

**Title:**

Internal documentation  
in an Elucidative environment

**Topic:**

Software Documentation  
for Developers

**Project period:**

7/2-2000 – 9/6-2000

**Project group:** DAT6, E1-207A

Max Rydahl Andersen  
Claus Nyhus Christensen  
Kristian Lykkegaard Sørensen

**Supervisor:**

Kurt Nørmark

**Number of appendixes:** 4

**Total number of pages:** 134

**Number of pages in report:** 90

**Number of reports printed:** 10

**Abstract:**

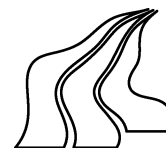
This master thesis deals with the structure and history of internal documentation in an elucidative environment.

Internal documentation is documentation which is produced by developers, used by developers and which documents the internal structures, such as algorithms or data structures of software.

Three main contributions are presented in the thesis: (1) The formulation of a documentation model, called the MRS-model, which divides the internal documentation into three interrelated categories: Motivations, Rationales and Solution descriptions. (2) A practical implementation of the MRS-model in an elucidative environment. (3) By using the MRS-model, a method is sketched for documenting the history of internal documentation and source code as a natural part of the internal documentation.

Based on these three contributions the report concludes that the structure of the MRS-model is both beneficial to the internal documentation and improves the presentation of the documentation to the reader. Furthermore it is concluded that the history of software can be documented, and be a natural part of the internal documentation.





**Titel:**

Intern dokumentation  
i en Elucidativ omgivelse

**Emne:**

Program dokumentation  
for udviklere

**Projekt periode:**

7/2-2000 – 9/6-2000

**Projekt gruppe:** DAT6, E1-207A

Max Rydahl Andersen  
Claus Nyhus Christensen  
Kristian Lykkegaard Sørensen

**Vejleder:**

Kurt Nørmark

**Antal appendiks:** 4

**Totalt antal sider:** 134

**Antal sider i rapporten:** 90

**Antal trykte rapporter:** 10

**Synopsis:**

Denne specialeafhandling behandler struktur af og historie i intern dokumentation i en elucidativ omgivelse.

Intern dokumentation er dokumentation som er produceret af udviklere, bruges af udviklere og som dokumentere de interne strukturer i programmer, så som algoritmer eller data strukturer.

Afhandlingen præsenterer tre hovedresultater: (1) En formulering af en model for dokumentation, kaldet MRS-modellen, som deler intern dokumentation ind i tre interrelaterede kategorier: Motivationer, Rationaler og Løsningsbeskrivelser. (2) En praktisk implementation af MRS-modellen i en elucidativ omgivelse. (3) Ved hjælp af MRS-modellen er det skitseret, hvordan dokumentation af programmers historie kan blive en naturlig del af dokumentation.

Baseret på disse tre resultater konkluderer afhandlingen at MRS-modellens struktur, dels er til gavn for den interne dokumentation, dels forbedrer præsentationen af dokumentationen for læseren. Derudover konkluderes det, at programmets historie kan dokumenteres, samt være en naturlig del af den interne dokumentation.



# Preface

This thesis documents the work made during our masters year at the Department of Computer Science, Aalborg University, Denmark. The report focuses on the work made during the second part of the thesis, from February 1, 2000 to June 9, 2000. It is founded in the report “The Elucidator — for Java” [Christensen et al., 2000], which documents the work done from September 2, 1999 to January 17, 2000 as the first part of the master thesis.

Small parts of the “The Elucidator — for Java” report is reproduced in this report, especially in Chapter 2. When this is the case it is clearly stated with a citation to the report.

**Chapter 1: Introduction** This chapter motivates our thesis. We place our work in the context of preserving quality of software, and two hypotheses are established which serves as the foundation for the remains of the thesis.

**Chapter 2: Earlier work** In this chapter the main results and conclusions from our earlier work on the elucidative environment are presented. Furthermore the chapter contains a description and evaluation of an documentation experiment, conducted with the Elucidative environment.

**Chapter 3: Analysis** Through a study of internal documentation in an ideal software process we identify two main roles in software development. Through this study we furthermore motivate, develop and describe a model, called the MRS-model, for writing and reading internal documentation in an elucidative environment. The chapter also presents a description of how the model is intended to be realized. Finally, the MRS-model is compared to other related documentation approaches.

**Chapter 4: Design** This chapter presents the detailed design for how the MRS-model is implemented in the elucidative environment. Three main issues are described: documentation nodes, links and navigation facilities.

**Chapter 5: Reflection** Through a small experiment, this chapter reflects upon the MRS-model and the elucidative environment. The reflection is qualitative and based on observations made during the experiment.

**Chapter 6: Conclusion** In this chapter we conclude on the two hypotheses stated in Chapter 1.

**Chapter 7: Future work** This chapter describes some ideas we believe would be interesting and rewarding to work on in the continuation of this thesis.

**Appendixes** In the appendixes (Appendix A through D) a description of the node types and link roles is presented. Following is this is a grammar for the EDoc language. Finally all templates used in the experiment and statically data from the experiment is presented.

Throughout the report figures and tables are enumerated successively in each chapter. When a figure is taken directly from a literature source this is marked with a citation in the figure caption.

The literature referred to in the report is listed in the bibliography. References are given on the form [Nørmark, 2000b], which means the piece of literature, marked by this label in the literature list, was used.

Further information about Elucidative programming and the examples in this thesis can be found at <http://dopu.cs.auc.dk>.

We would like to thank the following people:

*Amanuensis Thomas Vestdam* — for using one week to be a part of our experiment with the Elucidative environment and the MRS-model.

*Eastfork Object Space (EOS), especially Jørn Larsen and Kim Harding Christensen* — for providing us with tickets for the JAOO Conference 1999, held in Aarhus, September 20-22, 1999. The conference was a great inspiration to our project.

*Dr. Johannes Sameting* — for taking the time to talk with us in connection to his lecture at Aarhus University on November 5, 1999.

*Vincent Gay-Para and Thomas Graf* — for providing extensive support for the Kopi Java Compiler. Especially for devoting a whole week-end to incorporate our special wishes into the compiler.

Aalborg University, June 9, 2000.

---

Max Rydahl Andersen

---

Claus Nyhus Christensen

---

Kristian Lykkegaard Sørensen

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Maintaining quality of software . . . . .	1
1.1.1	Views on software . . . . .	2
1.1.2	The insufficiency of models . . . . .	2
1.2	From Literate to Elucidative Programming . . . . .	3
1.3	Motivation for the thesis . . . . .	5
1.3.1	Structuring documentation . . . . .	6
1.3.2	Documentation history . . . . .	6
1.3.3	Hypotheses . . . . .	7
<b>2</b>	<b>Earlier work</b>	<b>9</b>
2.1	The first part of the master thesis . . . . .	9
2.1.1	Main conclusions . . . . .	11
2.2	Initial experiment . . . . .	13
2.2.1	Experiment circumstances . . . . .	13
2.2.2	Gained experiences . . . . .	13
<b>3</b>	<b>Analysis</b>	<b>17</b>
3.1	Internal documentation in software development . . . . .	17
3.2	Limiting the number of participants . . . . .	20
3.2.1	Characterizing the reader and the writer . . . . .	21
3.3	Utilizing documentation . . . . .	22
3.4	The MRS-model . . . . .	23
3.5	Realization of the MRS-model . . . . .	24
3.5.1	Documentation structure . . . . .	25
3.5.2	Documentation nodes . . . . .	26

3.5.3	Relationships in documentation . . . . .	29
3.5.4	The usage of relationships to maintain the history . . . . .	31
3.5.5	Navigation in documentation . . . . .	32
3.6	Related documentation approaches . . . . .	33
3.6.1	Object-oriented Analysis and Design Documents . . . . .	34
3.6.2	Literate Programming . . . . .	35
3.6.3	Object-oriented Documentation . . . . .	35
3.6.4	The gIBIS hypertext tool . . . . .	36
<b>4</b>	<b>Design</b>	<b>39</b>
4.1	The Elucidative environment . . . . .	39
4.1.1	The languages in the Elucidative environment . . . . .	40
4.1.2	Entities in the Elucidative environment . . . . .	40
4.1.3	The three tools in the Elucidative environment . . . . .	41
4.1.4	Changes to the Elucidator . . . . .	43
4.1.5	The work flow in an Elucidative environment . . . . .	44
4.1.6	Entities in the hypertext model . . . . .	45
4.2	Designing the documentation nodes . . . . .	46
4.2.1	Documentation nodes in the MRS-model . . . . .	46
4.2.2	Thematic catalogs . . . . .	51
4.2.3	The internal structure of the documentation nodes . . . . .	52
4.3	Designing the links . . . . .	55
4.3.1	Link types . . . . .	56
4.3.2	Roles on links . . . . .	57
4.3.3	Structure imposed on the documentation by links . . . . .	59
4.3.4	Creation method . . . . .	59
4.3.5	Discussion . . . . .	60
4.4	Navigation . . . . .	62
4.4.1	Coloring . . . . .	63
4.4.2	Local navigation . . . . .	64
4.4.3	Neighborhood navigation . . . . .	66
4.4.4	Global navigation . . . . .	68



---

<b>5 Reflection</b>	<b>73</b>
5.1 Experiment circumstances . . . . .	73
5.2 Reflections upon the experience of the writers . . . . .	75
5.3 Reflections upon the reader experiment . . . . .	78
5.4 Discussion . . . . .	81
5.4.1 The MRS-model . . . . .	81
5.4.2 The Elucidator tool . . . . .	82
5.4.3 History in documentation . . . . .	82
<b>6 Conclusion</b>	<b>85</b>
<b>7 Future work</b>	<b>89</b>
<b>A Examples of different node types and link roles</b>	<b>91</b>
A.1 Examples of concrete node types . . . . .	91
A.2 Examples of link roles . . . . .	99
<b>B Grammar for the EDoc language</b>	<b>101</b>
<b>C Templates for documentation nodes</b>	<b>109</b>
C.1 Templates for Motivations . . . . .	109
C.2 Templates for Rationales . . . . .	111
C.3 Templates for Solution descriptions . . . . .	112
<b>D Statistics for the StregSystem project</b>	<b>117</b>
<b>Bibliography</b>	<b>119</b>
<b>Index</b>	<b>123</b>



# 1

## Introduction

This thesis deals with the notion of *internal documentation* in an Elucidative environment. By internal documentation we mean *documentation which are produced by developers, used by developers, and which documents the internal structures, such as algorithms or data structures, of software*.

The work done in the project is threefold: First we develop and present a theoretical model for structure of internal documentation. Next we implement this model, using a previously implemented Elucidator tool [Christensen et al., 2000]. Finally we make an experiment using the implemented model, in order to weaken or affirm the hypotheses stated later in this chapter.

This chapter introduces the project. First we discuss how to maintain quality of software, and places internal documentation in this context. Next we discuss our inheritance, which are Literate and Elucidative programming. Finally we present our motivations for the thesis and poses two hypotheses, which this thesis will try to weaken or affirm.

### 1.1 Maintaining quality of software

We see software as a highly complex artifact. The software emerges as a solution to a specific problem. That is, the software is a product of the different demands posed, in order to solve a problem and/or model some part of a real or imaginative world. Many details of the software depends on factors of the problem, world or other parts of the software. Regarding software as a artifact, we focus on the following qualities of software: understandability, modifiability and reusability. These qualities are all qualities in the domain of the software developer, i.e., they are not direct qualities for the user of the software.

Although good modularization paradigms exist, e.g., object-oriented programming, it is still an issue how to maintain the quality of a piece of software when it is changed or reused. We see this problem as a result of an emerging lack of consistency, both internally in the software and between the original design ideas and the changes made.

### 1.1.1 Views on software

Many different approaches have been attempted to describe and solve the problems of maintaining quality of software as it changes and is being reused. We will here give a short introduction to some of these approaches.

Peter Naur does not see the software as an artifact in it self [Naur, 1985], but rather as an implementation or manifestation of the developers knowledge about the solution or model. Naurs view on programming and software has some noteworthy consequences. The problem of degenerating software is seen as a consequence of developers not having the desired knowledge or theory of the software. The software is only one implementation of the theory and can, according to Naur, never convey the complete theory held by the original developers. Hence the problem of maintaining quality in software is a matter of educating the developers through direct communication between experienced and inexperienced developers.

Kent Beck agrees with Naur in the importance of direct communication. In his description of Extreme Programming [Beck, 1999] the quality of the software is maintained in a group of developers. These developers all share the same understanding of the problem at hand and the current solution. We will not discuss these approaches in detail, but just conclude that they address the problem of maintaining quality of software as a matter of sharing unformalized knowledge amongst a group of developers.

Other approaches focuses on the description and architecture of the software. Frameworks and component based programming focus on making generic software, that can easily be reused. Through clear definitions of responsibility, encapsulation and good modularization, these methods try to ensure quality of software. Common for these approaches are that they through better modeling, strive to make comprehension and reuse easier for the developers who uses the framework or component. Both approaches typically documents the interface to, and usage of, the framework or component.

### 1.1.2 The insufficiency of models

The approaches described above addresses the problem of maintaining quality of software from different angles. We see a number of problems in the different approaches. Naur and Beck both deemphasize documentation. Naur claims that it is impossible to document the theory held by the developers. Beck claims that documentation is seldom worth the effort, since the knowledge is held in the development team and can easily be transfered to new members of the team.

We agree that it might be difficult to document the knowledge held by a group of developers, but we disagree that it is impossible. Furthermore we find internal documentation relevant, since unwillingness from customers to throw away old software as well as changes in the development staffs *will* result in developers reusing old software written by other developers; Developers that are possibly unavailable.

The documentation produced for frameworks and components are targeted towards developers that use the software to produce their own software. We see frameworks and components

as a way to ensure quality of the software that is developed by using the framework or components. There is, however, still a lack of attention on the internal documentation in the developed software. We recognize that the object-oriented modeling paradigm can be used to document the internal architecture of the software, but we have objections to this being the only documentation. First of all, the object-oriented modeling paradigm can only describe certain aspects of the software. The success of design patterns underline this; An informal textual description is used to document aspects not directly visible in the object-oriented architecture. Secondly, and most important, the software and the object-oriented description of the software architecture, only communicate the final version of the software. They do not convey the designs that were tried but rejected nor the arguments from the developers that made crucial decisions during design and implementation. Hence we see a need for internal documentation that describe and argue the software.

## 1.2 From Literate to Elucidative Programming

In 1984 Donald E. Knuth, suggested that the time was ripe for significantly better documentation [Knuth, 1984]. To achieve this, he argued, that computer programs should be considered works of literature.

*“Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.”*

[Knuth, 1984]

Based on these thoughts, he developed *Literate Programming*, and a set of tools known as the *WEB tools*. This new paradigm relied on the code residing in the documentation rather than the, at that time, more commonly used solution, where the program was documented via comments written in the source code.

As we see it, Literate programming is well suited for detailed internal documentation, with a focus on the arguments and rationales behind the program. It furthermore has the side effect that, when used as a work method, it provides a powerful modularization mechanism for the program code. However, the advantage of this is not as big as it used to be, since modern programming language provides far more modularization mechanisms than programming languages did in the eighties.

Literate programming seems ideal for documenting algorithms or code fragments. However, Kurt Nørmark, points out several problems in this approach [Nørmark, 2000b]. Knuth’s WEB tools uses three languages: a documentation language, a programming language, and a language to bind the two in a literate document. Nørmark argues, that the “*mental load of using a WEB system is high*”. Firstly, one has to master and use three languages while keeping the focus on problem solving. Secondly, source code, as seen by the developer and the compiler, are different, causing problems when, e.g., syntax errors are to be located. Furthermore, the documentation for the code

*“...is almost exclusively oriented towards a paper [article] representation. Using today's media, a more online-oriented representation using hypertext concepts would be a big gain.”*

[Nørmark, 2000b]

Nørmark argues, that Literate Programming is well suited for producing publications of programs as technical literature, while the needs of the practical software developer are not met.

We agree with Nørmark on most of these problems. It should however be stated that the problems with locating syntax errors can be solved by using specific distributions of the WEB tools, together with specific languages (e.g. using cweb and the preprocessor mechanisms in C [Fischer and Jensen, 1990]).

In our opinion Literate programming has two main advantages:

**Proximity:** Literate programming provides proximity between the documentation and source code, both while the documentation and source code is written and when it is viewed after  $\text{\LaTeX}$  compilation. Since the source code and the documentation are to be placed in the same file, it becomes natural to write the documentation and the source code at the same time. In our experience this means the process of writing the documentation becomes easier and the quality of the documentation is heightened, since the arguments and rationales are fresh in mind.

Having the source code and the documentation in the same file is furthermore a special advantage while maintaining the source code, since it is easy to find the place in the documentation which needs to be updated as a consequence of a change in the source code. Since the source code and documentation is also presented in the same document after  $\text{\LaTeX}$  compilation, the proximity is also kept here.

**Modularization mechanisms:** The second main advantage of Literate programming lies in the modularization mechanisms. In Literate programming you are not limited to the modularization mechanisms provided by the programming language, but you are instead able to modularize your program as it is represented in your mind.

We do though think that this may cause some problems if you are using an object oriented programming language, since a big part of the understanding of programs written in languages from this paradigm relies on the structure of the program. We therefore believe that imposing further modularization on the source code, will lead to a somewhat obfuscated structure of the program.

As a consequence of the problems stated by Nørmark he introduces a branch of Literate Programming, called *Elucidative Programming* for documenting the understanding of programs.

To achieve this, Nørmark suggests that we keep the source code and documentation in separate files in order to remove the mental load experienced with the WEB tools. His primary concern is to maintain the program understanding for current and future developers. This

process should utilize the programming editor to bridge the gap between documentation and source code, by integrating Elucidative programming support in the editor. Furthermore, the output is not directed towards paper, but an online representation suitable for web-browsers.

The concept of Elucidative Programming is coined and described by Nørmark in the article *Requirements for an Elucidative Programming Environment* [Nørmark, 2000b] and an example of a specific implementation of an Elucidator is presented in *An Elucidative Programming Environment for Scheme* [Nørmark, 2000a]. Finally, the design and implementation of an Elucidative Programming environment for Java is presented in *The Elucidator — for Java* [Christensen et al., 2000].

### 1.3 Motivation for the thesis

Before stating the problem which we will attempt to solve in this thesis, we must consider in which surrounding we expect the solution to be situated. Nørmark states that elucidative programming is “*for documenting the understanding of practical programs in a software development project*” [Nørmark, 2000b]. We take this a little further and states that our target is software developers, situated in a software development company, which, for some reason, want to have internal documentation for their programs. Since our target user is the software developer, we suppose that he typically do not have especially good writing skills. We must therefore ensure that successful usage of the Elucidator tools is not completely dependent on the writing skills of the user of the tool.

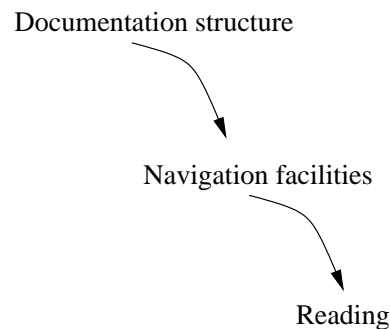
The overall purpose of writing internal documentation is to maintain the program understanding. This purpose is twofold. First, since the writer is writing down his understanding of the program, this gives him the opportunity to reflect on the decisions taken while developing the program. We believe this process will be a help to the developer, since it will provide him with a better understanding of the program he is developing. Second, the written documentation, is intended to help the reader when he tries to understand a piece of software.

In this project we have chosen to focus on the person reading the documentation. The main reason for this choice is that we want to make sure that documentation is actually read. It is important that the documentation is read for a number of reasons:

- If the documentation is not read, we do not exploit the full potential of the documentation, since only one of the purposes of elucidative programming is fulfilled.
- An amount of resources has been spend on making the documentation. If it is not read, the effort is not fully utilized, since the only value is gained in the writing process.
- Not many software developers will be willing to use time on writing documentation if they know in advance that nobody will read it.

### 1.3.1 Structuring documentation

Since we state that it is important that documentation is read, it seems natural to consider how to make sure the documentation is actually read. The direct answer is to make sure the documentation is readable. But how do we then make sure the documentation is readable? This question will be one of the main issues in this thesis. We intend to answer the question by using the model, illustrated in Figure 1.1.



**Figure 1.1:** *A model which illustrates the connection between structuring the documentation, and the readers possibility to have the documentation presented in a, for him, readable form.*

The basis of this model is that documentation is structured in a predefined way. This structure allows us to provide navigation facilities with the purpose of facilitating the reader.

The structure of the documentation can only come from one source: the writer. It is therefore important to make sure the structuring mechanisms is made in a way, which makes it attractive for the writer to structure his documentation. People might argue that developers are anarchists, and will not accept just filling out predefined structures. As we shall see later in the thesis, our experience tells us that this is not true.

### 1.3.2 Documentation history

The most important part of internal documentation are the rationales, since these tell why the program is developed like it is. It is therefore important these rationales are written down in the documentation.

Each time some change is made to a piece of software, this is done because of some rationale. In other words developers do not make a change without reason. It therefore seems like a natural step to document rationales each time a change is made to the software. This is one of the reasons that the history of a program is important. Another equally important reason, is if developers document the history of their program, they and others will be able to learn for errors made in the past.



### 1.3.3 Hypotheses

Based on the observations and opinions presented above we formulate two hypotheses, which serves as our problemization and focus of the project:

**To present internal documentation in order to facilitate the reader, it is necessary to structure it in a predefined way. This structure, combined with navigation facilities, will be beneficial to the internal documentation.**

**The history of the software is important since most changes in the software, are made as a consequence of a rationale. The history of the software can be documented and be a natural part of the internal documentation.**

In the remaining part of this thesis we will provide arguments, to weaken or affirm these hypotheses.



# 2

## Earlier work

The work presented in this master thesis is partly based on work and results from the first part of our work. This work is documented in the report: The Elucidator — for Java [Christensen et al., 2000]. In this chapter we will present the main results and conclusions from this work. Furthermore we describe and evaluate an documentation experiment, conducted with the Elucidator tool produced during the first part of our work.

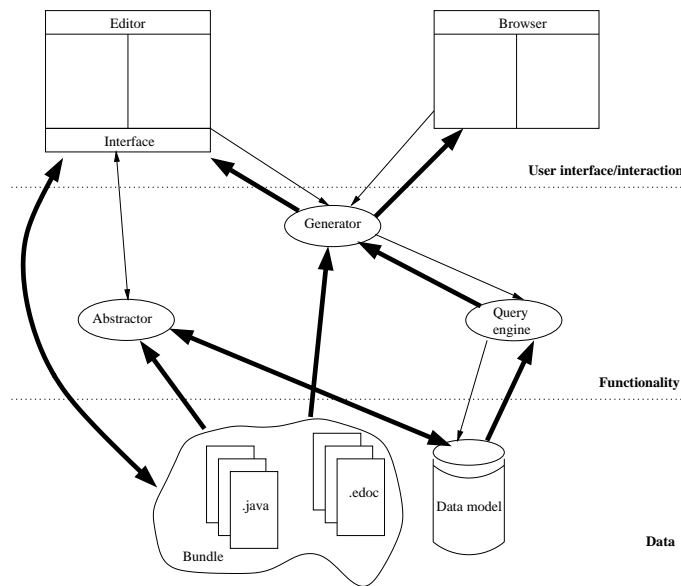
The implementation created in the first part of our work, and described in this chapter, is named the Elucidator 1 and the implementation created in this, the second, part is named the Elucidator 2.

### 2.1 The first part of the master thesis

The primary focus of the first part of our master thesis, were to design and implement an Elucidator for Java using the same principles as used in Nørmark's Scheme Elucidator [Nørmark, 2000a]. The work resulted in a prototype implementation of an Elucidator for Java (Elucidator 1). The prototype differentiates from Kurt Nørmark's earlier implementation on several points. It supports Java instead of Scheme, the HTML pages is dynamically generated and the overall design is more flexible.

Figure 2.1 on the next page illustrates the main components of the Elucidator 1. The roles of these components is briefly explained in the following paragraphs. For a more in-depth technical description see [Christensen et al., 2000].

**The Editor** The editor is used to edit source code and documentation files. Documentation is written in our own language called *EDoc* which has tags for defining chapters, sections and various link elements. The editor has been extended to give support for inserting links from the documentation to the source code and other documentation. The editor support was disregarded to a bare minimum, as opposed to the Scheme Elucidator by Nørmark, which contains more advanced editor facilities.



**Figure 2.1:** Design overview of the Elucidator 1.

At regular intervals the developer invokes the *Abstractor* which parses the source code and documentation. Derived information from this process is stored in the *Data model*. A screen capture showing the editor can be seen in Figure 2.2 on the facing page.

**The Browser** By using a HTML-browser the developer can view documentation and source code side-by-side. The documentation contains the text and links written by the developer. The source code contains links to allow jumping from the use of a symbol to its definition.

The browser also provides a *Navigation window* which lists all the locations to which a source symbol or documentation is related. This resembles features available in most reverse-engineering tools today. For example the navigation window can list all the locations where a method is documented, where it is used (invoked) and what entities the method itself uses.

A screen capture showing the browser can be seen in Figure 2.3 on page 12.

**The Generator** The editor and browser communicates with the *Generator* which is a servlet running on a web-server. The editor retrieves information to support link insertion and the browser retrieves source code and documentation in HTML format for browsing. It is, furthermore the *Generator* which automatically marks up the source code with links.

**The Abstractor** The abstractor extracts information from the Java source code and EDoc documentation. This derived information is stored in the *Data model*. This component is basically the only language dependent part of the Elucidator 1. Hence by providing a abstractor for another language, e.g., C++, the Elucidator 1 can be used for another language.

**The Data model** The Data model is similar to an entity/relationship model which contains a set of entities and their relationships. The use of an entity/relationship model is inspired by the work presented in [Chen et al., 1995] and [Korn et al., 1999].

Packages, classes, fields, methods and even parameters and variables is extracted from the source code as entities. Examples of relationships between entities is the *containment*-relationship between a method and its class, a *invoke*-relationship between the callee's definition and the calling method. The documentation is also represented in the Data model with chapter, section and link entities etc. where, e.g., "refers-to" relationships represents links in the documentation and source code.

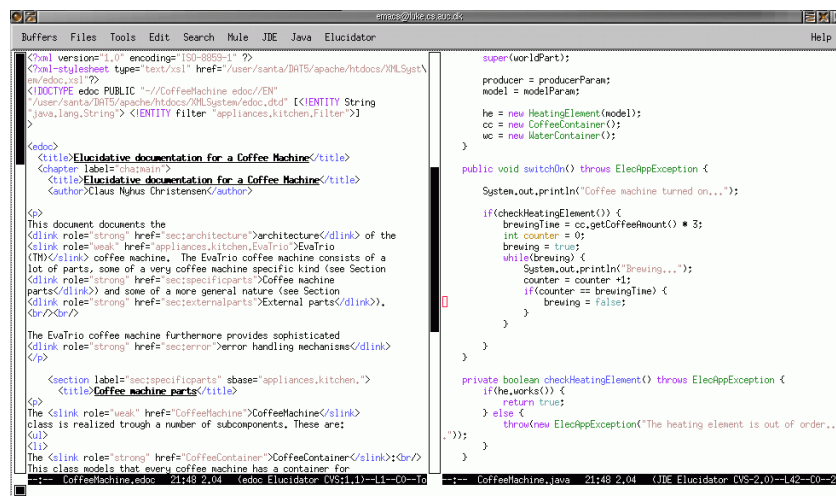


Figure 2.2: Screen capture of the Editor.

### 2.1.1 Main conclusions

As stated in [Christensen et al., 2000] we believe that the first part of our master thesis made a number of contributions. These are listed below, and are ordered with the contributions we found the most important at the top, and the minor contributions at the bottom.

**A prototype Elucidator for Java:** We have managed to show that an Elucidator for Java can be realized. We have furthermore implemented a prototype, which we find promising.

**An architecture of an Elucidator:** We have designed a modular architecture with well-defined standard interfaces. Among the strengths of this architecture is that it is very easy to change the Elucidator 1 to use another language, also non-object-oriented languages, or even make the Elucidator 1 use multiple languages.

