

*Software development line
IT University of Copenhagen
Copenhagen, Denmark
Supervisor: Kasper Østerbye
Spring 2001*

Elastic JavaDoc

An elastic information system for documentation of Java development

*Written by:
Kasper B. Graversen*

© 2001

Abstract

This report describes the design of a documentation presentation system. The system builds on the thesis that by using a database and impose structure to comments in the Java source code, the system can tailor the information to the users needs.

The report first investigates and groups requirements for documentation presentation systems set up by people conducting research on documentation . The groups are then compared to the requirements set up for the ElasticJavaDoc system (EJD). On the basis of the requirements and the thesis, the EJD is designed. The EJD consists of "server software" handling the requests and a large set of HTML-like tags for creating generic presentations. The tags lets the documentation designer, to a large extent, freely choose design, i.e. using colours and icons instead of textual representation. The tags are furthermore aware of the users current preferences thus reducing or expanding the information output according to this. After describing large parts of the EJD tags the report shows the development of a powerful presentation system named "navigator" which is build in a few pages of tags (roughly the same amount as building an ordinary homepage).

The report draws no well-authenticated conclusions. It should be regarded a preliminary report suggesting a model in which to perceive documentation and suggests a design for an implementation isomorphic with the model.

Note: the report is split into two separate reports, the "EJD" and the "EJD Technological background".

Keywords: Documentation presentation, Javadoc, JSP.

Preface

Project context

This report is the work of one person in the months February–May 2001. The report is rated 15 ETC points. The initial interest was learning JSP and making “building blocks” for large web systems. Through various articles on hypertext the idea for a documentation system was created, with the twist of making the presentation interface for the system highly customizable hence using custom tags and dynamic webpages.

One month was spent reading articles from various areas of computer science, and figuring out what exactly to build building blocks for. One day the idea struck that the combination of documentation presentation and hypertext research could be combined. The end result is a report describing the design of a system. Before the report was written a minimal prototype was developed. The prototype has a broad variety of functionality testing the design ideas.

Software

The software used in developing the EJD and this report was:

The EJD system was developed with Jikes (fast Java compiler), Allaire JRun Evaluation copy (JSP and web server), MySQL (database)

The report was typeset in L^AT_EX using the following packages

Package	Usage
a4wide	Use A4 paper dimensions.
fancyhdr	Changing the headers.
fancyverbatim	Displaying all the code listings.
graphicx	Inclusion of pictures in the report.
longtable	Tables spanning more than one page.
smartref	Referencing chapters, sections, ...
url	Typesetting URLs.
xr	Cross referencing between multiple documents.
BiB _T E _X	Managing the literature used.

Plus some home-made macros and definitions.

The illustrations were created in the freeware tool Dia.

Both LaTeX and Dia are recommendable in developing high quality reports. The Allaire JRun server was chosen over Tomcat as the installation and configuration was painless.

Contents

0	Introduction	1
0.1	Background	1
0.2	Problem statement	2
0.3	Terminology	2
0.4	Prerequisites	2
0.5	The reports structure	3
 Part I Rationale		 4
1	Background	5
1.1	Documentation's role in software development	5
1.1.1	API documentation	5
1.2	Improving documentation	6
1.3	Requirements of documentation and systems	8
1.3.1	Summary	11
1.4	The Javadoc	11
1.4.1	Problems with the Java API	11
1.4.2	Javadoc clones	12
1.5	Problem statement	13
2	The EJD system	14
2.1	Concept	14
2.2	Requirements	15
2.2.1	Elements of the documentation	15
2.2.2	The presentation of the documentation	15
2.2.3	The ease of adaptation to the new system	16
2.2.4	Documentation should be non-text based	16
2.3	Approaches	17
2.4	EJD seen from the users point of view	17
2.5	EJD seen from the documentation authors point of view	18
2.6	Design issues of the EJD	20
2.7	The current implementation	21

Part II	Design	22
3	Documentation from a technical point of view	23
3.1	Documentation in EJD	23
3.2	Documentation in the database	25
3.3	Technical design of EJD	27
4	The custom tag framework	29
4.1	Presentation	29
4.1.1	Restrictional use	31
4.2	How to make a presentation	33
4.3	Navigation	37
4.3.1	The link tag	38
4.4	How to make the class/interface list	39
4.5	The big <link> tag	39
4.6	Extending functionality with links	40
4.7	Internal linking	42
4.8	Configuration	42
4.9	The “navigator” assembled	43
4.10	Summary of the presented tags	45
Part III	Reflections	46
5	Reflections	47
5.1	Further work: General level	48
5.2	Further work: Concrete level	49
6	Conclusion	51

0. Introduction

This report describes the design of a new documentation system mainly for library documentation, that is documentation that describes an API (Application Program Interface) in an object oriented environment. We have created the system "Elastic JavaDoc" mostly referred to simply as "EJD" in the rest of the report. The key concepts of the EJD are

- The use of a dynamic web environment which enables dynamic responses and a high degree of customization of layout and output format.
- The use of an underlying database containing all the information which enables new and thrilling results which may create new views of documentation.

The name was chosen intentionally to give the user associations of a system conforming to "the shape of the user", as a rubber band conforms to the shape it is put around. A more entertaining angle would be to associate "elastic" with "the elastic in my pants that keeps them up".

The concept of placing information in a hypertext context is not a new invention in EJD, according to [Knudsen et al.: 1990], D. C. Engelbart used hypertext to integrate programs and documentation and published an article in 1978. Hypertext is the perfect medium for documentation as it enables "interactive reading". Interactive reading is defined by two properties, reading in a non-continuous way (jumping in the text) and information being tailored to the situation. The Javadoc tool is a positive step in this direction, but as this report shows, the concept is open to improvements.

It is our hope that the ideas presented in this report will inspire others to improve access to documentation in whole.

0.1 Background

The background for EJD set foot not in the area of documentation but in the hypertext world. It was the article "Structures Web Site Design" [Schwabe et al.: 1996] that gave rise to the idea. The article described an approach to building hypermedia applications called "OOHDM" (the case in the article was a site on art named "Portinari Project"). The concept of the OOHDM is to structure underlying information in a OO manner, completely separating information from layout and navigational features. With structured data they were able to reuse information in different situations, i.e. information on a painting (name, year, painting style) could be used in presenting the painting. The same information could just as well be used in building a "painters time line". With the information more than just named nodes on the time line could be created, it could be used to present a time line of paintings in a specific painting style. General information on art through time could easily be combined, and the user could easily combine "how art changed in time" with how the painter "changed in time" in order to find parallels (or maybe the opposite which could be just as interesting). Since the pages were dynamically generated, no special attention was needed on i.e. the time line when adding a new painting to the application since the time line was generated by requesting all paintings by the author sorted in various styles depending on the type of time line.

The concept gave rise to the idea that documentation needed those features as well. Our presumption was, that since programming languages tends be highly structured and cleansed of ambiguity, an implementation had potential of becoming "beautiful" (in technical terms).

To sum up: The “raw material” the EJD processes, the structure of Java is as follows: Packages contains classes and interfaces. Classes and interfaces contains constructors, methods and fields. Classes and interfaces can inherit other interfaces and classes. Constructors and methods takes zero or more parameters while only methods have a return value. Both methods and constructors can throw exceptions (which itself are just classes). On each level (package, class, interface, constructor, method and field) comments can be attached. In conjunction many extra parameters exists specifying type, access modifiers, synchronization issues and so forth. For more details on the language see [Joy et al.: 2000].

0.2 Problem statement

At this point we will not present the problem statement of the report. We feel further background information needs be given in order for understanding the context of the statement. The problem statement is placed in section 1.5 page 13.

0.3 Terminology

The following words have special meaning in the report. To eliminate confusion a short explanation is presented

Word	Meaning
User	Refers to the person or system making use of the EJD, that is browsing the documentation, or otherwise interact with the system.
Clace	Means classes and interfaces. The word is only used in the implementation hence also the custom tags developed. The word was invented since classes (cla) and interfaces (ce) have many properties in common.
EJD	Is both the name of the concepts and design idea described in this report, as well as an actual implementation of the same system.

Finally the end of an example is denoted by the following character: □

The illustrations are seeked held in the standard UML notation. Only non-standard notation which may be misunderstood are explained in the captions.

0.4 Prerequisites

The reader is expected to know the rudimentaries of HTML and the Java language. Having used the Javadoc or browsed the API at <http://java.sun.com/j2se/1.3/docs/api/index>. HTML is highly recommended.

With the report comes an additional report “technology background” which contains rudimentary descriptions of all the existing Javadoc tags. It also touches upon database connectivity, and generation of dynamic webpages using the servlet/JSP technology.

A glance at the descriptions before reading further in this report is recommended. However it can for some readers be necessary to read the additional report in order to understand this report.

Along with the technical prerequisites the report assumes the reader has prior knowledge in the field of documentation. The report touches upon issues on documentation and requirements for documentation presentation systems, but does not elaborate nor discuss the importance of, or the rationale behind, documentation.

0.5 The reports structure

The work has been separated into two reports, the current report and the “technology background” report. The later should be regarded as covering some background knowledge the reader could not be expected to know of.

The current report has been split into three parts.

Part 1, chapter 1–2, The rationale for the EJD system, covering issues on documentation, the requirements for documentation presentation systems and the requirements for the EJD.

Part 2, chapter 3–5, Focuses on the most relevant design issues in the EJD: The structure of comments, the database design and essential parts of the custom tag library.

Part 3, chapter 6–7 Deals with reflections over the EJD project.

Part I

Rationale

This part is the rationale for the EJD system, covering issues on documentation, the requirements for documentation presentation systems and the requirements for the EJD.

1. Background

The chapter presents a view on documentation and its role in software development. This is followed by requirements to documentation systems raised by various authors. The chapter is meant to be an opening to the next chapter, which will present the requirements of the EJD and its visions.

Documentation is important. Programming languages has come a long way in being more readable and being able to model problems in a more natural way. Conceptions and abstractions can be formed as objects, which can interact with each other. Though programmers have syntax highlighting editors there still is a need for documentation to give explanations in human language and preferably on many different levels of abstraction. This is especially true when systems spread over long periods and/or systems become a certain size. The following citation captures the essence.

“Brook’s law shows that the limiting factor in large software projects has stemmed more from communication limitations between programmers than from the difficulty of the programming itself.” [Cordes and Brown: 1991,p. 58]

1.1 Documentation’s role in software development

The need for good documentation is becoming increasingly important. This stems from the increasing size of systems and the increase of reusable code. The later must be ascribed to properties of the OO-programs.

Through time code has been gradually more and more readable and reusable. In the beginning programming was difficult and code were usually kept small. It was clear even back then, that programs divided into small chunks, were easier to develop and maintain. As programming languages developed these chunks of code became routines, subroutines, modules and finally objects. Code gradually became self-contained and easy to describe. Through objects code could be passed around and be reused in various places in a program. These are the historical observations done in [Weiss: 94].

1.1.1 API documentation

One of the strengths of the Object Oriented paradigm is the reuse of code. Samentinger puts it *“Typically, object-oriented systems are not build from scratch”* [Samentinger: n.d.]. Not only will this approach significantly improve productivity but also secure quality of the code. Efficient reuse of components, however, demands detailed information enabling the programmer to first identify proper components, then efficiently enabling him to create instantiations and invoke methods. One of Samentingers points are, that this necessitates a much stricter separation between user- and maintainer- documentation. Programmers must be able to find relevant information such as *“what does this class/method do?”* while maintainers needs information such as *“how does this class/method do X?”*

One of the strengths of Java language is its integral of a huge library of freely available code covering everything from a dynamic growing array to remote method invocation to the creation of dynamic webpages. This library goes by the name “the Java API”. The trenchancy of the API depends solely on how well written, structured and accessible the documentation is. Looking at the API over time (see table 1.1 page 6) this becomes evident. Reuse (in its true meaning, not just toy examples) is only possible when documentation exist.

Version	Packages	Interfaces	Classes
1.02	10	43	202
1.1	22	93	402
1.22	59	317	1462
1.3	76	422	1732

Table 1.1: *The table shows the size of the Java API through time. From release to release the figures roughly increase by a factor 2. Figures are counted from the API's on <http://www.java.sun.com>*

As Berglund notes, more time is spent in the API documentation than in the Java compiler [Berglund: 1999]. When focusing on nice development environments one should consider the need for a nice environment for reading and writing documentation, rather than the editors functionality or the speed of the compiler.

Summary:

Documentation facilities must be looked upon as just as important as any other development tool.

1.2 Improving documentation

In circles of documentation research there are three “truths” which keeps appearing in articles

- Up to 60% of an applications budget goes to maintenance, good documentation will drastically reduce these spendings.
- Programmers do not like producing documentation.
- No one ever reads the documentation available due to the low standards of it. Finding the right information is a tedious task.

We have not researched as to whether these “truths” are actual or have artificially become truth due to their repetition in the literature. Still we will take them seriously. We believe that solving bullet #2 automatically solves bullet #3. But how do we improve the process of making documentation? We believe Berglund touches upon a very important issue

*“If documentation is discussed, it is often viewed as something produced in the project rather than used in the programming task. In the book *Software Engineering with Java*, Schachs only advice is that documentation should be online.” [Berglund: 1999,p. 4]*

This is also touched upon in [Østerbye et al.: 2001] as their 4th of 11 requirements for documentation is “*Writting the documentation should be a natural part of each iteration of the development process*”.

