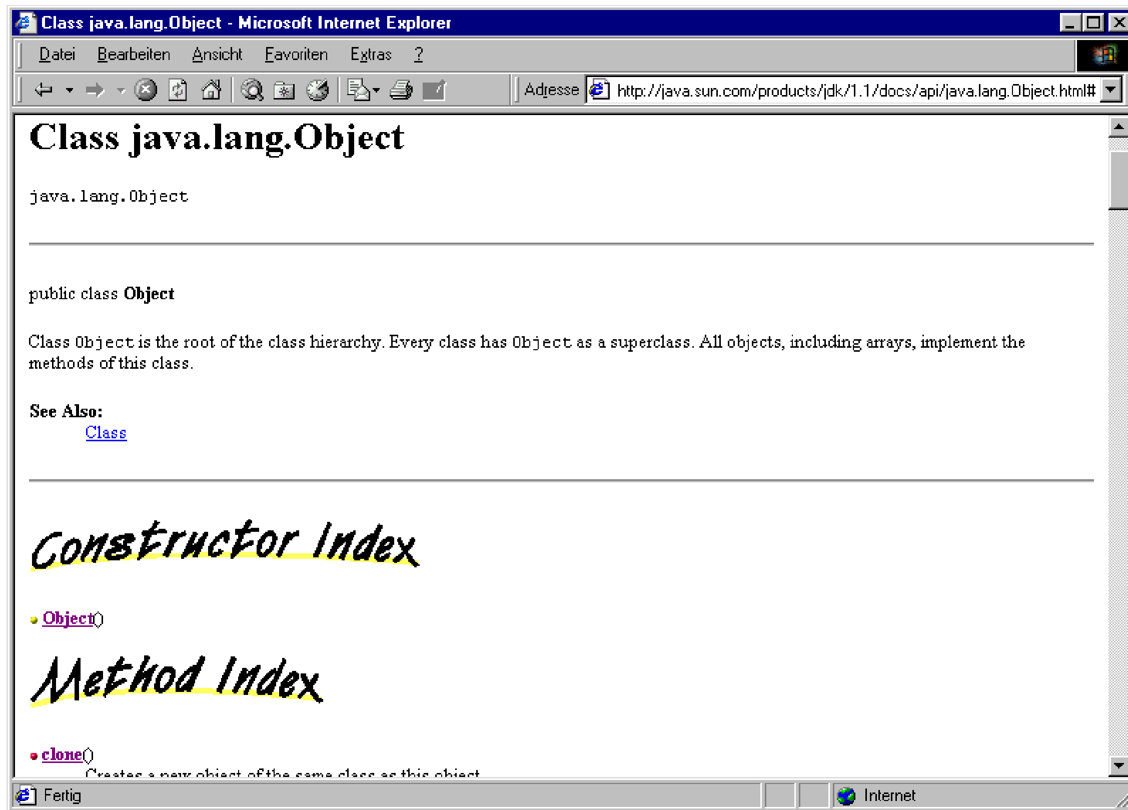


**Figure 2 – Javadoc 1.1 Screenshot**

Occurences of classes (e.g. in method signatures) are linked to the respective class, if the class is documented within the same run. @see tags are also linked to the documents referenced.

All pages have a navigation bar on top and on the bottom with varying entries.

## 2.4 Improvements in Version 1.2

### 2.4.1 Additional Features

The evaluation of user feedback on Javadoc lead to the implementation of frequently requested features in the second version. Main improvements are the ability to include comments for packages and an overview document, possibility to link to external Javadoc documentations, the inclusion of inherited members, subclasses or implementing classes.

The handling of large documentations has been improved by a hierarchical destination file structure, by the possibility to break up the index into one document per letter and a mechanism to automatically copy complementing files (e.g. pictures, examples) to the destination.