**Easy navigation in Java source code:** We have implemented and shown that when the Java source code is abstracted and stored in a data model it is possible to provide the user

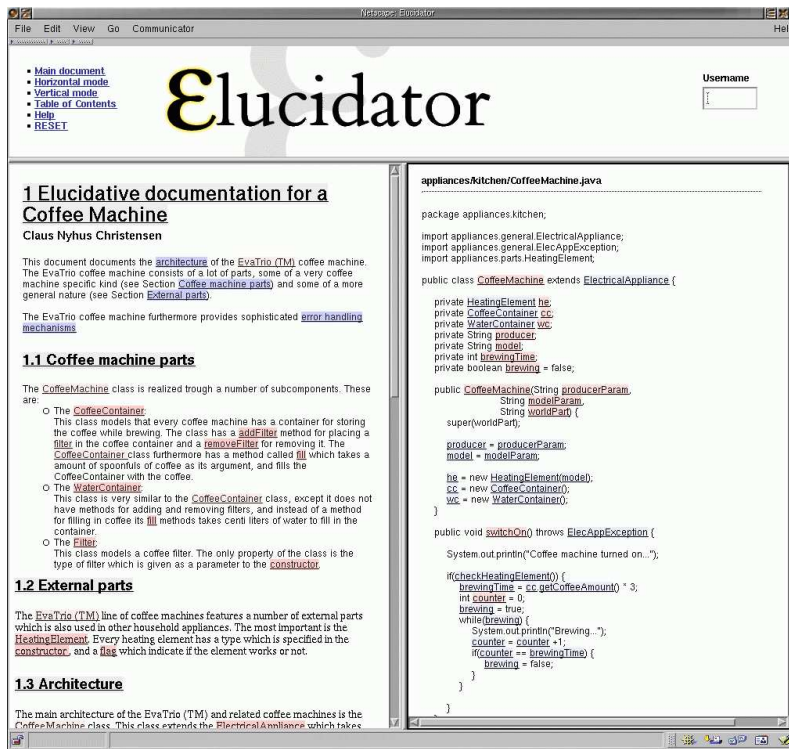


Figure 2.3: Screen capture of the Browser.

with a plethora of navigation possibilities. We have furthermore shown how these can be integrated with documentation.

**Flexible/configurable user interface:** The implementation of the Elucidator 1 makes it easy to change the look and feel of the user interface. This ensures a flexible solution which can easily be adjusted to new environments.

**Usage of standardized technologies:** This project shows that standardized technologies can be used when designing and implementing an Elucidator. It has furthermore been shown that the usage of these technologies has made it easy to use external tools in the realization of the implementation.

**Dynamic presentation of documentation and Java source code:** Our implementation of the Elucidator shows that a dynamic approach to presenting the documentation and Java source code in the browser is possible. We have furthermore shown that this solution is not slow but, on the contrary, rather fast.

The use of dynamically generated pages speeds up the processing time during abstraction of the Elucidator 1, in comparison to the Scheme Elucidator tool by Nørmark. E.g., the abstraction of the Elucidator tool itself takes about 20 seconds, as compared to the processing time of the Scheme Elucidator by the Scheme Elucidator which takes several minutes.

**Standard for Java entity names:** As a side effect of implementing the Elucidator 1 we have devised a standard for the naming of entities in Java source code. This standard solves the problem, with the normal name standard in Java being context dependent, by being non-context dependent.

**Markup of Java source code in a browser:** We have shown that it is not that difficult to markup and present Java source code in a browser.

## 2.2 Initial experiment

After finishing the work on designing and developing the Elucidator 1 tool we conducted an experiment to explore the strengths and weaknesses of Elucidative Programming and the Elucidator 1 tool.

In this section we will describe the circumstances which the experiment was conducted under, and discuss the experiences gained from the experiment.

### 2.2.1 Experiment circumstances

The strategy for the experiment was to use the Elucidator 1 to document the source code of the Elucidator 1 it self, and thereby gain experience with the tool, while documenting a real software project. The experiment was conducted by the three authors of this thesis during one week, approximately two months after the development of the Elucidator 1 was finished.

Each of the three authors was responsible for documenting different components of the system, corresponding to their main responsibilities during the development phase. This ensured that they knew the components they where documenting.

We acknowledge that the authors may have preconceived opinion towards the creation of documentation, and that this may somewhat color some of the gained experiences. It should furthermore be noted that the documentation process took place after the creation of the program was finished.

During the documentation phase the authors took notes, and thereby documented experiences gained both with the tool (the usability of features or the lack of certain features) and the documentation process. These noted constitutes the basis of the next section.

### 2.2.2 Gained experiences

The experiment added much to our understanding of Elucidative programming and our Elucidator 1 tool, as it was our first real intimate use of the paradigm together with the tool. It gave us several insights and revealed both good and bad sides with the tool and the paradigm as well — the most important is described in the following, with the ones we consider the most important at the top.

**The one long essay documentation style** When writing documentation using the Elucidator 1, the main structuring mechanisms was that the text could be split into chapters and section. This resulted in documentation which was structured as one long essay.

Explaining the programs in this form proved to be hard for both the writer and the reader. Writing easily turned monotone and it was hard to keep a consistent leitmotif in the story. It furthermore made it hard to update the documentation if sections were dependent on each other. The reader was often burden by too much information for some issues and too little for others.

**The free structure of the text** Neither the tool nor the paradigm presented limitations on how the writer chose to structure his documentation. As mentioned above it actually only required the writer to use chapters and sections as the basic building blocks and nothing said about how the contents inside these blocks should be structured.

Our initial belief was that this anarchistic kind of freedom was an advantage to the writer, but it turned out to be a burden in many situations. 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 lead to readers being left with the impression of a confusing and unstructured document.

**Parallel hypertext** The split window setup<sup>1</sup> which allowed the reader to perform parallel hypertext with documentation on the left and source code on the right proved to be an advantage while reading the documentation. It allows the reader to keep a persistent focus while reading the documentation and at the same time see different aspects of the source code. It would though be an advantage, if the reader was allowed to decide in which window he want the documentation and source code displayed. This would furthermore make it possible to view two documentation files at the same time, which we experienced would be useful.

**Tool support** During the writing phase we found tool support, especially editor support, is essential for a practical use of the Elucidator. The minimal support provided by the editor was appreciated. Especially the mechanisms for support while inserting links lessened the burden of creating links and furthermore minimized errors in the links.

We do, however, not believe this is enough and therefore other mechanisms for supporting the developer while producing Elucidative documentation, such a direct navigation via links in the editor or a more intuitive/context dependent method for insertion of links, is wanted.

Furthermore the process of abstracting the documentation and source code in order to populate the Data model, was found cumbersome. This was due to the fact that, whenever the documentation or source code was changed, all the documentation and source code files had to be abstracted in order to have them displayed correctly in the browser. A future implementation of the Elucidator 1 would therefore benefit from a

---

<sup>1</sup>This can be viewed as an example on parallel hypertext which Ted Nelson explains in his works on Xanadu [Nelson, 1999].



feature such as incremental update of the Data model. Incremental update of the Data model means only files that are actually change are abstracted during the abstraction process.

**The navigation window** Documentation produced during the experiment was primarily read from the browser as this had the best visual appeal and it was possible to jump between documentation and source at will.

During this, the navigation window was often used for finding relevant documentation about parts of the source code. It furthermore had the effect that source code was often looked at in the browser rather than in the editor, since the navigation window proved very useful while rediscovering the source code.

**Loose ends** As we have described previously the writers often found it difficult to keep a consistent leitmotif. This was especially the case when writing a part or aspects of a explanation, since the writer often came to think of some detail or related subject he needed to mention.

Being unable to make reference to a related subject, which was not described yet, made it necessary to create an empty section or chapter, with only a descriptive title. The writer then made a reference to this “loose end” and returned to it later.

The main problem with these loose ends where, that it was cumbersome to create and find a suitable place for the loose end. At the other end it was difficult to keep track of the loose ends in order to return to them.

**Post documentation** As mentioned above the documentation experiment took place about two months after the implementation of the tool had finished. Not surprisingly, this proved not to be the ideal time for the creation of documentation for source code. Even though the authors themselves had implemented the components they where documenting, and a detailed design report was present, they had big trouble remembering details of the implementation.

This experience showed us not surprisingly that documentation of source code should take place while the actual implementation is being carried out.



# 3

## Analysis

This chapter motivates, develops and describes a model, called the MRS-model, for writing and reading internal documentation in an Elucidative environment. This is done through a series of steps.

First the model is motivated. We start the motivation, by presenting a study on how internal documentation fits into the process of software development. Next we describe the participants in a software development project, and especially the two roles these participants find themselves in when it comes to internal documentation. Finally, we describe how we believe internal documentation should be utilized.

Based on this motivation we present and discuss the MRS-model for internal system documentation. Having presented and discussed the model, we next describe how we intend to realize the model in an Elucidative environment. Finally we compare the MRS-model to other related documentation approaches.

### **3.1 Internal documentation in software development**

We see documentation as a vital part of any software development process. This section presents how we view software development and how it has led us to our model of documentation.

Object oriented Software development is typically viewed as an iterative process which is divided into phases. These phases represent analysis, design and implementation, and are typically documented by a series of analysis and design documents [Mathiassen et al., 1997] which via text and/or special notation, e.g., UML describes the system and its intended architectural model. These descriptions lead to an implementation of the system.

We are primarily focused on developers and their need for documentation to uphold their program understanding. We therefore explore which situations a software developer can

be situated in when developing software, which again means that we focus on design and implementation.

We agree with Nowack [Nowack, 2000] in his identification of four abstract and generalized development cases. He divide all development into either *creation*, *examination*, *reuse* and *change*. In real world development the four cases is probably intermixed but here we view them as being ideal and pure. The advantage of this view is that we can discuss each case separately and focus on their individual characteristics.

The following will describe the four cases, what they represent and which kind of documentation is produced/needed for them.

## Software creation

Software creation is the traditional view on software development. When the software creation process is started, only an idea for the system exists. This idea, together with a set of requirements for the system is used to design a model for the solution. This model is then used to implement a solution.

**Documentation during software creation** Documentation in the software creation process normally involves the before mentioned analysis and design documents. It should however, be emphasized that the documentation should not only contain a factual description of the system. It should also document the ideas and requirements that has worked as motivations for, hopefully, rational decisions when choosing or rejecting solutions and alternatives.

A distinguishing fact from creation and the others cases is, that it is during this phase the initial documentation, as well as the first parts of the system is created.

## Software examination

Examination of software can be seen as exploration of an already existing system, with the purpose of understanding the system in question. The developer can reach this understanding by building up his own model, and thereby attempt to understand the software. This is typically done by reading the existing documentation, if any exist, or by exploring the software directly, possibly with the help of reverse-engineering tools. The model which represents the developers view of the software is then used to take some action, e.g., to perform a change, evaluate its quality/usability in a giving situation or perhaps to reuse it in another system.

**Documentation during software examination** In the examination of software, the existing documentation plays an important role as it can be used to save time for the developer while trying to understanding the software. This, of course, requires the documentation to be accessible and understandable for the developer.

Software examination is the only case where the developer is actually reading documentation, since the others cases focuses on the developer as a producer of documentation. The actual examination might not directly result in documentation, but it provides input to decisions made in the other cases and is therefore considered important.

#### Software reuse

Software reuse can be characterized as the reuse of a (sub)system in the composition of a system. It implies that examination has been performed as the developer need to have an understanding of the system to reuse.

**Documentation during software reuse** Information on how to do the actual reuse is found through examination of the system to reuse. In a ideal world the documentation of the system to reuse is already present and may even contains information on how to reuse the system. Hence, the documentation created when doing the actual reuse is not about documenting the system that is being reused, but about documenting why and how the reuse is actually done. The documentation should therefore focus on the resulting system and its interface to the reused component.

#### Software change

Change in software covers activities typically associated with software evolution, that is when developers change one version of their system to another version. Similar to reuse it implies examination, as developers need to have an understanding of the system they want to change together with a reason for why they want to change it. Change differ from reuse as it is more than just composition of two systems. Instead it consists of decomposing the existing system into parts. Some of these parts are then removed and new parts are created. The resulting parts is then composed into a new system.

**Documentation during software change** It is not enough in all situations to see only the result of changes, i.e., the description of the current system, to *make* changes. Documentation which describe past solutions, their alternatives and rationale is equally important.

Documentation of the rationales for a change is important because it states *why* something was changed. This information can be used to build and preserve developers understanding on why the system is currently implemented as it is. Description of solutions and alternatives only states *how* parts of the system is or could be realized. Thus when documenting changes it is important to not only document the chosen new solution, but also its alternatives, together with the motivations and rationales for the change.

## Discussion

This section has presented our views on software development, and how documentation should be created when viewing development in their four pure cases: creation, examination, reuse and change.

We found that all cases has a common denominator. They are all based on a rational design process similar to the one described by Parnas and Clements [Parnas and Clements, 1986] in which every decision is based on good reasons. Every solution a developer has selected or declined, is therefore to be based on a rational discussion which presents the arguments for and against the decision. Similarly, the examination of software becomes more fruitful when documentation is written with a clear distinction between rationales and solutions. This founding will later be used in the presentation of our documentation model.

Another aspect in finding how a developer needs documentation is in examining how he writes and read documentation. This will be discussed in the next section.

### 3.2 Limiting the number of participants

In a traditional software development project a number of different participants are involved, e.g., senior managers, project managers, software developers, customers and end users [Pressman, 1997]. The problems that arise when many people work together on the same software project are relevant for the field of software development in general. But since we focus on internal documentation we only consider *technical software developers*. However some of the general problems are still relevant in the context of internal documentation, e.g., does the project managers see the same need for documentation as the software developer? Given the focus for our work we choose to ignore these problems.

Instead we choose to focus on a small and generic set of roles which applies to the field of internal documentation. This focus allows us to discuss the involved participants, without muddling up the study with considerations on how relationships between the manager, customer and the software developer will effect the internal documentation. We remind the reader that internal documentation is documentation provided *by* software developers *for* software developers in a development team, and therefore do not have the same economical parameters as, e.g., user documentation, which will typically be a part of the product to be sold. We find it legitimate to leave out considerations on the relationship to, e.g., managers, since their main concern will be the documentation which is to be part of the final product. It furthermore has the noteworthy consequence that it will probably be the sole responsibility of the software developers to motivate the writing of the documentation.

We divide the software developers working in a development team into two roles; *writers* and *readers*. We define:

**The writer** to be the software developer that, during creation or reuse of some new software or during changes of existing software, *writes* internal documentation.

**The reader** to be a software developer which, during software examination, examines and hereby *reads* internal documentation in order to comprehend an existing system that is to be changed or reused.

It is important to recognize that software developers play both roles in the development team and often at the same time.

### 3.2.1 Characterizing the reader and the writer

In characterizing the reader and the writer we first look at their technical skills. Generally, participants in a development team will have different technical skills. Some might be highly educated with master or ph.d. degrees in computer science, some might have shorter programming educations and some again might be domain specialists, e.g., accountants or physicists, with additional programming education. This has the implication that terms which are familiar for one type of participant in the development team might be unfamiliar for another type of participant. An example could be that the term “observer pattern” is likely to be well known by many computer scientists but may be unknown to a domain specialist like an accountant.

Besides differences in technical skills, the software developers is likely to have different levels of experience with the system being developed or with the type of system in general. Other developers are the ones that originally created the software system at hand and will therefore understand the system much better than a software developer just assigned to the project. Some might be very experienced with a specific part of a large system but unexperienced with other parts of the system.

Despite the differences in level of experience and technical skills we believe that some common denominator of experience and skill exist. It is therefore not futile to write documentation which match all software developers with different experiences and skill level. The developer should however, still try to take the differences into account when the documentation is written.

Another matter is the writing skills of the software developer. Given that our focus is on the reader of the documentation, we make few assumptions about the writer. We recognize that ideally the writer should produce documentation of high quality. This would require the writer to have good writing skills and know and use pedagogical principles.

We see two reasons for this not being a feasible option: First, we do not believe that the typical software developer has particularly good writing skills, or know pedagogical principles that well. It is not an impossibility that he has these abilities, but we can certainly not require it from him. Second, we do not believe the writer will be especially motivated to spend a lot of time writing good, pedagogical correct, documentation in a typical development situation where he is under pressure. Today it is a problem to get the software developers to write *any* documentation at all.

As to the role of the writer, we therefore choose the lowest common denominator; The writer is unmotivated for writing documentation, he has no special writing skills and no knowledge

about pedagogical principles. When writing documentation the writer would furthermore typically be prejudiced to believe that the reader has the same technical skills and the same level of experience with the system as he has.

The other role a software developer can take in a development team is the role of the reader. We believe that it in general is difficult or even impossible to characterize readers uniformly, but we do find some common characteristics for the readers. We agree with Horn [Horn, 1992] in that there are two tendencies in readers: Holists and serialists.

**Holists** are readers that like to have a good knowledge of the big picture before proceeding to details. They read the documentation in pieces here and there. They experiment, make many hypotheses and examine them.

**Serialists** like to read the documentation step by step from start to end. They like to understand one detail before proceeding to the next.

By realizing that readers can have tendencies toward being holists as well as serialists, the internal documentation should support a “from A to Z” story about a subject as well as a large number of references to related subjects.

The readers are also differentiated on the type of work they do. As described in section 3.1 on page 17 we see four different cases of software development: creation of new software and reuse, change and examination of existing software. We acknowledge that the reader has different need for documentation depending on the type of software he is examining. For example when examining how to reuse some part of an existing piece of software the reader does not want detailed documentation of all previous version of the software. Instead he would be more interested in documentation of other attempts to reuse this specific part of the software.

This all together leaves us with a somewhat unclear and vague characteristic of the readers. They have different need for the documentation given their technical skills, their experience with the system, their tendency towards being either holists or serialists and the type of examination they are performing. Never the less, these factors will have to be dealt with in order to make the system usable for the reader.

### 3.3 Utilizing documentation

In this section we go into detail with our focus of this project. Others have already praised the effect of documentation of software, [Knuth, 1984], [Sametinger, 1992] and [Nørmark, 2000b]. We agree with these works in that *if* proper system documentation *is written* it will heighten the quality of the software. That is, the documentation will encourage the software developer to reflect on his own work and hence lessen the number of bugs and probably also make the software more comprehensible. At the other end we agree that *if* the documentation is actually *read* it can be used by software developers to reuse and change the software properly.



The problem of utilizing the documentation is twofold: First, documentation is rarely written in the first place. Second, if the documentation is actually written it is our impression that it is seldom read by others than the ones who originally wrote it [Fischer and Jensen, 1990]. We see two main reasons for these two problems: First, the software developer will not be motivated to write documentation if it is unlikely that the documentation will be read. Second, in our experience, it is frustrating to read documentation that does not suite the reading style (holistic/serialistic) of the reader, i.e., it is frustrating to read a lot of documentation in order to get a little amount of relevant information. Hence, the main motivation for this project is to utilize documentation for the reader in order to preserve the quality of the software.

By utilizing documentation we mean the software developer in the role of the *reader* should benefit from the documentation. Therefore the documentation should match the need of the reader, so the reader will be able to have his questions answered fast and easy. Preferably the software developer should have easy access to the documentation, since this will encourage him even more to write documentation.

How do we then create and provide documentation that is usable for the reader and writer ?

We start by improving the structure of the documentation. An improved structure affects both the writer and the reader. The writer can use the structure to focus his writing as the structure “guides” him. As for the reader, by having a common structure on documentation we can present to him a unified presentation of the contents. This enable the reader to faster comprehend and see what a piece of documentation is about.

By having a structure on documentation we furthermore have the possibility that the Elucidator tool can provide better navigation facilities and querying on the documentation. This makes the documentation more accessible for the reader as he can search for related documentation and explore other documentation easier.

## 3.4 The MRS-model

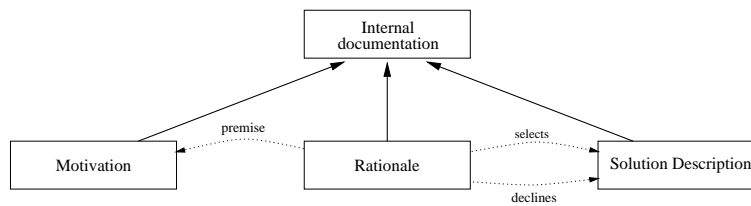
As discussed in Section 3.1 on page 17 we see software development to be based on a rational design process, in which every decision is documented by the developer by stating the motivations, rationale and selected/declined solutions.

We therefore introduces our *MRS-model* which categorize all internal system documentation into three deliberative<sup>1</sup> categories: *motivation*, *rationale* and *solution description*. Figure 3.1 on the following page illustrates the categories and their basic relationships.

**Motivation** is a description of a incentive, that gives occasion to a creation, reuse or change in a system. A typical motivation for a creation is the idea and requirements specifications. For change it can be a report about an error, or simply a fresh idea for an improvement for the system. All of these either influences or constrains the development of a system. Because of this, motivations acts as a premise to rationales.

---

<sup>1</sup>Deliberation: A discussion and consideration by a group of persons of the reasons for and against a measure [Merriam-Webster, 1997].



**Figure 3.1:** *The MRS-model.*

**Rationale** contains the arguments and other rationales for a selection or declination of specific solutions. Rationales should mention a number of motivations together with arguments for a chosen solution as well as alternative solutions that has been declined for some reason.

**Solution description** are factual description of the solutions that constitute the software system. These descriptions can have many different forms, e.g., UML diagrams, literate text etc.

The division into these deliberative categories is furthermore inspired by [Rüping, 1998] and [Sanvad et al., 2000]. Rüping and Sandvad et al. also state that the description of actual solutions and their rationales is equally important, but possible subjective arguments in rationales can muddle up a description of a solution.

Rationales is, as mentioned before, important when trying to build a complete understanding of the system. However as noted by [Parnas and Clements, 1986], if a developer just needs to use a specific part of the system, he is only interested in finding a description of the system explaining how it actually works and not in reliving the complete history of the system.

The opposite situation occurs when a developer wishes to change a system. In order to have the full understanding of the system it can be necessary to understand the evolution of the system in detail. In this situation, having documentation of the changes made to the system, will become important.

The categorization of documentation is not just a logical categorization, but also physical. A physical separation of documentation into motivations, rationale and solutions will help support the writer in focusing his writing, but even more important, help readers in finding correct information and support both holists and serialists.

### 3.5 Realization of the MRS-model

This section describes how the MRS-model presented above is intended to be realized. We do this by first describing two structuring methods which has served as inspiration to our choice of how to realize the MRS-model. We then relate these to the experiences gained during the initial experiment (see Section 2.2 on page 13 for detailed information on this experiment and its results). Having done that, we describe the three main components in the

realization of the model, namely documentation nodes, relationships in documentation and navigation in the documentation.

### 3.5.1 Documentation structure

In 1965 Tracey, Rugh and Starkey presented a method for doing Sequential Thematic Organization of Publications, also known as the STOP method [Tracey et al., 1999]. One of the characteristics of the STOP method, is that the documentation is divided into small stories, called *Topical Modules*. Tracey, Rugh and Starkey, gives the following description of these:

*Because it has obvious boundaries (both physical and editorial) and an appropriate capacity, the self-contained theme of two-page proportion becomes a prescription for thematic coherence that is more objective to the author and reviewer, while being compatible with the natural behavior of the author and reader.*

[Tracey et al., 1999]

A Topical Module is normally structured in a predefined way. It consists of:

- a topic title, that characterize and introduces the contents of the Topical Module, and not merely categorize it.
- an abstract, which will serve as a thematic window for the reader.
- a left side page with text
- a right side page with illustrations and/or text. Each text page contains no more than 500 words, yielding a maximum Topical Module length of 1000 words, if no illustrations are used.

Another approach for structuring documentation, is Structured Writing and the Information Mapping method [Horn, 1999]. Structured Writing relies on a systematic and complex view on how to create and structure documents. It relies on two main components: *information blocks* and *information types*.

Information blocks are considered the basic units in the approach. More than 200 common information block has been defined, for usage in different document types. Examples of information blocks are: table, fact, rule, decision table etc. Information types are then considered to be clusters of information blocks. Structured Writing defines seven information types: structure, concept, procedure, process, classification, principle and fact.

Another characteristic of Structured Writing is that it find it important to have properly defined topics for each block or cluster of information. This is important in order to help the reader quickly scan the contents and to understand the structure of the document.

According to Horn [Horn, 1999], the Structured Writing method has a number of similarities with the STOP methods, but at the same time it is very different. The two main similarities

between STOP and Structured Writing, is that they “*were both interested in better comprehension on the part of the reader, and both identified the method of writing as part of the problem*” [Horn, 1999].

We find both of the approaches has similarities to our work as well. The biggest difference between STOP and Structured Writing is the level of structuring detail the methods provide for the writer to use when writing documentation. The STOP method only provides topic, abstract and the usage of two-page chunks, while, as mentioned above, Structured Writing provides both a number of information blocks and types. As this thesis will show, our approach lay somewhere in between these two approaches.

The documentation for the STOP method [Tracey et al., 1999] is actually written by using the STOP method. This has given us firsthand experience with the method as a reader, and this experience is positive. The short stories made the documentation easy and comfortable to read, while the topic titles and abstract helped to provide a good overview of the documentation.

The structure of a STOP document is very different from the structure of the documentation produced in our initial documentation experiment (see Section 2.2 on page 13 for details on this experiment). However, one of the results of the experiment complies with the STOP method: Neither the writer nor the reader liked one long essay.

Another result of the experiment showed that with the structuring mechanisms used, the writer himself always had to come up with a structure for a given piece of documentation, and, while reading, this left the reader with the impression of “no structure”. This result fits well with both the STOP method and Structured Writing.

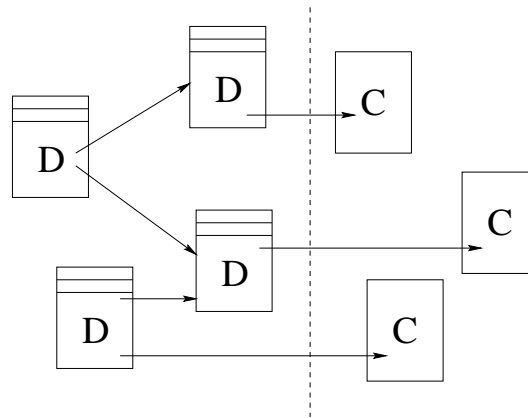
The two above mentioned methods and the lessons learned during the mentioned experiment, has inspired us to base the realization of the MRS-model on small self-contained nodes, tied together by links. In other words, we see it as natural to use hypertext to realize our model. Our definition and understanding of hypertext is the one presented by Jeffrey Conklin in [Conklin, 1987].

### 3.5.2 Documentation nodes

As mentioned, our documentation will be divided into hypertext documentation nodes. All though the source code is not considered to be documentation it also have to be a part of the hypertext network since we need to create links to it. We therefore think of source code entities (typically classes) as source code nodes. These nodes can be thought of as implicitly defined nodes. The documentation and source code nodes, as well as their relationship are illustrated in Figure 3.2 on the next page. The dashed line indicates that the documentation and the source code is physically separated. This comply with the third requirement for Elucidative Programming [Nørmark, 2000b], stating that the source code must be kept intact without surrounding documentation.

Although the source code are considered important, the main focus of this project is the documentation to be produced. Therefore, when referring to a node or hypertext node in the

rest of the thesis, it should be considered a documentation node, except when it is explicitly stated otherwise.



**Figure 3.2:** *Documentation and source code nodes. The dashed line indicates that the documentation and source code is physically separated as required by the Elucidative Programming paradigm.*

### Placement of the documentation nodes in the MRS-model

The MRS-model states that the documentation should be separated both physically and logically in deliberative categories. We have furthermore decided to separate our documentation nodes in hypertext nodes with a focused content. This leads us to a natural categorization of the documentation nodes.

We categorize the documentation nodes, such that they have a focused contents that apply in one of the deliberative categories; Motivations, Rationales and Solution descriptions, i.e., the contents of a documentation node will be either a motivation, a rationale or a solution description.

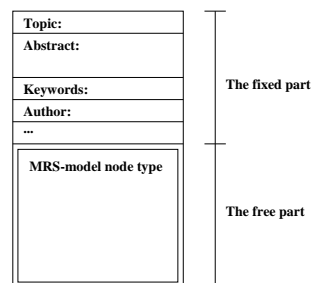
This leads us to introduce a node type for the documentation nodes. This type indicates the contents of the documentation node, e.g., a documentation node of type “Requirement” will have a motivating content. It will be possible to have several node types in one deliberative category. This will be discussed in further detail in the Design chapter, specifically in Section 4.2.1 on page 46.

### Internal structure of the documentation nodes

Inspired by the STOP method [Tracey et al., 1999], we urge, but does not demand, for the writer of documentation nodes to keep them short (no more than 1000 words and hopefully less), in order for the nodes to be manageable for the reader. If a subject to be documented is too big to fit in a documentation node, it can probably be divided into smaller nodes, which can then be connected by links.

Furthermore, also inspired by the STOP method, the writer should strive to make the nodes as self-contained as possible, meaning that whenever possible a node can be read without having to read other nodes first. The reason for this, is that it is far easier for the reader just to read one node, instead of having to switch back and forth between two or more nodes. This should however not restrain the writer from creating links to documentation nodes when needed, since it is also important that the information in a node is focused on the subject and without too many side-anecdotes.

Besides the textual guidelines to the contents, the documentation nodes can be characterized by their internal structure. This structure is prescribed by the type of the node, i.e., by the deliberative category. This internal structure of the documentation node is illustrated in Figure 3.3. The structure of a documentation node has two parts: The fixed part and the free part.



**Figure 3.3:** An illustration of the main internal structure of a documentation node. The fixed part contains common general information which is applicable for all documentation nodes, while the structure of the free part depends on which MRS-model node type is applied to the documentation node.