Teaching students the value of documentation and how to write it, and in general, introducing better habits to programmers are solutions both vaguely defined and also yields solutions on the long term scale. We believe that by simply providing better tools for preparing and presenting documentation is a big leap towards a solution. Writers must experience they produce a useful product. Readers must experience an increase in productivity (partly due to a better understanding of the system they are developing on, partly due to finding the right information quicker).

We believe the quality of documentation can be elevated simply by imposing structure on it. These structure definitions should come from the documentation presentation application. Some regard programmers as “anarchist” not wanting to conform to any pattern or template. This might hold for some, however it is our hope, that experiencing the strengths of structure will convince them to change their behaviour. Programmers already accommodated toward structure will find the EJD requirements “a system giving freedom under certain control”.

[Powell et al.: 1996] describes the situation where software engineers spend great time and effort preparing documents without concrete guidelines and with little assurance of the usefulness of these documents. The result being inadequate or inappropriate documentation, frustrated software engineers and complaints from users. Andersen et al describes a situation with several authors co-working in an unstructured lucidative documentation environment

“The writer always had to come up with his own structure for a given documentation task, and the resulting structure differed a lot depending on which writer produced the documentation. The combination of documentation from different writers led to readers being left with the impression of a confusing and unstructured document.” [Andersen et al.: 1999,p. 14]

Knudsen et al points out that extra semantics (rather than just the syntax and semantic control performed by a compiler) are to be used in order to write a program that uses existing classes correctly [Knudsen et al.: 1990].

The kind of structure we propose is to split an otherwise “flat” documentation into small easy identifiable chunks. By imposing structure within the documentation, **the writer is guided** in the writing process. Imagine being given a blank sheet of paper and told to describe the story of your life. How many details would be forgotten and not noted down or even knowingly ignored due to their “irrelevancy”. The example can directly be transferred to programming issues. The effect of “filling out forms” poses important questions to the code such as: “is a design pattern used? Does a pattern exist for this problem?”, “is this piece of code thread safe?” or “what are the post condition, and do they always hold in the way the code is used?”. An especially nice side effect of structuring the documentation is that it **creates navigation facilities** for the reader. The structure can easily be understood by a program which then can present the documentation according to the readers wishes.

The final argument for structure stems from programming experiences. When a person starts learning OO-programming, focus is on learning the language syntax and how to create and use objects. Getting more experienced focus is on structuring the code in hierarchies and packages (modules). The truly experienced programmer is taught that hierarchies are not enough. The requirements to the application being developed change over time, new requirements arise. The programmer is taught to think in general terms rather than taking a narrow focus on the problem being solved. But how many programmers evolve to also taking documentation to this level? How much documentation is written in a form that foresees the code might be reused in another context, or that the code must be adapted to new requirements not taken into account at development time? Our presumption is that documentation tends to be focused on the context in which the code is being written and what specific problem in that context it solves. Documenting thread issues or runtime complexity in code where those issues are irrelevant (at that point in time) will most likely not be written. **By providing a skeleton structure the documentation depends less on how the writer wishes to use the documentation.**

Summary:

We have argued structured documentation superior to unstructured documentation as it provides guiding to the author, navigability to the presentation system and finally ease reusability of code.

1.3 Requirements of documentation and systems

Much research has been conducted in the area of documentation. In general the literature agrees on differentiating documentation for different kind of users, however the granularity of this differentiation varies from author to author. More interestingly, in the context of this project, people set up requirements for what kind of information needed be presented to the programmer, and how it should be presented. All the points will not be formally covered when comparing it to the EJD system, rather this section is supposed to give rise to inspiration to this and other systems in development. Furthermore the section shows the diversity of demands a documentation system must respect. A lot of considerations must be taken into account. A documentation presentation system is not just some system spitting out information.

“Documenting Object Oriented Systems” [Knudsen et al.: 1990] talks about “descriptions” and “extra-semantic constraints”. That is natural language descriptions associated with program structure, and semantic constraints which are not expressed by the programming language. These are

Structural information Name, super class, instantiation-time parameters.

Description Two parts, first a one-sentence condensed description and a description in a more fully expanded form — both descriptions being in a natural language.

Categorization Divided into classes, patterns, exception patterns and such.

Usage The intended usage of a class, is it abstract (never instantiated), object, etc.

Terminology Establish a special local terminology regarding the concept and its model.

Configuration How classes are interrelated and intended to be instantiated.

Assertions Extra dynamic extra-semantic constraints, i.e. pre, post and invariant conditions

Operations Documentation of every method in a class, both parameters it is called with, and the result of the call.

Intentions The intentions of the code i.e. When is it used? How should it be extended.

Sequencing Describing the order for use of methods.

All these are needed to write a program that uses an existing class correctly. “Documentation in object-oriented systems and frameworks” [Østerbye et al.: 2001] sets forth 11 requirements for documentation.

1. The volume of documentation should be minimal.
2. Accessibility of documentation (which becomes more and more distributed geographically) should be taken into consideration.
3. New developers should be able to understand the static and dynamic structure of the system.

4. Documentation should be a natural part of each iteration cycle in the development process.
5. Minimize redundancies.
6. Traceability, i.e. it should be possible to identify how and where each requirement has influenced the design model.
7. The main abstractions and general design decisions should be easily recognizable.
8. The documentation should be separated in sections describing different aspects.
9. Documentation should include a domain model description.
10. A concise description of the systems interface (input/output).
11. Events, how and when does the system react on events.

“Building Documentation Generators” [van Deursen and Kuipers: 1999] focuses on the presentation of the documentation and introduces four criterions;

1. Documentation should be available on different levels of abstraction.
2. Documentation users must be able to move smoothly from one level of abstraction to another, without losing their position in the documentation.
3. The different levels of abstraction must be meaningful for the intended documentation users.
4. The documentation needs to be consistent with the source code at all times

They note criterion 3 makes documentation generation hard, as design information usually are not present in the source code and must be added manually (applies mostly to legacy systems which they concentrate on in their report). Criterion 4 implies more or less, that documentation is generated directly from the source code, which especially in legacy systems is violated.

“Pragmatic approach to software documentation” [Didrich and Klein: 1996] concentrate their efforts on setting goals that will make acceptance of the new documentation system easy as possible for the program developers. They propose seven essential goals their documenting system should adhere.

1. Exploit the inherent documentation as much as possible.
2. Provide a convenient documentation environment providing support to the documentation author. This is accomplished by embedding their system into existing systems.
3. The documentation should be configurable, i.e. by special ordering of modules, hiding subsystems etc.
4. Avoid sacrificing old code, the transition to the new system must be as smooth possible. The old code should work unmodified or at least be incorporated easily.
5. The system must be able to handle large structured implementations.
6. Provide different kinds of documentation, not tailored specifically to every person, but allowing the user to zero in on what they want to see.
7. The product must be impressive, convincing the new users that the system is useful and thus motivating them to using it. The system should be useable to as many people possible, but especially to the programmers who are writing the documentation.

Furthermore, the authors speak of four different kinds of documentation;

Ordinary Ordinary documentation.

Tagged Ordinary documentation with tags – tags can then lead to generating documentation for different audiences.

Sectioning Splitting up documentation in up to four levels.

Hidden Documentation that is not included in the generated documentation.

As the EJD works a little differently the types of documentation is reduced to only three: “Ordinary”, “Tagged” and “Hidden”.

“Concepts and tools”[Welsh and Han: 1993] elaborate on the output of online material, where the following points will significantly aid understanding.

1. Presenting documents as a sequence or hierarchy of meaningful views rather than as flat text.
2. Providing multiple alternative view formats for particular purposes.
3. Formatting each view with respect to its inherent structure.
4. Abstracting or suppressing detail to offset size limitations both in the screen used to display views and the brains to perceive them.
5. Enabling navigation between views via the implicit and explicit relationships than exists.
6. Enabling searching with respect to textual or semantic patterns.

Hitchcock in [Wang and Hitchcock: 1992] contributes with the following points (points already mentioned have been omitted).

1. Ability to recover from any mistake made either by a user or by the system
2. Consistency checking among documents.
3. Ability to create versions of a document type.
4. Application independence.
5. Data sharing among a group of users.

Finally Shih et al has an interesting twist in their requirements. They argue against the whole idea of having documentation imbedded in the code or put in a separate file!

“Text-based documentation is unfriendly and hard to understand. It makes original programmers unwilling to annotate and newcomers reluctant to read.” [Shih et al.: 1999,p. 286]

Moreover they claim that the textual format has two additional noteworthy drawbacks. First embedded comments are restricted in length and numbers — too long or too many comments blur the code. Nevertheless omitting comments may bring the documentation to an incomplete state. Secondly, and even worse, multimedia information can not be represented as text. Their solution is to propose a WYSIWYG system with support for pictures, sound and movies (called “DocMetrics System”).

The somewhat radical idea of having (parts of) the documentation as multimedia is an interesting suggestion! However we wish to keep the EJD a non-WYSIWYG system for reasons elaborated on in section 2.2.4 page 16.

1.3.1 Summary

We have presented requirements from many sources, set up by people with different backgrounds. Evidently specifications concentrated on the documentation itself are inadequate. Based on the many requirements, we classify the demands into four groups.

1. Elements of the documentation.
2. The presentation of the documentation.
3. The ease of adaption to the new system.
4. Documentation should be non-text based.

1.4 The Javadoc

The Javadoc tool is influenced by the literate programming idea[Friendly: 1995]. Java code and documentation is placed in the same file, which means there is a high proximity between documentation and code. All documentation within `/**` and `*/` is extracted by the Javadoc tool and placed in automatically generated HTML files which then can be browsed. These HTML files are also known as the Java API.

The Javadoc tool utilizes that Java is a structured programming language. The tool examines all classes and interfaces, and within these methods, parameters etc. While examining the internals of an element the tool also looks for the special comment above the element. If any is found, the documentation is placed with the element in the HTML file. Along with utilizing information read directly from the code, the javadoc tool has structured the documentation into named chunks. Describing a parameter `i` to method `F(int i)` is done by placing a `@param description text` in the comment just above the method.

1.4.1 Problems with the Java API

Based on our experience with using the Java API, we have identified problems which needs attention.

- Berglund points out that the Javadoc output contains excessive information in all work situations, thereby blurring the users view. [Berglund: 1999]
- The HTML output is static. If one has chosen to access only public classes, one can not choose to access i.e. private methods without first running the Javadoc tool with new parameters. This makes the output hard to use optimally for people with multiple roles, i.e. maintaining one part of the Java API but using another part of the API in the process.
- There are far to few named chunks. Clearly dividing the documentation into user and maintainer parts as Samentinger and others has suggested is impossible. Listing bugs, to-do's, labelling methods to be thread safe etc. is "impossible". The information can be noted but in an unstructured way which gives rise to the "excessive information" problem stated above along with sacrificing navigation facilities.

- The author has no control over the order of the documentation parts. All entities are sorted in alphabetical order disregarding the fact that less important information may show before more important information. It would be useful to let the author group information (i.e. methods and variables) into several different groups and let the reader choose the ordering of the information. The alphabetical sorting should however not be disregarded. Alphabetical ordering is useful in some situations i.e. grouping overloaded methods and manual search.
- As an extension to the previous point, there is no way to hide one or more methods or constants. Limiting the number of entities on the screen helps especially new reusers of the class. For instance, a student learning how to program finds the `java.lang.String` class for solving a simple exercise. In the v1.3 edition of the API the user are presented with 11 different constructors (of which two are deprecated and hence especially irrelevant for newcomers to the language), and a sheer 48 methods (of which one is deprecated).

Taking the thought even further; imagine being completely new to a programming language and being presented with a 2150 classes/interfaces big API (see table 1.1 page 6). It is an infeasible task to find the right class. Without guidance the user is completely lost. Admittedly experienced users have learned to travel the API, yet they too are set in a complicated situation, when they need to reuse a new class and is presented with seemingly hundreds of methods. To make things worse it is more common that a framework of classes are to be reused rather than a single class. This stems from the fact, that well-designed systems solving a given problem often consist of several classes.

An example: In the development of a tool (JavaMiner doclet) the part reading Java source code used more than 18 classes/interfaces from the API. Another part which did database connectivity used 5 different classes from the API.

It should be noted that documentation introducing i.e. the 18 classes/interfaces is *tutorial documentation*. Tutorial documentation is a completely different kind of documentation which is not covered in the EJD problem domain.

We suggest improving support for new reusers of a class/interface should be developed.

1.4.2 Javadoc clones

The Java language has support for programmers writing a plugin for the Javadoc tool (a so-called “doclet”). The package `com.sun.javadoc` has everything needed to complete the task (an elaboration on how to create doclets is found in chapter 3 page 19 in the technical report). From the Java doclet support homepage¹ one can navigate to the FAQ section² which presents all third party doclets and tools submitted to SUN. The submissions are split in “doclets” and “Javadoc Tools and Custom Tags”. In order for the reader to get an overview, we have categorized the submissions into five groups. The information for the categorization was gathered the May 3rd 2001. Note tools submitted which did not have documentation as their problem domain have been omitted from the list.

Output Changed the output to i.e. \LaTeX , GNU Emacs TexInfo, or RTF. This category is where the most doclets are categorized.

Convert Converted files to other vendors i.e. “Java2Rose” which converts Java files to Rational Rose format.