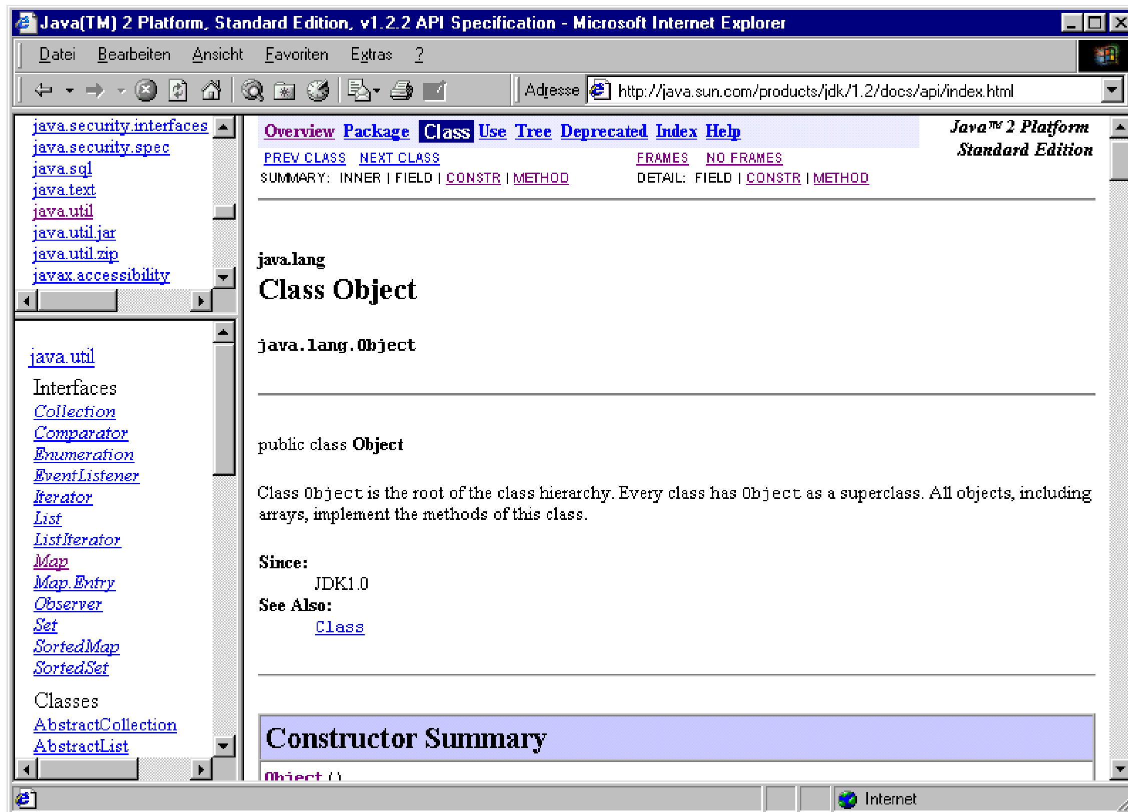


**Figure 3 - Javadoc 1.2 Screenshot**

## 2.4.2 Frame Layout

The most important improvement over the first version is the frame layout. It introduces many new possibilites to navigate the documents. As can be seen in Figure 3, the screen is divided into three frames. The frame in the upper left corner displays the packages and a link labeled "All classes". On clicking on one of these links, a list of all classes and interfaces in the interface (or all classes respectively) is displayed in the frame below. A click on a class or an interface in the classes frame displays the corresponding documentation in the main frame. The main frame is used like a normal browser window if the user clicks on links in this frame (e.g. the documentation of a referenced class is also loaded into the main frame).

## 2.4.3 Doclet Architecture

The first version of Javadoc had the output format hardcoded into it and an adaption of the output (e.g. style and file format) was virtually impossible. The second version adressed this by introducing a new architecture that separates between parsing of the input and production of the output. The source files are read by Javadoc and an internal representation is build. This data model, consisting of classes representing packages, classes and members, is passed to a Doclet, that can access the model via an API to create output in whichever format.

The following Doclets are currently available:

- The standard Doclet. This Doclet is provided by Sun and produces HTML pages. It will be the focus of the upcoming discussion.
- MIF Doclet. The MIF Doclet outputs the documentation in Adobe FrameMaker`s file format, which can be converted in serveral other formats. The Doclet is an experimental implementation by Sun targeting the production of printed documentations.
- Dynamic HTML Doclet. The result of a research project that uses the Dynamic HTML features of modern browsers to add new functionality to the documentation. The added features are mainly concerned with concentrating documents by hiding certain parts on the user`s request. This functionality might be of interest to the discussion on a new system.
- Simple RTF Doclet. This Doclet produces a documenatition in the Rich Text file format, that is supported by many applications (e.g. Microsoft Word) to created printed documentation.
- Emacs Doclet. Currently under development. Will produce output in the Info format used by the GNU Emacs editor.
- LaTeX2e Doclet. Developed as an internal tool by a consulting company, this Doclet produces code, that can be used as input to LaTeX to create a printed documentation.
- iDoclet. An extension to the standard Doclet, that adds support for three new tags (@pre, @post, @inv), which are used by a preprocessor (iContract) to create assertions.