**The fixed part** can be viewed as a header for the node. The purpose of this part of the node is to provide the reader with a set of common general information for all nodes. It could, e.g., contain information such as the topic of the node, an abstract for the node, the author of the node etc.

**The free part** The textual contents of the free part is at largely decided by the writer. It must however follow the structure of one of the categories of the MRS-model mentioned above. This structure is specified by the node type, and therefore all node of a specific type have a common structure for the free part. As an example, a documentation node in the Motivation category, could have the type “Requirement” which means that this particular node is documenting some requirement posed to the software project.

In the Design chapter (specifically Section 4.2.3 on page 52) the exact contents of the fixed part, as well as the node types of the MRS-model will be presented, described and discussed in detail.

This characteristic of the documentation nodes complies with Conklins description of hypertext nodes [Conklin, 1987]. Conklin state that hypertext nodes in most cases expresses

a single concept or idea, and they are typically smaller than a traditional text file (although nothing prevents the writer from making the nodes big). He furthermore describes that node may have types, and these are particularly useful to differentiate the nodes when they have an internal structure, as it is the case with our documentation nodes.

### 3.5.3 Relationships in documentation

In the MRS-model described in Section 3.4 on page 23 the Rationale category has a *premise* and it *selects* and/or *declines* solutions. In other words these three words describes the relationships between the three categories. In order to realize the model, we therefore need some mechanism to express this.

As we are using hypertext to realize the MRS-model, relationships between nodes will be expressed as links. In the following we will describe the most important aspects of how links are to be used in realizing the MRS-model. This is done by characterizing a number of properties of the hyperlinking concept we use.

#### Roles on links

From the description of the relationships in the MRS-model above, it is clear that it is not sufficient to just make links between the documentation node. We also need to give these links a meaning (such as *premise*, *selects* and *declines*).

Links in hypertext can have attribute/values pairs placed on them [Conklin, 1987]. We use this feature to introduce *roles* to our links. The roles on our links is used to express the kind of relationship two documentation nodes is involved in. An example could be that a Rationale node makes a link with the role “*premise*” to a Motivation node, and thereby expressing that the Motivation node is a *premise* for the Rationale node.

The number of link roles is not limited to the three before mentioned link roles, as it can be useful to have more specialized link types when expressing the relationship between two nodes. For example a link type which states where a solution description is implemented in the source code could have the role: “*implements*”. In this report we will present a number of link roles which will be implemented in the Elucidator tool, but the writer of the documentation is free to define new link roles if the need arises. The link roles presented in this report is described and discussed in the Design chapter (specifically in Section 4.3.2 on page 57).

#### Anchoring

All links in our model is considered to be directional. The Elucidator tool should however, provide navigation functionality to support going backwards along links.

Both the source and the destination anchor of a link is considered to be a region. In the case of the source of the link, the region will be a contiguous set of characters placed between the

begin and end link tags, and in the case of the destination of the link, the region will be either a whole documentation node, or some part of it such as a section.

Documentation nodes always has a unique id which is to be used when stating the destination of a link. Specific parts of a node, such as sections, can also be given a id to be used in anchoring. This id is used as a postfix to the nodes id to uniquely identify the specific part.

### Implicit and explicit links

Links can be created in two ways: Explicitly or implicitly. Explicit links are links manually created by the writer, while implicit links are created automatically by the system. In the following we will discuss explicitly and implicitly created links in detail.

**Explicit links:** The first method for creating links is to let the writer make them *explicitly*. We believe the writer will insert these links as the result of a reflection made while writing the documentation, and they will therefore express some comprehension held by the writer at the time the link is created. If this comprehension is transferred to the reader, the links will be of value to him. Therefore, the explicit links is potentially valuable links for the reader.

Since we believe that the explicit links are valuable for the reader, we need to encourage the creation of these links. A potential problem with the creation of explicit links are that they need to be created by the writer, while it is mainly the reader who benefits from links. A straightforward solution to this problem could be to make the links useful for the writer as well. This could, e.g., be done by using the links to provide navigation facilities, which can then be used by the writer, while he is developing the software. An example of this could be the Navigation Window found in the original Elucidator tool [Christensen et al., 2000]. The problem of navigating the MRS-model will be discussed in Section 3.5.5 on page 32. Another potential problem with the creation of explicit links is the burden of the work involved in the creation of the links. This problem can however be relatively easy lessened by providing good and easy editor support for the task.

**Implicit links:** As stated above a number of potential problems exists with the explicitly creation of links. A solution to these problem could be to create as many of the links as possible *implicitly*. This means that we let the system, instead of the writer, create the links.

As nice as this sounds, creating links implicitly is not without problems. First of all, since the links are created by the system and not by the writer, it is not likely that the comprehension held by the writer will be expressed and transferred to the reader in the same degree as we believe it will by creating explicit links. This results in implicit links being probably less valuable to the reader. Secondly, it is not an easy task to have the system determine when a links is appropriate, and to what other node it would be appropriate to link to.



Realizing this, we believe that implicitly created links are not meant to replace explicitly created links, but rather to compliment them. An example of how this could be realized would be to link keywords in a node to other nodes which have this specific keyword in either its topic, abstract or keyword list.

In conclusion, this means that implicitly links can be characterized as links who are meant to coexist with explicit links. They are created by the system and they tie nodes together which *probably* has some coherence.

### Organizational and referential links

The final characteristic of the links used in the MRS-model, is that they can be divided into two categories: organizational and referential.

The idea for these two categories is inspired by Conklin [Conklin, 1987]. The main idea is that organizational links are used to connect parent nodes with its children nodes. This results in a hierarchical devision of the nodes in a subgraph within the hypertext network, and can be used to express, e.g., that certain nodes documents details of other nodes.

The referential links is by far the most used links in the MRS-model. These links are used to tie the nodes together in a non-hierarchical manner. Examples of this category of links could be the premise, selects and declines links discussed above.

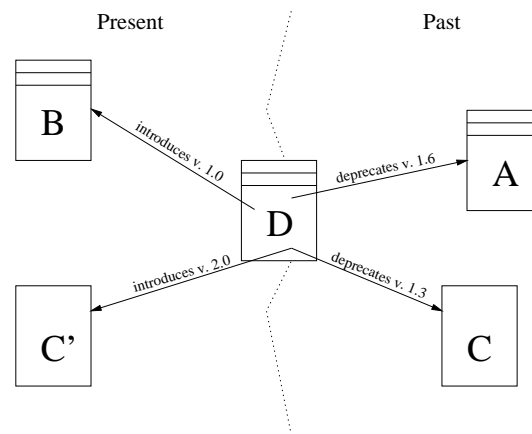
#### 3.5.4 The usage of relationships to maintain the history

As mentioned in Section 3.1 on page 17 documenting the rationales for a change is important. This means that the history of a program is to be retained in the MRS-model. In order for this historical information to be of value to the reader, it must somehow be possible to express it while writing the documentation and/or source code.

A writer would like to precisely state in his rationale description what is introduced and what is deprecated by a change. To do this we see it as natural to use the roles on links to express either the introduction or deprecation of a node, similar to selects and declines as described above.

Furthermore it would be naturally to be able to state which specific version of a node a link points to. The link could contain an attribute that contained version identification for a node which could be used to access the specific instance of a node through a version control system.

Figure 3.4 on the next page shows an example of a documentation node (D) which deprecates some documentation (A) and source code (C). It also introduces a new documentation node (B) and a newer version of the source code (C'). We have though limited our selves in this thesis to only use the roles to express history on links, and do therefore not take into consideration the problem of versioning.



**Figure 3.4:** *Maintaining history with links. A documentation node can use version numbers on links to model which specific version of node is deprecated (part of the past) and which versions is introduced (part of the present)*

### 3.5.5 Navigation in documentation

One of the problems arising by using hypertext to realize the MRS-model are what Conklin describes as the *disorientation problem* [Conklin, 1987]. Since hypertext allows you to organize your documentation in a somewhat complex manner, it can be difficult to know where you currently are in the network, and how to get to some other place in the network. Hence, you got a disorientation problem. Our main medicine to solve this problem is to provide navigation facilities to the reader. This section describes how navigation is to be realized in the MRS-model.

One of the problems of the disorientation problem, is to maintain the context of what you are reading when you follow a link. The setup in the current Elucidator tool help lessen this problem, since the two side-by-side windows allow you to look at both documentation and source code at the same time. See Figure 2.3 on page 12 for an example. A problem with this setup however exists. If you follow a documentation link from one piece of documentation to another, the original implementation of the side-by-side windows will not help you, since you can only view one piece of documentation at the time.

To improve on this we remove the strict separation with documentation on the left and source code on the right. Instead, the reader are to chose in which window he want to have the documentation or source code viewed. This will allow him to, e.g., look at two documentation nodes at the same time, and thereby keep the context he were in when he activates the link. This can introduce a higher mental load when following a link, as the reader now has to decide if he want to show the contents of the link in either the current or opposite window. We feel though, that with small means we can reduce this load to minimum. How this is specifically realized can be seen in Section 4.4.2 on page 64.

Another problem with navigation in hypertext, also raised by Conklin [Conklin, 1987], is when the reader are to decide if he wants to follow a link or not, he typically do not have

much information about the contents of the destination of the link (he can typically only see the name of the link). This makes the job of choosing to follow a link or not uncertain, and the reader will almost certainly end up following links to documentation with no interest for him.

To solve this problem, the realization of the MRS-model should provide to the reader, information about the destination of a link, before he actually navigates to the destination of the link. Examples of the provided information, could be the topic or the abstract of the destination node. He can then, with regard to this information, choose if he wants to follow the link or not.

### Views on documentation

Another way to provide the reader with navigation facilities is by providing what we call *views*. A view can be seen as a subset of the documentation nodes in the hypertext network, presented in a manner so only certain elements from the node, such as the topic or the abstract, is showed. The view furthermore contains a number of implicit links which takes the reader to the actual nodes. This notion of views is very similar to what Nørmark and Østerbye describes as an *outline presentation* [Nørmark and Østerbye, 1995].

We see two basic types of views: the Context views and the Index views.

**Context views:** As the name implies, Context views are views which show information on the context of a node. An example could be a view which show the topic and abstract of all the documentation nodes linked to by a specific documentation node. One could also imagine that the role of the links is used to refine the Context view. A Context view is always invoked on one specific documentation node, or on a specific element in a documentation node.

**Index views:** Index views are views which show some subset of nodes from across the whole hypertext network, according to some specified parameter. Typical examples of a index view could be a table of contents, or a list of all documentation nodes in the Motivation category.

No finite number of views exists. In this thesis we will present a number of views which will be implemented in the Elucidator tool. The views presented in this report is described and discussed in the Design chapter, specifically in Section 4.4.3 and Section 4.4.4 on page 68.

## 3.6 Related documentation approaches

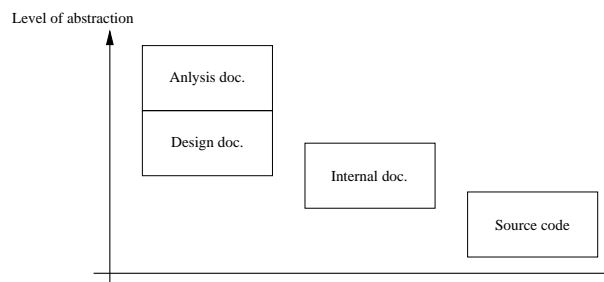
In the previous sections we have presented the principles of our MRS-model and how these are to be realized through the use of documentation nodes with short focused contents and links with roles. In this section we present a number of related approaches within software documentation. These all try, at various degrees, to maintain quality of software through the

use of documentation. The purpose of this presentation is to underline the differences and similarities between our model and these related approaches.

### 3.6.1 Object-oriented Analysis and Design Documents

In a software development process that follows the guidelines of Object-oriented Analysis and Design [Mathiassen et al., 1997], the software developers produce a number of different documents, e.g., requirements specifications, analysis documents, design documents and test specifications. These documents, together with the process that created and use them, all seek to maintain the quality of the software, with the exception that these documents ensure quality in a wider sense, i.e., they target the product that is to be delivered to the customer as well as quality of the process — not just the software source code. An example could be, that they address qualities as conformance with specifications, good performance and adherence to schedules.

The documents produced during object-oriented analysis and design also differ from internal documentation (as produced in an Elucidative environment) in that they address the software at a different level of abstraction, as illustrated on Figure 3.5. An example could be that they describe conditions in the problem world and not details, e.g., on how some logging mechanism is implemented. Still, there is some overlap between internal documentation and a design document, since these both describe the architecture of the source code.



**Figure 3.5:** *Internal documentation compared to analysis and design documents at their different levels of abstraction.*

In our opinion the traditional analysis and design documents have two important problems: First, they are often documents with no proximity or relations with the actual source code. This makes it difficult to maintain cohesion between the documents and the source code as the source code evolves. Second, they do not contain the knowledge gained at the implementation, which effects the implementation. Both these problems are targeted by the MRS-model and internal documentation in an Elucidative environment.

### 3.6.2 Literate Programming

As opposed to the documents from object-oriented analysis and design, Literate Programming makes documents that in more than one sense are more close to the source code. First of all, since the source code resides inside the documentation there is a high coupling between the two. As we described in Section 1.2 on page 3 this proximity in literate documents makes it easy to keep the documentation and source code coherent. Secondly, the documentation produced by Literate programming describes the actual source code.

A number of different variations of Literate Programming exists — Elucidative programming being one of them. If we examine the original Literate Programming as suggested by Knuth [Knuth, 1984], one of the major problems is that is based on a paper representation, which makes it cumbersome to navigate from one part of the documentation to a related part. This has been improved upon in one of the of the Literate variants. Markus Brown et. al. has suggested [Brown and Childs, 1990] and created [Brown and Czejdo, 1990] an interactive environment for Literate programming, which apply general hypertext concepts in order to make indexes with hyperlinks between related parts of the documentation.

The hypertext idea is extended by Kasper Østerbye [Østerbye, 1995], where the documentation and the source code are placed in hypertext nodes. In this tool the proximity is ensured with hyperlinks like the Elucidative environment. Experiences from small examples conducted in [Østerbye, 1995] show that both motivations and rationales appear next to the description of the solutions, which support the basis of our MRS-model. Although the tool offered hyperlinking these all had to be inserted manually which made the tool cumbersome to use.

For all of the presented Literate variants we see a number of problems. The main problem is that they all, although in different ways, change the inherent structure of the source code. Either by letting the source code be a part of the documentation or splitting the source code up in small hypertext nodes. The advantage of this approach is that in this way it is possible to emphasize some subpart of a entity in the source code, e.g., a subpart of a method. On the other hand, the problem with this approach is first of all that the language or tool mechanisms to control this modularization complicate the documentation process. Furthermore, the source code is so tightly entwined in the documentation tool that it becomes difficult to use other development tools on the source code. By moving both the reader and the writer away from the tools and mechanism they are accustomed to, such as, e.g., their favorite editor, we move them away from writing documentation.

Although experiments show that Literate documentation is used [Fischer and Jensen, 1990], these experiences appear to be from a small and homogeneous set of developers. Hence we still have doubts on the general usability of traditional Literate Programming.

### 3.6.3 Object-oriented Documentation

In 1994 Johannes Sametinger suggested a documentation scheme called Object-oriented documentation [Sametinger, 1994]. This work contributed in two main areas: The first area re-

sulted in a suggestion for a classification of system documentation. The second area focused on reuse of documentation, by using inheritance. In the following we only consider the first area.

As mentioned the first area of contribution resulted in the suggestion for a classification of system documentation, where system documentation is the documentation used by software developers. This classification divides the documentation into static and dynamic documentation on one axis, while overview, external and internal documentation is placed on the other, yielding six different categories of documentation. This is shown in Figure 3.6.

	<b>Overview</b>	<b>External view</b>	<b>Internal view</b>
<b>Static view</b>	Static overview	Class interface description	Class implementation description
<b>Dynamic view</b>	Dynamic overview	Task interface description	Task implementation description

**Figure 3.6:** Documentation scheme for object-oriented software systems [Sameting, 1994].

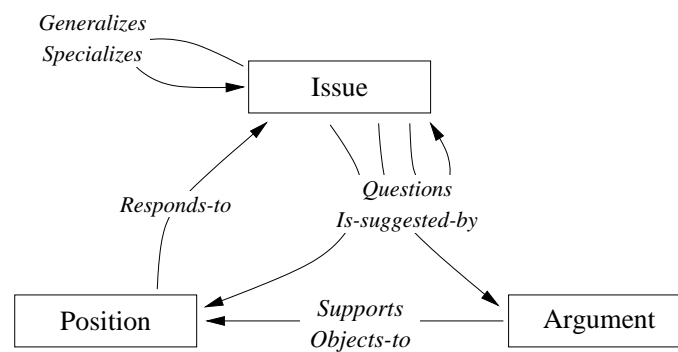
We believe that documentation made in an Elucidative environment (internal documentation) can be successfully used in all six categories. However some of the categories may be better suited than others. As an example, documentation to be placed at the internal level, both dynamic and static, seems to fit well with our notion of internal documentation written in an Elucidative environment. We do however, recognize that some other types of documentation may be better suited for some of the categories. Most obvious is the external documentation where interface documentation systems such as JavaDoc could be used with success.

### 3.6.4 The gIBIS hypertext tool

A tool that uses a model similar to the MRS-model is the gIBIS tool, created by Jeff Conklin and Michael L. Begeman [Conklin and Begeman, 1987]. gIBIS is a hypertext tool to support team design deliberation, using the IBIS method.

The IBIS method sees a design deliberation as a conversation among the participants. The model of these conversations focuses on *Issues*. The issues can have *Positions* that solve the issue, with possibly many mutually exclusive positions. The positions can in turn have one or more *Arguments* to support or object to that position. This is illustrated in Figure 3.7 on the facing page.

gIBIS works by supporting the creation of a network of hypertext nodes, being either an issue, a position or an argument. Typed hyperlinks are inserted between the nodes to state the role between the two nodes. The gIBIS tool works by one user starting a deliberation of an issue by adding an issue node to the network. The users of the gIBIS system then add a number of positions as a response to the issue raised and arguments to support or object to the position. The network can then be navigated using a graphical representation as well as indexes.



**Figure 3.7:** *The set of legal rhetorical moves in IBIS [Conklin and Begeman, 1987]. This is equivalent to the nodes and links in gIBIS, where the boxes are nodes and the arrows are hyperlinks.*

The IBIS model resembles our MRS-model, where the Positions in the IBIS model are equal to Solution Description in the MRS-model. The arguments equals the Rationales. The Issues resemble the Motivations of the MRS-mode, since they both initiate the construction of a position/solution description.

The main difference between the Elucidator tool and the gIBIS tool is focus of the tool. The gIBIS tool tries to support and control a design process, while the Elucidator focus on internal documentation.





# 4

## Design

This chapter present the detailed design for how the MRS-model is implemented in the Elucidator environment. The Elucidative environment presented to establish a foundation for the design and implementation. Having established a foundation, we move on to describe the three main issues of the implementation. First we describe how the documentation nodes is designed and implemented. Next, we focus on the links. Finally, we describe how the navigation facilities and the views are designed and implemented.

### 4.1 The Elucidative environment

The design and implementation presented in this chapter is build upon a environment which was designed and implemented in the first part of this master thesis. The main components of this environment is already presented in Section 2.1 on page 9. Further design details is given in Chapter 4 of [Christensen et al., 2000].

In this section we will focus on describing the Elucidative environment of both the Elucidator 1 and 2, as well as the terminology used in these environments. This is done in order to provide the reader of this thesis with background information on the environment.

The rest of the section is structured in the following manner: First the two languages used by the Elucidators is presented. Secondly, the entities contained in these languages is defined. Following this, the tools present in the Elucidator 1 and 2 environment is described. Next, the changes which will have to be made to the different component of the Elucidator 1, in order to implement the MRS-model and thereby evolve it to the Elucidator 2 is presented. Finally, we discuss the work flow when working using an Elucidative environment.

Other minor components/concepts than the ones described in this section exist in the Elucidative environment. If important, these will be dealt with in their appropriate section of this design chapter.

### 4.1.1 The languages in the Elucidative environment

In the Elucidative environment two languages is handled: Java and the EDoc language.

**The Java language:** The Java language is the only programming language supported in both the Elucidator 1 and 2 tools. The role of the Java language in the Elucidative environment, is to serve as the development language used by the software developers to implement the specific software project they are documenting in the Elucidative environment.

An important point concerning the Java language, is that every entity, such as a classes, methods or fields, in the source code must have unique names. These unique names are to be used when creating links from the documentation to a specific entity in the source code. We call this unique name the *idname*. As described and discussed in [Christensen et al., 2000, pp. 27–31] a number of problems arises when creating the *idname*, problems which we solve by introducing a naming standard for Java entities. We will not go into detail concerning this naming standard, but just state that it is also used in the Elucidator 2 tool.

**The EDoc language:** The EDoc language is a XML based mark-up language which is used to write the documentation. The purpose of the EDoc language is twofold: First, it provides a language which the writer uses to express structure and links in the documentation. Second, it provides information to the Elucidator tool, which can be used to present the documentation typographically.

### 4.1.2 Entities in the Elucidative environment

Entities is the fundamental elements recognized by the Elucidators. The Java and EDoc languages described above is used to define all source code and documentation entities known by the Elucidators.

Source code entities is source symbols that can be uniquely identified, thus packages, classes, methods, fields etc. is entities. For documentation basically all elements defined in the EDoc language is entities. Thus documentation nodes, sections, link anchors and all other structural parts of documentation nodes is entities. As with source entities, all documentation entities can be uniquely identified. The technical details about uniquely identification of both kind of entities can be found in [Christensen et al., 2000, Chapter 4].

Note that entities can contain other entities. This depict the fact the e.g. methods are inside classes and sections are inside documentation nodes. This result in a containment relationship between the two entities. Other relationships exists and this information about the entities in the Elucidators is stored in the Data model as mentioned in Section 2.1 on page 9. Again, more technical details can be found in [Christensen et al., 2000, Chapter 4].

### 4.1.3 The three tools in the Elucidative environment

The Elucidative environment consists of three tools: The Editor, the Browser and the actual Elucidator. In this section we will describe the role of these three tools in the Elucidative environment, and how they interact.

**The Editor:** The first tool in the Elucidative environment is the editor. The editor is one of the two components in the user interface to the Elucidator, the other being the browser. The editor is used by the writer to produce both the source code and the documentation. Conceptually, any editor which can be controlled by a programming language can be used. In the Elucidator 1 and 2, only the Emacs editor is supported though.

The editor is extended with a set of functionalities to support the writer working in the Elucidative environment. First of all, it features a split-view setup with separate frames for the documentation and source code, thereby making it possible for the writer to easily work on the documentation and source code at the same time. Since the insertion of links can be rather cumbersome, the editor furthermore provides functionality to help the writer do this. Finally, the editor provides the ability to start an abstraction process directly from the editor.

Besides functionality to support working in the Elucidative environment, the editor supports some common software development functionality. This is realized by using a number of third party packages. An example of this kind of support, is the usage of the Java Development Environment package [Kinnucan, 1999], to provide functionality such as lexical highlighting and the ability to compile the Java source code from within the editor. We also use a package which turns the editor into a structured xml editor, which support the writer in writing his documentation using the EDoc language.

**The Browser:** The browser is the second component in the user interface to the Elucidative environment. While the editor is used by the writer, the browser is used by the reader.

The purpose of the browser is to let the user of the Elucidator have an attractive view and functional way of reading the produced documentation and source code. It also provides him with navigation functionality. This functionality can be used in various ways, both internally in the documentation or source code, and between the documentation and the source code.

The setup of the browser found in the Elucidator 1 environment is similar to that of the editor, since it has a split-view, which presents the documentation on the left and the source code on the right. The navigation functionalities mentioned above, is basically implemented as links in the documentation and source code. More advanced navigation features is however implemented by the usage of a small extra window, called the Navigation Window. This window provides information on, e.g., which links are created in a particular section of the documentation, or which other sections makes links to this particular section. The information used to provide the navigation facilities is gathered by the Elucidator tool, as it will be described below.

In principle any Internet browser can be used. The Elucidator 1 environment has, e.g., been tested with Netscape Communicator on the Solaris, Linux and Windows

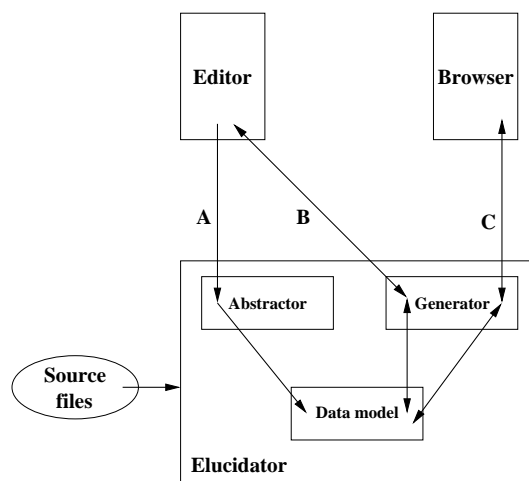
operating system and Internet Explorer on Windows. However, some of the navigation functionality to be implemented in the Elucidator 2 environment requires the usage of the Netscape Communicator browser.

**The Elucidator:** The Elucidator is considered the main tool in the Elucidative environment, since it provides the core functionality for the environment.

The Elucidator can be seen as three components which cooperate to provide the full functionality of the Elucidator. The first of these components is the Abstractor. The job of the Abstractor is to abstract information about entities, and relationships among these entities, from the documentation and source code. This information is then passed on to the next component; The Data model. The job of the Data model is to store the information provided by the Abstractor, and to enable the third component, the Generator, to query and access this data. The data is retrieved by the Generator in order to facilitate the two user interface tools mentioned above. Before delivering the data to these two tool, the generator formats it according to the needs of the two tools.

The Elucidator is implemented in Java, as a server, using the Servlet technology [Davidson and Coward, 1999]. It communicates with the two interface tools through the HTTP protocol [Fielding et al., 1999].

Having described the three tools in the Elucidative environment, we now take a more detailed look at how they interact. Basically three interaction scenarios exist. These are illustrated in Figure 4.1. In all three scenarios the Java source code and EDoc files are used. This is shown in the figure by the arrow from the source files to the Elucidator.



**Figure 4.1:** *Interaction in the Elucidative environment. In case (A) the writer asks for an abstraction of the documentation and source code. In case (B) the writer uses the information gathered by the abstraction to help him insert a link in the documentation, and finally in case (C) the reader has activated a link in the browser which means that a new page, produced by the Generator, will be shown in the browser.*

- (A) The first basic interaction in the Elucidative environment occurs when the writer, through the editor, asks to have the documentation and source code abstracted. The first step in this interaction, is that the Abstractor components of the Elucidator tool make an abstraction of the documentation and source code. The next, and final, step is that the Data model components stores the resulting data from the Abstraction.
- (B) The second basic interaction appears when the writer wants to insert a link with the help of the editor. This functionality is implemented in the following manner: First the editor asks the writer to supply some substring of the name of the destination entity the writer wants to link to. Next, this substring is passed from the editor to the Generator. The Generator asks the Data model to retrieve possible matches to the substring, which is in turn returned by the Data model to the Generator. This result is then passed from the Generator to the editor, which presents the result to the writer. The writer can then select the correct name for the entity he wants to link to.
- (C) The final basic interaction in the Elucidative environment takes place between the browser and the Elucidator, and is triggered by the activation of some link in the browser. When selecting a link, a new page containing the destination of the link should be shown. In the first step of this interaction, the browser passes an idname representing the destination of the link to the Generator. The Generator next contacts the Data model to retrieve the data necessary for it to present the destination of the link. After retrieving this data the Generator generates the page by using the information retrieved from the Data model, as well as the source files. Finally it passes the resulting document to the browser, which then presents it to the user.

#### 4.1.4 Changes to the Elucidator

In order to implement the Elucidator 2 tool a number of changes has to be made to the tools in the Elucidator 1 environment. This section present the most important ones.

**The EDoc language:** In order to express the MRS-model the EDoc language has to be changed. This means that some of the old tags such as `<chapter>` will be removed from the language, and a set of new tags will be introduced. The grammar for the changed version of the EDoc language can be seen in Appendix B.

**The Abstractor:** The only part of the Abstractor which needs to be changed is the part that abstracts the EDoc files. This is done in order to be able to abstract the new entities and relationships introduced in the extended EDoc language.

**The Data model:** Since the entities introduced in the extended EDoc language in some cases has different structure/data contents, compared to the one found in the first version of the language, the Data model has to be changed, in order to be able to store these changes. Furthermore, the Query engine, which is the part of the Data model which handles the retrieval of data, has to be change as well in order to be able to provide this new information to the Generator.