Validator Checks the “quality” of the java files i.e. finding methods missing the `@param` tag.

¹residing at <http://java.sun.com/j2se/javadoc/>

²residing at <http://java.sun.com/j2se/javadoc/faq.HTML>

Expansion Expands the “vocabulary” of the doclet to also understand i.e. @pre, @post, @inv tags.

Change Changes or improves the Javadoc. In this category falls only Erik Bergmans “DJavadoc” system.

The EJD project has its place in both the ‘expansion’ and ‘change’ category, by adding both new tags as well as take a new dynamic approach to documentation reading.

One must expect that the third party submission list more or less covers what has been developed, since the list is an easy way of exposing ones work. It seems as if people have not yet focused on Java documentation. Apart from looking at the submission page, we found only Andy Cockburns “Jaba” system[Cockburn: april 2000], which has a different approach being a source browser with fisheye, holophrasting and code views, rather than a documentation presentation system.

On the basis of this examination we find that there is a need for the EJD system, since the tools found in the examination do not address the problems presented in section 1.4.1.

1.5 Problem statement

It is clear that the many requirements for the documentation system can not be solved by static HTML pages in an acceptable way. We however do not regard the technique for producing API documentation inadequate. The high proximity between documentation and source code makes it easy to keep the two isomorphic.

It is our hypothesis that in utilizing a database for information retrieval in conjunction with a dynamic system, we can solve many of the problems stated in the last sections. Further, we believe that by providing a skeleton structuring the documentation, we can tailor the information to the needs of the user.

2. The EJD system

The chapter describes the EJD system from the following perspectives: Concept, Requirements, the user of the system and documentation from the authors point of view.

All the criteria presented in the former chapter are interesting, however not all makes sense implementing in the EJD system, i.e. Østerbye et al's criterion #1 which states documentation should be held minimal. We see this as a guideline to the writer. Instead of going through all the requirements one by one, we describe the requirements of EJD based on the four types found in the summary section 1.3.1 page 11.

2.1 Concept

Before going into detailed requirements we first lay out the concept and consolidation of the EJD. We hope that the concepts shines through the requirements set up.

The concept of EJD is that documentation systems are more than gathering information, it is structuring and presenting it. Structure (and a proper fetching machinery in the back end) brings searchable and terse documentation.

The presentation should be looked upon as a system which can give "intelligent" feedback by combining information from several places in the system.

There are five concepts which are the consolidation of the EJD system.

- By introducing many new comment tags, the comments become structured and navigable and acts as documentation rather than comments (which we see as inferior).
- By storing many syntactic and semantic structures, the user can make completely new and excitatory requests to the API. As "references" in a relational database "points both ways" one can easily find i.e. all classes implementing an interface, or find all methods containing a @bug tag (indicating bugs are present in that method).
- By having a system which can "reach" into other parts of the system, i.e. the system could implement Samentingers idea of code inheriting the comments from their ancestors.
- By having users on the system. This enables logging of users habits and conforming the presentation towards this. If the user only browses 10 classes on a regular basis he might find the list of those classes more relevant (most of the time) than the list of all classes in the system. Also user profiles could be joined showing i.e. the most browsed classes of the user and his co-workers working on the same part of the system.
- The presentation layer is a completely new framework of HTML-like tags that has been developed. This enables the user to totally configure the layout of the presentation — even substituting certain parts of textual information with graphics or translation into his native tongue.

A spin-off of this approach has made the framework more powerful than simply adjusting information. With the set of tags developed the user can create his own requests suited for the situation without coding new parts of the system. Additionally needed is the composition of a new JSP page which then must be put on the server.

2.2 Requirements

The following four subsections corresponds to the four groups of requirements found in the last chapters summary.

2.2.1 Elements of the documentation

This topic covers all the types of elements the documentation should consist of. These are both elements “physically” found in the documentation as well as indirect elements such as traceability of change requests.

- The system must bridge the documentation to process oriented aspects of software development. This could be domain models and change request traceability.
- Must support abstract design expressions such as design patterns, algorithms and data structures. The names of these structures are the vocabulary of the modern programmer. Further more it groups methods and fields belonging to the same abstract design.
- It must support short descriptions describing the system from many angles, i.e. internal documentation, multi threaded code, runtime complexity and blocking code.
- The system should **not** support FAQ, example and tutorial inlined in the sourcecode i.e. as @example, @tutorial, @FAQ since their size easily grows to many pages thereby blurring the code.

2.2.2 The presentation of the documentation

Which abilities should the system have in presenting the documentation.

- The system must conform the amount and type of information to the users needs, i.e. reuser, maintainer, project manager. Each of these groups may be further differentiated.
Some user roles may not be restricted to the documentation alone, they may benefit from seeing the sourcecode in cases where documentation is vague or missing.
Other user roles may benefit from being able to edit the source code. For instance when being presented with all the todo's or bugs in the system, easy access to editing facilities needs arise. This will promote the system to being both a “passive” information retrieval device, and an “active” system for i.e. bug handling.
- The system must be aware of its users and on the users request tailor its output to the users practice, i.e. at startup giving him the most browsed classes/interfaces rather than all the classes/interfaces in the system.
- The system must be able to output the information in several formats — the system could be used both as a hypertext information system as well as generating documentation ready for print. A free combination could be HTML as hypertext format and \LaTeX for printed documentation as both languages are textual markup languages.
- The system must be able to use information or documentation from other parts of the system. This enables “intelligent” queries such as “find all todos in the system with the author name J. Doe”, “how many methods throws this legacy exception”, “when having a reference to object of type A how to get a reference to some object of type B?” or inheriting documentation from super classes.
- It must display the information in a highly configurable way.

2.2.3 The ease of adaptation to the new system

How easy should it be for a new user to use the system.

- The EJD must be able to read and understand documentation tailored the original javadoc tool. However, To get the full potential out of the documentation, adding EJD tags can not be avoided. Likewise the documentation for EJD should be usable in the ordinary JavaDoc tool.
- The EJD tool should be able to run using standard software. A relational database and a JSP webserver is sufficient.
- The EJD system should have a role of being an “external system”, which does not force the programmer into using a new kind of editor (which most probably do not have the features found in the editor he currently is using). The system should also not run only on certain operating systems, as changing operating system often demands much more than changing an editor.
- The configuration possibilities of the system should be restrainable on user level. Some types of users, beginners to programming for instance, should not be presented with too advanced properties of the system or those aspects of the documentation.

2.2.4 Documentation should be non-text based

Should the underlying format for the documentation be based on text or a visual system.

We disagree with [Shih et al.: 1999] in that the documentation should be based on multimedia files rather than textfiles. Firstly, the text-based system works directly with existing Java code and does not enforce radical change to the working process. Using a visual system requires the programmer to adapt the new system into his routines of programming.

Secondly in case of the rise of a better system, switching systems is a matter of “search’n replace” (provided the new system also use textual configurations).

Third, as programmers wish for control, a clean textual environment is easy to backup, easy to recover damaged files and concurrent versions system such as CVS are fully serviceable. Finally all sorts of gadget tools can easily be developed: A company wanting traceability in the code could develop a type of control software that verifies every class, method, etc. contains an author and the name equals a current or a former employee.

We agree that using multimedia as part of the documentation arsenal is fruitful, however we believe the underlying system should be kept textual.

We suggest a hybrid solution: Multimedia parts be placed in separate files. Special tags could be developed which either inlined the multimedia parts or created links which launch external viewers. Using HTML and a modern browser inlining multimedia files should be unproblematic. Also an information file should accompany each multimedia file so the system could establish index and navigation facilities for these files on the same level as the textual documentation.

2.3 Approaches

The EJD uses two approaches to gather information. Deduction from the source code and introduction to new tags in the documentation. While this project only extracts the same information as the original JavaDoc tool does (simply because this tool was used to retrieve information), we are convinced that extracting even more information is a fruitful path to follow and should be investigated in the future. Collecting all calls to external classes could be used to visualize the dependencies in the program. It could reveal dead code or give suggestions as to how many parts of the system would suffer or need change when changing the behaviour of a certain class.

2.4 EJD seen from the users point of view

The EJD system consists of a database and a JSP webserver. Additionally, an editor is needed for developing JSP pages and a browser is needed to display the results generated by the JSP pages. From a users point of view the EJD is a two phase system: a preparation phase and an use phase. The phases are illustrated on figure 2.1.

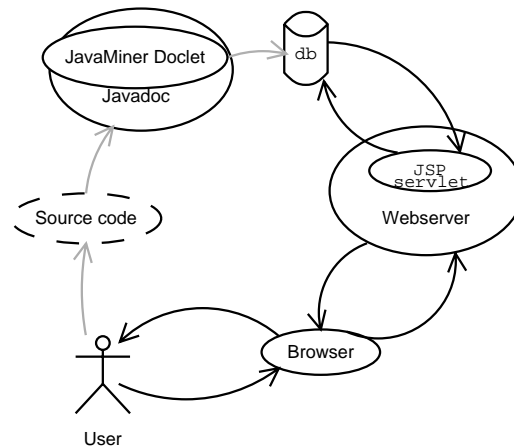


Figure 2.1: *The colour of the arrows denote the phase they belong to, where the bright colour denotes the first of the two phases.*

In the preparation phase the user feeds raw data to the system. This is done with the Javadoc tool (which comes with Java) and the javaminer doclet. The doclet, which is a plugin changing the behaviour of the Javadoc tool, interprets the Java source code and feeds it to the database.

The use phase is where the most time is spent. This is where the reading of documentation takes place. If the user wishes to change how detailed the output is, he can either change his settings in the EJD or develop a new JSP page for a different kind of presentation.

Comparing these steps to using the Java API, the process of turning code into API is not very different and involves just as many steps. Normally one would use the JavaDoc tool on the source code to generate static HTML pages, and afterwards browsing these with a browser.

2.5 EJD seen from the documentation authors point of view

From the authors point of view there is really no difference between EJD and the traditional system. Documentation is still inlined in the source code, and the documentation is divided into chunks by the use of the divider tag (`@name`). The only difference is that under the EJD system far more divider tags are available and the use of as many as possible is imperative to the quality of the presentations the system produces.

As minimum change is required from the author we now focus on the new divider tags. The following selection of tags has not systematically been chosen among a bigger set of tags. The tags are chosen to display how exuberant the system is capable of being and not by how revolutionising the feature of the tags are. Still a system supporting these tags (although there might be serious flaws in the selection of tags) the system would be far superior to the static HTML pages the Javadoc tool produce. Along with these tags the tags from the original Java API should be considered included. The list and explanations of the original tags are found in chapter 1 in the background report. The first line of every tag below shows the argument which must be given. Arguments inside [] are optional.

Miscellaneous tags

These tags did not fit into any group (or were one member groups) thus gathered here.

@bug [#] [minor|medium|severe] description

Denotes a bug in the system. A bug number, its severity and a description of the symptoms can be noted. Placing the comment in above a class denotes a bug somewhere in that class, placing it above a method denotes a bug in the method.

@complexity description

Denotes the complexity of the method either in O , Ω , Θ or other standard notation forms. This gives the user an idea about the cost of calling the method. The main usage is probably comparing to entities which performs the same thing. It should not be considered as a substitution for a profiler. Programmers are notoriously bad at guessing where time is spent in an application hence they should let a profiler calculate it. An example is that `java.lang.String.size()` returns the length of a string by returning a variable in the `String` object. The same method in C (`strlen()`) traversed the whole string character by character. A for-loop in C containing a string length as stop-parameter would be much slower than manually storing the size outside the loop. However, this optimization in Java would yield no performance gain.

@internal description

The tag marks the documentation internal, for the maintainers of the code. Although the documentation is internal it should not be mistaken for elucidative documentation as elucidative documentation covers far more than this.

@job # [description]

Specifying a codechange due to a change request with the specified number #.

@supercomment { taglist }

Is specified with a comma separated list defining which documentation chunks should be concatenated from the super class documentation into this documentation. The system might be set up to automatically include certain tags, i.e. the `@bug` tag.

@todo description

During the development of one class the need for changing or extending other classes can arise. With this tag notes can be attached to the code to be modified. Later these notes can be retrieved i.e. with a required value of the `@author` tag.

Abstract design

This group bridges the gap between abstract design vocabulary and source code. It should be noted, that these tags also do excellently in directing the presentation. Methods in a class could be grouped by their inter-relationship.

@algorithm [#] name [description]

Specify the name of the algorithm used. The optional number groups together elements with the same number.

@datastructure [#] name [description]

Specify the name of the datastructure used. The optional number groups together elements with the same number.

@domain name [description]

Specify the name of the domain the code is a part of. This refers to the domain model commonly developed before coding. No number is needed as the domain model parts have unique names.

@pattern [#] name [short description]

Specify the name of the design pattern used. The optional number groups together elements with the same number.

Multithread issues

These tags cover issues in multithreaded code. The `@threadsafe/@notthreadsafe` tags supplement the `synchronized` keyword in the Java language. Using the `synchronized` keyword slows down the code, so using it on thread safe code would be silly.

@blocking [description]

Denote that the call can take infinite time (i.e. a user request) hence the call should probably be made by a thread.

@threadsafe [description]

Notes the code is thread safe. An explanation as to the precautions taken to make the code safe can be made.

