Elucidative Programming in Java

Kurt Nørmark, Max Andersen, Claus Christensen,
Vathanan Kumar, Søren Staun-Pedersen, and Kristian Sørensen
Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg
Denmark
Email: normark@cs.auc.dk

Abstract

In this paper we describe the ideas of Elucidative Programming. With Elucidative Programming we are primarily concerned with documentation of program understanding, and presentation of such understanding on the World Wide Web. In a broader perspective, we are interested in support of any technical writing which needs to address source program constituents. We claim that Elucidative Programming may be helpful for these purposes, mainly due to a documentation model based on relations between places in the documentation and program constituents. The main section in the paper introduces a concrete Elucidator tool for Java. We use this tool as outset for a discussion of collaboration and teamwork centered in WWW based program presentations.

1 Introduction

Programs embody a wealth of knowledge captured by domain specialists, designers, and programmers. This knowledge is to be used throughout the lifetime of the software. Unfortunately, we are not good at preserving this information for the people who need it at a later point in time. Some information is not captured at all; Other information is documented early in the development phase, but not kept up-to-date in relation to the implemented software; And yet other information is represented in low level form, and perhaps scattered around in thousands of program source lines. We see this as a serious software quality problem.

We can envision a changed state of affairs in which we consistently preserve the program understanding in various kinds of written documentation. Such documentation can be seen as an investment which is likely to pay off during the lifetime of the software. More important, we see the documentation as a much needed contribution to improving the quality of the software that we produce. Tools and concrete mechanisms are not likely to cause significant changes in this area. We need to address working habits and not least the programmer's education in order to make progress. Nevertheless, the presence of good examples and attractive ways of communicating the program knowledge may be a motivational factor. This paper is intended to make a contribution in this direction.

Considering Java programs we have experienced a success with respect to documentation of the external interfaces of Java classes using the JavaDoc tool [6]. As part of our agenda we want to find out if internal program documentation can be supported by similar means. We focus on the following key aspects of the internal documentation problem:

- The representation of relations between program fragments and the program explanation.

- The presentation of internal program documentation in standard WWW browsers.

- The support of the programmers who write the documentation during the development process.

In this paper we will elaborate on a tool for Java - a Java Elucidator - which makes it possible to describe and discuss pieces of Java programs. The essential mechanism of the tool makes it possible to relate selected regions in the documentation with constituents of the program. We present the documentation and Java programs in a two-framed WWW browser which provides for mutual navigation in between the documentation and the program. The HTML pages are produced dynamically by the Java Elucidator tool.

There are many kinds of program-related descriptions and discussions. In the area of program documentation we will distinguish between the following three kinds:

1. maintenance documentation
2. proactive mental understanding
3. process documentation

The first kind of documentation is targeted at the maintainer. The purpose of such documentation is to convey sufficient understanding from the original developers to the people who will eventually make modifications to a program. Such documentation will often be written late in the implementation phase, because the program in this phase is stabilizing in its final form.

The second kind of documentation is oriented towards the program author who is likely also to be the author of the documentation. With this documentation the programmer writes to himself or herself, often being in the role of a program architect or designer. The purpose of such writing is to describe problems and solutions in the hope that *written formulations* provide for an improved starting point of the programming efforts. Such documentation tends to be written before the implementation starts, or side by side with the programming.

The third kind of documentation plays a role during the program development process as a means to keep the practical activities on track. Such documentation is often found in form of diaries and logs.

As a common element of all mentioned kinds of documentation we see a need to address program details in descriptions and discussions. The elucidative tools have been designed to alleviate exactly this need. However, it remains a challenge to come up with good guidelines for the elaboration of each kind of documentation; The discussion of this challenge is beyond the scope of this paper.

In Section 2 we will describe the background of and the motivation behind our work, in particular the impact of Literate Programming. In Section 3 we will discuss our notion of Elucidative Programming, mainly as a contrast to an existing set of Literate Programming tools. In Section 4 the Java Elucidator is described and exemplified. Finally, in Section 5 we will discuss the perspectives of elucidative program documentation with special emphasis on the teamwork and collaboration potential of Elucidative Programming. The paper is closed by the conclusions in which we also describe the current status of our work.

2 Background and Motivation

Our main inspiration is the work on *Literate Programming* which was initiated by Donald Knuth in the early eighties [9]. A literate program is structured according to the needs of the program reader, and not according to the rules of a programming language or to other software engineering concerns. Each piece of program is organized and physically represented in a section of the documentation. Program pieces can refer to each other, much in the style of syntax-directed program modularization [3].

