Nordic Journal of Computing

# ELUCIDATIVE PROGRAMMING

KURT NØRMARK
*Department of Computer Science*
*Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark.*
*E-mail: normark@cs.auc.dk*

**Abstract.**
In this paper we introduce Elucidative Programming as a variant of Literate Programming. Literate Programming represents the idea of organizing a source program in an essay that documents the program understanding. An elucidative program connects textual documentation with the abstractions and the details in the source program. The documentation and the source program are defined in separate files. Using Elucidative Programming, the relations between the documentation and the units of the program are defined without use of containment. The textual documentation is intended to be technical writing about the program, and as such an elucidative program can be seen as a reflected program. Elucidative Programming allows for documentation of transverse relations among the program constituents. In addition, Elucidative Programming is oriented towards on-line presentation of documentation and programs in a WWW browser.

## 1. Introduction

Program development is based on understanding. The understanding is embodied and encoded in the program. Unfortunately, it is not easy to recover the understanding from the program. Consequently, a great amount of efforts are used to reestablish the original understanding when the program needs updatings of various kinds. The widespread interest is reverse engineering tools, which are quite dominating in the program comprehension community [32, 4] is a clear evidence of this observation.

In this paper we recommend an investment in documented program understanding. The essential understanding, present among the people who write the program, should be captured and related to the relevant program constituents. Seen in perspective of the program life time it is not economical to forget the program understanding and to recover it repeatedly via detective work, which is difficult to support by effective tools.

The ideas about documented program understanding are not new. Literate Programming has been around for more than 15 years. Literate Programming is supported by a family of systems known as WEB systems [15, 18, 5, 33, 14]. Unfortunately, Literate Programming has not caused much impact on everyday software development practice. Part of the reason is that Literate Programming is extreme in several directions:

Received June 22, 2000.

- It is based on the ideas of breaking the program into fragments that are contained physically in the document which represents the program understanding. Thus, the program "lives in" the documentation. The concept of a source file (which is familiar to most programmers) does not exists in the literate programming paradigm.
- It mixes fragments of a text formatting language and fragments of a programming language in one "ugly" and monolithic file, the value of which is low in the development situation.
- It aims at documentation with literate value which can serve as technical literature in the same way as scholarly papers.

It is our hypothesis that Literate Programming is beyond reach of the average programmer. The ambition of literate programming is too high, the program artifacts are too far from mainstream, and current programming environments will suffer too much if adapted to the 'literate ideas' in a WEB-like elaboration.

With Elucidative Programming we keep the basic idea of documented program understanding from Literate Programming. However, we re-orient the approach in the following ways:

- In Elucidative Programming, the source program is left intact without embedded or surrounding documentation.
- The program understanding is described in a document which is firmly related to named constituents in the source program.
- The physical proximity between documentation and program, which is characteristic for Literate Programming, is traded for a navigational proximity in Elucidative Programming. This allows for explanation of aspects that crosses the boundaries between several program modules, such as design patterns in object-oriented programs.

Elucidative Programming is intended as as an alternative to Literate Programming in the area of internal program documentation and explanation. We realize, however, that the ideas behind the approach can be used in most situations where there is a need to *write about a program*. This includes program tutorials, program reviews, and student reports which need to address various program details.

In this paper we will discuss the tools involved in an elucidative programming environment. This includes both an elucidative programming environment for Scheme [21] and for Java [23]. In that respect, we are concerned with two overall goals. First, we want to orient the tools toward the medium of the Internet, WWW, and HTML. We have witnessed the great success of documenting class library interfaces using this medium in the Java Development Kit [6]. During our research we will like to find out whether program documentation of more internal nature can be made by similar means. Second, we want to integrate the support of Elucidative Programming in an existing editing environment. With this goal programmers

(1) The internal documentation must be oriented towards current and future developers of the program.

(2) The internal documentation is intended to address explanation which serves to maintain the program understanding and to clarify the thoughts behind the program.

(3) The program source file must be intact, without embedded or surrounding documentation.

(4) The programmer must experience support of the program explanation task in the program editing tool.

(5) The program "chunking structure" follows the main abstractions supported by the programming language.

(6) The documented program must be available in an attractive, on-line representation suitable for exposition in an Internet browser.

**Fig. 1.1**: Requirements for an elucidative programming environment.

can continue "programming as usual", but now in a *documentation enabled environment*.

We are aware of several obstacles that need to be overcome in order for Elucidative Programming to succeed in the area of internal program documentation. The obstacles rely on positive answers to the following questions:

- Is it realistic to expect that programmers retain their program understanding in a free style story or essay?
- Is it possible to keep the documented program understanding up-to-date and valuable during the maintenance phase of the program.