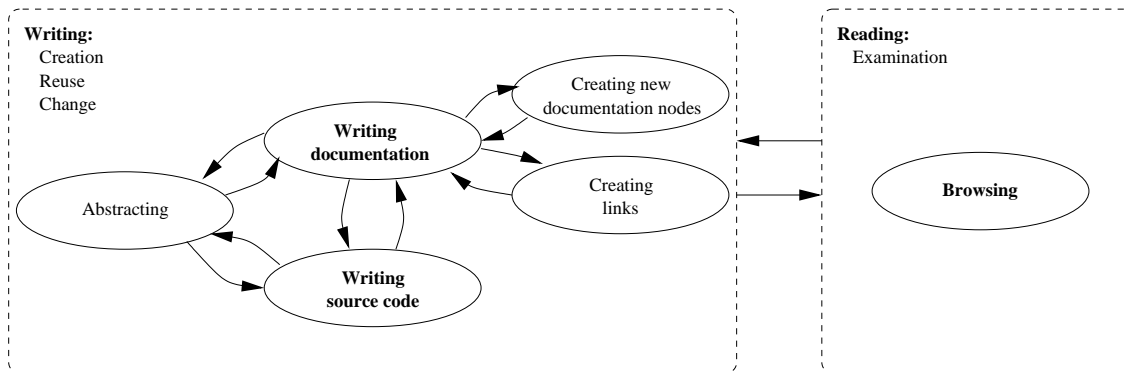
**The Generator:** The Generator is the component which has to be changed the most. As a consequence of the new structure of the documentation, the documentation has to be presented in a completely new way. New functionality, such as extended navigation features and views, has to be implemented as well.

**The Editor:** The changes to the editor will only be minor. The most important is support for templates for the creation of the documentation nodes.

#### 4.1.5 The work flow in an Elucidative environment

Having described the languages, tools, interactions and improvements to the Elucidator 2, we now turn to a description of how the Elucidative environment is typically used.

As mentioned in the analysis, the software developer is either in the role of the reader or the writer. The two main activities while using the Elucidative environment will thus be reading and writing. It was also mentioned that the software developer can be in both the reader and writer role at the same time. While using the Elucidative environment this means that he shifts between the two activities at will. While performing one of these activities, the software developer can furthermore be in one of a number of different states. This division into states can be seen in Figure 4.2.



**Figure 4.2:** An illustration of the work flow in the Elucidative environment. The work flow is depicted as a collection of states which the user can be in while using the Elucidator. The states can be divided into the two main tasks performed while using the Elucidative environment; reading and writing. While writing, you can be in a number of different states, but you can only be in one state while reading.

While engaged in the reading activity the software developer can only be in one state: *browsing*. By browsing is meant that the software developer is reading the documentation using the browser tool described earlier in this section. This activity complies with the *Examination* software development case described in Section 3.1 on page 17 in the Analysis chapter.

In contradiction to the reading activity, the writing activity contains a number of states, the two main states are *Writing documentation* and *Writing source code*. As the setup of the editor features a split-view with documentation and source code in separate windows, it is

easy for the writer to shift between these two states. Common for both states is that you can move to the *Abstracting* state from them, which is done when you invoke the abstraction command in the editor. While in the *Abstracting* state you can return to either of the two main states.

While *Writing documentation* the writer can move to two other states besides the *Abstraction* state: the *Creating links* and *Creating new documentation nodes* states. Both states can be viewed as sub-states to the *Writing documentation* state, but is described separately in order to provide a detailed picture of the work flow while creating documentation. While *Creating links* the writer creates links to either documentation or source code. Depending on how much new documentation and source code has been produced since the writer was in the *Creating links* state the last time, he will typically chose to make an *Abstraction* just before he starts *Creating links*. The reason for doing an abstraction before creating links is to make sure that the entities and relationships from the newly created documentation and source code is represented in the Data model, and thus include these in the possible matches made by the linking feature in the editor.

The *Creating new documentation nodes* state is also entered from the *Writing documentation* state. This is typically done for two different reason. The first is simply to create a new documentation node, and start writing documentation in it. The second, is to create a loose end. As described in Section 2.1.1 on page 11 of the Earlier work chapter, loose ends are made to be able to make references to a related subject which is not described yet. When *Creating new documentation nodes* the writer will therefore typically only fill in the topic, and then return to the *Writing documentation* state for the original node and return to fill out the loose end node at a later point.

Finally it should be noted that when the software developer is in one of the states of the writing activity, he is performing one of the three software development case, *Creation*, *Reuse* and *Change*, described in Section 3.1 on page 17 in the analysis chapter.

#### 4.1.6 Entities in the hypertext model

Since we need to reference specific details, that is entities, of the documentation nodes and the source code, in order to design the realization methods described above, we here give a short introduction to entities in the documentation and source code.

Common for all entities, both documentation and source code entities, are that they have an unique identity. This is expressed by there idname. This property of the entity is used, when a link to an entity has to be created.

In a documentation node all tags of the EDoc language is considered to be entities. The means that, e.g., a documentation nodes, such as a Requirement, a Rationale or a Task is considered a entity. Also, the different parts of the documentation nodes, as well as the links are considered to be entities. An important property of the entities in the documentation is that one entity can be contained in another entity. This means that en entity representing a specific part of a documentation node, is considered to be contained in the entity representing the documentation node.

When it comes to the source code, entities are considered to be source symbols like a class name, a method name or a variable name.

## 4.2 Designing the documentation nodes

Having described the environment in which the documentation is being produced, we now narrow our focus to the documentation alone, and especially to how the documentation nodes is designed.

We do this in three steps: First we describe and discuss how the documentation nodes fit into the MRS-model presented in the analysis. Next we introduces the notion of thematic catalogs and describe how these relate to the MRS-model and the documentation nodes. Finally, we present how the internal structure of the documentation nodes are designed.

### 4.2.1 Documentation nodes in the MRS-model

When describing how the documentation nodes fit into the MRS-model we first recall the illustration of the extended structure of the MRS-model, as presented in Figure 4.3 on the facing page. From this we identify three levels of abstraction.

At the highest level, above the dotted line, we have the pure presentation of the MRS-model with its three deliberative categories: Motivation, Rationale and Solution description.

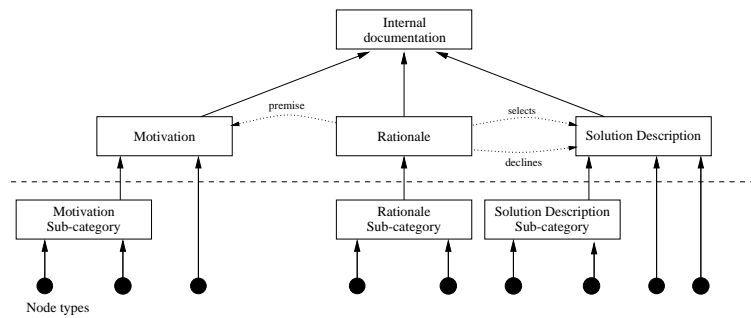
At the second level, just below the dotted line, we have a number of sub-categories. These sub-categories can be seen as optional details of the three categories in the MRS-model.

At the third and final level, concrete node types are placed. Examples of such are: Requirement, Change description and Task. These types can either be categorized directly within one of the three deliberative categories or in one of the sub-categories (and thereby indirectly within one the three deliberative categories of the MRS-model). That is, the Requirement node type mentioned above belongs to the Motivation category, the Change description to the Rationale category and the Task to the Solution description category.

The documentation nodes can be seen as instances of these concrete node types. This means that all documentation nodes has a type and can be categorized with respect to the three deliberative categories of the MRS-model, based on this type.

In Figure 4.3 on the next page a dotted line has been placed to separate the pure MRS-model from the sub-categories and node types. The dotted line has another purpose as well. It indicates two levels of importance. We considered the part above the dotted line to have conceptual importance for our work, while the part below the dotted line only has exemplifying importance. In other words, this means that we consider the pure MRS-model to be fixed, while the sub-categories and node types presented in this report is only considered as examples, and they could therefore be exchanged with other sub-categories and types as the user sees fit.





**Figure 4.3:** *The extended structure of the MRS-model. Above the dotted line the MRS-model with its three deliberative categories is presented. Below the dotted line, sub-categories to these deliberative categories as well as concrete node types in the categories is placed. Boxes represents categories (deliberative as well as sub-categories) and the black dots represents node types.*

In the rest of this section we will take a closer look at the three deliberative categories, in order to characterize the node types which could be placed in these categories. Furthermore a number of concrete suggestion for node types has been made. These are presented and described in detail in Appendix A on page 91. As mentioned these are only examples of node type, and they are selected/presented to demonstrate the principles and usability of the MRS-model.

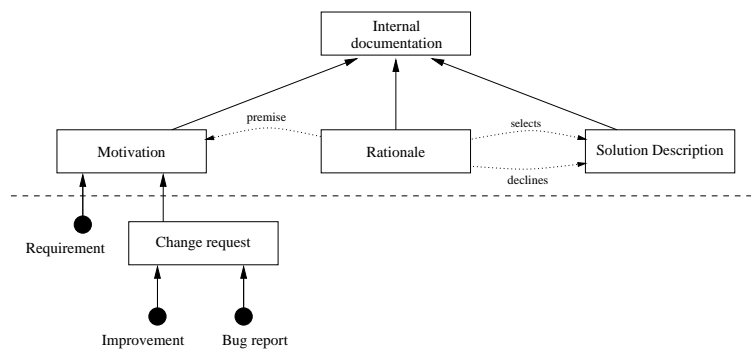
The Elucidator 2 implementation does not allow the user to alter the node types. In a fully implemented version of the Elucidator it should be possible for the development team using the Elucidative environment to change and extend the concrete node types to fit their specific documentation needs.

### Sub-categories and node types of the Motivation category

All documentation nodes with a node type from the motivation category contains motivations or incentives, that give occasion to a creation, reuse or change of a system. These motivations can describe facts from outside the system, that affect the development of the system, but also facts based internally in the system, that affect the development of other parts of the system.

In Figure 4.4 on the next page an example of a number of node types and sub-categories from the motivation category is presented. As it can be seen a *Change request* sub-category has been made to express that the two node types *Improvement* and *Bug report* both can be categorized as request for a change.

Another example of a node type in the motivation category is *Requirement*. Documentation nodes of this type will contain a description of a requirement that specifies or restrains the behavior of the design or implementation of the system. This can be seen as the information that will typically go into a system definition when using a Object-oriented Analysis and Design method [Mathiassen et al., 1997].

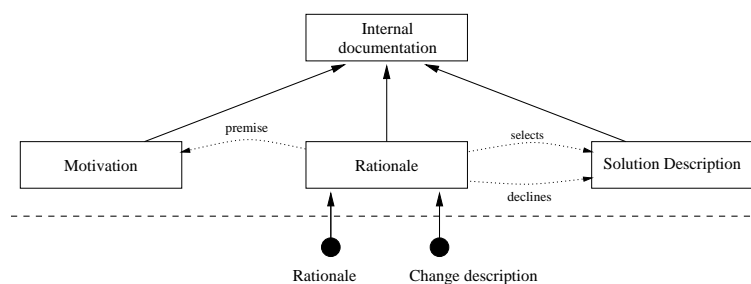


**Figure 4.4:** An example of a number of node types and sub-categories from the motivation category of the MRS-model. Boxes represents categories (deliberative as well as sub-categories) and the black dots represents node types.

Section A.1 on page 91 contains a detailed description of the three motivation node types mentioned above, and Figure A.1 on page 92 shows an concrete example of a *Requirement* type documentation node.

### Sub-categories and node types of the Rationale category

The purpose of a documentation nodes with a node type from the rationale category, is to document the arguments that selects and/or declines a given set of solutions. Documentation nodes of one of the rationale node types furthermore tie the motivations together with selected and/or declined solutions. This, as well as two examples of concrete node types from the rationale category, is shown in Figure 4.5. It should be noted that both a category in the MRS-model and a concrete node type is named *Rationale*, but that they are not considered to be identical.



**Figure 4.5:** An example of two node types from the rationale category. Boxes represents deliberative categories and the black dots represents node types.

A documentation node which has the node type *Rationale*, should contain a description of the forces that affect the arguments, i.e., “what drives us to a specific solution”. This will typically be realized through a short description of the motivations, that are relevant to the decisions made. The documentation node should furthermore also contain a description and

discussion of the selected and declined solutions. However, since the actual solutions would be described in a separate documentation node with a type from the solution description category, it should only contain a description of the properties that are important to the decision made.

Finally, a rationale will often contain a somewhat subjective discussion of the chosen and declined solutions with both subjective and objective assessments of the consequences of the choices made. Since this discussion is often subjective it would be an advantage to have a special place in the rationale for this discussion, in order to distinguish it from the rational decisions. An example of a concrete instance of the *Rationale* node type is presented in Figure A.2 on page 94 in the Appendix.

One of our hypotheses states that the history of a piece of software is important, and that it is possible to document the history as a natural part of the documentation. To fulfill this hypothesis we introduce the *Change description* node type.

The description of changes contain the same elements as a documentation node of the *Rationale* type, i.e., a description of the forces, the solution and a discussion of the choices made. The difference between a rationale node type and a change description node type is that the change description not only declines a set of alternative solutions but also deprecates a set of old solutions.

We believe that by giving the change description special attention in this way, we allow, the reader to identify the history of a piece of software, while at the same time make it more naturally for the writer to document the rationales for the specific solution.

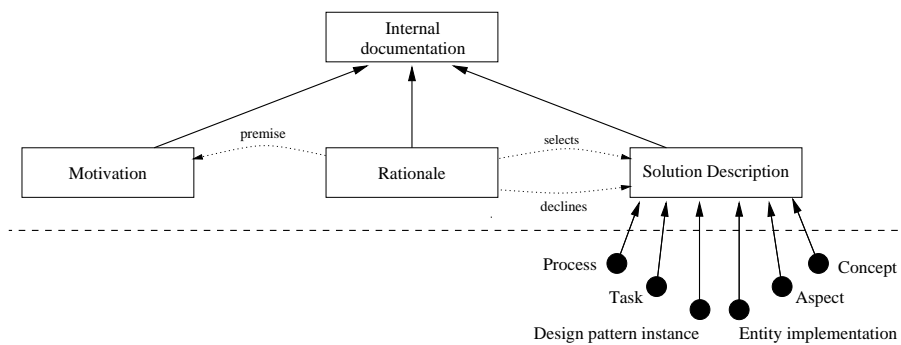
An example of a concrete instance of the *Change description* node type is presented in Figure A.3 on page 95 in the Appendix, and Section A.1 on page 92 contains a detailed description of the two node types presented in this section.

### **Sub-categories and node types of the Solution descriptions category**

Documentation nodes with a node type from the Solution description category are all documentation nodes that contribute to a factual descriptions of the solutions that realizes the developed system. The node types from the solution description category can be further categorized in various ways dependent of the system that is to be described and the development team using the documentation. In Figure 4.6 on the following page a number of examples of node types from the solution description category is shown. As it can be seen from the figure, no suggestions for sub-categories to the solution description category has been made, since such a categorization depends upon the development team using the Elucidative environment as well as the system being documented.

The examples has been chosen to demonstrate the contents of the documentation nodes in the solution description category. To further exemplify the contents of the node types from the solution description category, we here describe two of the six examples.

The first example is the *Design pattern instance* node type. The Design pattern instance node type is used to document the usage of design patterns. It is important to note that the



**Figure 4.6:** An example of six different node types, that all are a part of the solution description category. Boxes represents deliberative categories and the black dots represents node types.

documentation node documents an instance of a design pattern (which classes are involved in the instance of this pattern and so on), and not the design pattern itself, since this is already documented in the description of the design patterns, e.g., the “GOF-patterns” [Gamma et al., 1995]

We consider the design patterns instance node type to be important since it demonstrate the ability of the Elucidative environment to document a pattern in the system, that is not obvious visible in the source code.

An example of the usage of this node type could be to document that a Visitor Pattern is used in the implementation of the Abstractor components found in our Elucidator tool. In Figure A.5 on page 98 in the appendix, this is illustrated by a concrete instance of the *Design pattern instance* node type.

The second example of a node type from the solution description category is the *Concept* node type. This node type is used to separate and describe a subject or concept which has special meaning or importance for the rest of the documentation. A concept will typically be a technical term from the problem domain of the program being developed, but it could also be a concept which needs clarification because one or more developers in the development team is not familiar with it.

An example of the first type of concept could be a development team doing a accounting system. It would then be useful to have the concepts *debit* and *credit* described, since software engineers are not likely to know these in detail unless they have developed accounting software before.

Some might argue that a concept can not be categorized as a solution description. However, we believe that it is a part of the solution description category, since a concept can be seen as a factual description of something that constitute to the description of the system.

An example of a concrete instance of the *Concept* node type is presented in Figure A.4 on page 97 in the Appendix, and Section A.1 on page 92 contains a detailed description of all the six node types from the solution description category.

### 4.2.2 Thematic catalogs

As previously described, the MRS-model enforces a deliberative categorization of the documentation, into three categories: Motivations, Rationales and Solution descriptions. For each of these categories a number of node types is specified, and documentation nodes are considered to be instances of these node types. The division of the documentation into these node types and categories says something about how the documentation as a whole should be structured in terms of deliberative categories and node types, but it does not say anything about how it should be physically and thematically represented. In this section we take a look at this problem.

The first thing to consider is how each documentation node should be stored. Inspired by Java, and in order to secure flexibility when it comes to saving the documentation entity to persistent media, has led us to state that one documentation node equals one physical file in the file system. The name of the files are decided by the author, but they should be prefixed with `.edoc` for easy recognition of the data type.

Next we turn to the problem of which structure to use when saving these files in the file system. A solution would be to make three directories named Motivations, Rationales and Solutions and then save the files in these according to their node type. This is however not an optimal solution for a number of reasons. First of all, if the number of documentation nodes are high, the directories will be “crowded” and it will be difficult for the writer to find the documentation node he needs. Second, a division of the nodes into the three categories of the MRS-model, is a bit crude. Often systems consist of a number of parts or components, and it will typically be an advantage to separate the documentation according to these parts or components, in order to preserve the general view. Finally, different development teams may have different preferences for how they like their directory structure.

In order to solve these problems we introduce the notion of *thematic catalogs*. A thematic catalog is considered a collection of documentation nodes with some common theme or subject. The thematic categories are created by the writer and can be nested. They can be made according to some standard decided by the development team using the Elucidative environment or be just as the writer sees it fit. For example the thematic categorization could follow the logical structure of the system such as Abstractor, Generator and Data model. Finally, it could be a possibility to, e.g., use one or more of the categorizations presented in the literature, such as the one presented by COT [Sanvad et al., 2000] or Sametinger [Sametinger, 1994].

As we have chosen to implement the documentation nodes as physical files, the thematic categorization is implemented through the use of directories in the file system. By using this implementation the name of a directory becomes the name of the thematic category, and it is possible to create nested thematic catalogs.

Having introduced the notion of thematic catalogs, we now have two ways of structuring our documentation. Using these two methods together results in the structuring of the documentation nodes according to two axes. This is illustrated in Figure 4.7 on the following page. On one of the axes we have a physical and thematic structure of the documentation, while the other axis presents a deliberative categorization of the documentation nodes.

Deliberative categorization Thematic categorization	Motivations	Rationales	Solution descriptions
Theme A	□ □ □	□	□ □ □ □ □
Theme A.1	□	□ □	□ □ □
Theme A.2	□	□	□ □
Theme B	□	□	□ □
Theme B.1	□ □	□ □	□ □ □ □
Theme B.2	□	□ □	□ □ □
Theme C	□	□	□
Theme C.1	□ □ □	□ □ □	□ □ □ □ □ □ □ □
Theme C.2	□	□	□ □ □

**Figure 4.7:** *The overall structure of the documentation. The documentation is placed in documentation nodes, which is categorized according to the three deliberative categories of the MRS-model. Each documentation node (the squares) is represented as a separate file, which is saved into a number of, possible nested, thematic categories, represented by directories in the file system.*

### 4.2.3 The internal structure of the documentation nodes

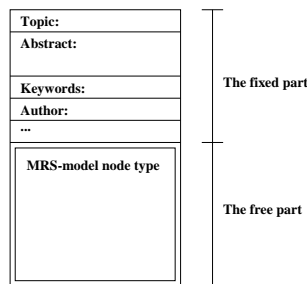
In the previous section we have discussed a categorization of documentation nodes in two dimensions: the deliberative categories and the thematic catalogs. These categorize the documentation at node level by the nature of the contents of the documentation nodes. In the following section we focus on how the structure internally in the documentation nodes is designed.

This internal structure has a twofold purpose. First, it helps the writer to structure his documentation. Second, it makes the documentation easier to read since it is structured uniformly, and with a focus on categorizing the internal structure.

As discussed in the analysis, a documentation node is divided into two main parts: a fixed part and a free part. These are illustrated in Figure 4.8 on the next page. The fixed part, can be seen as a header for the documentation node, and therefore contains information that is applicable to all documentation nodes, regardless of their node type. Since the information of the fixed part is applicable to all node types, a common structure of this part can be created. The free part, can be seen as the body of the documentation node, and it therefore contains the actual documentation in the node. The structure of this documentation is dependent upon the node type, and a common structure for all node types can therefore not be created. Thus, a separate structure for each of the node types will be created.

The internal structure of the documentation nodes is implemented by the EDoc language. This is done by introducing a number of new tags to the EDoc language, which makes the language able to express both the fixed and the free part of a documentation node. The grammar for EDoc language can be seen in Appendix B on page 101.

In order to make it easier for the writer to create a documentation node, an number of annotated templates, each expressing one of the node types, has furthermore been created. A template can be seen as an empty documentation node, which contains only the EDoc tags



**Figure 4.8:** An illustration of the two main parts of a documentation node. The figure is the same as Figure 3.3 on page 28.

needed to realize the node. Whenever the writer wants to make a new documentation node, he selects one of these templates from within the editor. This results in a new editor buffer, containing a copy of the template. It is then the job of the writer to fill out the template with the documentation to be placed in the node. The templates are annotated with guidelines that describes the purpose of the specific tags. These guidelines are implemented as comments above each tag in the templates. This is done in order to help the writer fill out the templates. The templates for the different node types can be seen in Appendix C on page 109.

In the rest of this section we will take a closer look at the contents and templates for the fixed and the free part respectively.

### Structuring the fixed part of the nodes

As mentioned above the fixed part of the documentation node can be seen as a header for the node. It contains a number of elements which is applicable to all documentation nodes, regardless of their node type. Below, a list of elements in the fixed part in the Elucidator 2 is presented. These elements should be seen as our suggestions for reasonable meta-information about the entire node, but other elements may be applicable as well.

**Topic** The topic should introduce the thematic contents of the node. The topic is inspired by the STOP method [Tracey et al., 1999], which argues that *“it is important to recognize that the topic title must characterize and introduce the thematic contents [of the documentation node], not merely categorize (label) the theme body”*. The STOP method furthermore states that *“topic titles are more likely to be representative and topically faithful if they are (1) constructed as sentence fragments and (2) rewritten after composition of the theme [the documentation node]”*.

**Abstract** The abstract is like the topic inspired by the STOP method. It should strive to give a short summary of the contents of the node. It should contain the most important words or phrases of the node, these either being the main arguments, important properties of a solution or special conditions for a motivation. In general the abstract should be about 3–5 lines of text. As with the topic the abstract is likely to be representative and topically faithful if it is written after the actual documentation node is written.

**Keywords** This is a list of important keywords that relates to the contents of the node. The writer should strive to use keywords that are spelled uniformly throughout the documentation, as the keywords are used for the creation of implicit links, as we will describe in Section 4.3.4 on page 59.

**Status** The status element of the fixed part, is a marking of the status for the node. Possible values are: *new*, *in progress* or *finished*. The status is used to make it easier for the writer to manage loose ends in the documentation, since they make it easier to find these, e.g., by looking for documentation node with the status set to new. Besides this, it also give the reader information about the reliability of the contents of the node.

**Author** The name of the author. This can both be the name of the original author or the name of the person who last edited it. In the Elucidator 2 tool the author is set to the last person who edited it.

**Created** The time of creation of the node.

**Updated** The time of the last update of the node.

As an example of the fixed part of a documentation node, consider Figure 4.9. On the left side of the figure (Figure 4.9(a)) the template for the creation of the fixed part is show (the annotations has been removed for the sake of simplicity). On the right side of the figure (Figure 4.9(b)) the result of a filled out fixed part for a documentation node with the node type *Change description*, is shown as it is presented in the browser.

```

<head>
<topic >
</topic >

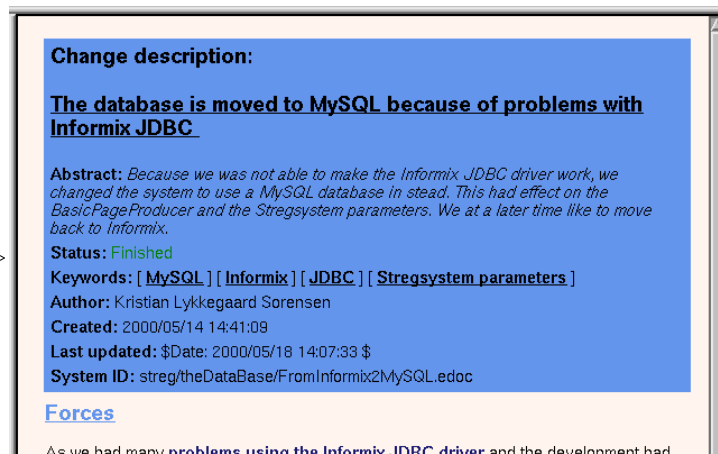
5 <abstract >
  </abstract >

  <status ><new/></status >
10 <keywords ><kw></kw></keywords >

  <author ></author >

  <created ></created >
15 <last -updated ></last -updated >
</head >

```



(a) The template for the fixed part.

(b) An example of how the fixed part is presented in the browser.

**Figure 4.9:** An example of both the template and the final result of the fixed part of a documentation node.



### Structuring the free part of the nodes

The second main part of a documentation node is the free part. This part contains the actual textual contents of the node. In contrast to the fixed part, no common structure exist for the free part. Instead the structure of this part depend on the node type of the documentation node. This means that a unique structure and template has been created for each of the node types. The structure of the node types can be seen in the grammar for the EDoc language as presented in Appendix B, while Appendix C contains the templates of all the node types.

An example of the structure and the corresponding annotated template used to realize the Rationale node type is furthermore presented in Figure 4.10.

```

<rationale >
  <!-- The forces section should contain a short description of the -->
  <!-- driving forces and motivation for this rationale . The description -->
  <!-- should contain a number of dlinks , role :premise to motivation -->
5 <!-- nodes , describing the motivations in detail , and/or a number -->
  <!-- slinks , to special parts of the system that motivates this -->
  <!-- rationale . Finally the section should contain argumentation for -->
  <!-- the selected and declined solutions .-->
  <forces >
10 </forces >

  <!-- The solution part presents the selected solution . If alternative -->
  <!-- and/or declined solutions exists they are mentioned to . This part -->
15 <!-- will typically contain dlinks , role :selects and role : decline to -->
  <!-- detailed documentation of the selected and/or declined -->
  <!-- solutions .-->
  <solution >
20 </solution >

  <!-- A discussion of consequences of the select /declined solutions -->
  <!-- including personal subjective assessments .-->
  <discussion >
25 </discussion >
</rationale >

```

**Figure 4.10:** *The annotated template for the Rationale node type.*

It should be noted that our work should not be considered a study on how to make an ideal internal structure of the documentation nodes. The structure of the templates for the node types should therefore only be considered examples of structure. In fact we believe that no perfect structure for these node types can be made. The structure should instead be refined and changed in continued iterations, by the development team using the Elucidative environment, in order to suite their needs.

## 4.3 Designing the links

Having described how the documentation nodes are designed, we now focus our attention on the relationships between these nodes. In Section 3.5.3 on page 29 of the Analysis chapter,

it was described that we need links in order to express the relationships between documentation nodes in the MRS-model. In this section we describe how links are designed and implemented in the Elucidator 2 tool.

The links used in the Elucidator 2 can be described according to four characteristics: their *type*, their *role*, the *type of structure they impose on the documentation* and the *creation method* to create the link. In the following sections we will discuss each of these characteristics in detail. We realize that, as a first impression, the characteristics of the links may seem overwhelming and complicate. We do however, not believe this to be the case, and it will be explained why in the final section which contains a discussion of how these characteristics apply to the links.

### 4.3.1 Link types

As the first characteristic of the links used in the Elucidator 2 tool, every link has a type. Four types of links exist, corresponding to the four types of data which the link can have as its destination.

In the following a brief description of the four link types will be presented.

**Links to documentation:** Links to documentation has both the source and destination anchor in a documentation node or some part of a documentation node, and thereby links two documentation nodes, or parts of, together.

**Links to source code:** A link to source code has its source anchor some place in a documentation node or the source code, while the destination anchor is always a source code entity, such as a class, a method or a field. This means links to source code, relates a documentation node or source code entity to a source code node.

**Links to external entities:** A link to an external entity has, as the links to documentation, its source anchor in a documentation node. The destination of the link is some entity which is considered external to the internal documentation. An example could be an object-oriented analysis and design document. The destination of a link to an external entity is always expressed as an URL.

**Links to views:** Links to views has their source anchor in either a documentation or a source code node. The destination of the link is a view. Views will be dealt with in detail in Section 4.4 on page 62.