@notthreadsafe [description]

Denotes the code is not thread safe. Some code either can not be synchronized or the wish to achieve speed is stronger than the need for thread safe code. The description field is where an explanation as to why `synchronized` is not used can be provided. Example of such code could be the `java.awt.Graphics.drawLine()`.

Ordering the presentation

The following four tags are for controlling the ordering of entities. Along with these tags the tags in "Abstract design" can be used for grouping as well.

@category name

Groups more constants, methods and constructors. The user can choose to see listings ordered by grouping rather than alphabetic ordering. The tag is opposite the `@domain` which groups classes.

@marshall #

The number supplied with the tags determines the position of the entity in the list. The tag lets the user see the important methods before other methods. It can also be used to organize constants in the hierarchy they represent.

The tag operates solely on numbers while the @category is for textual categorization of entities.

@level 1

The tag describes the level of importance the entity plays in understanding the class. Level 1 methods and fields are the small set of methods and fields which are essential to the new reuser of the class. The `java.lang.String` class mentioned in section 1.4.1 could use such a tag to present a total of maybe 8 methods. Currently only one value has been assigned a meaning, in the future more levels might show valuable

@sequence name|none

The sequence tag can be used to describe the a required order of calls to method. The tag may also prove useful to make “flowcharts” of the inner workings of a class. It could reveal all the entry points of the class and show the possible ways further from here.

The “name” is a list of methods to be called before the current method. If there are no predecessor the value “none” should be assigned.

Design by contract

Lately more tools supporting “design by contract” (DBC) has come to the attention of the public. As Java does not support specification of pre-, post-conditions and invariants, the tools enable programmers to place such code in the @pre, @pos and @inv comments above the methods. When compiling code with their tools, the conditions are inlined into the code and the final compilation can be made.

There is no reason why EJD should not support these actions towards the design by contract idea. However, we take it one step further. As documentation is code on a higher level, and as some conditions can be difficult to express, we suggest three additional tags. The @predoc, @postdoc and @invdoc which are used for specifying the conditions in textual form.

@pre description

@predoc description

@post description

@postdoc description

@inv description

@invdoc description

2.6 Design issues of the EJD

There lies many design issues ahead when building a system such as the EJD. We have split the design into eight parts, of which most will be focused upon in the following chapters.

- The design of tables in the relational database.
- A JavaMiner doclet which reads the source code and inserts it into the database.

- The basic framework which is the “skeleton application”. It consists of a few servlets and JSP pages and a database fetching mechanism.
- A Custom Tag framework.
- A query language minimising superfluous data transfers.
- How to extract “flat” information from the relational database and return it as structured information.
- Users in the system and their configuration abilities.
- Custom formatation codes in the documentation to enable formatted cross-platform documentation.

2.7 The current implementation

The current implementation has not fully implemented the design. At the present stage, we have created the JavaMiner doclet, set up the application skeleton and made a few tags. It is possible to see a list of all classes in the system, clicking a class name brings up a list of all methods in that class in another window. At this point the implementation can only establish credibility that the design will work.

Part II

Design

This part focuses on the most relevant design issues in the EJD. These are:
The structure of comments in Java source code, the database design and essential parts of the custom tag library.

3. Documentation from a technical point of view

3.1 Documentation in EJD

Before discussing design issues it is imperative to first understand fundamentals of Java documentation. The following schema is important since it laid ground for the design of the EJD. Its shape illustrates how we perceive documentation and its structured.

Due to the complexity of how and where comments can be placed in Java code, describing by example is an impossibility. We found that describing documentation in EBNF (Extended Bachus-Naur Form) resulted in a terse description. In a sense this approach regards the structure of comments as a language itself. To reduce the size of the table we have removed most @tags since they all would be put in the <comment> definition and this not adding new information.

The information contained in the table is very condensed, so reading and fully understanding it may take longer than expected.

Nonterminal	Definition
S	→ {<package>} {<class> <interface>}
package	→ [<comment>] Name
class	→ [<comment>] [InPackage] Superclass [(implements interfaces)] [abstract] <modifiers> Name {<constructor>} {<method>} {<field>} [inherited methods]
interface	→ [<comment>] [InPackage] [(implements interfaces)] <modifiers> Name {<method>} {<field>} [inherited classes]
constructor	→ [<comment>] [synchronized] <modifiers> Name <signature> [(throws)]
method	→ [<comment>] [synchronized] [abstract] <modifiers> ReturnType ReturnTypeDimension Name <parameter> [(throws)]
throws	→ classname
field	→ [<comment>] [IsTransient] [IsVolatile] <modifiers> Type Name
parameter	→ {Type dimension qualifiedName}
comment	→ FirstSentence comment {@author} [@deprecated] {@exception} ...
modifiers	→ [static] [final] [public private protected package]
inherited-classes	→ 'for each class upto and including Object:' classname
implements interfaces	→ {interface name }

Table 3.1: The EBNF of how we conceive it.

As can be read from the table, the set of documentations (S) consist of zero or more packages, zero or more classes and zero or more interfaces. A package contains a comment and a name. A class consist of a comment, it may optionally reside in a package, have a superclass, implement zero or more interfaces, may be abstract, have modifiers, a name (name including package name), have zero or more constructors, zero or more methods, zero or more fields, and finally may inherit methods from its superclass. Deducting the content of the rest of the nonterminals is for the reader to do.

Graphically this is illustrated in the following figure. Note that for simplicity reasons all arrows to the comment entity has been removed (all other entities has an arrow to the comment entity).

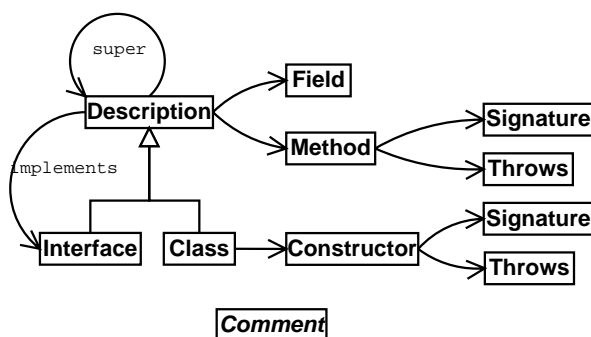


Figure 3.1: A graphical overview of how we conceive the structure of Java comments. For simplification arrows to the comment entity is omitted, as all other entities has an arrow to it.

3.2 Documentation in the database

We now take a brief look at the design of the database. Understanding the structure of the database gives an understanding of how easy (or hard) information retrieval can be.

The design of the database is a straight forward implementation of table 3.1. For every non-terminal in the table, a table in the database is created. Every terminal in the table resulted in the creation of a field in the database table. However automatic this process may seem the design was limited by two important restrictions.

- As the HTML version of the Java API is the size of 93 MB and the source code 18 MB, redundant data should be avoided.
- As the number of “@tag” probably would grow in the future, thus the comment table should be expansive.

As methods and constructors are overloadable a method/constructor is only identifiable in conjunction with its parameters. To prevent having the primary key be a combination of zero or more entries in another table, we gave the methods/constructors an id. All the fields were given unique names (by concatenating part of their entity name) in order to shorten the length of request made when extraction of data is to be made. To clarify some of the fieldnames we will shortly explain some of the abbreviations.

com.id A id of a comment table.

cm.id A Constructor-Method id, to avoid any confusion the two tables were enumerated with the same counter.

displayname The fully qualified name of methods, constructors, classes and interfaces stripped down to only contain the name.

Redundancy

When focusing on redundancy in the database design many tables are created, and so many tables must be joined for most requests. On the other hand by gathering the data in big tables, requests could first of all be unnecessary hard to express. Secondly, it would require a lot of manual joining and reduction of the results — jobs performed more efficiently by the database. EJD needs to be suitable for almost any kind of request we chose to normalize the tables.

The comment table

At first we designed the comment table as any other table in the system, for every @tag we made an entry in the table. We soon ran into problems with tags containing more than one chunk of information, i.e. @param. For those “special tags” we created an additional table to express this multiplicity.

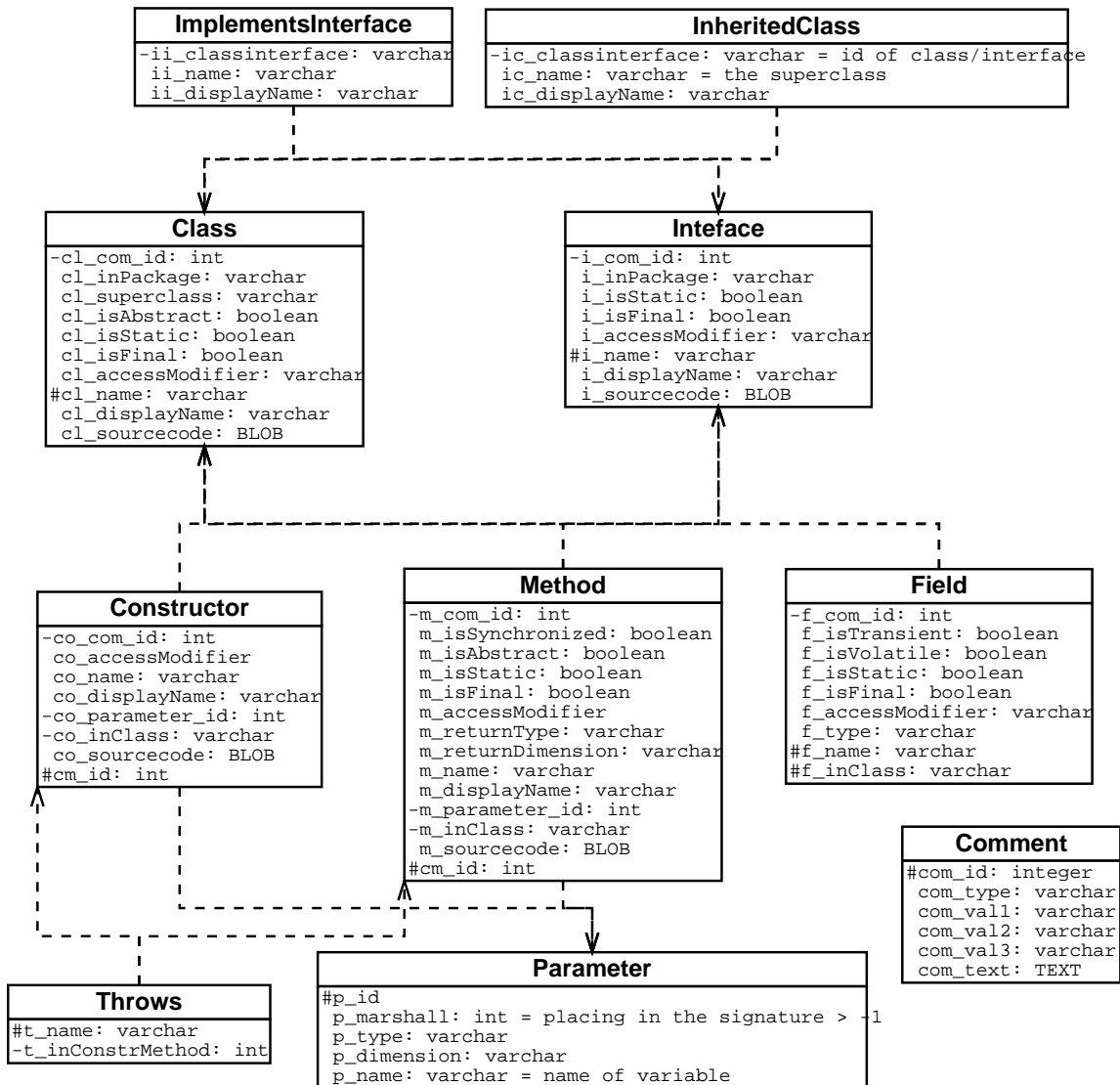


Figure 3.2: The design of the relational database. The “-” denotes a foreign keys, while “#” denotes the primary keys. The arrows denote the cardinality of the entries. Note all arrows pointing a t table comment are omitted since all tables has an arrow to the table.

It was clear that the design was not only clumsy it also necessitated a re-write of the table for every new tag emerging in the EJD. The solution was to regard the comment table as a generic container. We changed the table into a six-entry table. An id, an entry describing the type of tag (this being the tagname excluding the @), and finally four entries of various type to hold information. Most fields are of type varchar, although limited in size to 255 bytes, they allow efficient search facilities. For storage of more information the com_text field allows up to 4 gig of data (this figure varies somewhat from database to database). table 3.2 page 27 shows how different tags are contained in the comment table.

The only drawback of the design is that it limits the tags to having only four kinds of different data.

Type	Val1	Val2	Val3	Text
firstline				The first line of the comment outside any “@tag”-block.
comment				The full comment stripped from any “@tag”-blocks.
author	name			
bug	number	severity		description
deprecated	deprecated text			
domain	domain name			
internal				internal doc. text
param	parameter name			description
pattern	name	affected		Pattern desc.
serialfield	field name	field type		field desc.
todo				todo text

Table 3.2: The table shows how the comment is split into five groups. The splitting is straightforward; the type goes in column 1. The description goes in text and the rest are filled in the order given to the @tagname.

3.3 Technical design of EJD

Based on of servlets and JSP (described in chapter 1 page 3 in the technical report), we now roughly illustrate how the EJD handles a request from a user already logged into the system.