Sun also uses two additional Doclets internally: (1) A DocCheck Doclet, that scans the comments for errors and inconsistencies and generates a report. (2) An Internationalization Doclet that substitute the comments in the source code with translated versions in separate files.

# 3 DISCUSSION OF JAVADOC'S USABILITY

Fillion and Boyle [5] identify nine significant issues, that influence usability and effectiveness of hypertext systems:

- Orientation
- Interface/Presentation
- Search
- Navigation
- Index
- Links
- Literary Paradigm
- Electronic Augmentation
- Customization/Personalization

These issues will provide a framework for the following discussion of Javadoc`s weaknesses.

## 3.1 Orientation

Orientation has been recognized as a very important topic in hypertext systems. In contrast to printed documentations, hypertext does not have a linear ordering and the user might easily get lost. Hypertext systems have to provide mechanisms, that show users where they are with respect to related documents and to some static reference point.

In the case of the Java API documentation, the hierarchic package structure is one possibility to structure the documents. Displaying the current position in the package structure is one way to give orientation to the reader. Classes in a common package are usually interelated and presenting an overview of the package together with the current document could further improve orientation.

In the first version of Javadoc, the only way to position the current document with respect to the package structure is the package name as presented on top of the page. In order to find classes in the same package, the user had to go back to the page that lists all classes and interfaces in the package.

Version 1.2 introduced the use of frames and slightly improved orientation. Clicking on a package in the package frame showed all classes in this package in the classes frame below. After choosing a class from the classes frame, the classes frame is still showing the related classes and helps the user to put the class documentation currently read in context.

Unfortunately, the current Javadoc implementation does not update the classes frame if the user navigates to another document by clicking on a hyperlink in the class documentation. After this, the class documentation frame might show a class from a different package, while the classes frame still shows the classes from the previous package. This is highly likely to confuse users not familiar with the frame update mechanism and does severely contribute to disorientation.

## 3.2 Interface/Presentation

Reading text on the screen is very different from reading from paper. The presentation of text on the screen and the presentation of link anchors are issues that arise in this context.

With the widespread use of the Internet, users have become accustomed to the HTML interface as rendered by current browsers. Their presentation of text and links has evolved and has been widely accepted and should hardly be an issue here.

Almost any modern hypertext user interfaces uses graphical icons to represent links. This has never been done in Javadoc though. The first version used images for headings (e.g. "Constructors", "Methods") and for bullet points. The bullet points were color-coded to represent certain characteristics of methods and fields (Magenta=instance variable,

blue=static variable, yellow=constructors, red=instance methods, green=static methods [6]). Clearly, this scheme is not at all intuitive and it does not come as a surprise that it was dropped in the second version. The images depicting section headings were also left out, because they could easily be substituted by HTML code and did not provide any additional benefits. Also, the image files had to be in a certain place on the file system relative to the document and this required copying of the files by hand. They also had to be distributed with every documentation. All these reasons lead to the fact that no images are incorporated into the second version of Javadoc. This decision might not have been a good one considering all the possible benefits one could get from using images wisely. A new distribution mechanism might also simplify handling of the files.

## 3.3 Search

Keyword search can provide an efficient way to retrieve documents from hypertext systems and is a major advantage over paper documentation.

The Javadoc system does not provide such a search mechanism. It`s possible though to extend the system by using any of the widely available search engines that index HTML pages. However, these search engines are external products that have to be purchased, installed and maintained separately and may therefore not be an option to many users. They also suffer from not being tightly integrated into the system. For instance, they do not capture the semantics of the documentation (a search for a method name will return the index entry, the overview entry and the method comment, though only the method comment is being searched for) and usually have difficulties with the frame format (the search engine will link to the file of the document only and will therefore not recreate the frames layout).

## 3.4 Navigation

The nonlinear form of hypertext arises the need for tools that faciliate navigation. An issue is how the user reaches the information she or he is looking for. Index and keyword search are solutions but one might imagine other, more sophisiticated methods (e.g. involving graphics).

In addition to an index, Javadoc 1.2 provides the frame layout to navigate by going through the package structure. Obviously, a search mechanism is missing. The frame layout can also be improved (see Chapter 3.1 for a discussion).

## 3.5 Index

Indexes complement the keyword search. When doing a search, the user must know and provide keywords, while the index lists all available keywords. Indexes are created by the documentation writer and can therefore give valuable additional information (the writer will not include irrelevant hits and might give the entries some kind of ordering by importance).