Knuth made a system for Literate Programming called the WEB system [10]. During the eighties and nineties a family of Literate Programming systems appeared [17; 4; 7]. Using these, the program pieces are assembled to a whole program by a tool called *tangle*. Sections of the documentation are organized in an ordinary hierarchy of chapters and sections. The *weave* tool formats the literate program as a document with embedded program fragments which refer to each other as explained above. The weave tool of most WEB-based systems are oriented towards production of book-like volumes on sheets of paper.

A literate program can be seen as a traditional program inversed with respect to its comments. Thus, in a literate program, the program fragments are annotations of the explanations, whereas in a an ordinary program the explanations (comments) are annotations of the program. The physical *proximity* between the documentation and the program fragments is an important characteristic of literate programming.

Although the proximity mentioned above is important for literate programming, the notion of proximity can also be blamed as a serious problem for the success of Literate Programming. It has been the experience of the authors from numerous (but minor) student projects using Literate Programming that it is very difficult to convince a programmer to split the source program in many fragments, and to organize these in a shell of documentation. In projects, where the students are encouraged to use Literate Programming, the students usually develop programs in a conventional way. The literate documentation and the necessary fragmentation of the program are done 'as the last thing on the agenda', and it never becomes a natural organization of the total program knowledge, as possessed by the group of students.

As a consequence of these observations we hypothesize that a variant of Literate Programming, which leaves the program unaffected in source files and directories, will have a better chance of success. In the next section we will introduce the concept of Elucidative Programming, and we will explain an alternative model for organization of explanations and programs.

3 The Elucidative Programming Model

The Elucidative Programming model is characterized by *program entities* and *documentation entities* related by a number of binary relations. The program entities represent the natural, named abstractions found in the supported programming language. The sections and subsections in the documentation make up the documentation entities. In the Elucidative Programming model we do not deal with specially named program fragments as required by most Literate Programming systems.

The most important relation connects particular places in the documentation entities with program entities. In the Elucidative Programming model we do not rely on any kind of containment for representation of the connections between the documentation and the program.

The documentation is written in an XML-based language called *Edoc*. The Edoc language provides slink, dlink, and xlink elements (tags) for links to program source entities, documentation entities, and external pages respectively. An href attribute of the slink tag makes use of a *naming scheme* that allows the author of the documentation to address Java program constructs in an unambiguous and context independent way. Typographical markup is done by means of HTML tags. There are no requirements to the Java program files. If, however, we want to refer to specific places or regions within a program entity, it can be done by means of so-called *source markers*. Source markers are represented by XML markup in Java comments.

With this outset the road is open for a *hypertextual organization* of program and documentation. By this we refer to a model with relatively fine grained nodes holding either documentation or program pieces, and relations represented as anchored hyperlinks. This organization has been explored in some of our earlier research [15; 16], and in that work we found it problematic because of the extensive linking efforts needed to create a valid web. The Elucidative Programming model therefore insists on a very coarse grained node structure. On the documentation side there is only one large node (or at least very few such nodes) with internal hierarchical, sectional structure. On the program side, the nodes correspond to the usual source files which are organized in natural directories.

Our main linking challenge is the connection of program related descriptions in the documentation with the program entities being described, such as methods and classes. It is of paramount importance for our approach to come up with flexible means for definition of such connections, since lots of them will exist.

From a user interface point of view, the central idea is to present the documentation and the program side by side in two relative large frames in an Internet browser, see Figure 2. Mutual navigation in between the two frames can take place. By following a link anchored in one frame the other frame scrolls to the destination anchor of the link. These characteristics of the elucidative user interface provides for a new kind of proximity between explanations and programs which we call *navigational proximity*.

Navigational proximity is weaker than the physical proximity between documentation and program in Literate Programming. However, the navigational proximity makes it possible to describe several program entities in one section of the documentation. In addition, a single program entity may be discussed in several different sections of the documentation. Thus, the user interface ideas in Elucidative Programming mediates a many-to-many correspondence between documentation entities and program entities. This is a powerful organization, but it is also an organization that will become a challenge for the maintainers of an elucidative program.