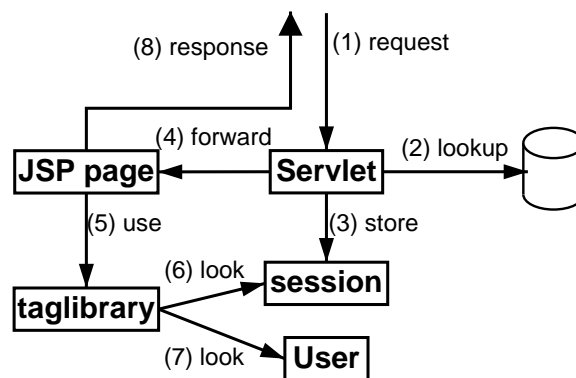


Figure 3.3: The illustration shows the handling of a request by a user already logged into the system.

All the request to the system results in a response (meaningful response or error messages). We see on figure 3.3 the request hitting the webserver. Along with the request parameters can be sent. The server passes the request to the servlet according to its configuration settings (1). The servlet parses the parameters and take appropriate actions according to the parameters (i.e. changing users preferences or process a request). The figure illustrates a user requesting a page hence the servlet contacts the database with proper SQL expressions (2). The result of the request is stored in the users session object (3). Afterwards the servlet forwards the user to the JSP page specified in one of the request parameters (4). Before the JSP renders its output it makes use of an EJD custom tag library (5). The tag library first fetches the requested data from the session object (6) and generates output according to the settings in the user object (7). Finally the JSP page return data to the webserver which then sends response the users request (8).

There are several advantages to this design. First of all by having only one point of entry, the system is easily kept in a valid state (there is simply nothing that could interfere). Secondly the design makes the application scalable. The system could round-robin between several databases for each request made — or even redirect the request to another server to distribute the load.

For the curious reader the EJD is illustrated on implementation level in chapter 2 in the technology background report.

4. The custom tag framework

The chapter illustrates the design ideas behind the framework of custom tag libraries developed for the EJD and shows how to make an API presentation.

The tags are essential to the EJD as they are handling the documentation presentation. A badly designed presentation layer either makes too limited presentations or is too complicated to use. In both cases the use of such a system will be limited.

There are two approaches to designing tag libraries; either the tags output both formatation and information or they just output information. Both approaches are good, but for different situations. Tags outputting formatation and information are good for maintaining a consistent layout, but require reprogramming when the layout changes or when similar but different layout is needed. The other type of tags are easy to use, but require additional formatation codes in order to have a layout. The technical aspects on how the JSP tags are developed and work internally is explained in section 1.6 page 11 in the technical report.

The main purpose of the EJD system is to make the documentation conform to the user, but emphasis has also been put on easy presentation creation. The approach presented here is an approach where programming elements have been abolished. There are no methods, pointers nor variables. The only notion that something “advanced” is taking place, is the link specification where minor knowledge of the database design is required.

The EJD tag framework consists of three groups of tag libraries, each having their own purpose. They all have in common that they shield the internals from the JSP designer.

Presentation These tags are used for “getting information on the screen”. The presentation taglib consists of roughly three kinds of tags which gives the JSP designer great control over the presentation.

Navigation These tags are used for establishing navigability in the pages being presented. The library are capable of creating clickable links as well as frameset defining how the browser should divide its canvas (like frames in HTML browsers).

Configuration These tags are used for changing the users settings. The tags create radio and check-buttons and take care of the parameters names and values.

As the presentation tags are the most extensive we choose to mainly dwell on these. In addition the link tags are special since they describe how to fetch data from the database and thus are explained.

4.1 Presentation

The presentation tags are utilized to create and form the presentations. The goal of the presentation tags is to create generic documentation presentations on the basis of the content of the database without using SQL statements, variable or other programming elements.

The combination of presentations are too many to show here. Through this chapter, we will instead develop some basic JSP pages which we will call the “navigator”. Although simple to code, it has somewhat the same functionality as the Java API. The navigator is a three window presentation with a list of all classes and interfaces in the system in the leftmost window. A list of all constructors and methods in a selected class is presented in the middle window. Finally the rightmost window is an elaboration window on a selected class. The navigator is illustrated on figure 4.1 page 30. Before we start creating the “navigator”, the essentials behind the presentation tags must be explained.

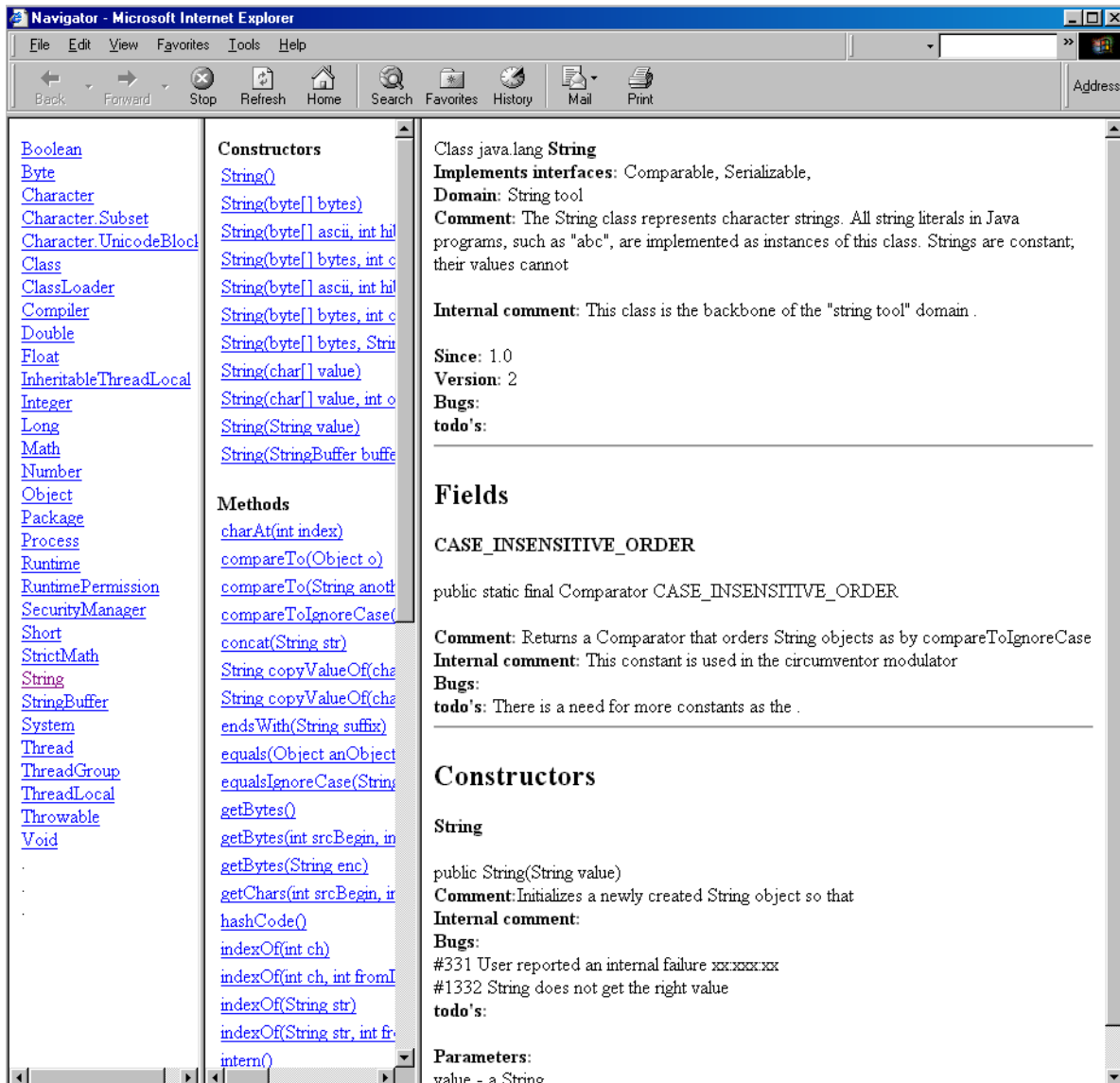


Figure 4.1: A snapshot of the “navigator” developed in EJD.

The presentation tags are divided into three groups: **iterator**, **if** and **information**. The ‘iterator’ iterates over a result (fetched from the database), and for each iteration, executes its body. The following code is the code for showing all private methods.

```

<method:iterator>
  <method:if-private>
    "some presentation..."
  </method:if-private>
</method:iterator>

```

The ‘if’-tags (here `<method:if-private>`) only execute their body if the current iteration element is of the if’s type. For instance if only public methods were iterated, the some presentation would not be executed at all. In addition, a confirmation from the users preference settings are required. Has the user selected not to see private methods, the code above would yield no output whether there were iterated over private methods or not.

The example is now extended so an actual presentation may occur. In order to see the names of the methods being iterated, ‘information’-tags are used. The code will produce a list of names of private methods, with one method name pr. line. The text “private” is placed in front of the method names.

```

<method:iterator>
  <method:if-private>
    private <method:name> <br>
  </method:if-private>
</method:iterator>

```

Many variations over this theme can be spun, i.e. colouring the method name or using pictures instead of the textual representation as the “private” used above. The following two examples show the different approaches.

```

<method:if-private>
  <font color="red"><method:name></font>
</method:if-private>

```

and with pictures depending on the access modifier:

```

<method:if-private>
   <method:name>
</method:if-private>

<method:if-public>
   <method:name>
</method:if-public>

```

4.1.1 Restrictional use

Many tags exist in the presentation tag libraries, however they can not be used interchangeably! Although documentation is “elastic” in the EJD, documentation is still tied to entities (classes, methods etc). This is reflected in the design of the taglibraries; Method information tags can not reside outside a method iterator, method iterators can not reside outside clace iterators etc. This should not come as a surprise, as methods can not exist outside classes/interfaces in the Java programming language.

Instead of thinking of these rules as restrictions, we prefer to think of them as “tags provides other tags”: A `<clace:iterator>` provides a `<method:iterator>`. Besides sounding less negative, the choice of word reflects the implementation. Tags ask their parent tag for information. Each time a method iterator starts iterating it asks the clace iterator for all the methods residing in the current class/interface in the iteration cycle. As the class/interface iterator always is the first tag to be used, the tag fetches the database query result from the session.

In the following the hierarchy of the tags available in the EJD will be presented. The bold words are the tags and the following shows what tags they provide. Tags ending with a slash ("/") are tags with no body. All other tags have a corresponding end tag not shown. The prefix x: are definable by the JSP author, but in order to distinguish different tags, we have defined prefixes according to the nonterminals in table 3.1 page 24.

```

<clace:iterator>
<class:if> <interface:if>

<implementsinterface:iterator> <method:iterator> <field:iterator>
<clace:doc-iterator>

<clace:inPackage/> <clace:if-isStatic> <clace:if-isFinal> <clace:final/>
<clace:accessModifier/> <clace:if-private> <clace:if-protected> <clace:if-package>
<clace:if-public> <clace:name/> <clace:displayName/> <clace:sourcecode/>

<class:if>
<constructor:iterator>

<class:superclass/> <class:if-isAbstract> <class:abstract/>

<interface:if>
(empty)
<constructor:iterator>
<throws:iterator> <parameter:iterator> <constructor:doc-iterator>

<constructor:accessModifier/> <constructor:if-private> <constructor:if-protected>
<constructor:if-public> <constructor:name/>
<constructor:displayName/> <constructor:inClass/> <constructor:sourcecode/>

<method:iterator>
<throws:iterator> <method:doc-iterator>

<method:if-isSynchronized> <method:synchronized/> <method:if-isAbstract>
<method:abstract/> <method:if-isStatic> <method:static/> <method:if-isFinal>
<method:accessModifier/> <method:if-private> <method:if-protected>
<method:if-package> <method:if-public> <method:returnType/>
<method:returnDimension/> <method:name/> <method:displayName/>
<parameter:iterator> <method:inClass/> <method:sourcecode/> <method:cm_id/>

<parameter:iterator>
<parameter:type/> <parameter:dimension/> <parameter:name/>

<throws:iterator>
<throws:name/> <throws:displayname/>

<field:iterator>
<field:doc-iterator>

<field:if-isTransient> <field:transient/> <field:if-isVolatile> <field:volatile/>
<field:if-isStatic> <field:static/> <field:if-isFinal> <field:final/>
<field:accessModifier/> <field:if-private> <field:if-protected> <field:if-package>
<field:if-public> <field:type/> <field:name/> <field:inClass/>

<implementsinterface:iterator>
<implementsinterface:method-iterator>

<implementsinterface:name/> <implementsinterface:displayName/>

<implementsinterface:method-iterator>
<implementsinterface:method-name/> <implementsinterface:method-displayName/>

<*:doc-iterator type="">
<*:doc field="[val1|val2|val3|text]" />

```

The root of the hierarchy is the `<clace:iterator>` tag. From here shared information among interfaces and classes can be found, including iterators for methods, fields, etc. Special information on classes or interfaces are reached through the `<class:if>` and `<interface:if>`. The rest of the hierarchy is based on the same concept. The only exception is the `doc` tags which are designed to be generic. The `doc-iterator` requires an argument determining what comment type to iterate over (which is the `@tagname` excluding the `@`). The `<doc>` tag requires a parameter `val` which determines what part of the comment to extract (see table 3.2 page 27 for an elaboration).