Javadoc automatically produces an index page. Version 1.2 also allows to split the index into separate pages per character, which is useful for large projects. The index lists all packages, classes, interfaces, methods and fields, sets them into context (e.g. gives the containing class for a method or field) and links to the document describing the entry. There is however no mechanism to include other entries (e.g. definitions of terms on overview pages) in the index. It also suffers from not being adapted to the Java naming conventions: Methods manipulating protected fields are prefixed by "get" and "set". Obviously, the entries for the letters "g" and "s" get very large and no mechanism exists, to improve navigation inside these pages.

## 3.6 Links

The number of links presented to the user and the granularity of referenced objects also has an impact on usability. If too many links are shown, the user has problems finding relevant ones. On the other hand, not including some links might make navigation more difficult because the user has to find the object by other means (e.g. index or keyword search). Granularity refers to the question, whether links reference the whole document, that includes the relevant information, or whether only the specific part (e.g. diagram, paragraph, sentence) is linked to. If the granularity is too large, the user might have problems finding the relevant information.

Javadoc links occurences of classes (e.g. in method signatures or in an inheritance tree) to their documentation. These links are essential and need no further discussion. It should also be out of question that some other links (e.g. overriding method linking to the method overridden, classes link to subclasses, interfaces to implementing classes) are useful while not overloading the documentation and should therefore be included. On the other hand, one category of links can be considered controversial: While version 1.1 of Javadoc did not link to inherited methods and fields (see [6] for the rational behind this design), version 1.2 does include a short form of both in the method or field index respectively. The documentation of a class, that has a deep inheritance structure, is likely to get blown out of proportion (see the documentation of class javax.swing.JRadioButton for an example). To make things worse, the method signature is omitted. Links for overloaded methods therefore all have the same label, making them practically useless.

In addition to the links generated automatically, documentation authors also have the possibility to reference arbitrary classes, methods and other documents by either using the @see tag or by including HTML links in the comment itself. The author might not make the effort to include links to relevant documents though, but this can hardly be considererd a flaw of Javadoc.

Concerning granularity, occurences of classes are linked to the document describing that class. References to methods and fields are linked to their detailed description inside the class documentation. This design can hardly be improved. It is an issue though, whether methods and field should all be documented in one document per class or whether each method or field should be described in a seperate document. With the current design, documentations of large classes tend to get too long, worsening orientation. On the other

hand, documentations of single methods or fields are usually very short and the navigation among these seperate documents might introduce an overhead.

## 3.7 Literary Paradigm

Hypertext offers many new advantages over paper, but is also very different. The writing style has to be adapted to the new medium.

It appears that the Java API documentation is well suited for the new medium and hardly needs any specific adaptations. One question that remains though, is the issue of node granularity as discussed at the end of the previous chapter.

## 3.8 Electronic Augmentation

The hypertext media does not limit the user to static text and the integration of sound, graphics and animation, if wisely used, can increase the value and usability of the documentation.

The standard doclet does not produce data of any other format than HTML text. Version 1.1 did make use of simple graphics for bullet points and headings but did this in a very inefficient way (see Chapter 3.2). Instead of improving the icons and usage though, the current version does not use any graphics at all. Given all experiences with graphical user interfaces, it should be considered highly likely, that usability can be increased by the use of icons.

As introduced in Chapter 2, the documentation writer can link to any type of information by using the @see tag or by inserting HTML links into the comments. This allows to augment the documenation with almost any media that can be captured electronically and displayed by a web browser.

## 3.9 Customization/Personalization

The Java Documentation is used by users with different intentions and varying skill levels. It is therefore highly unlikely, that one style fits the needs of all. An application developer using the API is only interested in the public methods for instance, while someone working on the documented system itself is also interested in the private methods.

Javadoc does provide certain customizations at the time of production. It is possible to specify, whether private and package private methods are included and which tags (e.g author, version) should be taken into account. The HTML interface on the other hand does not not allow to customize any aspect of the documentation at the time of reading.

A possible solution is, to create separate sets of documenation for every group of users, but considering the necessary efforts to configure and distribute these sets it should be obvious, that this approach is not desireable.

# 4 REQUIREMENTS FOR AN ENHANCED SYSTEM

As introduced in Chapter 2, the Javadoc system consists of two parts: The Javadoc program, that translates the source code comments into HTML pages, and a web browser, that is used to view the produced HTML pages. This chapter assumes that any new system will use a similar architecture consisting of two components: A producer and a viewer. This is not stated as a requirement but helps ordering the following discussion.