As it can be seen from the descriptions above the type of the link can be derived, based on the type of data the link points at. We have however chosen to express each link type explicitly by four different names. This results in the following three link tags being used in the EDoc language: `<dlink>`, `<slink>` and `<xlink>`. Besides these three types we also have the `<vlink>` type. This link type is not present in the EDoc language, and can therefore not be inserted by the writer. Instead it is only used by the system to create implicit links to views. The notion of implicit links will be dealt with later in this section. We have chosen to write

the name of this link as a tags even though it is not present in the EDoc language in order to have an uniform appearance for the four link types. We realize that naming the links types explicitly means that we have redundancy in the system. However, the choice was made to provide the writer with a practical way of distinguishing the different types of links. If we had chosen to not give each link type an explicit name, the writer would instead have to distinguish the type of the links by looking at the destination of the link, and then derive the type from this information. Thus, we believe that by giving each link type a explicit name it is easier for writer to use them when producing documentation.

### 4.3.2 Roles on links

Having discussed the four types of links we now move on to describe the roles on these links. A role on a link is an attribute on the link, which is used to express the nature of a relationship between two nodes (this can be both documentation and source code nodes). Furthermore, a link can only have one role.

The links in the Elucidator 2 tools are considered to be directed. However, when two documentation nodes are to be linked together, it is up to the writer to decide which of the two nodes should contain the source anchor of the link, and which should contain the destination anchor. This scenario demands for the names of the roles to be symmetrical [Conklin, 1987]. This is best illustrated by an example: A writer wants to express that some part of a documentation node (A) is described by another node (B). He can then place the anchors of the link in two ways: He can create a link *from B to A* with the role *describes* or he can make a link *from A to B* with the role *described-by*. Both solutions are allowed, and the result is that the name of the role is symmetric, namely *describes/described-by*. For the sake of simplicity the roles are normally mentioned with only one of the two symmetric names.

#### Link roles for maintaining the MRS-model

A subset of the link roles used in the Elucidator 2 tool has a special meaning since these roles are used to *maintain the MRS-model*. These roles, which are considered to be *key roles*, are: *Premise*, *Selects*, *Declines*, *Introduces* and *Deprecates*. By maintaining the MRS-model, is meant that links with these roles ties the three categories of the MRS-model together.

To understand how these five key roles are used to maintain the MRS-model, we first describe them.

**Premise/Premise-for:** The *Premise* role is placed on links between two documentation nodes of the Motivation and Rationale categories, and is used to specify that some documentation node serves as a premise to another documentation node.

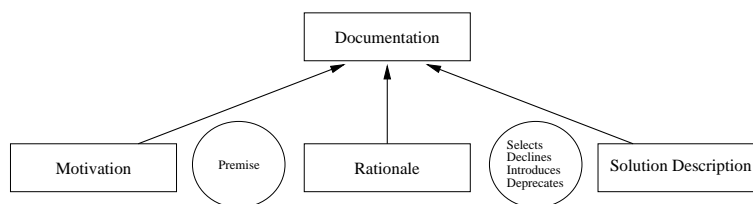
**Selects/Selected-by:** The *Selects* role is placed on links between two documentation nodes of the Rationale and Solution description categories, and is used to express that, based on some argument, the rationale selects a certain solution.

**Declines/Declined-by:** As the name implies, the *Declines* role, does the opposite of the *Selects* role, namely, expressed that a rationale declines a solution.

**Introduces/Introduced-by:** This role is used when the writer wants to express the history of the documentation/source code. The *Introduces* role is used to express that some documentation node is introduced somewhere in the history of the program, that is, it was not present in the initial version of the program. The introduces role will be placed on links going from a change description node to a solution description node.

**Deprecated/Deprecated-by:** This is the final of the five key roles used to maintain the MRS-model. As with the *Select/Decline* roles the *Introduces/Deprecates* roles are also opposites. The *Deprecates* role is used to express that, due to some change, the destination node of the link is now deprecated, and therefore not used anymore. The destination is not simply deleted, since this would make it impossible to track the history of the system.

The description of the five roles revealed that they can be further categorized, since the *Premise* role is used to express relationships between Motivations and Rationales, while the remaining four key roles are all used to express relationships between Rationales and Solution descriptions. This is illustrated in Figure 4.11.



**Figure 4.11:** *The usage of the five key roles in the Elucidator 2. The Premise role is used to express relationships between Motivations and Rationales, while the four other roles are used to express different relationships between the Rationales and the Solution descriptions.*

### Other link roles

Besides the roles used to maintain the MRS-model, a number of other roles can be used. Common for these roles are that they are not essential to the realization of the MRS-model, but they can be useful to express certain relationships between the nodes. The number of these roles are not finite. The roles presented in this report can therefore be considered examples of roles which the authors of this report found useful.

In this section we would like to present two of these roles which we find to be of a so general nature that they will be useful in any Elucidative environment. The other examples of roles of this category is presented in Appendix A.2 on page 99.

**Describes/Described-by:** The *Describes* role can be used in almost any case. The intended usage of the role is to be able to specify that something is either a description of

something else, or that something is described elsewhere. Furthermore it can be noted that this role corresponds to the strong link found in the Elucidator 1 tool [Christensen et al., 2000], and it is often used on links to the source code.

**Mentions/Mentioned-by:** The *Mentions* role is a somewhat weaker version of the *Describes* role. The *Mentions* role should be used to point the reader in a direction which may shed some additional light to a subject, but which is not especially necessary in order to understand the documentation being read. The *Describes* role on the other hand, should be used to link to documentation which is important for the understanding of the documentation being read. The *mentions* role is similar to the *wear* link found in the Elucidator 1 tool [Christensen et al., 2000].

**Containment/Contained-in:** Finally the *Containment* role is used to express that some entity, that be both documentation and source code entities, is contained within another entity. This roles is normally not used by the writer, but instead by the system to, e.g., express that some documentation node is contained in a specific thematic catalog.

### 4.3.3 Structure imposed on the documentation by links

When creating links in the documentation, these links imposes a structure on the documentation. This structure can be either hierarchical or non-hierarchical, corresponding to the *organizational* and *referential* categories presented in Section 3.5.3 on page 29 of the analysis.

In the Elucidator 2, organizational links is specifically used in the hierarchical index view, to express the containment relationship between two thematic catalogs or between a thematic catalog and a documentation node. The hierarchical index view will be described in detail in Section 4.4.4 on page 68. The referential links on the other hand, will typically be used throughout the whole documentation.

### 4.3.4 Creation method

The final characterization of links is the method used to create the links. Two methods exists: The links can be either *explicitly* or *implicitly* created. While Section 3.5.3 on page 29 of the analysis, contains a detailed description and discussion on explicitly and implicitly created links, this section take a look at how these are realized in the Elucidator 2.

#### Explicitly creation of links

To create a link explicitly means that the writer creates it manually. Specifically, explicitly created links are links that are inserted directly into the documentation nodes, with the help of the editor. To do this the writer uses the three links tags provided by the EDoc language: `<dlink>`, `<slink>` and `<xlink>` to link to documentation, source code and external entities respectively. The start and end tags are placed around a region of text in the documentation

node, and the destination of the explicitly created link is specified as an attribute on the link. In Figure 4.12 an example of the usage of the explicitly created links is presented.

```
In this example we first make a link to a class named
<mlink href="elucidator.datamodel.DBBundleController">DBBundleController
</mlink>. Next we make a link to a documentation node which argues
<dlink href="elucidator-2.0/datamodel/need_for_views.edoc">why we need
views</dlink> in the Elucidator 2 tool. Finally we make a link to the
<xlink href="http://dopu.cs.auc.dk">home page of the DOPU project</xlink>.
```

**Figure 4.12:** *Example of the usage of explicitly created links.*

Since the names of the destinations of the explicitly created links, tend to be rather long, it can be a tedious job to create these explicit links. In order to reduce this problem, the editor provides support to help the writer insert links. This support is implemented in the Elucidator 1 and will be present in Elucidator 2. The functionality is furthermore described in greater detail in Section 4.1.3 on page 41.

### Implicitly creation of links

Links in the Elucidator which is not explicitly expressed by the writer in the EDoc files is implicitly created by the Elucidator tool. They are created by the Elucidator tool in order to realize the different navigation facilities in the browser.

Examples of implicit links could be links to documentation nodes from views or links to a view from locations in the documentation nodes, e.g., from the topic of a node to a Context View for the node.

Another, and more interesting, usage of implicit links is to use keywords and terms to present related information to the reader. The writer can specify a list of keywords in the fixed part of the documentation node and can furthermore mark-up words with the <term> tag in the free part of the documentation node to indicate that these words has special meaning.

The Elucidator 2 tool places implicit links from these keywords and terms to a Subject Index View (the Subject Index Views will be dealt with in detail in Section 4.4.4 on page 68) which list documentation nodes that presumably contains related information about the actual keyword or term. The related documentation nodes is found by matching the textual contents of a keyword or term to the topic and abstract of documentation nodes. This makes it, e.g., possible to mark-up the word “Abstraction” as a term and thereby implicitly link to a Concept node explaining the Abstraction concept.

### 4.3.5 Discussion

Having described the four characteristics of the links we now turn our attention to how these apply to the links. The four characteristics along with their possible values is summarized in Table 4.1 on the facing page.

Characteristic	Possible values
Type	<dlink>, <slink>, <xlink> and <vlink>
Role	Premise, Selects, Declines, Introduces, Deprecates, Describes, Mentions . . .
Imposed structure	Organizational and referential
Creation method	Explicitly and implicitly

**Table 4.1:** Summary of the four characteristics of the links in the *Elucidator 2* tool.

The first thing to notice is that all links must have exactly one value from each of the four characteristics attached to it. This means that every link has a type and a role, it implies a structure on the documentation, and it is either created explicitly or implicitly.

We illustrate this with a couple of examples:

First consider a situation where the writer wants to express that some documentation node (A) from the Motivation category acts as the premise of some other documentation node (B) from the Rationale category. The Writer will then make an explicit link from A to B, with the type set to <dlink>, the role will be *Premise* and the link will impose a referential structure on the documentation.

Next consider another situation where the writer has just created a new documentation node (A), which has been placed in a catalog (B). An implicit link will then be created between A and B, the type will be <dlink>, the role will be *Containment* and the link will impose an organizational structure on the documentation.

It should be noted that not all combinations of the values of the four characteristics makes sense. For example, the writer is not allowed to make an explicit link to a view, that is making an explicit link with the type set to <vlink> (although this feature might be possible in a future version of the tool). Another example could be that an implicit link with the type set to <xlink> cannot be made. Recall that a <xlink> is a link to some entity which is considered external to the internal documentation, it then is not possible for the Elucidator to be able to create links to something outside and unknown to it.

One might believe that it is very complicated and confusing for the writer to create a link, since he will have to choose values from the four different characteristics. We do not believe this to be the case. First of all, one should notice that the choice of whether to create the link explicitly or implicitly is not one made by the writer. Secondly, the writer will neither have to choose between making an organizational or referential link, since this is implicitly given depending on the link he creates. It should furthermore be noted that explicitly created links in almost any situation will be referential. Finally, we do not believe that choosing the type of the link is complicated or confusing for the writer, since this is just a matter of considering the type of the data he is linking to.

This leaves only one real choice: the role of the link. Some help can be found while making this choice as well. First of all the writer knows that only the Premise roles can be used between Motivations and Rationales. Similar, he knows that he only has to choose between

four different roles when making links between Rationales and Solution descriptions. Finally, the rest of the proposed link roles are just examples or suggestions so the writer is free to neglect these if he find using to many roles confusing.

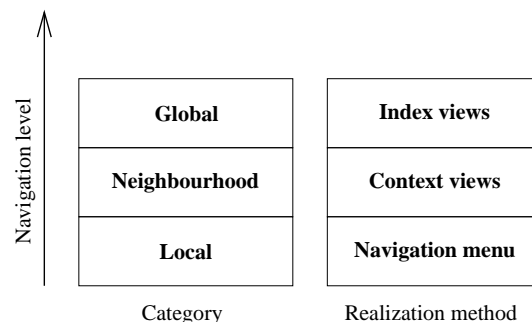
## 4.4 Navigation

As mentioned in Section 3.5.5 on page 32 of the analysis, one of the problems by using a hypertext system is that you tend to get disoriented while using it, and often end up missing the context of the documentation you are presently reading. As described, our main medicine against this problem is to provide navigation facilities to the reader. In this section we describe how the navigation facilities are designed and implemented in the Elucidator 2.

In a hypertext system, such as the Elucidator, the documentation may be read at different levels. One reader may be reading and jumping between specific documentation nodes at a detailed level, while another reader may try to survey the whole network at a higher level in search of relevant information. The problems concerning disorientation and lack of context are present at all these levels, but a divergence of needs between readers at different levels exists.

Readers at one of the lower levels will typically need support for recognizing element in the documentation nodes, such as links or special parts of the node. Such a reader will also need to be able to find his navigation possibilities. Readers at a higher level also have the need for understanding the contents of a documentation node and to identify his navigation possibilities. Besides this he also tries to comprehend the whole network and therefore needs assistance to navigate and search through all the documentation and source code.

In order to deal with these problems, we have divided the readers navigation needs into three levels: *Local*, *Neighborhood* and *Global*. For each of these levels we present a suggestion for a method of realization. The three levels and the suggested realization methods is illustrated in Figure 4.13.



**Figure 4.13:** An illustration of the three navigation levels, and their suggested realization methods.

Local navigation, is navigation internally in a documentation node, or between two documentation nodes. The level of navigation will be realized through the navigation menu.



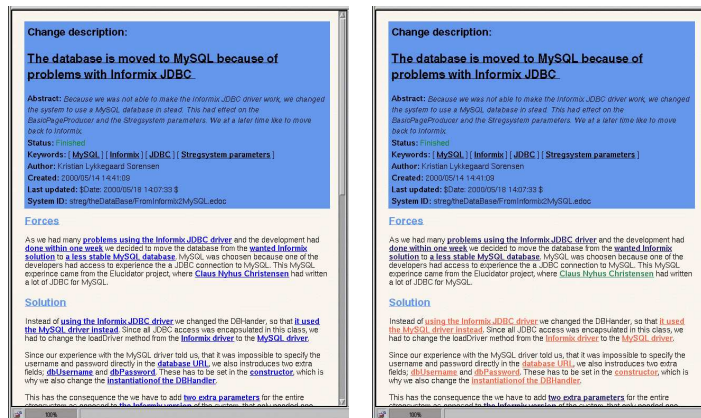
Neighborhood navigation is navigation between a documentation node and a number of documentation nodes, which are all related directly to the documentation node in question. This level of navigation will be realized though the Context view. Finally, Global navigation, is navigation in the whole network of documentation nodes, and which will be realized though the Index views.

Common for all these realization methods are that they use colors. The next section will therefore describe how these are used in the design of the Elucidator 2. Following this, the rest of the section will describe these three levels and their suggestion for a realization method in detail.

### 4.4.1 Coloring

Colors are used throughout the presentation of the internal documentation in the browser for two main purposes: To let the reader easily distinguish the different types of links, and to heighten the awareness of the MRS-model.

First we consider the purposes of using colors. Figure 4.14 show two browser screen shots containing internal documentation. On the first screen (Figure 4.14(a)), the same color is used on all the links. On the second screen (Figure 4.14(b)), the links are colored according to their type. The types used, in the figure and throughout the Elucidator 2 are: `<slink>`: red, `<dlink>`: blue, `<xlink>`: green and `<vlink>`: black. As it can be seen, the usage of the different colors on the links, makes it easy to distinguish between, e.g., links to documentation and links to source code.



(a) Node with the same color on all links

(b) Node with different color on links

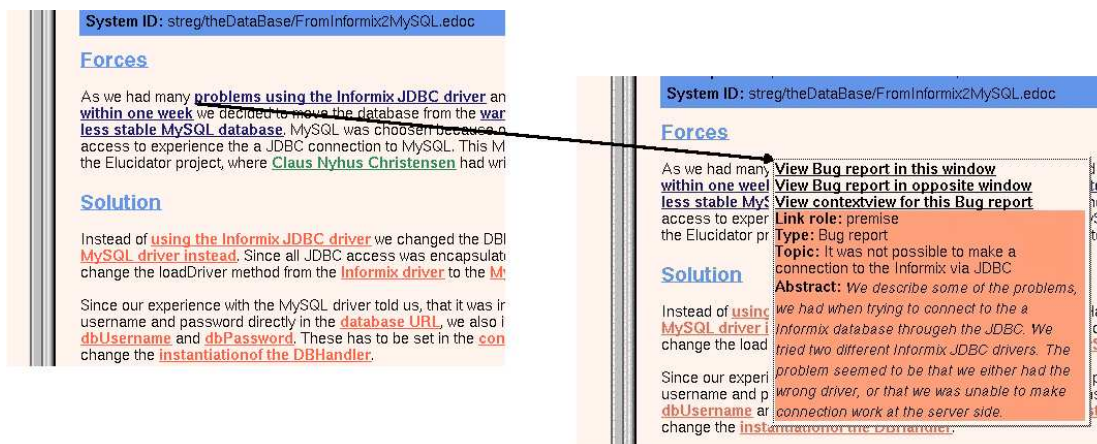
**Figure 4.14:** Illustration of the usage of colors as visual cues. (a) A node which is rendered with only one color for links. (b) The same node, but with links rendered with different colors dependent on their type.

The second purpose for using colors was to heighten the awareness of the MRS-model. This is realized by using a different color for each of the three categories in the MRS-model: motivations uses *orange*, rationale uses *light blue* and solution descriptions uses *green*. These *category colors* are applied in a number of ways. The most notable being the background of the header (the fixed part) of the documentation node, but the colors are also used, e.g., in the navigation menu, as we shall see in the next section.

## 4.4.2 Local navigation

The first of the three levels of navigation is Local navigation. Our suggestion for a realization of navigation at this level is the *navigation menu*. The navigation menu is a context dependent menu which is designed to help the reader decide if he wants to follow a link or not. The menu is shown when activating a link, for which it is feasible to view the contents of the destination of the link. This destination can both be documentation and source code, which means that the navigation menu is applied both on `<dlink>` and `<slink>`.

The navigation menu has two parts. The first part contains a number of possible actions to perform, while the second part contains information of the destination of the activated link. An example of the navigation menu applied to a `<dlink>` is shown in Figure 4.15.



**Figure 4.15:** Activation of the Navigation menu. When the reader activates a link, the navigation menu pops up directly upon the link. It contains a list of possible actions to perform, and a short summary of the contents of the destination node of the activated link.

### The first part of the navigation menu

In earlier versions of the elucidative environment, the frame setup in the browser dictated the documentation to be shown in the left frame and the source code in the right frame. As described in the Analysis chapter, we wish to remove this restriction and thereby let it be up to the reader to decide where he wishes to view the documentation and source code.

This is realized through the first part of the navigation menu, since this part lets the reader choose in which frame he wants to have the destination of the activated link shown. This means that the first part of the navigation menu contains a list of implicit links for the reader to choose from. The selection of links depends on the type of the activated link. For links to documentation three possibilities exist: view destination in the current window, view destination in the opposite window or show the context view for the entity. This is shown as the upper part of the navigation menu presented in Figure 4.15 on the preceding page, which shows a navigation menu for a link whose destination is a Bug report documentation node.

Similar actions are presented for source code entities, but it has the extra feature of allowing direct access to the interface documentation for the specific entity. For Java this means linking to JavaDoc generated pages. This provides a direct coupling to the interface documentation and therefore provides a utilization of the internal documentation together with JavaDoc.

### **The second part of the navigation menu**

When choosing whether to follow a link or not, information about the destination of the link may prove valuable. In this way, the reader has some information about the contents of the destination, and may therefore be able to make a more qualified assessment of whether to follow the link or not. This is the purpose of the second part of the navigation menu.

For links to documentation, the navigation menu shows the node type, the topic and the abstract of the documentation node in which the entity linked to is contained. It furthermore shows the role of the link. For links to source code the role of the link, as well as the type and name of the entity linked to, is shown. In order to improve the awareness of the MRS-model, the background of the navigation menus placed on links to documentation, is furthermore set with respect to the category color of the destination node. This can be seen in Figure 4.15 on the facing page, where the color of the navigation menu is orange since it describes a motivation.

### **Implementation**

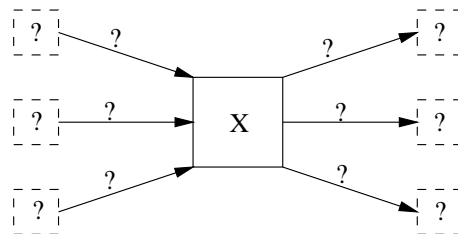
As a technical note, the navigation menu is realized by using dynamic html (DHTML). The menu is displayed by a JavaScript function that is invoked when clicking on a link. The function dynamically retrieves the contents for the menu from the Generator, based on the destination anchor, and displays the result in a layer on top of the main html-document.

This implementation of the navigation menu has the noteworthy consequence that the choice of browser is limited to only support Netscape Communicator. The navigation menu could be implemented to also support Internet Explorer, but since the implementation of Elucidator 2 is done as proof of concept, this was not prioritized.

### 4.4.3 Neighborhood navigation

The second of the three levels of navigation is neighborhood navigation. In order to realize this level of navigation we introduce the *Context view*.

The purpose of the context view is to provide the reader with information on the context concerning a specific documentation or source code entity, such as a documentation node or a class in the source code. Therefore a context view applies to a documentation or source code entity in the hypertext network. A context view displays a selection of information about the links from and to the entity applied on. These links are also called the outgoing and incoming links. It can therefore, e.g., be used to find which motivations nodes acts as premise for a specific rationale node. A conceptual drawing of the context view is shown in figure 4.16.



**Figure 4.16:** *Conceptual drawing of the context view. The context view is applied on a specific entity (X), which is used as a base to find and display all the relationships held between this entity and other entities in the network.*

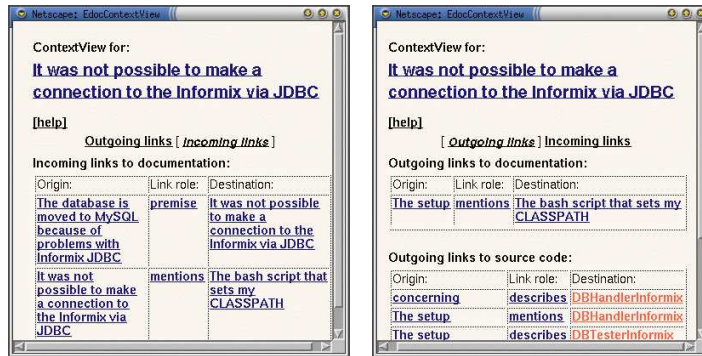
Two different kind of Context Views exists — one for documentation and one for source code.

#### Context view for documentation

The context view for documentation is implemented in a separate window. This window contains two panels, one for outgoing links, and one for incoming links. The reader is then able to switch back and forth between these two panels. A screen shot showing a context view for documentation with its two panels is shown in Figure 4.17 on the facing page.

Outgoing links contain links to both source code and other documentation entities (Figure 4.17(b) on the next page), while incoming links shows which documentation entities link to the entity<sup>1</sup> (Figure 4.17(a) on the facing page). Both the outgoing and incoming links is presented in a table, with one outgoing or incoming links in each row. For each link the following information is presented: the role of the link, an implicit link to the location where the link is placed, and an implicit link to the entity in which the link is contained (typically a documentation node or part of a documentation node).

<sup>1</sup>Source code does not contain links to documentation — hence no source code links is shown in incoming links for documentation

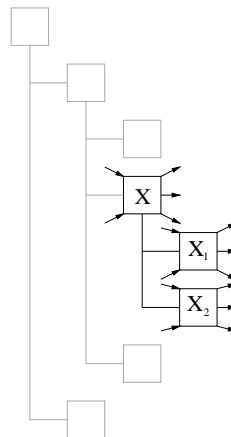


(a) The incoming links panel

(b) The outgoing links panel

**Figure 4.17:** A context view for a documentation node. (a) shows a list of entities which link to this node. (b) show entities which this node links to.

A context view for documentation can be shown for all types of documentation entities. This means everything from catalogs, documentation nodes to entities inside these nodes. When a context view is shown it not only presents the outgoing and incoming links for the entity it is applied on. It also presents the outgoing and incoming links for all the entities inside the entity the context view is applied on. Thus when showing outgoing links in the context view for a catalog the list of links is found by recursively searching for outgoing links in the documentation nodes inside the catalog. Figure 4.18 illustrates the process.

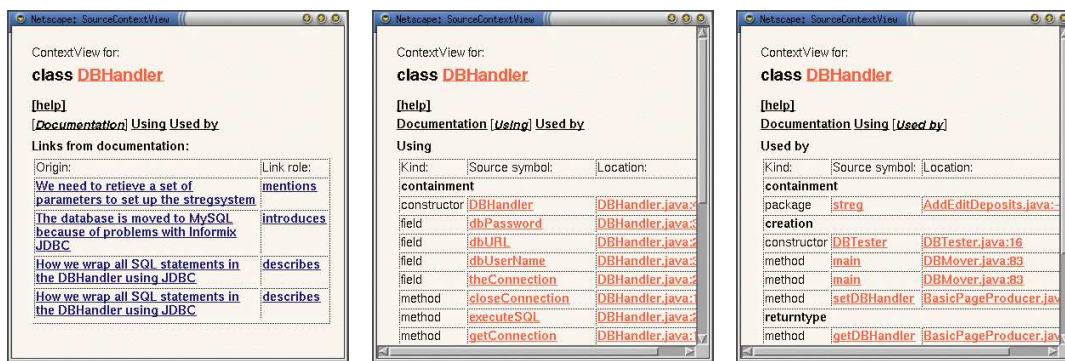


**Figure 4.18:** A context view applied on an entity is considered recursively in nature. The context view is applied to a specific entity ( $X$ ) which it then used as a basis to find outgoing and incoming links to and from  $X$  but also to and from  $X$ 's children ( $X_1, X_2$ )

## Context view for source code

The context view for source code is like the context view for documentation implemented in a separate window, with a number of panels. Instead of two panels, the context view for source code has three panels, and the contents of these are different than those of the context view for documentation.

Figure 4.19 shows these three panels. The first panel shows which documentation entities links to the selected source entity, thus presenting a list of where the selected source entity is documented (Figure 4.19(a)). The two other panels shows relationships in the source code. One panel shows which source code entities the selected entity uses, e.g., which fields is defined in a class (Figure 4.19(c)), while the other shows which other source code entities uses the selected entity, e.g., which methods instantiates a class (Figure 4.19(b)).



(a) The documentation panel

(b) The using panel

(c) The used by panel

**Figure 4.19:** The context view for the source code entity `DBHandler`. (a) shows documentation entities which link to the `DBHandler`. (b) shows the source entities using the `DBHandler`. (c) shows the source entities used by the `DBHandler`.

## 4.4.4 Global navigation

The final level of navigation is global navigation. This level of navigation is realized using *Index views*. All of these views are conceptually like normal table of contents and indexes in text files.

The purpose of the index views is to provide the reader with a survey of both the documentation and the source code. Therefore the index view applies to the whole hypertext network. As the name implies, the presentation of the view has the form of an index, which means that it is presented as a list of implicit links to thematic catalogs and nodes. A number of different index views could be imagined. We have chosen to design and implement three: the *hierarchic index view*, the *entity index view* and the *subject index view*. The hierarchic index

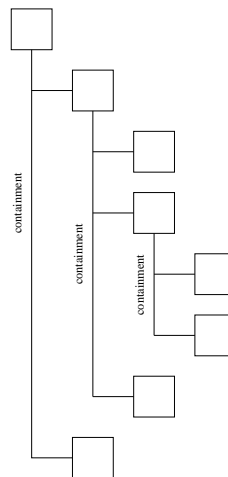
view can be compared to a table of contents and the subject index view is comparable to a normal subject index. The three views will be described in detail in the rest of this section.

### Hierarchic Index Views

Both the documentation and source code is physically organized in a hierarchic manner. The documentation nodes is stored in, possible nested, thematic catalogs, while the source code is organized in Java packages. The purpose of the *hierarchic index view* is to present this organization to the reader, while at the same time allowing him to navigate it. Since the documentation and source code is organized in separate hierarchies, separate index views are applied to them.

The two hierarchical index views is constructed by following the implicit containment links. This is illustrated in Figure 4.20. For the documentation this means that the hierarchical index view starts by displaying a list of implicit links to all the thematic catalogs in the project. By activating one of these links, the reader can traverse downward into the catalogs and show their contents, which will be documentation nodes and possible nested thematic catalogs. When viewing the contents of a thematic catalog the topic, abstract and node type is shown for the documentation nodes type, as well as a implicit link to the documentation node in question. Finally, the navigation menu presented in Section 4.4.2 on page 64, is available on all the implicit links in the view.