Almost all tags are self-explanatory, the few exceptions are explained here. The `<name>` returns the full name (including package or class name) while `<displayName>` is the stripped down version, i.e. class or method name. Usually `<name>` is used in conjunction with the `<link>` tag (explained in section 4.3 page 37) while `<displayName>` is used in the presentation. The `accessModifier` returns either “public”, “package”, “protected” or “private” according to the entity, its a textual shortcut to the if-tags. Other shortcuts are `static`, `final`, `abstract`, `transient` and `volatile`. They either output nothing or the word “static” “final” etc. The `cm_id` is a unique identifier for methods and constructors and is used in `<link>` tag (as their names are not unique).

4.2 How to make a presentation

Designing pages for the EJD system is a bit like designing webpages. The author never knows what preferences the user has chosen (likewise the web designer never knows the number of colours or screen resolution of the visitors). The user can at runtime decide which kind of information he wants to see (i.e. public methods or internal documentation) and select the order in which the entries must be showed. It should cause no problems, as long as the designer does not expect to make “Designer pages” (in the true meaning of the word).

Begin presented only with the hierarchy in the previous section, it can be hard to conceive the strength of the tags. For this reason we now show how to build the main window (the rightmost) of the “navigator” (shown in figure 4.2 page 34). To make things simple we target the pages towards HTML browsers and leave out navigation issues and explain those later. In order not to blur the code the HTML is restricted to `<h2>`, `
` and `<hr>` which are the codes for “big font”, “new line” and “horizontal ruler”.

We will explain the code line for line (excluding the initial HTML code such as `<HTML><head> . . .`).

line 1: The first line shows whether the entity is a class or an interface, the package it is placed in and finally its name written in bold. In EJD this is written as:

```
<clace:iterator>
<class:if>Class </class:if>
<interface:if>Interface </interface:if>
<clace:inPackage/>
<b><clace:displayName></b>
<br>
```

line 2: Shows the interfaces implemented as a comma-separated list

```
<b>Implements interfaces: </b>
<implementsinterface:iterator>
  <implementsinterface:displayName/>,
</implementsinterface:iterator>
<br>
```

line 3: Displays the name of the domain the class is categorized in

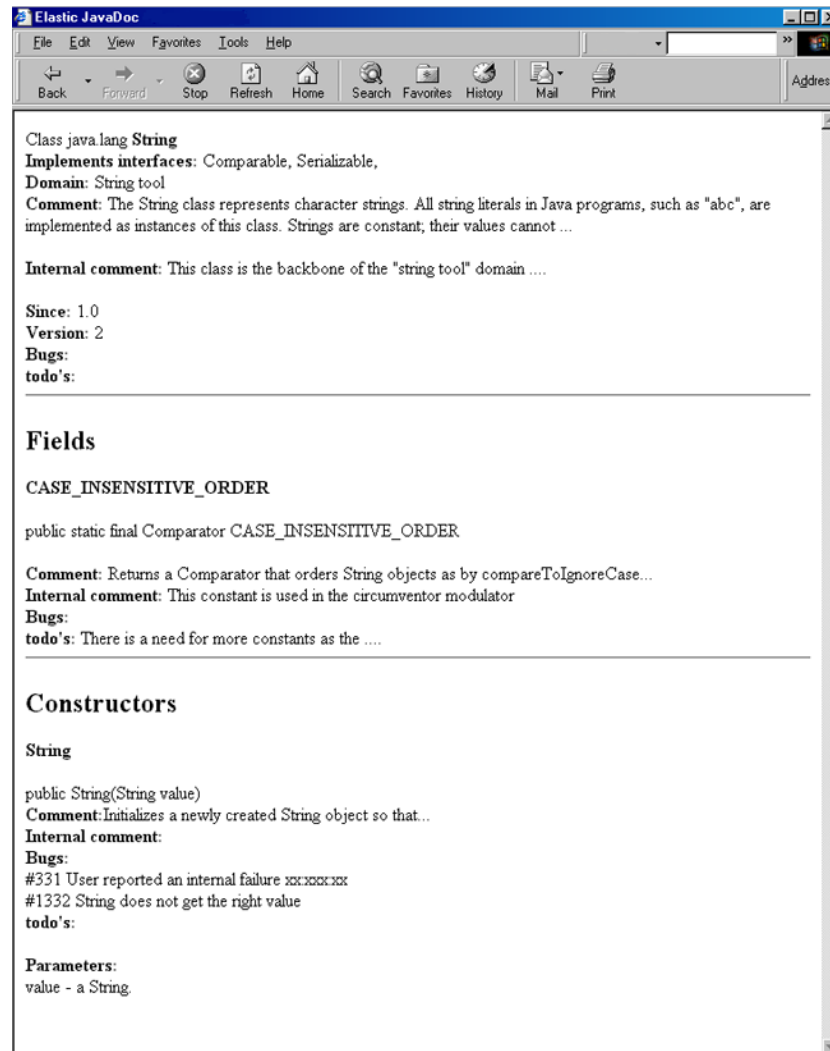


Figure 4.2: The screen shows the elaboration screen of a class made in the EJD system.

```

<b>Domain:</b>
<clace:doc-iterator type="domain">
  <clace:doc val="vall">
</clace:doc-iterator>
<br>

```

line 4: Displays the class comment

```

<b>Comment:</b>
<clace:doc-iterator type="comment">
  <clace:doc val="text">
</clace:doc-iterator>
<br><br>

```

line 6: Displays the internal comment

```
<b>Internal comment:</b>
<clace:doc-iterator type="internal">
  <clace:doc val="text">
</clace:doc-iterator>
<br><br>
```

line 8: Displays the class since-comment

```
<b>Since:</b>
<clace:doc-iterator type="since">
  <clace:doc val="vall">
</clace:doc-iterator>
<br><br>
```

line 9: Displays the class bug number and description

```
<b>Bugs:</b>
<clace:doc-iterator type="bug">
  # <clace:doc val="vall"> <clace:doc val="text"> <br>
</clace:doc-iterator>
<br><br>
```

line 10: Displays the class todo comment and a horizontal line

```
<b>Todo's:</b>
<clace:doc-iterator type="todo">
  <clace:doc val="text">
</clace:doc-iterator>
<br><hr>
```

line 11–17: Displays “Field” and all the fields in the class.

```
<h2>Fields</h2>
<field:iterator>
  <b><field:name></b>
  <br><br>

  <field:accessModifier/> <field:static/> <field:final/>
  <field:type/> <field:name/>
  <br><br>

  <b>Comment:</b>
  <field:doc-iterator type="comment">
    <field:doc val="text">
  </field:doc-iterator>
  <br><br>

  <b>Internal comment:</b>
  <field:doc-iterator type="internal">
    <field:doc val="text">
  </field:doc-iterator>
  <br><br>

  <b>Bugs:</b>
  <field:doc-iterator type="bug">
    # <field:doc val="val1"> <field:doc val="text"> <br>
  </field:doc-iterator>
  <br><br>

  <b>Todo's:</b>
  <clace:doc-iterator type="todo">
    <clace:doc val="text">
  </clace:doc-iterator>
  <br><hr>
</field:iterator>
<br><hr>
```

line 18- : Display "Constructor" and all the constructors.

```

<h2>Constructors</h2>
<constructor:iterator>
  <b><constructor:name></b>
  <br><br>

  <constructor:accessModifier/> <constructor:name/> (
  <parameter:iterator>
    <parameter:type/> <parameter:dimension/> <parameter:name/>
  </parameter:iterator>
  )
  <br><br>

  <b>Comment:</b>
  <constructor:doc-iterator type="comment">
    <constructor:doc val="text">
  </constrcutor:doc-iterator>
  <br><br>

  <b>Internal comment:</b>
  <constrcutor:doc-iterator type="internal">
    <constrcutor:doc val="text">
  </constrcutor:doc-iterator>
  <br><br>

  <b>Bugs:</b>
  <constrcutor:doc-iterator type="bug">
    # <constrcutor:doc val="vall"> <constrcutor:doc val="text"> <br>
  </constrcutor:doc-iterator>
  <br><br>

  <b>Todo's:</b>
  <constrcutor:doc-iterator type="todo">
    <constrcutor:doc val="text">
  </constrcutor:doc-iterator>
  <br><br>

  <b>Parameters:</b>
  <constrcutor:doc-iterator type="param">
    <constrcutor:doc val="vall"> - <constructor:doc val="text">
  </constrcutor:doc-iterator>
  <br><br>
</constructor:iterator>
<br><hr>
</clace:iterator>

```

As the code shows, creating presentations is a straight forward job. The tags are easily learned due to their repetitive nature. As the left and the middle window of the navigator contains links, they will be explained in the next section.

4.3 Navigation

The navigation tags handles the navigation issues, which currently is the creation of clickable links and dividing the browser in frames. The main issue of this section is the link tags. The link tags provides a way to make a region clickable, either text, images or a combination of both — just as the `<a href . . >` in HTML.

4.3.1 The link tag

Clicking a link makes one or more windows change content. As a minimum the link tag must provide information on which frame is to be changed, and what JSP file must take care of the presentation. Along with this basic information, specification of what data needs be presented must also be applied in the link. Due to the many information needs being specified, the link tag seems fairly complex at first.

Creating a list of all classes in the system where each class is clickable and will result in a display of that class in frame `right` by the file `info.jsp` yields the following (preliminary) code:

Codelisting 4.1:

```
1 <clace:iterator>
2   <nav:link>
3     <nav:part source="/info.jsp" target="right">
4       <nav:param entity="clace" field="name">
5         <clace:name>
6       </nav:param>
7     </nav:part>
8     <clace:displayName>
9   </nav:link>
10  <br>
11 </clace:iterator>
```

The program shortly explained: On line 2 we start the link. Line 3–7 defines a part in the link. Line 8 is what is displayed as the link (notice the “display friendly” tag is used). Line 10 sets a line break so the next iteration cycle will print on a new line.

A `<nav:part>` tag is used for every frame in the browser to be effected when clicking the link. The `<nav:param>` specifies conditions under which the information extraction in the database takes place. This specification is crucial to the presentation. The “main window”, explained in the last section, blindly iterates over all classes in the database result. If this result is not restricted the JSP file would display in-depth information on **all** methods and constructors in the system! The `<nav:param>` at the lines 4–6 narrows the result to all classes with the name of the current iteration of the `<clace:iterator>`. This is a unique name hence the JSP page will display only information on one class. The reason for placing `<clace:name>` inside the body of `nav:param` is that tags can not be given as values for parameters. Legal `entity` and `field` names are found in table 3.1 page 24.

Multiple `<param>` tags can be specified as they are joined before the database query, however they can each only have one value.

4.3.1.1 The `include/exclude` attributes

In an ideal situation no extra parameters needs be specified. Whenever a request for i.e. a class or a method was received all tables in the database would be joined and sent to the JSP page to be formatted. Unfortunately performance does not allow this. Imagine asking for the list of classes and interfaces (which is a very common request). This yield first joining and then transferring 100 MB data to the JSP tags which strips this down to a mere 1800 words ~ 34 KB (with the names estimated to an average length of 19 characters). For this reason there is a need for making further restrictions to minimize the superfluous data transfers.

Instead of having to specify every table not needed or every table needed (both configurations yields cases where unreasonable many tables needs be specified). We propose **attributes** to the `<nav:part>` tag for including and excluding tables, but with an extra semantic feature: Including tables yields all tables are excluded and only the specified tables are included. Excluding tables yields all tables are included except for the specified tables. The tables to be included or excluded are specified with a comma-separated list. If all tables are needed (i.e. for creating a presentation alike when clicking a class in the Java API) a `*` is given as argument to the include tag.

4.4 How to make the class/interface list

To build the complete class/interface list placed in the left window of the navigator we need to extend the code in codelistings 4.1 page 38 with the extra include/exclude specification. Also we need to update both the right and the middle window. This is easily done by having two `<part>` tags instead of one. The full code listing looks like

Codelistings 4.2:

```
1 <clace:iterator>
2   <nav:link>
3     <nav:part source="/info.jsp" target="right" include="*">
4       <nav:param entity="clace" field="name">
5         <clace:name>
6       </nav:param>
7     </nav:part>
8
9     <nav:part source="/MethodCon-list.jsp" target="middle"
10      include="clace,method,constructor">
11       <nav:param entity="clace" field="name">
12         <clace:name>
13       </nav:param>
14     </nav:part>
15
16     <clace:displayName>
17   </nav:link>
18   <br>
19 </clace:iterator>
```

4.5 The big `<link>` tag

When not knowing the internals of the EJD it may seem strange that the two links in codelistings 4.2 creating two HTML `..` requires no less than 16 lines of code. The reason is that the JSP pages generate output blindly on the basis of the database result. In the link we restrict the number of results the database gives. These restrictions, nonetheless, just as well be placed in the beginning of the JSP page itself. Doing so, however, limits the JSP pages to only one presentation. Placing the restrictions in the link instead enables the JSP page to be used for several different presentations. An example of this is showed in the next section.

4.6 Extending functionality with links

Now that we have shown the basics of creating links, we can now extend the functionality of the main window in the navigator. The page presented all bugs in the class/method/constructor. We now extend this presentation with the following links “show all bugs numbered X”, “show all bugs in domain Y” and “show all severe bugs in the system”. The first link is useful if bug notification is placed several places in the code. The second link may be of use to the programmer programming a class, as he usually has the responsibility for a package or maybe a domain. Finally, the last link gives a status of the system. The project manager can get some idea as to whether a release within short time is realistic.