## 4.1 Improvements to the Issues Discussed in Chapter 3

Chapter 3 discovered several shortcomings of the current system. Some of them have obvious solutions, that will be stated as an requirement here. Others are less easy to solve and will not be included as explicit requirements. They should however be kept in mind and be considered when making design decisions.

Orientation must be improved by implementing a consistent update mechanism for separate views, i.e. on clicking on a link in a class documentation, a tree view used for navigation must in some way be updated to respect the new location.

Graphics should be used to augment the interface and navigation. The distribution mechanism must be adapted to faciliate the handling of these graphics.

The new system must provide a keyword search mechanism, that ideally captures the semantics of the documentation. It should be able to at least distinguish between different types of documents (classes/packages) and recognize certain special hits (not show the index entry but the document it links to instead).

The viewer should allow a number of possible customizations and personalizations at runtime. The user must at least be able to specify which documents to include in overviews (e.g. only public members), and be able to create bookmarks.

## 4.2 Backward Compatibility

To have the slightest chance of being adopted, an new system must be compatible with existing components. In the specific case of the Javadoc system, this requirement comes in three varieties: (1) It would be impractical and inefficient to rewrite every comment to be able to use the new system. Therefore, the producer that extracts the new documentation format from the source code must be able to read comments written in the current format. (2) Some documentations might only exist in the HTML format produced by the current tool. This leads to the requirement that the new viewer should be able to read and render HTML that has been produced by the current standard doclet. As these documentations suffer from the weaknesses discussed in chapter 3, the viewer does not have to provide the same navigational enhancements for this format as it does for its proprietary format. (3) Some existing environments, IDEs for instance, might rely on the current architecture. Therefore, it should still be possible to use a web browser to view the documentation.

## 4.3 Platform Independence

The fact, that the Java Compiler is written in Java, allows to use any platform with an available Virtual Machine to be used as a development environment. The exisiting documentation system requires the existence of a web browser, but with the growth of the Internet, browsers have become so widespread, that they can be considered available and installed on every platform in question. The documentation tool is an integral part of the development environment and the new documentation system should therefore be available on any platform as well, preferably without installation of any additional components.

## 4.4 Efficienct Distribution

There currently are two ways in which the HTML documentation for Java programming interfaces is being distributed: (1) The documents are stored on a Internet/Intranet web server and can be downloaded one at a time by a remote user. (2) The documents are packaged into an archive and send to the user over a network as a whole. The user then extracts the documents and browses them on his or her local file system.

The first approach suffers from the delay that is introduced by the downloading of documents on demand. On the other hand, it minimizes the amount of data transfered over the net and it allows to keep the documentation up to date by centralizing it. The second approach requires to download possibly large files (the compressed Java 2 Standard Edition API documentation is almost 20 MB large), relies on compression software installed on the users`s machine and uses up large amounts of disk space on the local machine.

The new system should support both modes but should aim at making both more efficient and user friendly.

## 4.5 Reusability

IDEs can greatly benefit from integrating a documentation system. The new system`s architecture should reflect this by providing a way to be integrated into these environments.

# 5 DESIGN OF A JAVADOC VIEWER

The following chapters addresss the requirements by making design decision for a prototype of an enhanced Javadoc system.

The separation of producer and viewer is a natural one since writer and reader are often different persons. As in the chapter before, this architecture will be assumed for the implementation. The viewer might be extended later to control the producer so documentation can be created on the fly.

The best choice for implementing the producer is to create a custom Doclet. This minimzes the implemenation effort and nicely integrates into the existing architecture. To accomplish platform independence of the viewer, it will also be written in Java. Another option for the implementation of the viewer is, to extend the functionality of a regular web browser with Applets and JavaScript. It is highly unlikely though, that this approach can account sufficiently for all requirements.

The following discussion will assume the Java 2 Standard Edition Software Development Kit on the producer platform and the Java 2 Runtime Environment on the viewer platform. Java 2 has been chosen, because it is a major consolidation of the Java platform and because it includes many useful libraries (e.g. the Swing Java Foundation Classes). It has also been released suffiently long ago to assume widespread deployment to the likely user base.

Backward compatibility will be achieved by producing HTML pages and augmenting them by additional functionality. The Doclet will capture additional information about the API documented to implement the search, index and navigational functions. For a prototype, the Doclet can invoke the standard Doclet to create the HTML, though a more sophisticated presentation might be designed later. The additional information will be made persistent (serialization will do for the prototype) and shipped with the HTML.