A variety of navigational possibilities are supported by a *navigation window*. In contrast to the two large frames the navigation window is only visible when a documentation or program entity is explicitly selected in one of the two frames. One of the many functionalities of this window is to present the many-to-many correspondence between the documentation and program entities. This is done by listing all outgoing and incoming relationships to the selected entity. This means that the Navigation window can be used to show all the places in the documentation where the selected entity is documented. An example of this is shown in Figure 1.

Finally we introduce the concept of a *documentation bundle* in order to keep the parts of an elucidative program together. The documentation bundle defines the name and location of program source files and the documentation file. In the current version of the Java Elucidator the documentation is represented in a single file. The source files of the documentation bundle are defined by stating a directory, which is traversed recursively to identify all Java files.

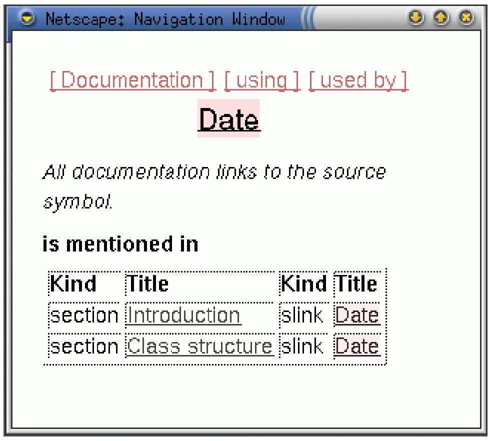

Figure 1: *An example of a navigation window.*

4 The Java Elucidator

The Java Elucidator was created in the first part of a master thesis project [1] by the five last authors of this paper, and supervised by the first author. The Java Elucidator follows the ideas and requirements, as laid out in [14] and as implemented in an Elucidator for Scheme [13]. In the sections below we first present an example of an elucidative program, which is discussed relative to the possibilities of the elucidative browser. Following that some underlying design and implementation issues are discussed.

4.1 An example

In order to be concrete an example which illustrates the approach will be discussed. The example is concerned with the development of a program that can decode the number of seconds elapsed since a particular point in time (midnight of January 1, 1970) to a record of year, month, day, hour, minute, and second. The same example has been used to illustrate the Scheme Elucidator in [13]. Figure 2 shows a snapshot of a browser which presents the result produced by the Java Elucidator. For a better presentation, please consult the online version of the example at