From the “line 9 explanation” found on page 35, we know how to present all bugs in the current entity. The result could look like figure 4.3.

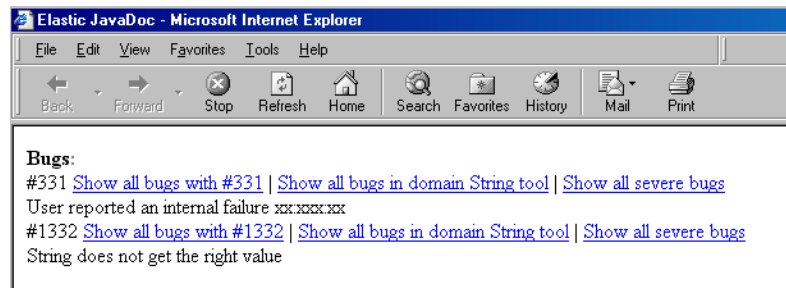


Figure 4.3: How the extensions to the navigator may look.

Codelisting 4.3:

```

1 <b>Bugs:</b>
2 <clace:doc-iterator type="bug">
3     # <clace:doc val="vall">

```

Here we start the iterating over all bugs in the entity and print the bug number.

```

4 <nav:link>
5     <nav:part source="/buglist.jsp" target="this" exclude="parameter,implementsinterface">
6         <nav:param entity="comment" field="vall"> <clace:doc val="vall"> </nav:param>
7     </nav:part>
8     Show all bugs with #<clace:doc val="vall">
9 </nav:link>

```

Here we create the first link. On line 5 we specify we want all tables joined except parameters and implementsinterfaces. On line 6 we specify in the param that the comment field 1 must match the current bugnumber. On line 8 we generate a dynamic linktext.

```

10 <clace:doc-iterator type="domain">
11     <nav:link>
12         <nav:part source="/buglist.jsp" target="this" exclude="parameter,implementsinterface">
13             <nav:param entity="comment" field="vall"> <clace:doc val="vall"> </nav:param>
14         </nav:part>
15         Show all bugs bugs in domain <clace:doc val="vall">
16     </nav:link>
17 </clace:doc-iterator>

```

Here we create the second link. To get hold of the domain information we need to start a new iterator. By starting it before the link creation, the link will only be created if there is any domain information (the `<doc-iterator>` only iterates if any elements exist). Apart from this trick the code is the same

```

18 <nav:link>
19   <nav:part source="/buglist.jsp" target="this" exclude="parameter,implementsinterface">
20     <nav:param entity="comment" field="val2">severe</nav:param>
21   </nav:part>
22   Show all severe bugs
23 </nav:link>

```

The third link is created the same way link one is created. The only difference being we want all bugs marked “severe” rather than a specific bug number.

```

24
25   <br> <clace:doc val="text"> <br>
26 </clace:doc-iterator>

```

Finally we print the bug description.

Looking at lines 5, 12, 19 we see that it is the same JSP file doing the presentation! This is only possible when the restrictions for the database query is placed in the link rather in the JSP file doing the presentation.

Codelisting 4.4: “buglist.jsp”

```

1 <clace:iterator>
2   <clace:doc-iterator type="bug">
3     <b>bug:</b> # <clace:doc val="val1"> <clace:doc val="text"><br>
4     <b>in:</b> <clace:name>
5     <br><br>
6   </clace:bug-iterator>
7
8   <constructor:iterator>
9     <constructor:doc-iterator type="bug">
10    <b>bug:</b> # <constructor:doc val="val1"> <constructor:doc val="text"><br>
11    <b>in:</b> <constructor:name>
12    <br><br>
13  </constructor:bug-iterator>
14 </constructor:iterator>
15
16 <method:iterator>
17   <method:doc-iterator type="bug">
18     <b>bug:</b> # <method:doc val="val1"> <method:doc val="text"><br>
19     <b>in:</b> <method:name>
20     <br><br>
21   </method:bug-iterator>
22 </method:iterator>
23 </clace:iterator>

```

For every class, method and constructor we print the list of bugs. Of course the page should be punctuated with links, i.e. the class name should be a link to the class. We have chosen to omit the links to make the code clearer.

4.7 Internal linking

EJD also supports the creation of internal links. Internal links are build from two things, a clickable link, and an invisible bookmark which is jumped to when clicking the link.

The bookmark is created with the tag `<nav:mark name=" " >` where the name must be a unique name on the page.

The internal links are created with the tag `<nav:ilink to=" " >` where `to` must equal a bookmark name on the page. We have chosen to distinguish links from internal links due to their different nature in syntax and action. By having two tags which each has only required parameters instead of one tag with many optional parameters, we ensure that most errors made by the JSP designer will yield compilation exceptions rather than runtime exceptions (i.e. when the tag discovers a parameter is missing).

Creating a link to a marker on an external page is done within a `<to>` body in the body of the `<part>` tag.

To develop the middle window should now be fairly simple. The code will look a lot like codelisting 4.2. The only difference is we only update one window and we make reference to bookmarks.

4.8 Configuration

We find it extremely important that the users preferences themselves are configurable. Beginners to programming should not be baffled with the possibilities and configurabilities of the EJD. They have a hard time learning the basic programming, and should ignore essential as well as fancy documentation features. By tailoring the preference appearance the EJD is more lenient to new users than the Java API.

The configuration part could use colours i.e. green to signal novice features and red to signal advanced features. Advanced features could also be completely left out. We want to depart from situations like teachers having to explain a simple "hello world" example (in Java) to the students and having to tell them to ignore the "class", "public", "static", "void", "main" and "[]" (which is essentially 90% of the code). EJD should become a natural part of the programming environment the users are in, thus EJD features should match the users knowledge.

To develop configuration presentations as easy as possible, EJD is supplied with configuration tags which create radio buttons and check boxes. Figure 4.4 and codelisting 4.5 shows how the tags nicely can be grouped within a HTML table.

Access	Elements	Comment		Sorting/Grouping	
<input checked="" type="checkbox"/> public	<input checked="" type="checkbox"/> constructors	<input checked="" type="checkbox"/> @bug	<input type="checkbox"/> @pattern	<input checked="" type="radio"/> Alpha	<input type="radio"/> Algorithm
<input checked="" type="checkbox"/> package	<input checked="" type="checkbox"/> fields	<input checked="" type="checkbox"/> @user	<input type="checkbox"/> DBC	<input type="radio"/> Category	<input type="radio"/> Pattern
<input checked="" type="checkbox"/> protected	<input type="checkbox"/> abstract	<input type="checkbox"/> @internal	<input type="checkbox"/> @blocking	<input type="radio"/> Marshall	<input type="radio"/> Datastructure
<input checked="" type="checkbox"/> private	<input type="checkbox"/> no overload	<input type="checkbox"/> @todo	<input type="checkbox"/> @threadsafe	<input type="radio"/> Sequence	
<input type="checkbox"/> Only Lvl 1		<input type="checkbox"/> @job	<input type="checkbox"/> @notthreadsafe		<input type="button" value="Submit"/>

Figure 4.4: An example on how the tags can be grouped by a HTML table.

Codelisting 4.5:

```
1 <pref:form>
2 <table>
3 <tr><th>Access</th><th>Elements...
4 <tr><td>
5     <pref:showPublic/> public<br>
6     <pref:showPackage/> package<br>
7     <pref:showProtected/> protected<br>
8     <pref:showPrivate/> private<br>
9     <pref:onlylvl1/> Only Lvl 1<br>
10    </td>
11    <td>
12     <pref:showConstructors> constructors<br>
13     :
55 </pref:form>
```

Note that the tags are inserted in the order the JSP designer wants. The tags themselves takes care of setting their initial values based on the setting in the user object. A the `<pref:form>` takes care of creating a form which on submit will be sent to the right place. Notice that some tags appear as checkboxes while others as radiobuttons. This choice is solely up to the tag, depending on whether one or many values can be chosen. When the form is submitted the EJD system updates the user profile and saves the settings.

4.9 The “navigator” assembled

Through out the chapter we have developed parts of the navigator. We now integrate the configuration window with the 3-split window already shown. We do this as the users needs may change repeatedly and frequently. We do not want the configuration to be hidden, for it is the setting of preferences that makes the information flow elastic. Figure 4.5 page 44 shows “the navigator” as we have built it. Note that “the navigator” should be regarded a sample application — an example of what the EJD system is capable of doing.

As the EJD system has separate user profiles the system can log the most often visited classes/interfaces and let the user choose to have these shown instead of a list of all classes/interfaces. A third very useful option could be that a plain textfile could contain “recommended classes/interfaces” which could be chosen the same way. This would be a great way for new users to either programming or maybe a big development project to see the most necessary classes in the system.

The only thing not showed is how to divide the browser into many frames. The EJD tags for this functionality resembles the procedure in HTML and is thus omitted in the report.

The screenshot shows a Microsoft Internet Explorer browser window displaying a web application titled "Navigator". The browser's address bar is empty. The application interface is divided into several sections:

- Left Sidebar:** A vertical list of Java classes including Boolean, Byte, Character, Character.Subset, Character.UnicodeBl, Class, ClassLoader, Compiler, Double, Float, InheritableThreadLoc, Integer, Long, Math, Number, Object, Package, Process, Runtime, RuntimePermission, SecurityManager, Short, StrictMath, String, StringBuffer, System, Thread, ThreadGroup, ThreadLocal, Throwable, and Void.
- Main Content Area:**
 - Constructors:** Lists constructors for the String class, such as String(), String(byte[] b), String(byte[] a), String(byte[] b), String(byte[] a), String(byte[] b), String(char[] v), String(char[] v), String(String v), and String(StringB).
 - Methods:** Lists methods like charAt(int index), compareTo(Object o), compareTo(String s), compareToIgnoreCase(String s), concat(String s), String copyValueOf(char[] data), String copyValueOf(char[] data, int offset, int count), endsWith(String suffix), equals(Object obj), equalsIgnoreCase(String s), getBytes(), getBytes(int start, int end), getBytes(String charsetName), and hashCode().
 - Class Information:**
 - Class:** java.lang.String
 - Implements interfaces:** Comparable, Serializable
 - Domain:** String tool
 - Comment:** The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class. Strings are constant; their values cannot be changed after they are created.
 - Internal comment:** This class is the backbone of the "string tool" domain.
 - Since:** 1.0
 - Version:** 2
 - Bugs:**
 - todo's:**
 - Fields:**
 - CASE_INSENSITIVE_ORDER:** public static final Comparator CASE_INSENSITIVE_ORDER
 - Comment:** Returns a Comparator that orders String objects as by compareToIgnoreCase
 - Internal comment:** This constant is used in the circumventor modulator
 - Bugs:**
 - todo's:** There is a need for more constants as the .
 - Constructors:**
 - String:** public String(String value)
 - Comment:** Initializes a newly created String object so that
 - Internal comment:**
- Bottom Control Panel:** A table with four columns: Access, Elements, Comment, and Sorting/Grouping. Each column contains checkboxes for various filtering options.

Access	Elements	Comment	Sorting/Grouping
<input type="checkbox"/> public	<input type="checkbox"/> constructors	<input type="checkbox"/> @bug	<input type="radio"/> Alpha
<input type="checkbox"/> package	<input type="checkbox"/> fields	<input type="checkbox"/> @user	<input type="radio"/> Category
<input type="checkbox"/> protected	<input type="checkbox"/> abstract	<input type="checkbox"/> @internal	<input type="radio"/> Marshall
<input type="checkbox"/> private	<input type="checkbox"/> no overload	<input type="checkbox"/> @todo	<input type="radio"/> Sequence
<input type="checkbox"/> Only Lvl 1	<input type="checkbox"/> @job	<input type="checkbox"/> @notthreadsafe	<input type="radio"/> Algorithm
		<input type="checkbox"/> @pattern	<input type="radio"/> Pattern
		<input type="checkbox"/> DBC	<input type="radio"/> Datastructure
		<input type="checkbox"/> @blocking	
		<input type="checkbox"/> @threadsafe	
		<input type="checkbox"/> @notthreadsafe	

Figure 4.5: A screenshot of "the navigator" developed through last chapter.

4.10 Summary of the presented tags

A lot of tags and categories have been presented. We will try to summarise the chapter in a terse form. First tags has three purposes

- Presentation - The presentation of information
- Navigation - Navigation of the EJD
- Configuration - Configure the configuration presentation

Each of these “purpose groups” each have several partitions.

- Presentation
 - ▷ iterator - Iterate over a set of data and provides tags
 - ▷ if - Executes its body if the current element is of the type the “if” represents.
 - ▷ information - Tags with no body which output raw (unformatted) information.
- Navigation
 - ▷ <link>
 - <part> - For every frame to affect a “part” is used
 - <param> - Setup restrictions to the database query.
 - <to> - Used if linking to a bookmark on an external page.
 - ▷ <ilink> - Creating links linking to a bookmark on the page.
 - ▷ <mark> - Create an invisible bookmarks to be used by “ilink”.
- Configuration
 - ▷ <form> - To encircle the configuration tags
 - ▷ config tags - Producing either radio- or checbx-buttons. They automatically initialize to the users current settings.

Part III

Reflections

The last part of the report deals with reflections over the EJD project.