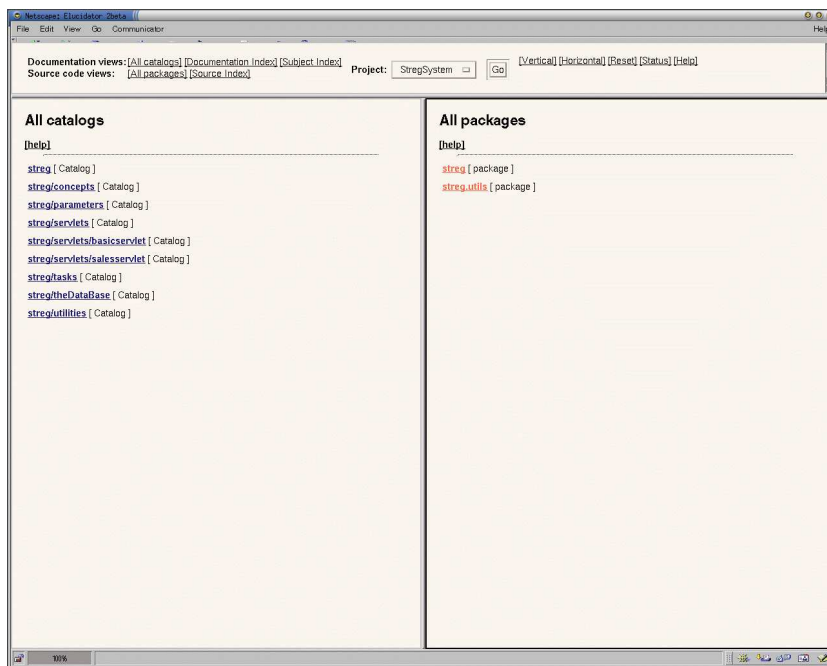
The hierarchic view for the source code is similar to the corresponding view for documentation. Instead of thematic catalogs and documentation nodes, it show all packages and classes which can also be traversed to browse their contents.



**Figure 4.20:** A hierarchic index view is used to browse the hierarchic structure applied to both the documentation and source code, by the thematic catalogs and packages respectively.

The two hierarchic views is used as the default view when the reader starts to browse a project. This setup is shown by a screen shot in Figure 4.21 on the next page.





**Figure 4.21:** *The hierarchical index views of documentation (the left frame) and source code (the right frame). Both views has implicit links which can be used to traverse further down into catalogs or packages.*

## Entity index views

Although the hierarchical index view presents a survey of the documentation and source code to the reader, it does not use any information from the MRS-model, or the structure of the source code, to do that. The *entity index view* is, however, based on this information.

As with the hierarchical index views, two separate entity index views are provided for the documentation and source code respectively. The entity index view for documentation presents an, alphabetically ordered, index of documentation nodes while the view for source code, presents a index of source code entities.

These two indexes can be filtered with respect to either the MRS-model or the structure of the source code. For the documentation view it is possible to filter on three categories of the MRS-model, the node types and the status of the node (new, in progress or finished). This filtering mechanism can be used to, e.g., show all documentation nodes in the rationale category which has the status “Finished”. The filter on status is particular useful in regard to loose ends, since by applying a filter to find documentation nodes with the status set to “new”, the result will be an index of loose ends.

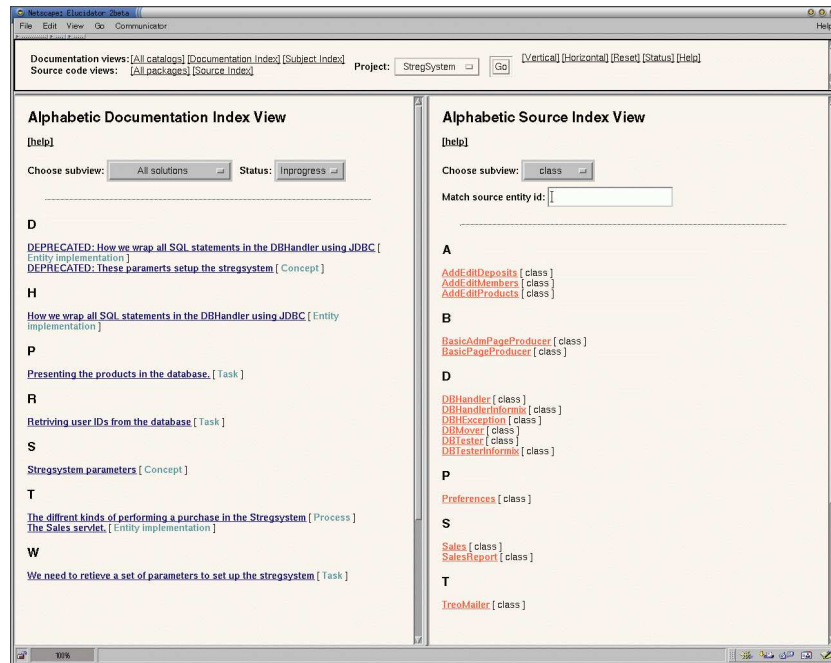
The source code index view can be filtered to show all source entities of a specific type, e.g., all classes or methods. As the number of source code entities in a project is typically high<sup>2</sup> the index of source code entities tend to get rather large and confusing. Therefore a search

<sup>2</sup>In the Elucidator 2 source code there are approximately 4000 entities.



facility has been implemented in this view. The entered search string is matched against the idname of the entities, and can therefore, e.g., be used to show entities inside a specific package, class, etc.

An example of the usage of the two entity index views, is presented in the screen capture in Figure 4.22. The documentation view (on the left) presents documentation nodes in the solution description category of the MRS-model, with the status “in progress”, while the source code view (on the right), presents a index of all the classes in the project.



**Figure 4.22:** A screen capture illustrating the two entity index views.

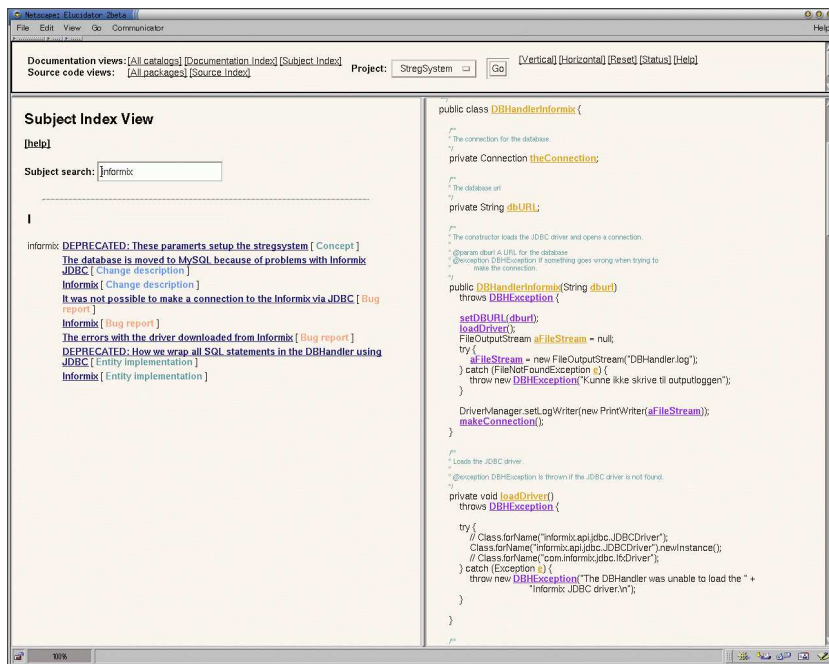
### Subject index view

The final index view concerns information gathered internally in the documentation nodes. This is called the *subject index view*, and as the name implies this view presents a index of subjects found in the documentation.

Concretely the subject index is a index of the keywords and terms placed in the documentation, for which a match could be found. Recall that keywords are special words listed in the documentation nodes fixed part, and terms are special words marked up inside the nodes free part. How a match for these two types of special words is found is described in Section 4.3.4 on page 59 earlier in this chapter.

In order to enable the reader to search for special words not explicitly stated by the writer as terms or keywords, the subject index view furthermore provides a searching mechanisms where the reader can search the topic, abstract and keyword list of the documentation nodes

for specific subject. This is illustrated in Figure 4.23 where the subject index view is used to search for the subject “informix”.



**Figure 4.23:** A screen capture illustrating the subject index.

# 5

## Reflection

In order to evaluate the MRS-model and the Elucidator 2 tool, a small informal experiment is conducted. In this chapter we describe this experiment and present the experiences gained by it.

First we present the circumstances under which the experiment was conducted. Next we present reflections on the experience gained through the experiment. These reflections are divided into two sections with respect to the experience gained by the writer and the reader. Finally we discuss the experiences gained by the experiment together with our own assessment of the MRS-model and the implemented Elucidative environment.

### 5.1 Experiment circumstances

The strategy for conducting the experiment was to document, by using the Elucidator 2 tool, the development of a project called StregSystem. The rest of this section describes what StregSystem is and the circumstances under which the experiment was conducted.

#### **Project StregSystem**

StregSystem is a system used to record purchases of products stored in two refrigerators located in a shared coffee room at our department at the university. The system works by having a terminal placed in the coffee room which is used to operate an internet browser which sends and retrieves data to and from a server. The server has a list of users and records their purchases and payments. It also keeps track of which products are offered, their prices and how much there are in stock of each product. The server uses a relational database for persistence and runs from a standard web server.

StregSystem currently exists in an old implementation which does not provide all the wished functionality and there is no documentation for the implementation. The development of

a new StregSystem with the Elucidator tool thus has a twofold purpose — to provide a better implementation with more features and, by using the Elucidator, create a system that is well documented. It is important that the StregSystem is documented as the maintainers of the system will be students which only reside at the University for a limited time.

### **Writer experiment**

Development of the system were performed by Thomas Vestdam, Amanuensis at the Department of Computer Science Aalborg University, together with one of the authors of this thesis. Vestdam had never used the Elucidator tool before, but had some knowledge of its capabilities. Thus the amanuensis was briefly introduced to the MRS-model and the functionality of the tool by the assisting author shortly before doing the actual development.

Vestdam had been designing the overall system before he started using the Elucidator 2, but had not done any detail design and implementation of the StregSystem. The experiment was conducted in a period of one week.

After this week the assisting author conducted a small informal interview with Vestdam where they discussed and collected their experiences using Elucidator 2 for documentation. The information collected in this interview is used in our evaluation of the model and tool, seen from the writers perspective.

### **Challenges for the writers**

The writers successfully produced documentation for the design and implementation of the StregSystem, but two problems reduced their efficiency.

Vestdam and the assisting author had been informed how the relational database, to be used for the StregSystem, should be incorporated, but due to technical difficulties and misinformation the relational database was not usable. Thus significant development time was used on finding a solution to circumvent this problem. This caused a delay in the development of the StregSystem, but gave good experiences on how the tool handled the documentation of changes to the system and its design.

A further infliction was, that Elucidator 2 tool was being actively developed during the span of the experiment. The complete development of Elucidator 2 was only finished a day before the experiment was finished. This did, however, not affect the developers notably as the development on the Elucidator tool only affected navigational issues in the browser. It did though have the effect that the browser was not used as extensively by the writers as expected.

### **Reader experiment**

After the development of the StregSystem was finished the two remaining authors of this thesis was allowed to read the documentation. This was done with the purpose of evaluating the model and tool from the readers perspective. The documentation used in the reading experi-

ment was the result of one week of development. Therefore, some parts of the StregSystem was not completely developed and consequently the documentation was in the same state of flux. We do though, not consider this a big disadvantage as this would properly also be the case for internal documentation written in any initial software project.

### **Project size**

In order to give an idea of the size of this project this section presents some statistical material on the size of the project. Vestdam and the assisting author produced 26 EDoc and 15 Java files during the one week of development. The implementation contained 2 packages and 15 classes. The documentation spanned 9 thematic catalogs and 26 documentation nodes was created. The writers explicitly created 32 documentation links, 75 source links and 27 external links — a grand total of 134 links.

A more detailed report on the statistical material from the project can be found in Appendix D on page 117.

We are aware that, due to the limited size of the experiment, the gathered statistical material can not be used as a basis to make concrete conclusions. We are instead primarily using the experiences gained by the writers and readers of the StregSystem project in order to indicate certain points.

## **5.2 Reflections upon the experience of the writers**

In this section we will present and reflect upon the most important experiences we have gained during the development of the Stregsystem. The experiences is presented in no particular order. We reflect on these experiences and in some cases propose solutions that can remedy possible found problems.

### **Solving problems using the MRS-model**

Before the experiment started we expected that documentation nodes would be produced and structured at the same time as problems where being solved during development. The experiment showed that this was not always the case when using the MRS-model.

In the preliminary phases of solving a problem both writers had problems dividing the documentation into segments. This was the case for dividing documentation into motivation, rationales and solution descriptions. It was also an issue when the writers had to choose a specific node type for the documentation nodes.

They experienced that it were easier to just write down their thoughts and action unstructured in the preliminary phases and then later place these thoughts and action in the correct documentation nodes. This had the effect that the documentation often was first written after the problem was solved or when writers could grasp the solution for the problem.

We do not see this as a major problem for the MRS-model and internal documentation. As long as the developer writes down his thoughts and structure the documentation while he has his knowledge about the actual problem solved fresh in mind the internal documentation will not suffer.

A contrary experience gained by the writers were that they considered it an advantage to be forced into segmenting the documentation when they actually wrote the nodes. It focused their writing and enabled also the reuse of, e.g., one motivation node in multiple rationales.

### Using the templates

The templates devised for use in internal documentation was from the start not intended to be neither ideal nor complete. Our experiment confirmed that this was the case.

When Vestdam was confronted with the templates and guidelines in the beginning of the test period he often found the guidelines confusing and became in doubt of whether he had chosen the right node type. Later on he adapted his writing to the templates but he did not feel that they all matched his needs for documentation.

The problems was not experienced by the other developer. Once he was on top of the problems he was solving, he used most of the documentation nodes and experienced it natural to chose the right node type. This should be seen in the light of the fact that this developer had contributed in the creation of the templates and their guidelines.

We see the experiences reported above as a question of the writers not having enough experience with the MRS-model and the templates used, and as a question of the style of the templates not matching with the expectations of the writer. This confirm our believes in that the templates used by a team of developers should be gradually changed in order to facilitate the need and styles of the developers in the team.

### The number of link roles

The intention of link roles where to strengthen the expressiveness of the documentation. We therefore devised eleven (22 if you count their symmetrical names) link roles. Unfortunately the number of roles for the links confused the writers by their sheer numbers. The writers also felt some roles had close semantic meaning. This made it a complex task to choose the right role for links.

A reason for the confusion could be that the writers where unfamiliar with the link roles and could not directly see the purpose of them. The size of the StregSystem was relative small and had a low complexity which indicates there were no direct need for expressing complex relationships.

A response to the presented problem is to reduce the number of link roles. One of the ways to reduce the number of roles will be to reduce roles with close semantically meaning to just one, e.g., *describes* instead of *describes* and *detailed-by*, *selects* instead of *selects* and *introduces*.

Another reduction would be to limit the roles to a basic set of roles, e.g., *premise*, *selects*, *declines*, *deprecates*<sup>1</sup>, *describes* and *mentions*. These basic link roles would in the same way as the templates be modified and possibly extended through continued iterations over the use and evaluation of the Elucidative environment.

### Validation of links

As we have discussed above both writers found it difficult to find a proper structure for the documentation early in the problem solving phase. As the problems where grasped the proper structure emerged. This had the consequence that documentation was moved from one node type to another and nodes that in the beginning where placed in one catalog where moved to another catalog.

The movement of nodes and their contents could inflict that links that linked to these nodes would become invalid and hereby useless. This result in an instance of the classical hypertext problem of keeping links consistent.

Fortunately this problem can be remedied by the Elucidative environment because it has information about the structure of both documentation and source code and their linking mechanisms. This can be used to validate links when performing the abstraction process. A link validation process can be used by the writer to detect errors in the documentation similar to errors from a compiler. The information could also be used to support the writer in moving nodes and provide semiautomatic update of possible invalid links.

It was also experienced that there were cases where the documentation surrounding source code links that had become invalid were incorrect. We anticipate that with a link validation process we could increase the proximity between source code and documentation when errors occur. This could be used by the writer not just to correct the link but also check if the surrounding documentation still applies.

### Writers motivation

One of the means we hoped would motivate the writers to write documentation was immediate pay back, i.e., that they would benefit from their own writing. It turned out that none of the writers ever used their own documentation in the test period.

The main motivational factor for the writers was instead to use the documentation written by the other writer and to see their own documentation being useful for others, e.g., the time spent writing good abstracts for documentation nodes where greatly appreciated when the documentation was read by the other. Thus the simple fact that the writer could see the documentation in the browser and felt it could be useful for others, especially the next generation of the developers on the StregSystem, motivated both writers.

---

<sup>1</sup>The difference between *declines* and *deprecates*, is that a solution that has been declined may be relevant in another part of the system, where as a deprecated solution is rejected throughout the system.

### 5.3 Reflections upon the reader experiment

This section will present the most important experiences gained through the reading experiment. The experiences is presented in no particular order. Similar to the reflection presented in the previous section we present the experiences and reflect upon them.

#### Starting point

The first displayed information for a project in the browser of the Elucidator environment is the hierarchal view of documentation and source code. As both readers were unfamiliar with the StregSystem project and the views only showed the names of catalogs and packages the readers found it hard to figure out where to start reading the documentation. A missing starting point consequently complicated the readers possibility to form a general view of the system from the start. This is especially a problem for readers with a serialistic reading style.

We do not see it as a major problem that the readers of the StregSystem could not immediately form a general view of the StregSystem. The StregSystem was not completely developed and large part of the documentation was not completed, but still a method should be found to enable readers to have a starting point for the system.

This could be done by having a documentation node in each catalog that is to be displayed each time a user enters the catalog. This node would be written by the writers of the system and would be a natural place to present an overview of the contents of the thematic catalogs and provide links to follow. Thus the Elucidative environment can support the writer in providing a overview which can be used as a starting point for readers — serialists as well as holists.

#### The usage of views

As a consequence of readers not being able to find a starting point at first glance, they both started to traverse the hierarchal view of thematic catalogs. When entering the catalogs their contents were listed and each documentation node presented with their full topic and abstract together with possible sub-catalogs.

Both readers used the hierarchal view to read the topic and abstract of nodes that seemed interesting. This enabled the readers to start comprehending which documentation was available for the StregSystem without going back and forth between the nodes.

As both readers were aware of the capabilities of the Elucidator 2 they started using the Documentation Index View. This view proved to be useful as it allowed the readers to filter documentation nodes based on their type and category. This was used to find motivations, continue on to the rationales and then finally solution descriptions, i.e., the reading became based on the MRS-model. In each step they studied the nodes and followed links to the extent they found it necessary. The readers experienced that the MRS-model and the use of relatively short documentation nodes was useful for doing this form of (holistic) reading.



The readers also felt they actually were reverse-engineering the documentation with the help of the views and navigational facilities of the Elucidator to grasp the system. It was noted by the readers that if they have had previous knowledge of the system, e.g., been active developers on the StregSystem, they felt that the navigation facilities also were useful. They experienced that specific parts of the system was easy to find with help of the views.

### **Information on links**

The Elucidator 2 environment and the MRS-model urge the use of links. This can result in a high number of links inside documentation nodes. As seen in the statistics in Appendix D on page 117 134 explicit links were created even in this small project. We have made an effort to reduce the readers need to unnecessarily explore links by providing information about the destination of a link though the navigation menu.

The readers all appreciated the distinction between link types through colors as it made the reader aware whether a link would lead to documentation, source code or external information. This was often used as the readers typically either were looking for source code or documentation.

When following links a navigation menu is displayed and a short summary of the destination is shown. This was of great help for the readers as it saved time while reading. Especially when using views that displayed many implicit links it saved time as, e.g., the topic and abstract for documentation nodes provided enough information to decide if it was worth the effort to follow the link.

### **Parallel hypertext**

When exploring the system and following links as described above, the readers made use of the possibility to view documentation and/or source code nodes in parallel.

The possibility to perform parallel hypertext with documentation and source code was powerful. Both readers used the parallel hypertext feature with a documentation node that explained the overall system with links to more concrete details. It allowed readers to open the documentation node in one side and used this as a reading-guide and as a fixed point for navigation. When following links the reader then consistently viewed the links destination in the opposite window to keep the context. This was true for both viewing source code and documentation.

In our initial experiments with the Elucidator 1 tool, documentation and source code were always shown in parallel and hence source code was always visible. A side effect of parallel hypertext with no constraints on the location of source code or documentation were that the focus was moved from examining source code to examining documentation.

## Visibility of the MRS-model

The readers utilized their knowledge of the MRS-model to start reading the documentation as described previously. The readers was aided by the Elucidator in utilizing the MRS-model as the documentation nodes were colored according to their category.

A further aspect of the MRS-model is the roles on links. The readers were not using the role on links when following the links from inside a documentation node. This is probably caused by the fact that the role of a link is only visible when it is activated. In the navigation menu the role was displayed but was not emphasized enough to be useful. Still, roles were used, not when doing normal browsing, but in the Context View of documentation and source code. Here the link roles were used to find, e.g., the premises for rationales. Without the link roles the readers would not be able to distinguish a simple reference to a node from a special usage of a node. Thus the use of links were advantageous in reflecting the MRS-model for the readers.

Another aspect was the use of change descriptions to track changes and thereby document the history of the system. Both readers found the documentation of the before mentioned technical problems to be of good use in an examination situation. The information was useful and the use of “declines”, “deprecates” and “selects” to express what was rejected and what was accepted were used by the readers. A feature missing from the system though was the possibility to show the documentation nodes in chronicle order to give a better historical view of the changes.

## Trustworthy documentation

Both readers were at several times “annoyed” by the documentation. The annoyance was caused by some documentation nodes which were only half finished and furthermore by links which pointed nowhere. However by applying link validation to the documentation it seems that this problem can be reduced. This was also noted in Section 5.2 on page 77.

Documentation nodes containing only a hint on what could be expected and sometimes nothing at all was to great distress, as the readers felt information was being held back or that the information was incorrect.

We do not believe this problem to be confined to this small experiment. Inconsistency and lack of documentation is a general problem concerning documentation of software. We imagine that if the internal documentation for a piece of software generally has these problems readers using the documentation will spend more time resolving the inconsistencies than actually benefiting from the documentation. We therefore suggest that to increase the trustworthiness of internal documentation, continuous documentation reviews are needed.

## 5.4 Discussion

The following section contains a discussion of the experiences gained through the experiment together with our own assessment of the MRS-model and the implemented Elucidative environment. The section is divided into three parts. First we discuss the MRS-model, next the implementation of the MRS-model in the Elucidative environment, and finally we discuss the usage of the Elucidator tool to document the history of software and documentation.

### 5.4.1 The MRS-model

We see the MRS-model as one of the main contributions of our work. This is based both on our own assessment of the model and the experiences gained through the experiment.

The MRS-model was used by both the writer and the reader. As stated previously in this chapter, the writer had difficulties dividing the documentation into the three categories while at the same time trying to solve a software problem at hand. They experienced that it was easier to just write down their thoughts and then structure them later. We recognize that this might be a problem. However, the first time you make a class hierarchy for a software program, it is not easy either. Therefore, we see a part of this problem as a matter of getting used to the MRS-model. A contrary experience was also gained by the writers: They considered it an advantage to be forced into segmenting the documentation, since it focused their writing of the documentation.

As for the roles on links, it was expected that the relatively large number of roles would benefit the expressiveness of the documentation. This was however not the experienced case, since the writers got confused by their sheer numbers, and therefore found it confusing to choose between them.

As for the readers, they experienced only minor problems with the usage of the MRS-model, and it was therefore quickly an advantage for them. This was for instance shown by the fact that the MRS-model was actually used to navigate the documentation. By starting with motivations documentation nodes, the readers navigated to rationale nodes and from there to solution description nodes. On the down side the MRS-model does not provide a natural place for the reader to start reading the documentation. This was especially needed when the readers started from scratch on a new and unknown system.

It should furthermore be noticed that conceptually the MRS-model is applicable on other documentation types than internal documentation. It is furthermore both language and paradigm independent. This means that all documentation situations where it is useful to represent, relate and capture motivations and rationales together with their actual description of solutions can therefore benefit from the MRS-model.

In conclusion we believe the experiment indicates that the MRS-model is particularly useful for the readers but also acceptable for the writer.

### 5.4.2 The Elucidator tool

It is our position that in order for the MRS-model to become successful it is dependent on good tool support. If the internal documentation and the MRS-model is to be useful, it not only need to be practical manageable for the writer to write the documentation, it also has to be accessible in a form that provide navigational facilities to grasp the possible large amount of documentation. The Elucidative environment implemented in this thesis is a suggestion for such tool support. We believe that the experiment confirmed this need for tool support. The tool support is therefore seen as the second main contribution of our work.

One of the consequences of our focus on the reader, was that the reader experienced better support by the tool than the writer. However, the experiment still showed that the writers used all the features in the editor and stated that it would be impossible to create the documentation without these features. Particularly the support for link insertion was praised. The selection of editor support was found short of the target though, and as a consequence better and more advanced editor support was queried for.

As for the reader, especially the navigation facilities was appreciated and extensively used. Furthermore, the usage of colors to increase the visibility of the MRS-model was found useful. As to the problem of no starting point in the documentation, the tool did not provide a direct solution. It did though, help to lessen the problem, by providing the navigation facilities.

On the technical front, we see the implemented elucidative environment as independent on multiple fronts:

Physically the Elucidative environment does not confine the users to a single development environment. The documentation is placed in EDoc files which are physically separated from the source code. This enables the free use of the source code files in other environments without modification.

Architectural the environment is also flexible. Each component in the implementation is low coupled to the other components. This means that e.g. the environment is not dependent on a specific editor, database or server and they can all relatively easily be replaced.

Finally the environment is also language independent. Currently there only exist a Java abstractor for the environment, but since the Elucidator have a Data model that is based on entities and their relationships it is possible to create an abstractor for other languages. The only requirement is that it is feasible to uniquely identify entities that should be documented via links in the Elucidative environment.

### 5.4.3 History in documentation

The final topic for discussion is the usage of history in the internal documentation. Not many experiences was gained on this subject, since the period in which the experiment was conducted was only one week. However, due to unexpected difficulties with the relational databases, some experiences related to changes was never the less gained.

Initially it was planned to use the Informix database system. Therefore source code and documentation was produced for this solution. It however turned out that this database could not be used, and a switch was made to the MySQL database system. This switch was documented through the usage of a Change description node, and thus, the history of the program was maintained. At a later point, the problems with the Informix database system was resolved, and it was experienced to be relatively easy to find the documentation for this solution.

We are aware that, based on this experience, nothing conclusive, concerning the importance and usability of the history of the documentation, can be said. However, the method provided by the MRS-model for documenting the history of the software was experienced to be useful for both the writers and the readers. In our opinion this indicates that the history of the software can be documented and be an natural part of the internal documentation.



# 6

## Conclusion

This chapter concludes our master thesis. This is the place where we look back at the work done and results accomplished throughout the project and, based on these, conclude on the two formulated hypotheses.

In order to weaken or affirm these two hypotheses an investigation with five steps was conducted throughout the project. 1) First the notion of internal documentation in software development was examined. The main result of this examination was that internal documentation becomes *more fruitful when it is written with a clear distinction between rationales and solutions*. 2) Next we divided participants working with internal documentation into two roles: the *reader and the writer*. The reader was chosen to be the focus of our further work, and a characterization of him, showed him to have tendencies towards being *holist* as well as *serialist*. As for the writer it was stated that *no demands for special writing skill*, could be placed on him.

3) Based on these first two steps, the *MRS-model* was formulated. The main concept of the MRS-model, is a division of the internal documentation into three interrelated categories. 4) Furthermore an important part of the MRS-model is its *realization in the elucidative environment*. This was done by using the elucidative environment developed in the first part of the master thesis. 5) Finally, an experiment, called the StregSystem experiment, was conducted in order to *show the effect of the model* when applied to internal documentation in a software development project.

The main results of the investigation presented above is the *formulation of the MRS-model* and the *practical realization of this model in the elucidative environment*, that is, the Elucidator 2 tool. These two results, together with the results from the StregSystem experiment, will be used to affirm the two hypotheses.

The first hypothesis states:

**To present internal documentation in order to facilitate the reader, it is necessary to structure it in a predefined way. This structure, combined with navigation facilities, will be beneficial to the internal documentation.**

This hypothesis actually contains two sub-hypotheses, both centering around structure in internal documentation. We therefore conclude on these two sub-hypothesis separately.

The first sub-hypothesis concerns the necessity of a predefined structure of the internal documentation, with the purpose of *presenting* the documentation. By applying the MRS-model to internal documentation, traceability in the presentation of the documentation is provided to the reader, since this model makes a clear distinction between motivation, rationales and descriptions of the solutions. By using the concrete realization of the MRS-model in the elucidative environment, visibility of the three different categories of documentation is presented to the reader. This is accomplished by the implementation, e.g., by using a coloring scheme which reflects the three categories. Finally, the internal structure of the documentation nodes also makes an uniform presentation of the documentation possible.