`http://dopu.cs.auc.dk/version1/elucidator`

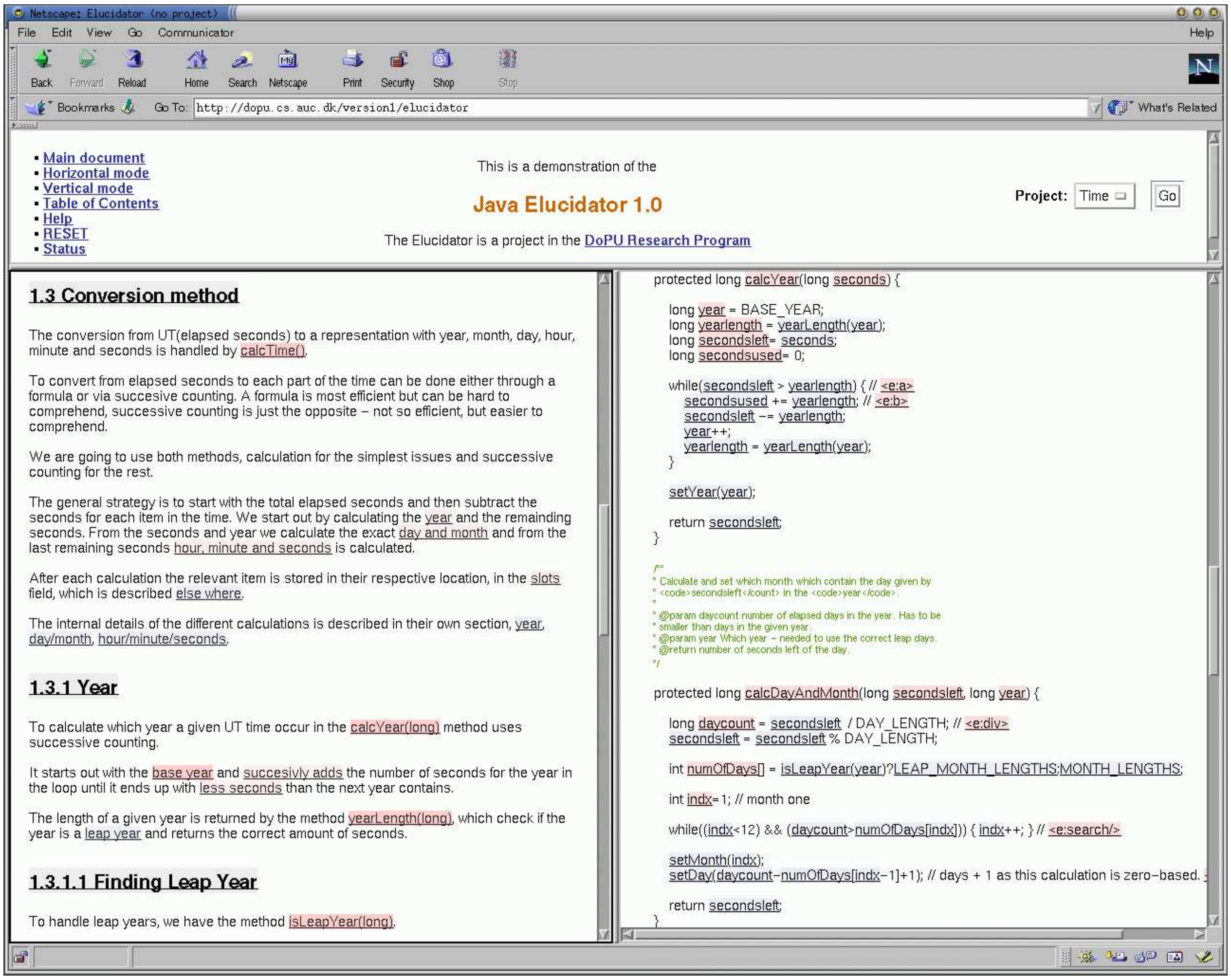

Figure 2: *Screen capture of the browser in the Java Elucidator.*

The author of this elucidative demo program first briefly introduces the problem, and a reference is made to the native solution of the problem in the Java class libraries. Next a number of organizational and algorithmic concerns are brought up. Throughout the documentation a number of references are made to methods and classes in the program. At some points there are also references to particular details in a program, via the use of a source markers which denotes positions or regions in the source program. The natural steps in the problem solving process are discussed. The essay ends with a reference to a testing class and JavaDoc documentation of the Date class.

The documentation shown in the left hand frame of the browser in Figure 2 presents the documentation, which is structured in sections, subsections, etc. Thus, the documentation is a conventionally structured textual document. The most noteworthy element of the documentation is the numerous connections between the documentation and the program. These are all represented as links which are anchored in selected regions in the documentation text. These anchors typically correspond to the names of abstractions in the program. At some places in the documentation a particular program unit is explained or mentioned. At other places there are anchored links

Anchors in the documentation		Anchors in the program	
Red	Links to an *explained* program entity	Red	Links a defining name to the navigation window
Light red	Links to a *mentioned* program entity	Light blue	Links an applied name to a defining program entity
Light blue	Links to another documentation entity		
Light green	Links to an external page		

Figure 3: *Coloring conventions in the documentation and program presentations.*

which serve as cross references within the documentation, or as references to external WWW pages. Figure 3 summarizes the graphical conventions used in the presentation of the documentation and the programs.

The program shown in the right hand frame represents a Java source program file. All applied names are linked to their definitions provided that these are part of the current documentation bundle. The defined names are linked to a navigation window which, among other possibilities, allows navigation to the places in the documentation where the definition is explained or mentioned (se section 3).

It should be kept in mind that the referred example remains a very brief demonstration of the elucidative approach. As mentioned in the introduction, a variety of different documentation styles and purposes exists. It is not important to categorize the example in relation to these styles. It *is* important, however, to emphasize the possibilities of relating descriptions and discussions to a variety of program entities and to details in these. Any technical writing with such needs is hypothesized to benefit from the Elucidative Programming style.