The viewer will make use of the Swing Java Foundation Classes to build the user interface. Among other things, Swing includes an HTML renderer, that allows to view the pages created by the producer. This also makes it possible to view exisiting documentations as required.

HTML pages can be compressed quite well and the JAR file format can therefore be used to enhance distribution. The JAR library is also include in Java 2. Four modes of distribution come to mind:

- Do not use compression and keep both the serialized information and the HTML pages on either a remote host or on the local machine. Download or open the serialized information into the viewer and have the viewer load HTML pages when needed.
- Keep the serialized information compressed on a remote host. Open this file into the viewer and load the uncompressed remote HTML pages on request. This reduces the time required to download the serialized file.
- Download the compressed serialized information and keep them on the local machine. Browse this information and download HTML-pages over the network when needed. This approach allows for instant navigation while requiring little download time and hard disk space.
- Download the serialized information and the HTML pages as a single JAR file and open this JAR file directly into the viewer. This approach might require a large download but reduces the amount of hard disk space needed. It also frees the user from having to uncompress the file manually.

Figure 4 illustrates these distribution mechanisms.

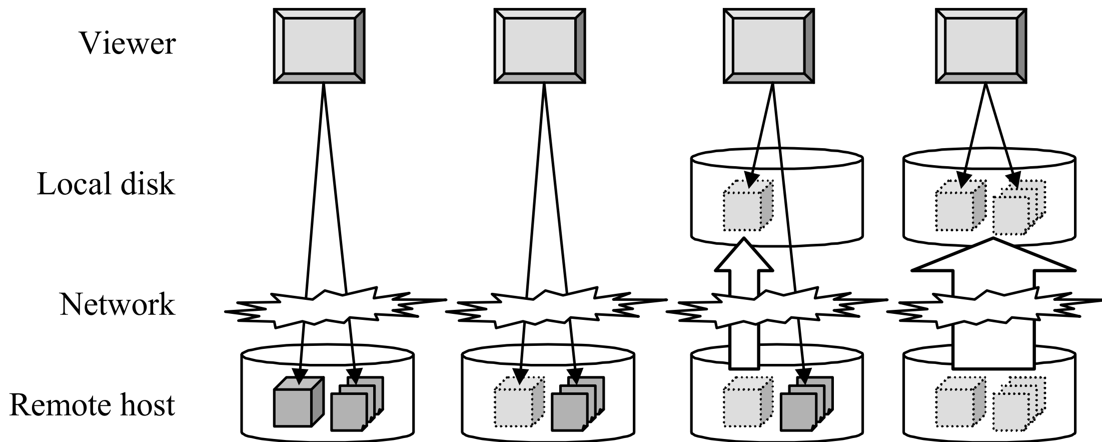


**Figure 4 – Possible Distribution Mechanisms**

# 6 PROTOTYPE STATUS

The current prototype does implement some of the requirements, though some (e.g. the search engine) were out of the scope of this early implementation. The following sections give some details on the implementation.

It should be mentioned, that the JAR-file containing the Doclet, the viewer, the application and all currently used images is only 18 KB small.

## 6.1 The Doclet

The prototype Doclet builds a model for a tree view and an index. This model is serialized and written to disk. It also invokes the standard Doclet to create the HTML pages. The doclet passes all command line options to the standard Doclet and thereby maintains all previously possible customizations. The distribution mechanisms using JAR files are not yet implemented.

## 6.2 The Viewer

The viewer is a JPanel, that exposes a number of methods so it can be used as a component in other Java applications. No attempt has been made to respect the Java Beans architecture, though creation of a Bean should be possible. A Java application that uses the viewer panel and that can be invoked from the command line has been implemented.

The viewer panel is divided in two halfs by a JSplitPane. The left side contains a JTabbedPane having a tree view of the package structure and the index as tabs (the search function could also be added here). The right side of the screen is made up by the browser window showing the hypertext documents. A floatable JMenuBar containing buttons for basic functions ("Open", "Go Back", "Go Forward") is placed on top of both.

The tree view shows the package structure as constructed by the Doclet. Packages make up the root elements of the tree, their children are the classes and interfaces contained in the package. The classes and interfaces have their fields, constructors and methods as children. The elements are depicted as icons, that allow to distinguish them easily. A click on an element in the tree invokes loading of the respective document into the browser window.