The StregSystem experiment did not affirm nor weaken the *necessity* of structure, but our experiences with the structure-less documentation produced with the Elucidator 1 tool indicate it. With regard to the roles on links the writers did not appreciate the full expressiveness of the MRS-model, since they were burdened with the number of roles. We believe that this indicates that the proposed expressiveness is not needed and the number of links roles can be reduced.

The second sub-hypothesis concerns the *benefit* accomplished by applying structure to the internal documentation. By applying the MRS-model to the internal documentation it will benefit from the clear distinction between motivation, rationales and descriptions of the solutions, and the documentation will thereby avoid being muddled up. By using the Elucidator 2 tool, it is possible to extract implicit information, such as the navigation menu and the views, which also adds to the benefit of the internal documentation.

The uniform presentation of the internal structure of the nodes was not experienced to be as important as expected. The overall structure of MRS-model on the other hand together with the navigation facilities proved to be beneficial to the documentation for the readers. In general both the overall structure and the internal node structure imposed by the MRS-model, helped the writers to structure their documentation. During the StregSystem experiment it was indicated that the usability for the writer was limited by two factors. The first being the writers unfamiliarity with the MRS-model. The second factor was that the writers the preliminary phases of solving a problem in the software, were unable to structure thier thoughts and hence use the MRS-model.



The second hypothesis stated:

**The history of the software is important since most changes in the software, are made as a consequence of a rationale. The history of the software can be documented and be a natural part of the internal documentation.**

As with the first hypothesis this one can also be split into two sub-hypotheses. Both these sub-hypothesis centers around the history, and how this relates to internal documentation.

The first sub-hypothesis concerns the *importance* of the history of software in connection to rationales. That is, will the internal documentation benefit from the presence of documentation of the history of the software. Based on the work done in this project we have not been able to neither weaken nor affirm this hypothesis. In fact we have come to believe that the question of whether this hypothesis is correct or not, is a matter of opinion and point of view. However, our work shows that if one believes that the hypothesis is correct, we can affirm the second sub-hypothesis.

The second sub-hypothesis concerns the question of whether the history *can be documented* and furthermore be a *natural part* of the internal documentation. Our work affirms this hypothesis. As a part of the realization of the MRS-model, a change description node type was created. This node type, together with the two link role targeted at the history; introduces and deprecates, made it possible for the writer to document the history. Besides this, the experiences gained by the StregSystem experiment showed that the documented history was actual useful, and was considered to be a natural part of the documentation.

- o -

In conclusion, the work presented in this report has not proven, but indicated that it is a necessity to structure the documentation in a predefined way in order to present it so it facilitates the reader. Arguments have however been provided which show that if the documentation is structured in a predefined way, it can be presented so it facilitates the reader. It has furthermore been shown that the internal documentation do benefit from applying a structure to the documentation.

Besides this, we have not been able to neither weaken nor affirm our claim that the history of software is important. We have however, showed that given the history is considered important, it can be documented, and be a natural part of the internal documentation.



# 7

## Future work

Having worked with the notion of elucidative programming for a whole master thesis, a lot of ideas to extension and further development of the work presented in this report has emerged. In this chapter we present three ideas for subjects which we believe would be interesting and rewarding to work on in the future.

### **Experiments using the elucidative environment**

In this project we have performed two small experiments to show that the elucidative environment and MRS-model is usable in our own surroundings.

Seen in the light of the relatively small size and short duration of these experiments it would be natural to conduct one or more experiments that explore the use of Elucidative programming and its environment to a larger extent. By larger extent we mean the duration, complexity and participants of the project has to be considerably higher than it was the case with the experiment performed in this project.

The purpose of these experiments should be threefold: First, they should seek to answer if developers productivity increases by writing internal documentation in an elucidative environment. Secondly, they should try expose if software, for which internal documentation exist, has a higher quality, i.e., is more understandable, modifiable and reusable. Finally, it could be interesting to try to show if the history of software is as important as claimed in this master thesis.

### **Motivating the writer**

Until now the writer present in our work has been solely motivated for writing internal documentation through his own personal satisfaction or professional obligation. Either he has been personally interested in producing internal documentation for his software or his development team has obliged him to do so.

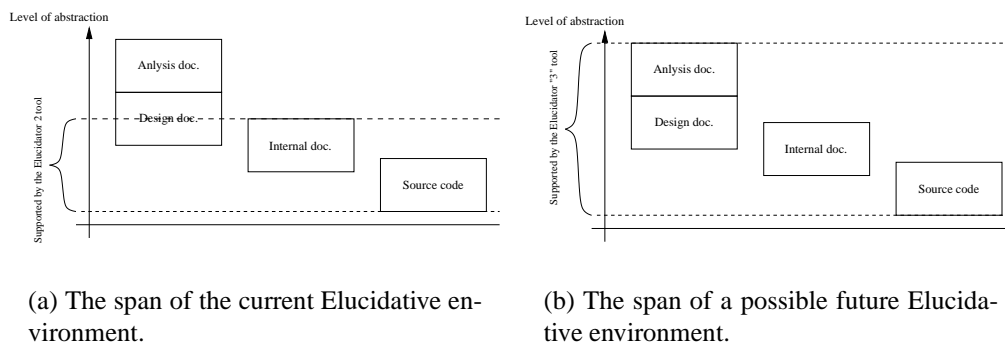
If the only motivation for the writer to produce internal documentation is rules stated by the development team, and he personally cannot see the advantage of producing internal documentation, it will be hard for him to write quality documentation and even so quality software.

Throughout our work we have attempted to reduce the burden placed on the writer by providing him with an elucidative environment with, e.g., navigation facilities and editor support. This is however only a consoling measure. It could therefore be interesting to investigate which non tool related measures can be devised to motivate writers to voluntarily engage in the process of creating internal documentation.

### Elucidative environment in a process

Our focus in this thesis has been on internal documentation of software. This type of documentation is mainly created during design and implementation. This is illustrated in Figure 7.1(a).

We have disregarded the documentation created during other phases of the classical object oriented development process. We think that an interesting challenge would be to explore if, and how, an Elucidative environment could be used for documenting all these phases, while at the same time maintaining the close relation to the source code. This is illustrated in Figure 7.1(b).



**Figure 7.1:** *Illustration of the span of the currently implemented Elucidator tool, and a possible future implementation of the Elucidator tool.*

The documentation presented in the analysis and design documents should typically be changed over time in the iterative process. This process will typically also include reviews (both source code and hopefully documentation reviews), milestones and an extended set of roles (such as managers and customers). Therefore, by extending the span of the Elucidator tool to also include analysis and design documents, it becomes important to provide support for this iterative process. At the moment the Elucidator 2 tool does not have any support for processes, but we find it both natural and interesting to extend the Elucidator tool with this capability.

# A

## Examples of different node types and link roles

In this appendix we present a number of different node types and link roles. These are all concrete examples of possible node types and link roles, that were found useful in the experiment we have conducted. They should not be regarded as archetypical examples but merely as possible implementations of the MRS-model. Examples of the use of the nodes in the experiment is included as screen captures.

### A.1 Examples of concrete node types

#### Node types in the Motivation category

**Requirement** is a node type that describes a requirement or a constraint that specifies or restrains the behavior, design or implementation of the system. This can be seen as the information that would go into a normal system definition in traditional Object-oriented Analysis and Design [Mathiassen et al., 1997]. It can also be an internal requirement specified by a developer on the software project.

A requirement contains a description of the requirement and a list of the parties that has specified the requirement.

**Bug report** is simply a report of a error in the system. The bug report will contain a description of which part of the system that is affected by the bug.

The actual description of the bug contains a description of the circumstances under which the error occurs, a description of the error and possibly suggestion on how to fix the bug. The description may contain debugger traces, excerpts from log files etc.

**Improvement** is a suggestion for an improvement of the system. The suggestion contains a description of which part of the system that is considered in the improvement and a actual description of the improvement. An example of an improvement could be that we would like to use another and more stable database for the StregSystem.

**Requirement:**

**The stregsystem is developed by two developers with limited time**

**Abstract:** *The development of the stregsystem is constrained by a number of facts: It is developed mainly by two developers; Thomas Vestdam and Kristian Lykkegaard Sørensen. The development is done as part of a masters project, that investigates documentation of software. The main part of the development was to be done within one week.*

**Status:** Finished

**Keywords:** [ [stregsystem](#) ] [ [constraints](#) ] [ [development](#) ] [ [documentation](#) ] [ [dopu](#) ]

**Author:** Kristian Lykkegaard Sørensen

**Created:** 2000/05/14 14:46:18

**Last updated:** \$Date: 2000/05/14 15:42:53 \$

**System ID:** streg/conditionsForTheDevelopment.edoc

**Description**

The [stregsystem](#) is developed because the old stregsystem, had served it's time. It was difficult to maintain and difficult to understand. Hence was it difficult to change the system. At this time three master studends at the department, is working on a development tool to support the documentation of internale system understading. This masters project is a part of the [DOPU research programme](#).

It was determined that the new stregsystem should be developed by two developpers, one Amanuensis [Thomas Vestdam](#) and one of the three masters students [Kristian Lykkegaard Sørensen](#).

One week was the estimated deelopment time of the project. The system was not intended to be ready for "production" in this week, but most of the development was to be placed in this week.

During the development the two developers should document the development and reflect on the experince with the documentation tool.

**Specified by**

The contraits where set by Thomas Vestdam, Max Rydahl Andersen, Claus Nyhus Christensen and Kristian Lykkegaard Sørensen.

**Figure A.1:** A concrete example of a documentation node of the type "Requirement".

## Node types in the Rationale category

**Rationale** Is a node type that describes the arguments that leads to a chosen solution and a set of declined solutions. We structure the rationales as first a presentation of the driving forces and motivations that affect the rationales. These motivations are described short, since they are possibly described in a motivation type node.

For each motivation that appear in some other node, a link is given with a short description of the specific part or aspect of the motivation that makes it relevant here and affect the rationale. Some of the driving forces may also very well be solution description of other parts of the system, if those parts affect the decisions made in the rationale.

The rationale contains a short description of the chosen and declined solutions. As the solutions and possibly alternatives are described in a solution description type node, this description focuses on the properties of the solutions that affects the decision.

At the end of a rationale node is a discussion of the choices made. This can be a discussion of the consequences of the selected or declined solutions. This part may also include personal subjective assessments.

**Change description** The change descriptions structure is the same as the general rationale node. The difference is that this explicitly defines a change in the system. It is through this type of nodes, that the history of the changes to system is preserved.

The forces section of the change description describes the reasons for the change. The solutions section should describe the old system as well as the new system.

## Node types in the Solution description category

**Concept** The concept node type is used to separate and describe a subject or concept which has special meaning/importance for the rest of the documentation. A concept will typically be a technical term from the problem domain of the program being developed, but it could e.g., also be a concept which needs clarification because one or more developers in the development team is not familiar with it.

An example of the first type of concept could be a development team doing a accounting system. It would then be useful to have the concepts *debit* and *credit* described, since software engineers are not likely to know these in detail unless they have developed accounting systems before.

**Aspect** An aspect node is used to document a facet of a system which is used in various parts of the system.

Examples of the usage of the aspect node, could be to document how information is written to a log file throughout the whole system, or how the loading and saving of files is handled.

**Design pattern instance** Design Patterns [Gamma et al., 1995] has become increasingly popular when creating programs using object-oriented programming languages. We therefore introduces the design pattern instance node type for the documentation of the usage of design patterns. It is important to note that the node documents an instance of a design pattern (which classes are involved in the instance of this pattern and so on), and not the design pattern itself, since this is already documented in the description of the design patterns.

**Rationale:**

**We use Java Properties instead of the Servlet Context Parameters**

**Abstract:** *We use Java Properties in stead of the Servlet Context Paramerts, bacause we have no where to place the Servlet Context Paramerts.*

**Status:** *Finished*

**Keywords:** [ [web.xml](#) ] [ [properties](#) ] [ [paramerts](#) ] [ [servlet api](#) ] [ [servlet context](#) ]

**Author:** Kristian Lykkegaard Sorensen

**Created:** 2000/05/10 15:28:59

**Last updated:** \$Date: 2000/05/28 08:23:23 \$

**System ID:** streg/parameters/whyWeUse.JavaProperties.edoc

**Forces**

We would have liked to use the Servlet API to specify the global parameters for the [stregsystem](#). But there is no place in the web.xml file, to place parameters that are global to the [context](#). So instead we use the normal Java way to specify Properties, which is to use the [Properties](#) class.

**Solution**

The alternative solution would have been to use the Servlet [context](#). When a Servlet is initialized with its [init](#) method it revieves a ServletConfig. Throug this is it possible to [fetch the Servlet Context](#). And with this, shoud it be possible to acces paramerts that are global to the servlet context. -- In theory.

Since this is not possible we use standard [Java Properties](#).

**Discussion**

It might be possible in a later release of the Servlet API, to specify parameters global to the context in the web.xml file.

**Figure A.2:** A concrete example of a documentation node of the type “Rationale”.



**Change description:**

**The database is moved to MySQL because of problems with Informix JDBC**

**Abstract:** *Because we was not able to make the Informix JDBC driver work, we changed the system to use a MySQL database in stead. This had effect on the BasicPageProducer and the Stregsystem parameters. We at a later time like to move back to Informix.*

**Status:** Finished

**Keywords:** [ MySQL ] [ Informix ] [ JDBC ] [ Stregsystem parameters ]

**Author:** Kristian Lykkegaard Sorensen

**Created:** 2000/05/14 14:41:09

**Last updated:** \$Date: 2000/05/18 14:07:33 \$

**System ID:** streg/theDataBase/FromInformix2MySQL.edoc

**Forces**

As we had many **problems using the Informix JDBC driver** and the development had **done within one week** we decided to move the database from the **wanted Informix solution to a less stable MySQL database**. MySQL was chosen because one of the developers had access to experience the a JDBC connection to MySQL. This MySQL experince came from the Elucidator project, where **Claus Nyhus Christensen** had written a lot of JDBC for MySQL.

**Solution**

Instead of **using the Informix JDBC driver** we changed the DBHandler, so that **it used the MySQL driver instead**. Since all JDBC access was encapsulated in this class, we had to change the loadDriver method from the **Informix driver** to the **MySQL driver**.

Since our experience with the MySQL driver told us, that it was impossible to specify the username and password directly in the **database URL**, we also introduces two extra fields; **dbUsername** and **dbPassword**. These has to be set in the **constructor**, which is why we also change the **instantiation of the DBHandler**.

This has the consequence the we have to add **two extra parameters** for the entire stregsystem as opposed to **the Informix version** of the system, that only needed one parameter for the database.

**Discussion**

There was a very good reason for the choice of the Informix database. The local Informix database is used by the systemadministration, and is therefor much more likely to be supported in the future. The MySQL database on the other hand is used for experiments

**Figure A.3:** A concrete example of a documentation node of the type “Change description”.

An example of the usage of this node type could be to document that a Visitor Pattern is used in the implementation of the Abstractor component (see Section 2.1 on page 9 for further information on the component) found in our Elucidator tool.

**Process** A process node should describe the interaction and flow internally in the software system. This means that this type of node will e.g., document how different components in a system interact with each other. The process node will often use links to task node to describe the interaction and flow.

As an example of a process node in the description of the interaction between the Abstractor, Data model and Generator components in the Elucidator tool. See Section 2.1 on page 9 for information on the functionality of these components).

**Task** The task node is used for documenting a limited job the system should be able to perform. This mean that this node type will typically be used to divide the documentation into manageable chunks. The task node type is often used in connection with the process node type described above.

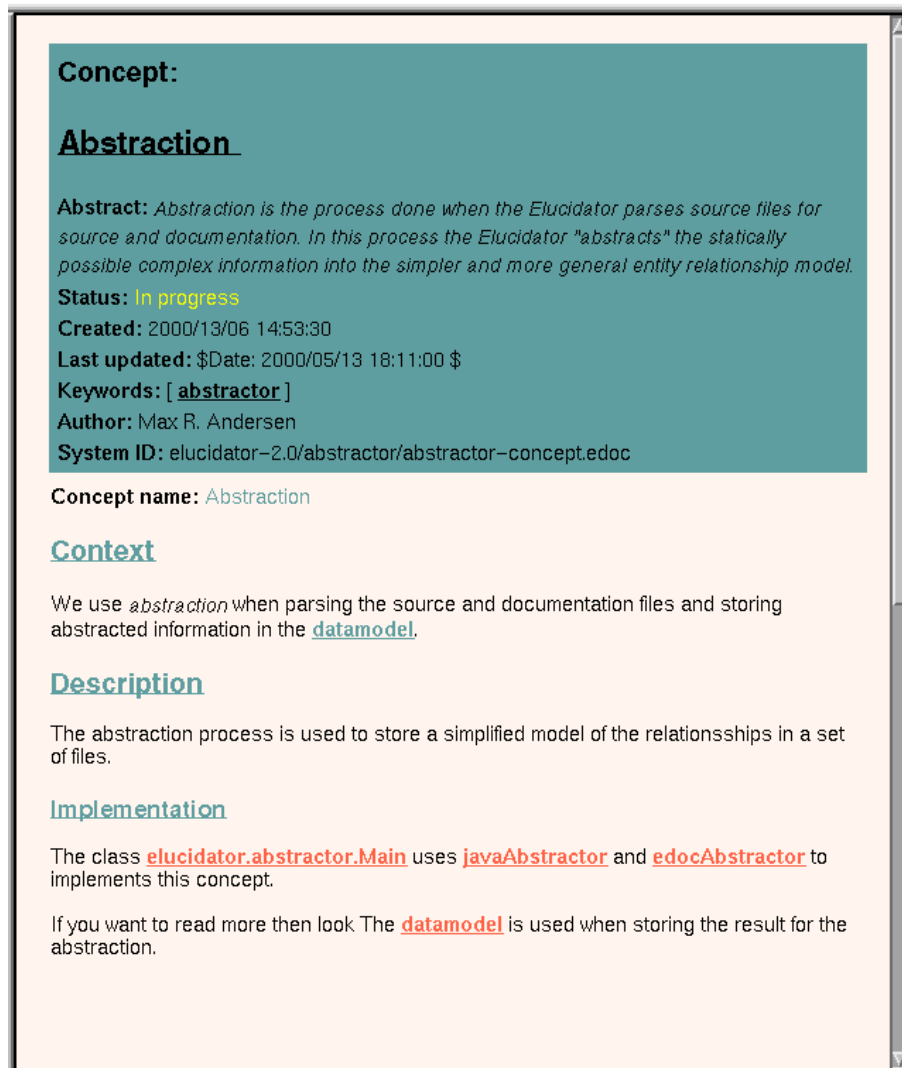
An example of a the usage of the task node could be a system which stores its preferences in a file. The task of loading these preferences will then be documented in a task node (and the node may link to an aspect node describing how the loading of files are generally handled in the system).

**Entity implementation** This node is used to describe details of the implementation of a entity (such as a package, a class or a method). It is important to note the difference between the entity implementation nodes and interface documentation produced with the help of tools such as JavaDoc. Entity implementation node focuses on the actual implementation (used data types, algorithms and so on) while interface documentation focuses (or should focus) on the usage of the entities by specifying the interface to the entities.

An example of an entity implementation node could be the documentation of the XSL-ProcessorPool class found in the implementation of the Elucidator tool. This class manages the accounting of a list of objects. In doing so it used mutex concepts and these will then be documented in the entity implementation node.

**Essay** Our final example of solution description nodes is the essay node type. This node type is special in that is has no inherent structure and it is used if no other node type in the solution description category seems to fit.

An example of an essay node could be a overview over the StregSystem. This documentation node explains the main components of the StregSystem and gives a UML class inheritance diagram to explain the overall placement of responsibilities in the system.



**Concept:**

### Abstraction

**Abstract:** *Abstraction is the process done when the Elucidator parses source files for source and documentation. In this process the Elucidator "abstracts" the statically possible complex information into the simpler and more general entity relationship model.*

**Status:** In progress

**Created:** 2000/13/06 14:53:30

**Last updated:** \$Date: 2000/05/13 18:11:00 \$

**Keywords:** [ [abstractor](#) ]

**Author:** Max R. Andersen

**System ID:** elucidator-2.0/abstractor/abstractor-concept.edoc

**Concept name:** [Abstraction](#)

### Context

We use *abstraction* when parsing the source and documentation files and storing abstracted information in the [datamodel](#).

### Description

The abstraction process is used to store a simplified model of the relationships in a set of files.

### Implementation

The class [elucidator.abstractor.Main](#) uses [javaAbstractor](#) and [edocAbstractor](#) to implements this concept.

If you want to read more then look The [datamodel](#) is used when storing the result for the abstraction.

**Figure A.4:** A concrete example of a documentation node of the type “Concept”.

**Design pattern Instance:**

**Use of the Visitor design pattern in the Abstractor**

**Abstract:** *The Visitor design pattern is used in the abstractor to provide maximum flexibility independent of the language being abstracted. Most parsers can provide a AST which can be processed by an abstractor, for this the visitor pattern is excellent.*

**Status:** In progress

**Keywords:** [ abstractor ] [ AST ] [ visitor ]

**Author:** Max R. Andersen

**Created:** 2000/05/28 17:34:33

**Last updated:** \$Date: 2000/05/08 15:28:44 \$

**System ID:** elucidator-2.0/abstractor/visitor-pattern.edoc

**Design Pattern name:** Visitor pattern

**Context**

The Visitor pattern is applied to the abstraction of Java source code. It could also be applied to the abstraction of edoc and/or other languages that the Elucidator needs to abstract.

**Purpose**

The purpose of the Visitor pattern is to have a separate class outside the [KJC Java Compiler](#) that can traverse the AST provided by their Java compiler.

**Roles**

[AbstractVisitor](#) defines the interface for any ConcreteVisitor that wishes to traverse the Elements in the AST. The Elements are all JPhylum's from the KJC package which defines the accept(Visitor) method.

The actual visiting is done by the [ConcreteVisitor](#) for Java.

**Collaborations**

The visitation is started by the [javaAbstractor](#). The traversal is initiated in the run method when all the AST's has been generated for each Java file.

Note that the AbstractVisitor handles the default traversal of the AST nodes in a left to right manner. Thus there is no need to implement visitor methods for Elements which is irrelevant for the ConcreteVisitor.

**Description**

The implementation is straightforward and based on the pattern described in the GoF book. The only remark is that the AST from the KJC compiler contains Elements that represent comments which is not true visitable nodes. This is probably fixed in the newer versions of KJC, but they do not include all the changes we need to have complete access to the AST. The handling of comments is therefore processed specially in the visitor.

**Figure A.5:** A concrete example of a documentation node of the type "Design pattern instance".

## A.2 Examples of link roles

The following description of link roles is supplementary to the other link roles described in the Design chapter in Section 4.3.2 on page 57.

**Uses/Used-by:** The *Uses* role is used to specify that the documentation presented in some documentation node, in order to explain something, uses another documentation node. Symmetrically, the *Used-by* link is used to specify that some documentation node is used by another documentation node.

An example of the usage of this role, could be that a documentation node describing the main Abstractor components of the Elucidator tool, wants to specify that the Abstractor uses two other components (that is, the JavaAbstractor and the EDocAbstractor), which are documented in separate nodes.

**Detail-of/Detailed-by:** The *Detail-of* and *Detailed-by* roles are similar to the Uses/Used-by role. The main difference is that this role, contrary to the Uses/Used-by role, expresses a difference in the level of detail in the two documentation nodes.

An example of the this role could be that a documentation node describing the Abstractor might mention that it is implemented by using a Visitor Pattern, and a link with the role Detailed-by is then made to a documentation node describing the details of how this instance of the Visitor Pattern is implemented.

**Reuses/Reused-by:** The *Reuses* role is used in development situations where the writer wants to explicitly state that some part of the developed system was reused from another system. In this case he makes a link with the role Reuses from a documentation node describing how the piece of software is reused, to a documentation node which is documenting the software which is being reused.

**Implements/Implemented-by:** The final link role is *Implements*. As the name implies this role is used to express that something described in the documentation is implemented by a source code node. Typical examples, could be key components of a system which is documented as concepts of the system. In these concept nodes links with the role *Implemented-by*, and destination anchor in a source code node, could be placed.

An example of the usage of this role is present in the documentation for the Elucidator tool. In this documentation, a concept node describing abstraction is placed. In this node a link with the role Implemented-by is made to the class which implements the abstraction concept.



# B

## Grammar for the EDoc language

This is the grammar for the EDoc language. The grammar is specified as a DTD [Bray et al., 1998].

### edoc2.dtd

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- DTD for the Elucidator 2.0 -->
5 <!-- The Elucidator is a part of the DOPU research programme, see -->
  <!-- http://dopu.cs.auc.dk -->
  <!-- The edoc element is the root element. The element corresponds to -->
  <!-- a node in the Elucidator. It contains one element that specifies -->
10 <!-- the type of the node.-->
  <!-- In this version we only support the essay node type.-->
  <!ELEMENT edoc ( essay | requirement | bug-report | improvement |
    rationale | change-description | concept | aspect |
    task | process | entity-implementation |
15    design-pattern-instance )>
  <!-- ===== -->
  <!-- HEAD ELEMENT AND SUBELEMENTS -->
20 <!-- ===== -->
  <!-- The head element is to appear in all different types of nodes. It -->
  <!-- contain the fixed part of the node: Topic, Abstract, Keywords and -->
  <!-- Status.-->
25 <!ELEMENT head ( topic , abstract , status , keywords?, author?,
    created , last-updated )>
  <!-- The Topic element contains a topic heading for the node. From -->
  <!-- the STOP method we have the following recommendations for the -->
30 <!-- contents of the topic:-->
  <!-- The topic heading must characterize and introduce the thematic -->
```

```

<!-- content , not merely categorize (label) the node body. Topic -->
<!-- headings are more likely to be representative and topically -->
<!-- faithful if they are (1) constructed as sentence fragments and -->
35 <!-- (2) rewritten after composition of the node. -->
<!ELEMENT topic (#PCDATA)*>

<!-- The author element just contains the name of the author for a node. -->
<!ELEMENT author (#PCDATA)*>
40

<!-- The abstract element contains a abstract for the node. From the -->
<!-- STOP method (again) we have the following recomandations for the -->
<!-- contenst of the abstract (Thesis sentences in the STOP -->
<!-- method).: -->
45 <!-- It is supposed to be a thematic window into the contents of the -->
<!-- node. Some more bla . on how to write a good abstract. -->
<!ELEMENT abstract (#PCDATA)*>

<!-- The keyword element contains a list of important keywords , from -->
50 <!-- the node. The kw element contains the single keywords. -->
<!ELEMENT keywords (kw)*>
<!ELEMENT kw (#PCDATA)*>

<!-- Elements to keep track of important dates and times for the -->
55 <!-- node. -->
<!ELEMENT created (#PCDATA)*>
<!ELEMENT last-updated (#PCDATA)*>

<!-- The status element contains a textual representation of the -->
60 <!-- status of the node: NEW, INPROGRESS and FINISHED. Additional text -->
<!-- can be written. -->
<!ELEMENT status (new | inprogress | finished)>
<!ELEMENT new EMPTY>
<!ELEMENT inprogress EMPTY>
65 <!ELEMENT finished EMPTY>

<!-- ===== -->
<!-- STANDARD ELEMENTS -->
70 <!-- ===== -->

<!-- The section element contains a title and more text. -->
<!ELEMENT section (title , (p)*)>
<!ATTLIST section
75      label      ID      #IMPLIED
      sbase      CDATA   #IMPLIED
>

<!-- The title element just contains a title for a section or -->
80 <!-- figure. -->
<!ELEMENT title (#PCDATA)*>

<!-- The ordinary paragraphs -->
<!ELEMENT p (#PCDATA | figure | itemize | enumerate | desc |
85      eimage | xlink | slink | dlink | term)*>

<!-- The figure element are for figures in the edoc document. -->
<!ELEMENT figure (body, title)>
<!ATTLIST figure
90      label      ID      #IMPLIED
>

<!-- The eimage element are for images that are a part of the -->
<!-- Elucidative documentation . Since we do not validate the -->
95 <!-- documents , the writer is free to add any attributes . We do -->
<!-- require that the href attribute points to some file in the edoc -->
<!-- bundle. -->
<!ELEMENT eimage EMPTY>
<!ATTLIST eimage

```



```

100         src      CDATA      #REQUIRED
>

<!-- The body of the figures -->
<!ELEMENT body (#PCDATA | itemize | enumerate | desc |
105         eimage | xlink | slink | dlink | term)*>

<!-- The itemize, enumerate and desc elements are typographical -->
<!-- elements. -->
110 <!ELEMENT itemize (item)*>
<!ELEMENT enumerate (item)*>
<!ELEMENT desc (pair)*>
<!ELEMENT pair (name, item)>
<!ELEMENT name (#PCDATA)*>
115 <!ELEMENT item (#PCDATA)*>

<!-- The term element is used for making indexes -->
<!ELEMENT term (#PCDATA)*>

120 <!-- The xlink is a URL to an external document -->
<!ELEMENT xlink (#PCDATA)>
<!ATTLIST xlink
        role      ( detail-of | detailed-by |
125                 describes | described-by |
                    mentions | mentioned-by |
                    implements | implemented-by |
                    uses | used-by |
                    premise | premise-for |
130                 declines | declined-by |
                    selects | selected-by |
                    deprecates | deprecated-by |
                    introduces | introduced-by |
                    reuses | reused-by)      #REQUIRED
        href      CDATA      #REQUIRED
135 >