4.2 Tool design and implementation

The Java elucidative programming environment has two tangible tools: The editor and the browser. The browser is a standard Internet browser. The editor is used to edit both Java source programs and documentation files in the Edoc language. The editor is supposed to make

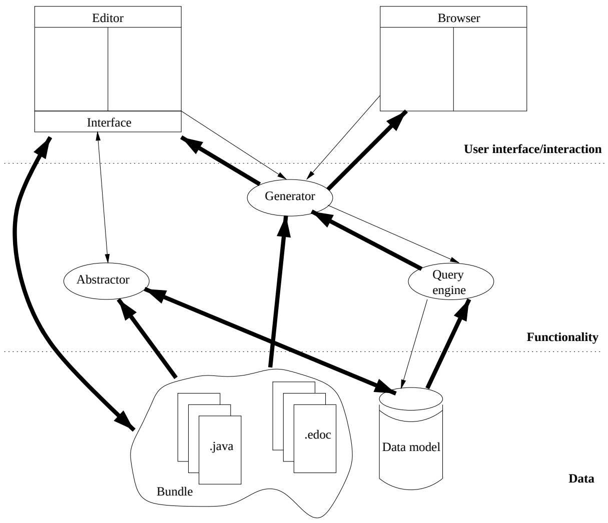

Figure 4: *Design overview of the Java Elucidator.*

it flexible and secure to define the relations between places in the documentation and units of the program. We use a customized Emacs editor for elucidative Java programming. Alternatively, we can imagine that commercial, integrated development environments can be extended to support the writing of the inter-linked documentation and program.

Both the editor and the browser tools depend on structural knowledge of the documentation and the program files in the documentation bundle. Figure 4 shows a design overview involving the editor and the browser together with the documentation bundle. The bold lines in the figure represent data flows, and the thin are message passings. In the middle layer of the figure we illustrate three important components of the Java Elucidator: The abstractor, the query engine, and the generator. The roles of these components will now be discussed.

The *abstractor* parses and extracts information from the Java classes and the Edoc documentation. The abstractor passes information to the data model, which also is illustrated in the figure. The abstractor makes up the only language dependent component of the Elucidator. Consequently, another programming language can be supported by the architecture outlined in Figure 4 by providing an abstractor for the new language.

The Data Model represents an entity/-relationship model which is inspired by the work on Ciao and Chava from AT&T [5; 11]. The Data model contains information about all named program and documentation constituents together with their mutual relations. The Java Elucidator uses a relational database for its Data Model. Java packages, classes, fields, and methods are examples of extracted entities from the program. Examples of program relations include the containment of a method in a class, and the 'invoke' relationship between methods. The documentation is represented as chapter and section entities in the database.

Both the editor and the browser need information from the data model. This information is extracted by the *query engine* and aggregated by the *generator*. The editor needs the information to support flexible and secure definition of the relations between the documentation and the program. The browser needs the information to render both the documentation and the classes together with the anchored links in between them. The generator runs as a Java servlet on the WWW-server. Thus, the generator is activated each time an HTML page is requested from the browser. As such, the Java Elucidator produces the browser content on demand as opposed to producing a set of static HTML pages once and for all. The production relies on standard WWW technologies including Extensible Markup Language (XML), Extensible Stylesheet Language (XSL), and Cascading Stylesheets (CSS).

5 Perspectives

As already discussed earlier in this paper we envision that the Java Elucidator will make it easier and more attractive to produce technical documents that addresses classes, methods, and detailed regions or positions in a Java program. We see a broad need for such documents throughout the program development process, for instance as design descriptions, maintenance documentation, code reviews, and diaries. It may be necessary to develop slightly different variants of the elucidator tools for these various purposes, but the elucidative mechanisms that tie the documentation and the program together will make up the kernel of all such tools.

The Elucidative Programming model is especially useful for documentation of *transverse themes*, such as *design patterns* which involve several constituents of a Java programs. This is an important observation for documentation of object-oriented programs because of the logical fragmentation of such programs. It may be noticed that Literate Programming tools cannot match Elucidative Programming with respect to documentation of transverse themes. The Literate Programming systems are first and foremost strong in explaining a single and isolated program aspect.

The online WWW availability of the documentation as well as the program source code (Java classes) seem to be very attractive in situations where the development involves a geographically distributed team of programmers. A *reflected program* with carefully related descriptions and discussions may be a very useful online resource for a group of cooperating developers.

The vision described above is only a minor contribution to a collaborative development environment for a geographically distributed team of developers. Cooperative development tools and repositories are likely to form the kernel of a full-fledged environment. We believe, however, that the ideas outlined above may be an attractive beginning in order to explore the potential of WWW mediation of programs and program related documents.

6 Status and conclusions

In this paper we have described a variant of Literate Programming which we call Elucidative Programming, and we have applied the ideas of Elucidative Programming on Java. Elucidative Programming remedies some weaknesses in Literate Programming tools which we have identified in a large number of medium-size student projects.