5. Reflections

The underlying model for handling documentation presented in the EJD project is illustrated on figure 5.1. Although we do not have a fully working implementation we are convinced that using a database to producing documentation presentations is a wise choice. We believe that the database enables interesting combinations of documentation parts. Similar to the “time line of a painter” example from the introduction chapter, the database enables new and exciting presentations just by combining the “raw material” differently than what has been done up till now.

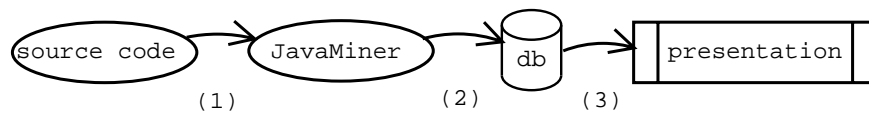


Figure 5.1: A low-detail model of the EJD system.

With the amount of information currently extracted from the source code many new features missing in the Java API can be built. An exciting feature is the automatic generation of class diagrams. Combined with the `@level 1` tag the content of the boxes in the diagram is limited to the most necessary methods, fields and constructors. The hardest part in the development of the EJD was to understand the framework of servlets, JSP and the Doclet. The technologies were new to us at the start of the EJD. Reading textbooks on the technologies gave some understanding, but it was only after creating (and maintain through the development) the simple diagrams (1.4 page 9, 1.5 page 13 and 3.1 page 20 in the technology report) that we were able to fully understand the possibilities and limits of the technologies. Having had auto generated diagrams from the beginning would have helped a lot.

As to whether enough information is extracted from the source codes is debatable, and were touched upon already in chapter 2.3 page 17. Increasing the amount of information, makes the EJD system useful to new kinds of software. Currently all static information is extracted which is of high use for the user of a library. The reuser is interested in information such as “what is the signature for a method performing X ” and cares less about how X internally is carried out. As all static information currently is extracted, gathering more information yields gathering a new type of information — the dynamic information. Such information could be invocations of methods locally and in foreign objects. The dynamic information could be utilized to spot dead code, build sequence diagrams, visualize the dependencies in the program etc. In library code, and most GUI code for that matter, these types of visualizations are impossible to make mechanically as these types of code holds the special property that most of its public methods are never invoked in the code. We do not know whether the Javadoc tool was targetted library code only, but it is used for many kinds of code. Gathering dynamic information therefore seems highly useful. In addition the maintainer of complicated library- or GUI-code might very well have an advantage in seeing the structure of the non-public code.

From where this report stop further work needs be conducted. The following two sections will elaborate on the many things we find interesting. These points should give inspiration to which areas efforts can be put into, on both a general and a concrete level. However, the points should not be regarded as a huge to-do list. We have this semester created the basis for a model through which documentation can be perceived. The model (and tool) perceives documentation as searchable, joinable and indexable chunks of various comments. And when a fully operational version of the EJD is finished there is access to test new ideas in a tool which represents the model. Empirical work with the EJD will be a “reliability-platform” on which further research experiments can be conducted. It is therefore crucial that the EJD is finished soon. In this perspective, an “EJD II” with an extended set of features is not planned. We have designed the EJD, and we are left with many problems in many directions. Building an “EJD II” will only leave us with a new product and maybe new problems in many directions. From certain points of view real progress can be hard to detect. Rather we wish to take a much broader perspective concerning documentation presentations. We wish to research not a few new features but try to put into perspective the many possibilities of extensions and what impact they may have — constructing multi-planed graphs where the planes are features in the arsenal of documentation, figuratively speaking. However, this research can not be conducted without the use of the EJD or other such systems, otherwise the risk of being speculative is too high. For what good is it to produce spectacular theories, when people using the EJD do not find it particularly useful?

5.1 Further work: General level

Usability The usability of the EJD needs be investigated. Are people actually finding the EJD useful? To what extend the system will gain functionality if more data was stored in the database?

Programming language As touched upon in section 5.2 the set of tags offered may need extension — Or maybe even more radical; Maybe what is needed is a *programming language* for creating documentation presentations. By having methods, variables and such enable the designer to create complex and less stiff presentations. The thought is interesting but will probably require a lot of work since a compiler and “execution machinery” needs be developed. Additionally before inventing the programming language the JSP model should be utilized to gain experience as to where the documentation programming language should have its forces and what tasks are most common in developing presentations.

Dynamic client Currently emphasis on keeping the EJD system dynamic. Erik Berglunds “DJavaDoc” system focuses on a having a dynamic client (browser) instead. A merge of the two approaches probably yields good results. Unfortunately utilizing dynamics on the client side when using the current HTML browsers tends to yield hackish solutions. HTML browsers are built to serve as dynamic clients in a degree we could wish for. We suggest building a browser which has dropdown menus, multiple frames, links and automatic syntax highlighting in-built. The browser should use a mark-up language to be compliant with the EJD. There are many advantages of this approach, first of all dynamics in the client would yield less hackish code, secondly some of the functionality currently built into the server could be moved to the client. Things like dynamic hiding and showing of elements or how the entities are sorted could be handled by the client. The data needed be transferred would grow, but there may be less request for the server to take care of, and many operations would be performed by the client thus giving the user a more responsive end product.

Other parts of the language Support for the “new types” of classes such as servlet classes, custom tag classes and Java Enterprise classes. As these kinds of classes are tailored an external framework, in which they are executed, new kinds of information are needed. This could be describing the sequence of servlets used in an order in a web shop or somehow put into the documentation the fact that the classes are build to be used in web pages.

In addition, adding support for tying classes. Testing framework such as the JUnit has by now enforced a “standard” where the testclass of a class is named after the original class and appending the word “Test” to the name. Making EJD aware of this enables special formatting for classes having testcode, or giving estimates to how much code is being tested when tests are conducted.

Integrating other documentation type Support for other types of documentation such as tutorial documentation. The tutorials could refer to figures generated by the EJD on the basis of the database information. The tutorials could also refer to classes which when displayed are greatly reduced in number of methods and constructors — suited for the description in the tutorial.

Support multiple users. There are several reasons as to why this is important. For the EJD system to be known and used it needs to be available to students in the educational system. To ask students to set up a database and a webserver may easily be too big a hurdle for most. In addition they may even not have permission to take such actions.

In addition people installing a new tool which they are unsure of whether is worth spending time on, the threshold for solving troubles can be very low. If a system requires a new user to setup and get acquainted with three different technologies (Database, JSP server and EJD) the risk that the user will never get the system running is very high. A central server should offer new users to be able to upload and use a small number of classes to be convinced of the powers of the EJD.

Production teams need to be able to upload only the part of the system they are developing and still making it accessible to other members of the team. If every member needs keeping a separate copy of the API the time spent on maintenance could easily outnumber the benefits of the EJD system.

Query language There is a need to invent a query language suitable for more complex queries, yet having a simple structure for the presentation designer. It should also try to shield the designer from knowing the underlying database design.

5.2 Further work: Concrete level

It can be discussed as to whether the proposed tags are easy to understand and are sufficient. We feel the tags are fairly obvious — easy to understand and learn. As to whether the tags are sufficient, or due to their structure too often result in clumsy unreadable code, we can not conclude. We hope to soon carry out research both in areas of usability for the user and for the designer of the presentations.

We have identified four areas of deficiencies. Here they will be laid out and will be commented in the discussion chapter.

Missing variables and sizes There are no variables or sizes available to the presentation designer. Showing the count of some entities i.e. the numbers of todo’s in the system. Neither can calculation on sizes be carried out, at the top or end of the class it could be nice to have a sum of total bugs and todo’s maybe.

Missing iteration tags There is no tags for empty iterations, an iterator just skips its body if there are not elements to iterate. Also there is a need to casing out the last element of the iteration as it usually requires a different formatation. As iterators can be highly nested and combined in many ways the problem might be more difficult than what it seems. The solution might be introducing a variant of an “if-else” mechanism. Due to time constraints we have chosen to ignore the problems till a working version of EJD is ready to conduct experiments on.

Information block in nested doc-iterators Nesting doc-iterators from the same entity the values from the outer doc-iterator is blocked since there is no unambiguous reference to it.

```
<clace:doc-iterator type="job">
  <clace:doc-iterator type="bug">
    <clace:doc val="vall">
```

Line 3 will give the bug number not the job number, and there is no way to retrieve the job number but to close the bug iterator.

Wrong structure The idea of having the tags follow the structure of the language it represents may turn out to be a bad idea. currently the restrictions prohibit certain presentations, i.e. we can not present the names of all the entities connected to a certain job number (the @job tag used for tracing changes) as this information has nowhere "to be connected to". The structure easily handles one or more classes having one or more methods, but if i.e. a method needs be joined with several other methods, classes and other things, the datastructure falls short.

Some of the situations can be solved with a clickable link, which when pressed, presents the entities (which again need be conforming to the class-method-etc.-structure). The @job example would work just fine this way, however this requires an unnecessary click from the users side.

Also the codelisting "buglist.jsp" (list 4.4 page 41) showed the need for attacking the database query result from the angle of the bug comment rather than the structured way from class/interface to constructors and methods.

6. Conclusion

We have in this report presented requirements for documentation presentation systems suggested by a wide variety of people. On basis of these we have set up the requirements for our system and described the design of the system. We have created a working prototype which, although limited, establishes credibility that our design ideas can in fact be implemented.

We have based our implementation on the web language JSP which is a part of the Java language. We have argued the existence of problems using the JSP technology and the simple tag-hierarchy we have designed. We have suggested a programming language as a solution to these problems. Despite the problems we think the JSP-approach was the right choice compared to developing a full fledged programming language. Not only did it enable us to develop a running system in short time, in the future it will also work as a “laboratory” to gain experience as to where the documentation programming language, if developed, should have its forces and which tasks are most common in developing presentations.

Since the implementation is not fully working it is hard to conclude whether the EJD comply with the requirements set up. With the requirements in mind we designed the EJD, we are therefore confident that the requirements should be a noteworthy obstacle.

The future development will be finishing the EJD as it is proposed in this report. With the EJD as a “reliability-platform” we wish to conduct research in the following fields: first, to which groups is the EJD useful, and how useful is it? If the outcome of these studies are positive we plan three different directions. Assimilate the rationale for new documentation elements, Find patterns or typical usage of the JSP tags as a basis for implementing a programming language for documentation, and finally study the use of dynamic clients and maybe developing a EJD browser.

Whether the producer of the documentation presentations are JSP or some other language we firmly believe the use of a database as shown in figure 5.1 is the way to proceed. Through the design of the EJD we are confident that the thesis stated in section 1.5 page 13 holds.

Bibliography

- Andersen, M. R., Christensen, C. N. and Sørensen, K. L. (1999). *Internal documentation in an elucidative environment*, Master's thesis, Aalborg University.
- Berglund, E. (1999). *Use-oriented documentation in software development*, Master's thesis, Linköping university, Linköping Sweden.
- Cockburn, A. (april 2000). Supporting tailorable program visualisation through literate programming and fisheye views, Preprint submitted to Elsevier Preprint.
- Cordes, D. and Brown, M. (1991). The literate-programming paradigm, *Computer* pp. 52–61.
- Didrich, K. and Klein, T. (1996). A pragmatic approach to software documentation, *Technical report*, Technische Universität Berlin, Germany.
- Friendly, L. (1995). The design of distributed hyperlinked programming documentation. Presented at The International Workshop on Hypermedia Design in Montpellier, France. June 1995.
- Joy, B., Steele, G., Gosling, J. and Bracha, G. (2000). *The Java Language Specification*, second edn, Addison-Wesley Pub Co. ISBN: 0201310082.
- Knudsen, J. L., Hedin, G., Magnusson, B., Trigg, R. H. and Sandvad, E. S. (1990). Documenting object oriented systems, *Technical report*, Aarhus university, Denmark and Lund institute of technology, Sweden and Science park Aarhus, Denmark. Mjølner II working note "MjII-DK-MIA-90-1(1.0).
- Østerbye, K., Madsen, O. L., Sanvad, E., Bjerring, C., Kammeyer, O., Skov, S. H., Hansen, F. O. and Hansen, F. (2001). Documentation of object-oriented systems and frameworks — cot/2-42 – v2.3, Centre for object Technology, Denmark.
- Powell, A. L., French, J. C. and Knight, J. C. (1996). A systematic approach to creating and maintaining software documentation, *Applied Computing* pp. 201–208.
- Samentinger, J. (n.d.). The role of documentation in programmer training. University of Linz, Australia.
- Schwabe, D., Rossi, G. and Barbarosa, S. D. J. (1996). Systematic hypermedia application design with oohdm, *Hypertext 96, 7th. ACM Conference on Hypertext*.
- Shih, T. K., Chow, L. R., Keh, H.-C. and Lin, Y. C. (1999). Multimedia documentation environment supports well-engineered software development and maintenance, *Computers and artificial intelligence* **18**(3): 285–312.
- van Deursen, A. and Kuipers, T. (1999). Building documentation generators, *IEEE* pp. 40–49.
- Wang, B. and Hitchcock, P. (1992). Linking object oriented database and hypertext to support software documentation, *ACM Journal on Computer Documentation* pp. 149–156.
- Weiss, E. H. (94). Isomorphism between oop and documentation: reflections on samentinger's "object-oriented documentation", *ACM Journal on Computer Documentation* **18**(2): 8–9.
- Welsh, J. and Han, J. (1993). Concepts and tools, *Technical Report 23–93*, Software Verification Research Centre, the university of Queensland, Brisbane, australia.