The index view is enhanced by a combo box, that allows to select the elements presented in the index ("All elements", "Classes only" being the currently available options). The same icons as in the tree view are used to make it easy to distinguish the type of an index entry. A click on an element again invokes loading of the documentation into the browser.

In both the tree view and the index view, tooltip text is used to give a short description of the elements. The tooltip text shows the first sentence of the documentation, which is normally used in the overview section of the HTML pages. This should speed up navigation as the user no longer has to load the respective document into the browser (possibly over a WAN) to scan the overview comments.

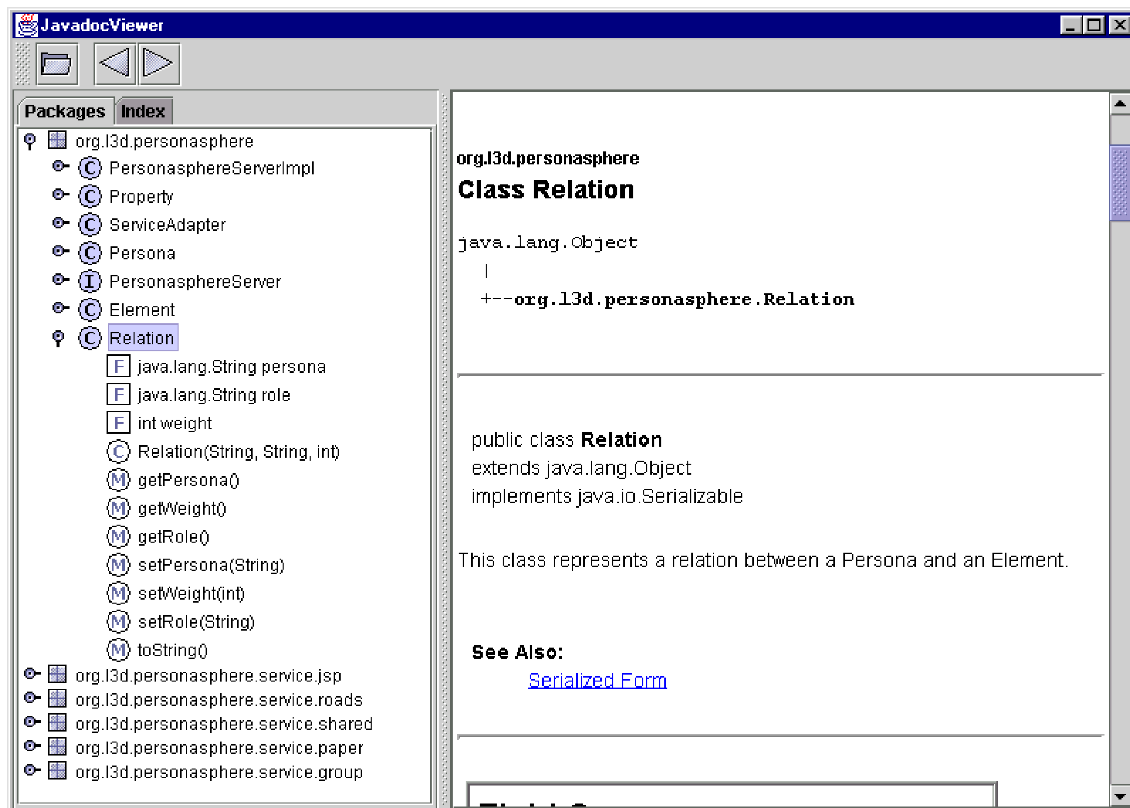


**Figure 5 – Screenshot of the Prototype**

The browser displays HTML documents from arbitrary sources. This not only allows to follow links to external sources (e.g. to a tutorial on the WWW), but also satisfies the requirement, that the viewer should be able to display exisiting documentations.

If the user clicks on a link in the browser that references another element in the same set of documents, the view on the left hand side of the viewer window is updated to show the new position. This is an attempt to improve orientation, but it remains to be proven that it actually does.

The menu bar currently holds three buttons: The "Open" button brings up a file chooser dialog to open a new documentation. The "Back" and "Forward" button implement the usual history behaviour as used in any web browser. The menubar will be extended to hold bookmarking functionality and to add further customizations of the display.

## 6.3 Case study

The prototype has been tried on a small programming project in an attempt to quantify improvements. The project consist of 17 classes in 6 packages, accounting for 250 index entries.

Figure 6 shows some measured file sizes. The numbers show, that the use of the JAR file format reduces the file sizes by a factor of 6, which is a significant improvement. The compressed meta information only increases the total amount of data by less than 10 percent, which is easily made up for by the use of compression. It should be mentioned though that no search index is included yet. This will increase the size of the additional information.