<!-- The slink is a link to some symbol in java source, included in -->
<!-- the elucidator bundle. -->
<!ELEMENT slink (#PCDATA)>
140 <!ATTLIST slink
        role      ( detail-of | detailed-by |
                    describes | described-by |
                    mentions | mentioned-by |
                    implements | implemented-by |
145                 uses | used-by |
                    premise | premise-for |
                    declines | declined-by |
                    selects | selected-by |
                    deprecates | deprecated-by |
150                 introduces | introduced-by |
                    reuses | reused-by)      #REQUIRED
        href      CDATA      #REQUIRED
>

155 <!-- The dlink is a link to some node in the edoc bundle. -->
<!ELEMENT dlink (#PCDATA)>
<!ATTLIST dlink
        role      ( detail-of | detailed-by |
160                 describes | described-by |
                    mentions | mentioned-by |
                    implements | implemented-by |
                    uses | used-by |
                    premise | premise-for |
                    declines | declined-by |
165                 selects | selected-by |
                    deprecates | deprecated-by |
                    introduces | introduced-by |

```

```

reuses | reused -by)      #REQUIRED
href      CDATA          #REQUIRED
170 >

<!-- ===== -->
<!-- ELEMENTS USED IN THE NODES -->
175 <!-- ===== -->

<!ELEMENT description ( p | section )*>
<!ATTLIST description
180      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>
<!ELEMENT specified -by ( p | section )*>
<!ATTLIST specified -by
185      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>
<!ELEMENT concerning ( p | section )*>
<!ATTLIST concerning
190      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>

<!ELEMENT forces ( p | section )*>
<!ATTLIST forces
195      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>

<!ELEMENT solution ( p | section )*>
200 <!ATTLIST solution
      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>

205 <!ELEMENT discussion ( p | section )*>
<!ATTLIST discussion
      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>
210
<!ELEMENT dictionary ( p )*>
<!ATTLIST dictionary
      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
215 >

<!ELEMENT concept -name ( #PCDATA )*>

<!ELEMENT pre -condition ( p | section )*>
220 <!ATTLIST pre -condition
      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>

225 <!ELEMENT post -condition ( p | section )*>
<!ATTLIST post -condition
      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
>
230
<!ELEMENT applicable -to ( p | section )*>
<!ATTLIST applicable -to
      sbase      CDATA      # IMPLIED
      label      CDATA      # IMPLIED
235 >

```

```
<!ELEMENT design-pattern-name (#PCDATA)>

<!ELEMENT context (p | section)*>
240 <!ATTLIST context
      sbase      CDATA      #IMPLIED
      label      CDATA      #IMPLIED
    >

245 <!ELEMENT purpose (p | section)*>
    <!ATTLIST purpose
      sbase      CDATA      #IMPLIED
      label      CDATA      #IMPLIED
    >

250 <!ELEMENT roles (p | section)*>
    <!ATTLIST roles
      sbase      CDATA      #IMPLIED
      label      CDATA      #IMPLIED
255 >

    <!ELEMENT collaborations (p | section)*>
    <!ATTLIST collaborations
      sbase      CDATA      #IMPLIED
260      label      CDATA      #IMPLIED
    >

    <!-- ===== -->
265 <!-- THE ESSAY NODE -->
    <!-- ===== -->

    <!-- This node type only contains a head and no additional element for -->
    <!-- the content.-->
270 <!ELEMENT essay (head, (p | section)*>
    <!ATTLIST essay
      sbase      CDATA      #IMPLIED
    >

275 <!-- ===== -->
    <!-- THE REQUIREMENT NODE -->
    <!-- ===== -->

280 <!ELEMENT requirement (head, (description, specified-by))>
    <!ATTLIST requirement
      sbase      CDATA      #IMPLIED
    >

285 <!-- ===== -->
    <!-- THE BUG REPORT NODE -->
    <!-- ===== -->

290 <!ELEMENT bug-report (head, (concerning, description))>
    <!ATTLIST bug-report
      sbase      CDATA      #IMPLIED
    >

295 <!-- ===== -->
    <!-- THE IMPROVEMENT NODE -->
    <!-- ===== -->

300 <!ELEMENT improvement (head, (concerning, description))>
    <!ATTLIST improvement
      sbase      CDATA      #IMPLIED
    >
```

```

305 <!-- ===== -->
    <!-- THE RATIONALE NODE -->
    <!-- ===== -->

310 <!ELEMENT rationale ( head , ( forces , solution , discussion ) )>
    <!ATTLIST rationale
        sbase CDATA #IMPLIED
    >

315 <!-- ===== -->
    <!-- THE CHANGE DESCRIPTION NODE -->
    <!-- ===== -->
    <!ELEMENT change-description ( head , ( forces , solution , discussion ) )>
320 <!ATTLIST change-description
        sbase CDATA #IMPLIED
    >

325 <!-- ===== -->
    <!-- THE CONCEPT NODE -->
    <!-- ===== -->

    <!ELEMENT concept ( head , ( concept-name , context , description , dictionary ? ) )>
330 <!ATTLIST concept
        sbase CDATA #IMPLIED
    >

335 <!-- ===== -->
    <!-- THE ASPECT NODE -->
    <!-- ===== -->
    <!ELEMENT aspect ( head , ( applicable-to , description ) )>
    <!ATTLIST aspect
340 sbase CDATA #IMPLIED
    >

    <!-- ===== -->
345 <!-- THE TASK NODE -->
    <!-- ===== -->
    <!ELEMENT task ( head , ( context , purpose , pre-condition , post-condition ,
        description ) )>
    <!ATTLIST task
350 sbase CDATA #IMPLIED
    >

    <!-- ===== -->
355 <!-- THE PROCESS NODE -->
    <!-- ===== -->
    <!ELEMENT process ( head , ( context , purpose , description ) )>
    <!ATTLIST process
        sbase CDATA #IMPLIED
360 >

    <!-- ===== -->
    <!-- THE ENTITY IMPLEMENTATION NODE -->
    <!-- ===== -->
365 <!ELEMENT entity-implementation ( head , ( slink , purpose , description ) )>
    <!ATTLIST entity-implementation
        sbase CDATA #IMPLIED
    >
370

```

```
<!-- ===== -->
<!-- THE DESIGN PATTERN INSTANCE NODE -->
<!-- ===== -->
375 <!ELEMENT design-pattern-instance ( head, ( design-pattern-name, context ,
                                             purpose, roles, collaborations ,
                                             description ))>
    <!ATTLIST design-pattern-instance
          sbase    CDATA    #IMPLIED
380 >
```



# C

## Templates for documentation nodes

In the following sections the different templates for the node types, that are implemented in the Elucidator 2 environment is presented. The comments (marked with `<!-- comment -->`) are the guidelines that instruct the writer in using the templates.

The template for the fixed part is placed between the `<head>` tags. Since this part of the templates is duplicated in all the templates, we only print the full header in the first template in order to save space. In the remaining templates three dots ( `...` ) is placed where the fixed part template should go.

### C.1 Templates for Motivations

#### Requirement

```
<edoc>
<requirement>

<head>
5
  <!-- The topic must characterize and introduce the thematic contents of -->
  <!-- the node, not merely categorize (label) it. The topic should aim -->
  <!-- to convey positions and results. Topics are more likely to be -->
  <!-- representative and topically faithful if they are (1) constructed as -->
10 <!-- sentence fragments and (2) rewritten after composition of the node. -->
  <topic>

  </topic>

15 <!-- The abstract is supposed to be a thematic window into the -->
  <!-- contents of the node. The abstract must boil down the node body -->
  <!-- to typicalle 3-5 lines of text. It should expose rationales, results -->
  <!-- and main characteristics of content of the node. -->
```

```

<abstract>
20 </abstract>

<!-- The status of the node, being new, inprogress or finished.-->
<status><new/></status>
25 <!-- A list of representative keywords for the node.-->
<keywords><kw></kw></keywords>

<!-- The name of the author that has last change the node ( Will be -->
30 <!-- filled in automatically ).-->
<author> </author>

<created > </created >

35 <last -updated > </last -updated >
</head>

<!-- A description of the requirement.-->
40 <description >

</description >

45 <!-- A report of the parties that has specified the requirement.-->
<specified -by>

</specified -by>

50 </requirement >
</edoc>

```

## Bug report

```

<edoc>
<bug-report >

<head>
5 ...
</head>

<!-- This section of the bug report should contain information about -->
10 <!-- which areas ( classes ) of the system, that are affected by the -->
<!-- bug. It would be a good idea to make slinks, role : mentions to -->
<!-- mark these.-->
<concerning>

15 </concerning>

<!-- The description of the bug report should contain information -->
<!-- about the nature of the bug. It should also state the conditions -->
<!-- under which the error occurs, and possible suggestions for -->
20 <!-- correction of the error. Debugger traces etc. could also be -->
<!-- included here.-->
<description >

</description >
25 </bug-report >
</edoc>

```

## Improvement



```

<edoc>
<improvement>

<head>
5 ...
</head>

<!-- A overview of which specific part of the system the improvement -->
10 <!-- suggest to improve. This will typically be expressed with the help -->
<!-- of a number of slinks, role: mentions to the involved entities. -->
<concerning>

</concerning>
15 <!-- The actual description of the suggested improvement. -->
<description>

</description>
20 </improvement>
</edoc>

```

## C.2 Templates for Rationales

### Rationale

```

<edoc>
<rationale>

<head>
5 ...
</head>

<!-- The forces section should contain a short description of the -->
10 <!-- driving forces and motivation for this rationale. The description -->
<!-- should contain a number of dlinks, role:premise to motivation -->
<!-- nodes, describing the motivations in detail, and/or a number -->
<!-- slinks, role:premise to special parts of the system that -->
<!-- motivates this rationale. Finally the section should contain -->
15 <!-- argumentation for the selected and declined solutions. -->
<forces>

</forces>

20 <!-- The solution part presents the selected solution. If alternative -->
<!-- and/or declined solutions exists they are mentioned to. This part -->
<!-- will typically contain dlinks, role:slects and role: decline to -->
<!-- detailed documentation of the selected and/or declined -->
<!-- solutions. -->
25 <solution>

</solution>

<!-- A discussion of consequences of the select/declined solutions -->
30 <!-- including personal subjective assessments. -->
<discussion>

</discussion>

35 </rationale>
</edoc>

```

## Change description

```

<edoc>
<change-description>

<head>
5 ...
</head>

<!-- The forces section should contain a short description of the -->
10 <!-- driving forces and motivation for this change description. The -->
<!-- description should contain a number of dlinks, role:premise to -->
<!-- motivation nodes, describing the motivations in detail, and/or a -->
<!-- number slinks, role:premise to special parts of the system that -->
<!-- motivates this change description. Finally the section should -->
15 <!-- contain argumentation for the selected, declined and deprecated -->
<!-- solutions.-->
<forces>

</forces>
20
<!-- The solution part presents the selected solution, and the -->
<!-- solution that where deprecated by the change. If alternative -->
<!-- and/or declined solutions exists they are mentioned to. This part -->
<!-- will typically contain dlinks, role:introduces, role:depricates, -->
25 <!-- role:declines to detailed documentation of the selected, -->
<!-- depricated and/or declined solutions.-->
<solution>

</solution>
30
<!-- A discussion of consequences of the select/declined solutions -->
<!-- including personal subjective assessments.-->
<discussion>

35 </discussion>

</change-description>
</edoc>

```

## C.3 Templates for Solution descriptions

### Concept

```

<edoc>
<concept>

<head>
5 ...
</head>

<!-- The name of the concept. May not contain any markup. -->
10 <concept-name>

</concept-name>

<!-- If the concept resides in a special context, this context is -->
15 <!-- described here. This will typically be though a short overview of the -->
<!-- place in the system where the concept is relevant. It may be a good -->
<!-- idea to use slinks, role:mentions, role:describe or -->

```

```

<!-- role : implemented-by to provide this overview.-->
<context >
20 </context >

<!-- The actual description of the concept.-->
<description >
25 </description >

<!-- OPTIONAL. Any citations from dictionaries , that will help to -->
<!-- clarify the concept.-->
30 <dictionary >
  <p>

  </p>
  </dictionary >
35 </concept >
  </edoc >

```

## Aspect

```

<edoc >
<aspect >

<head >
5 ...
</head >

<!-- This section should state which parts of the system the aspect is -->
10 <!-- applicable to. It may also contain arguments and/or references to -->
  <!-- arguments placed in rationale nodes , as to why the aspect is -->
  <!-- applicable . References to arguments should be done with a -->
  <!-- dlink , role : selected -by -->
  <applicable -to >
15 </applicable -to >

  <!-- The actual description of the aspect. This may also contain a -->
  <!-- description of special conditions in relation to the aspect , as -->
20 <!-- well as special consequences and exceptions.-->
  <description >

  </description >
25 </aspect >
  </edoc >

```

## Design pattern instance

```

<edoc >
<design-pattern-instance >

<head >
5 ...
</head >

<!-- The name of the design pattern . May not contain any markup.-->
10 <design-pattern-name >

  </design-pattern-name >

  <!-- This part should give the reader a orientation of the part of the -->

```

```

15 <!-- system, where the design pattern instance resides.-->
   <context >

   </context >

20 <!-- A short description of why, and for what purpose, the design pattern -->
   <!-- was selected. It will typically contain a number of dlinks,-->
   <!-- role: selected-by, declined-by, introduced-by, deprecated-by and -->
   <!-- reused-by to rationale or change-description nodes in order to provide -->
   <!-- further information on why the design pattern was selected.-->
25 <!-- If necessary it may also contain a number of dlinks, role: implements -->
   <!-- to aspect or concept nodes.-->
   <purpose >

   </purpose >

30 <!-- States which classes in the system play the different roles of -->
   <!-- the design pattern. The writer should briefly outline each role -->
   <!-- and how it contributes to the patterns essence. This part may -->
   <!-- contain a number of slinks, role: implements. We recommend that the -->
35 <!-- text in the slink, be the name of the role in the design pattern.-->
   <roles >

   </roles >

40 <!-- This part should contain a report of the interaction of the -->
   <!-- system and the design pattern instance. This part may contain a -->
   <!-- number of slinks, role: describes and mentions.-->
   <collaborations >

45 </collaborations >

   <!-- A description of the instantiation (implementation) of the design -->
   <!-- pattern. This may be special adaptations and changes to the -->
   <!-- original design pattern. This part may contain a number of slinks,-->
50 <!-- role: describes and mentions.-->
   <description >

   </description >

55 </design-pattern-instance >
   </edoc >

```

## Entity implementation

```

<edoc >
<entity-implementation >

<head >
5 ...
</head >

   <!-- A link to the entity being documented here. The can be a package, -->
10 <!-- a class, a method or perhaps even a field, if the structure -->
   <!-- of the field requires special explanation.-->
   <slink ></slink >

   <!-- The purpose and responsibilities of the entity. This part may -->
15 <!-- contain a number of dlinks, role: selected-by, declined-by, -->
   <!-- introduced-by and deprecated-by to rationale or change descriptions -->
   <purpose >

   </purpose >

20 <!-- A description of the entity. This could e.g., be how it works, -->
   <!-- difficult algorithms or any special requirements that the entity -->

```

```

<!-- impose on the rest of the system.-->
<description>
25 </description>

</entity-implementation
</edoc>

Process

<edoc>
<process>

<head>
5 ...
</head>

<!-- The context section should give an overview over the part of the -->
10 <!-- system, that the process is a part of.-->
<context>

</context>

15 <!-- A report of the rationale and purpose of the process. This part -->
<!-- may contain a number of dlinks, role: selected-by, -->
<!-- role: declined-by, role: introduced-by, role: deprecated-by and -->
<!-- role: reused-by to rationale or change-description nodes. It may -->
<!-- also contain a number of dlinks, role: implements to aspect or -->
20 <!-- concept nodes.-->
<purpose>

</purpose>

25 <!-- A description of the process, the involved parts of the system -->
<!-- and how they collaborate. The process description may naturally -->
<!-- refer to a number of task with dlink, role: detailed-by or -->
<!-- role: uses.-->
<description>
30 </description>

</process>
</edoc>

```

## Task

```

<edoc>
<task>

<head>
5 ...
</head>

<!-- The context section should give an overview over the part of the -->
10 <!-- system, that the task is a part of.-->
<context>

</context>

15 <!-- A report of the rationale and purpose of the process. This part -->
<!-- may contain a number of dlinks, role: selected-by, -->
<!-- role: declined-by, role: introduced-by, role: deprecated-by and -->
<!-- role: reused-by to rationale or change-description nodes. It may -->
<!-- also contain a number of dlinks, role: implements to aspect or -->

```

```
20 <!-- concept nodes.-->
   <purpose>

   </purpose>

25 <!-- A description of the special pre conditions that the task -->
   <!-- requires.-->
   <pre-condition>

   </pre-condition>
30 <!-- A description of the conditions the are ensured after the task is -->
   <!-- performed.-->
   <post-condition>

35 </post-condition>

   <!-- A description of the task, the involved parts of the system and -->
   <!-- how the purposes of the task are accomplished. The task may refer -->
   <!-- to a process with dlink, role:detail-of or role:used-by.-->
40 <description>

   </description>

   </task>
45 </edoc>
```

## Essay

```
<edoc>
<essay>

<head>
5 ...
</head>

</essay>
10 </edoc>
```

# D

## Statistics for the StregSystem project

The following tables shows statistical for the StregSystem experiment. The data is gathered by querying our Data model. Table D.1 and Table D.2 on the following page shows information concerning documentation and source code entities. Table D.3 on the next page shows how often the three link types, <dlink>, <slink> and <xlink> was used and which roles they were assigned.

Documentation entity	Count
<b>Edoc Files</b>	26
<b>Documentation entities</b>	317
Catalogs	9
Documentation nodes	26
Links	134
Other entities	148
<b>Documentation to documentation relationships</b>	927
implicit relationships	897
explicit relationships	30
<b>Documentation to Source relationships</b>	75
explicit relationships	75

**Table D.1:** *Documentation entity Statistics for the StregSystem*

Source entity	Count
<b>Java files</b>	15
<b>Source entities</b>	291
Packages	2
Classes	15
Fields	21
Methods	71
Parameters	120
Variables	51
Source markers	3
<b>Source to source relationships</b>	824
containment	169
access	514
invoke	76
creation	22
throws	22
typeof	13
extends	7
returntype	1

**Table D.2:** Source entity Statistics for the StregSystem

Type/Role	<dlink>	<slink>	<xlink>	<b>Total</b>
Premise	4			<b>4</b>
Selects	4			<b>4</b>
Introduces	2	6		<b>8</b>
Declines	1		4	<b>5</b>
Deprecates	2	2		<b>4</b>
Described-by	(1)6		4	(1) <b>10</b>
Describes		(7)48		(7) <b>48</b>
Mentions	(1)6	(4)19	12	(5) <b>37</b>
Detail-of	3			<b>3</b>
Detailed-by	4			<b>4</b>
Uses			7	<b>7</b>
<b>Total</b>	(2) <b>32</b>	(11) <b>75</b>	<b>27</b>	(13) <b>134</b>

**Table D.3:** Link type and role statistics for the StregSystem. Numbers in parenthesis are number of links that had an invalid destination



# Bibliography

- [Beck, 1999] Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison Wesley Publishing Company.
- [Bray et al., 1998] Bray, T., Paoli, J., and Sperberg-McQueen, C. M. (1998). Extensible markup language (xml) 1.0. <http://www.w3c.org/XML>.
- [Brown and Childs, 1990] Brown, M. and Childs, B. (1990). An interactive environment for literate programming. In *Structured Programming*, volume 11, pages 11–25. Springer-Verlag, New York Inc., Computer Science Department, University of Alabama, Box 870290, Tuscaloosa, AL 35487-0290.
- [Brown and Czejdo, 1990] Brown, M. and Czejdo, B. (1990). A hypertext for literate programming. In Akl, S. G., Fiala, F., and Koczkodaj, W. W., editors, *Advances in Computing and Information*, Department of Computer Science, University of Alabama, Box 870290, Tuscaloosa, AL 35487-0290.
- [Chen et al., 1995] Chen, Y.-F. R., Fowler, G. S., Koutsofios, E., and Wallach, R. S. (1995). Ciao: A graphical navigator for software and document repositories. In *International Conference on Software Maintenance, 1995. Proceedings.*, pages 66 – 75. AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill NJ 07974.
- [Christensen et al., 2000] Christensen, C. N., Andersen, M. R., Kumar, V., Staun-Pedersen, S., and Sørensen, K. L. (2000). The elucidator — for java. Technical report, Aalborg University, Department of Computer Science. Can be found via: <http://dopu.cs.auc.dk>.
- [Conklin, 1987] Conklin, J. (1987). A survey of hypertext. In *ACM Hypertext on Hypertext*. ACM. Available online through: <http://www.ai.univie.ac.at/%7Epaolo/lva/vu-htmm1999/>.
- [Conklin and Begeman, 1987] Conklin, J. and Begeman, M. L. (1987). gibis: a hypertext tool for team design deliberation. In *Proceeding of the ACM conference on Hypertext*, pages 247–251.
- [Davidson and Coward, 1999] Davidson, J. D. and Coward, D. (1999). Java servlet api specification, version 2.2. <http://java.sun.com/products/servlet/download.html#specs>.

- [Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol — http/1.1. <http://www.w3.org/Protocols/>.
- [Fischer and Jensen, 1990] Fischer, L. P. and Jensen, F. (1990). Literate programming in an industrial environment. Unpublished.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.
- [Horn, 1992] Horn, R. E. (1992). Commentary on the nurnberg funnel. *The Journal of Computer Documentation*, 16(1):8.
- [Horn, 1999] Horn, R. E. (1999). Two approaches to modularity: Comparing the stop approach with structured writing. *The Journal of Computer Documentation*, 23(3):7.
- [Kinnucan, 1999] Kinnucan, P. (1999). Java development environment for emacs. <http://sunsite.auc.dk/jde/>.
- [Knuth, 1984] Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111.
- [Korn et al., 1999] Korn, J., Chen, Y.-F. R., and Koutsofios, E. (1999). Chava: Reverse engineering and tracking of java applets. pages 314 – 325.
- [Mathiassen et al., 1997] Mathiassen, L., Munk-Madsen, A., Nielsen, P. A., and Stage, J. (1997). *Objekt orienteret analyse og design*. Marko Publishing.
- [Merriam-Webster, 1997] Merriam-Webster (1997). *Merriam-Webster's Collegiate Dictionary*. Merriam-Webster, Incorporated, Springfield, Massachusetts, U.S.A., 10th edition.
- [Naur, 1985] Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, (15):253–261. Also appears in the book “Computing: A Human Activity”. Addison-Wesley Publishing Company.
- [Nelson, 1999] Nelson, T. (1999). Xanalogical media: Needed now more than ever. <http://www.sfc.keio.ac.jp/~ted/XU/XuSum99.html>. This article is still being revised. It has been tentatively accepted for the ACM Computing Surveys hypertext issue.
- [Nørmark, 2000a] Nørmark, K. (2000a). An elucidative programming environment for scheme. In *Proceedings of the Ninth Nordic Workshop on Programming Environment Research, Norway*.
- [Nørmark, 2000b] Nørmark, K. (2000b). Requirements for an elucidative programming environment. In *Proceedings of the 8th International Workshop on Program Comprehension, Ireland*.

- [Nørmark and Østerbye, 1995] Nørmark, K. and Østerbye, K. (1995). Rich hypertext: a foundation for improved interaction techniques. *International Journal on Human-Computer Studies*, (43):301–321.
- [Nowack, 2000] Nowack, P. (2000). *Structures and Interactions — Characterizing Object-Oriented Software Architecture*. PhD thesis, The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark, Odense University, Campusvej 55, DK - 5230 Odense M, Denmark.
- [Østerbye, 1995] Østerbye, K. (1995). Literate smalltalk programming using hypertext. *IEEE Transactions on Software Engineering*, 21:138 – 145.
- [Parnas and Clements, 1986] Parnas, D. L. and Clements, P. C. (1986). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257.
- [Pressman, 1997] Pressman, R. S. (1997). *Software Engineering. A Practitioner's Approach*. McGraw-Hill Series in Computer Science. McGraw-Hill, 4 edition.
- [Rüping, 1998] Rüping, A. (1998). The structure and layout of technical documents. <http://www.coldewey.com/europlop98/Program/workshop3.html#Rueping1>.
- [Sametinger, 1992] Sametinger, J. (1992). *DOgMA: A tool for the documentation and maintenance of software systems*. Verband der wissenschaftlichen Gesellschaften Österreichs (VWGÖ).
- [Sametinger, 1994] Sametinger, J. (1994). Object-oriented documentation. *Journal of Computer Documentation*, 18(1):3–14.
- [Sanvad et al., 2000] Sanvad, E., Østerbye, K., Madsen, O. L., Bjerring, C., Kammeyer, O., Skov, S. H., Hansen, F., and Hansen, F. O. (2000). Documentation of oo systems and frameworks. COT/2-42-V1.2, Unpublished.
- [Tracey et al., 1999] Tracey, J. R., Rugh, D. E., and Starkey, W. S. (1999). Sequential thematic organization of publications (stop). *The Journal of Computer Documentation*, 23(3):7.



# Index

- A**
  - abstract..... 53
  - abstractor ..... **10**, 42
- B**
  - Beck, Kent..... 2
  - browser ..... **10**, 41
- C**
  - change description..... 49
- D**
  - data model ..... **11**, 42
  - deliberative categories ..... **23**, 46, 51
  - developer ..... 20
  - disco..... 1972
  - documentation node ..... **26**, 46
    - fixed part ..... **28**, 53
    - free part ..... **28**, 55
    - internal structure ..... **28**, 52, 76
- E**
  - editor ..... **9**, 41
  - EDoc language..... 40
  - Elucidative Programming ..... 4
  - Elucidative environment ..... 41
  - Elucidator tool..... **42**, 82
  - entity..... 11, 40, **45**
    - entity/relationship model ..... 11
    - name standard ..... 12
- F**
  - free structure ..... **14**, 26
- G**
  - generator..... **10**, 42
  - gIBIS ..... 36
  - guideline ..... 53
- H**
  - history ..... 6, 31, 49, 82
- holist..... 22
- hypertext ..... 26
- I**
  - internal documentation..... 1
- J**
  - Java..... 40
- K**
  - keyword ..... **54**, 71
  - Knuth, Donald E..... 3
- L**
  - link..... **29**, 56
    - anchor ..... 29
    - explicit..... **30**, 59
    - implicit ..... **30**, 60
    - organizational..... **31**, 59
    - referential ..... **31**, 59
    - role..... **29**, 57, 76
    - type ..... 56
    - validation ..... 77
  - Literate Programming ..... **3**, 35
  - long essay ..... 14
  - loose ends ..... 15
- M**
  - motivation ..... **23**, 37, 47
  - MRS-model ..... **23**, 27, 75, 80, 81
    - deliberative categories..... 23
    - link ..... 29
    - motivation ..... 23
    - rationale ..... 24
    - relationship ..... 29
    - solution description..... 24
    - sub-category ..... 46
- N**
  - Naur, Peter..... 2
  - navigation ..... 6, 23, 32, 62

- context view ..... 66
  - disorientation ..... **32**, 62
  - entity index view ..... **70**, 78
  - global ..... 68
  - hierarchal index view ..... 78
  - hierarchic index view ..... **69**
  - index view ..... 68
  - local ..... 64
  - navigation menu ..... 64, 79
  - neighborhood ..... 66
  - subject index view ..... 71
  - navigation window ..... **10**, 15
  - node type ..... 46
  - Nørmark, Kurt ..... 3
- P**
- parallel hypertext ..... **14**, 79
  - post documentation ..... 15
  - proximity ..... **4**, 35
- Q**
- quality of software ..... 1
- R**
- rationale ..... 6, 20, **24**, 37, 48
  - reader ..... 5, 14, **21**, 44, 78
  - relationship ..... **29**, 55
- S**
- serialist ..... 22
  - software development ..... **17**
    - analysis ..... 17
    - change ..... 19
    - creation ..... 18
    - design ..... 17
    - examination ..... 18
    - implementation ..... 17
    - OOA&D ..... 34, 90
    - reuse ..... 19
  - solution description ..... 20, **24**, 37, 49
  - STOP method ..... 25
  - StregSystem ..... 73
  - structure ..... 6, 23, 26
  - Structured Writing ..... 25
- T**
- templates ..... **52**, 76
  - thematic catalog ..... 51
  - tool support ..... **14**, 30, 82
    - abstraction ..... 14
    - direct navigation ..... 14
    - incremental update ..... 15
    - link insertion ..... 14
  - topic ..... 53
- V**
- view ..... 33
    - context ..... 33
    - index ..... 33
- W**
- writer ..... 6, 14, **20**, 44, 75, 89