The most important contributions in our work are (1) the principle of elucidative description or discussion of a program without affecting or disturbing the source code, (2) the support of documenting transverse themes which involves several fragments of Java programs, and (3) the idea of presenting both the documentation and program source files in a conventional WWW browser.

The shift from physical proximity to navigational proximity gives both advantages and disadvantages. On the positive side, we are able to document transverse themes which otherwise will be hard to capture in the documentation. On the negative side, it is less trivial to ensure a proper updating of the documentation upon program modifications, and vice versa. The problem is amplified by the fact that a single program fragment may be addressed at multiple places in the documentation.

Besides the Java Elucidator we have developed another Emacs-based Elucidative Programming environment for Scheme [8]. As a contrast to the Java Elucidator, the Scheme Elucidator produces HTML pages on a static basis. Both tools are in local but limited use at the Computer Science Department at Aalborg University.

Recently we have developed a second version of the Java Elucidator [2]. The main extensions are threefold: First, as opposed to the existing Elucidative Environment, the documentation is divided into small hypertext nodes, with focused contents. Second, these documentation nodes are organized with respect to a documentation model called the MRS-model, which divides the documentation into three interrelated deliberative categories: Motivations, Rationales and Solution descriptions. Finally, the MRS-model is utilized in the Java Elucidative Programming Environment by the implementation of a coloring scheme and extensive navigation facilities.

As the first step of getting experience with the Java Elucidator, we plan to introduce it in the introductory object-oriented programming projects at Aalborg University. In these projects there is a massive need for the students to address various Java methods, classes and program details in project reports and in additional program documentation. After that we wish to try the tools out in an industrial setting.

As noticed in the introduction, we do not believe that the introduction of a tool is sufficient to make a real impact on the programmer's documentation habits. Consequently we find it necessary also to include program documentation topics (how and what to document in a program development process) in the introductory curriculum of Computer Science.

References

[1] Max Rydahl Andersen, Claus Nyhus Christensen, Vathanan Kumar, Søren Staun-Pedersen, and Kristian Lykkegaard Sørensen. The elucidator - for Java. Preliminary master thesis report, January 2000. Available from http://dopu.cs.auc.dk.

[2] Max Rydahl Andersen, Claus Nyhus Christensen, and Kristian Lykkegaard Sørensen. Internal documentation in an elucidative environment. Master's thesis, Aalborg University, June 2000. Available from http://dopu.cs.auc.dk.

[3] B. Möller-Pedersen B. B. Kristensen, Ole L. Madsen and K. Nygaard. *Integrated Interactive Computing Systems*, chapter Syntax-directed program modularization, pages 207–219. North-Holland, Amsterdam, 1983.

[4] Preston Briggs. Nuweb, A simple literate programming tool. Technical report, Rice University, Houston, TX, USA, 1993.

[5] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *International Conference on Software Maintenance*, pages 66–75, 1995.

[6] Lisa Friendly. The design of distributed hyperlinked programming documentation. In Sylvain Frass, Franca Garzotto, Toms Isakowitz, Jocelyne Nanard, and Marc Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France*, 1995.

[7] Andrew L. Johnson and Brad C. Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.

[8] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[9] Donald E. Knuth. Literate programming. *The Computer Journal*, May 1984.

[10] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison Wesley, 1993.

[11] Jeffrey Korn, Yih-Farn R. Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of java applets. In *The Sixth Working Conference on Reverse Engineering*, pages 314–325, 1999.

[12] Kurt Nørmark. The elucidative programming home page. http://www.cs.auc.dk/∼normark/-elucidative-programming/, 1999.

[13] Kurt Nørmark. An elucidative programming environment for Scheme. In *Proceedings of NW-PER'2000 - Nordic Workshop on Programming Environment Research*, May 2000. Available via [12].

[14] Kurt Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*. IEEE, June 2000. Available via [12].

[15] Kurt Nørmark and Kasper Østerbye. Rich hypertext: A foundation for improved interaction techniques. *International Journal of Human-Computer Studies*, (43):301–321, 1995.

[16] Kasper Østerbye and Kurt Nørmark. An interaction engine for rich hypertexts. In *European Conference on Hypermedia Technology 1994 Proceedings*, pages 167–176. ACM Press, September 1994.

[17] Ross Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, 1992.