The current status of the prototype did not allow any meaningful time measurements, but

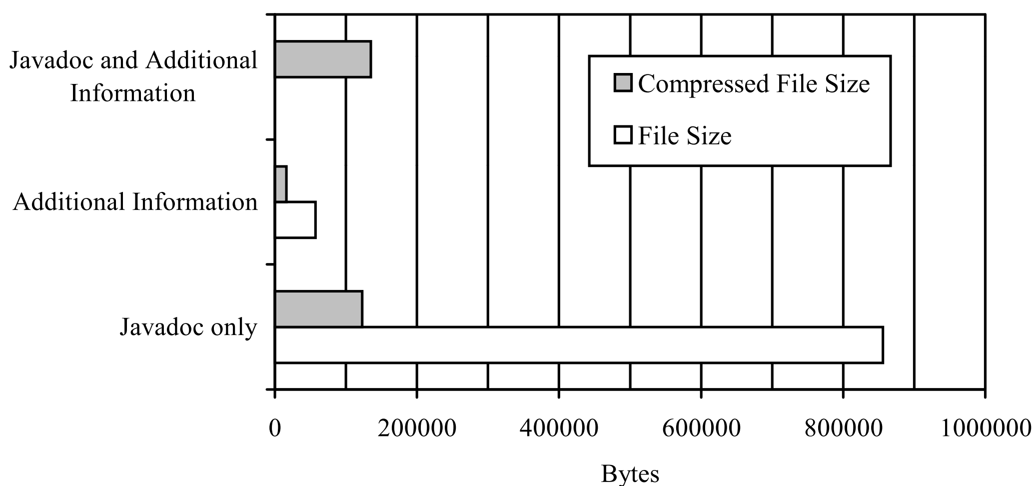


**Figure 6 – File sizes**

the first impression is, that both the Doclet and the viewer perform well.

It showed, that the new architecture easily integrates into existing environments. During production, the only change necessary is the specification of the custom Doclet when invoking Javadoc. The invocation of the viewer is also very easy, it does not even require any parameters (the documentation can be opened from the graphical user interface). Packaging of the viewer as an executable JAR file will further simplify the process.

Figure 5 shows a screenshot of the viewer displaying the project`s Javadoc information. First impressions are, that navigation and orientation have been improved. The tree view and its use of icons seem to make an significant difference.

# 7 CONCLUSIONS

This paper discussed the weaknesses of the Javadoc system and deducted a collection of required improvements. This lead to the design of a new system based on a custom Doclet and a Java viewer.

Though the prototype is in a too early state to make any substantive conclusions, the approach seems promising. The system integrates nicely into the existing environment and is easy to handle. It not only implements some of the obvious improvements over the weaknesses of the existing Javadoc system but the possiblity to add several other features (e.g. the distribution mechanisms as presented in chapter 4) suddenly arises.

A significant increase in usability and, therefore, developer efficiency can be expected from the new system as proposed in this paper.

## REFERENCES


1. *Javadoc Tool Home Page*. http://java.sun.com/products/jdk/javadoc/

2. *Javahelp 1.0 Specification*. Sun Microsystems, Palo Alto, CA, 1998.

3. CONKLIN, J. *Hypertext: An Introduction and Survey*. IEEE Software, September 1987.

4. BROOKS, F. *The Mythical Man-Month*. Addison-Wesley, Reading, Mass., 1995.

5. FILLION, F. and BOYLE, C. *Important issues in Hypertext Documentation Usability*. Proceedings of the ACM Ninth Annual International Conference on Systems Documentation, 1991, Pages 59 – 66.

6. FRIENDLY, L. *The Design of Distributed Hyperlinked Programming Documentation*. International Workshop on Hypermedia Design, 1995.

7. GOSLING, J., JOY, B. and STEELE, G. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1995.

8. GREEN, R. *Component-Based Software Development: Implications for Documentation*. Proceedings on the 17th Annual International Conference on Computer Documentation, 1999, Pages 159 – 164.

9. KRAMER, D. *API documentation from source code comments: a case study of Javadoc*. Proceedings on the 17th Annual International Conference on Computer Documentation, 1999, Pages 147 - 153.

10. PRIESTLEY, M. *Navigation Issues in Hypertext: Documenting Complex Hierarchies with HTML Frames*. Proceedings of the 15th Annual International Conference on Computer Documentation, 1997, Pages 223 – 235.