We are not presenting substantial answers to these questions in the present paper. However, these questions are central in our ongoing research, and future papers from our group are expected to address these important issues.

In section 2 we will introduce the ideas and the concepts of Elucidative Programming. In section 3 we present a concrete example of an elucidative program. A discussion of the tools in an elucidative programming environment follows in section 4. The paper ends with a description of related work, status of the research, and conclusions.

## 2. Elucidative Programming Concepts

To *elucidate* is to throw light on something complex and to make clear or plain, especially by explanation. We introduced the idea of Elucidative Programming in an earlier paper [22]. In that paper we discussed Elucidative Programming and Literate Programming in relation to each other, and we came up with six requirements for an elucidative programming environment. These requirements are summarized in figure 1.1. The paper *An Elucidative*

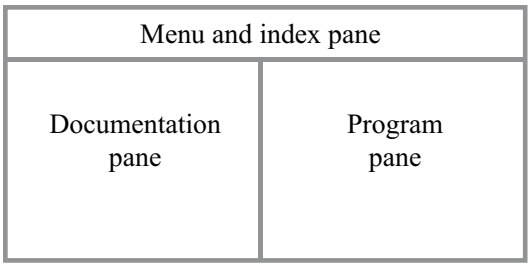


**Fig. 2.2**: *The layout of panes in an Elucidator.*

*Programming Environment for Scheme* (which is an earlier version of this paper) introduces a set of tools for Scheme - a programming language in the Lisp family [21]. Similarly, the paper *Elucidative Programming in Java* describes facilities for Elucidative Programming in Java [23]. The Java Elucidator was designed and implemented by a group of five master students [1] based on the preliminary experiences with the Scheme Elucidator.

### *2.1 User interface*

From a user interface point of view, the central idea in Elucidative Programming is to present the program and the documented program understanding as hypertext in two relative large panes of a window, see figure 2.2. In that way the documentation may be presented in a pane on the left half part (or top) of the screen and the program in the right half part (or bottom) of the screen. A concrete example can be seen in figure 3.4, which will be discussed in section 3 of this paper.

The proximity of documentation and program gained by this setup of the user interface forms an interesting contrast to the WEB systems that supports Literate Programming. The proximity supported by the elucidative user interface will be called *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. This is useful for discussion of transverse program themes, such as design patterns in object-oriented programs. Furthermore, 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.

### *2.2 Hypertext aspects*

Seen as hypertext, the nodes of an Elucidator are very coarse grained. The entire documentation is a single node in which sectional units are embed-

ded into each other. Similarly, each source program file can be seen as a node where the units of the programs are composed and embedded according to the rules of the programming language. In that respect, Elucidative Programming runs counter to other hypertext-based programming environments [25, 24] where more fragmented models seem to dominate. The hypertext links are derived from relations among program and documentation entities. This is explained in more details below.

### *2.3 Central model*

From a modelling point of view, the program and the documentation are broken into *entities*. At the program side the entities are the overall building blocks of the program (named abstractions) such as classes, procedures, and functions. At the documentation sides the entities are sections and subsections of the program explanation. As an integral part of the entity concept there must exist a *naming scheme* which allows the documentation to address specific constituents of a program. In a more general sense, the naming scheme provides for definition of relations between documentation entities and program entities.

In the Scheme environment, the most important program entities are top-level `define` forms. The naming scheme in the Scheme Elucidator allows addressing of top-level abstraction in a specific source file such as

```
file$function
```

In the Java environment, the naming scheme is more elaborate. A name of a Java program entity includes information about package, class, method, and types of formal parameters of methods (in order to distinguish overloaded methods from each other). The following serves as an example:

```
package1.package2.class@method(class3,type)
```

In the Scheme environment the documentational entities are identified and named with specialized markup. The left hand box of figure 2.3 shows an example of two sections named 'intro-section' and 'attack-plan'. The Java environment uses XML markup instead of a specialized markup. This is illustrated in the right hand box of figure 2.3. XML markup is mainstream standardized markup today, but as can be seen it is more voluminous than the specialized ad hoc markup used for the Scheme Elucidator. Both the Scheme and Java systems use plain HTML markup for typographical details.

Program and documentation entities can be connected to each other by means of a few natural *relations*, all of which are binary. If we assume that P (together with P1 and P2) are program entities and D (together with D1 and D2) are documentation entities we can describe the meanings of the relations in the following way:

- **The *strong doc-prog* relation**:
  An element (D,P) in the *strong doc-prog* relation represents that the program entity P is explained in the documentation entity D.

```
.SECTION intro-section
.TITLE Introduction
.BODY
  Introductory text which may refer to
  section [attack-plan].
.END

.ENTRY attack-plan
.TITLE The plan of attack
.BODY
  Documentation describing the plan of attack which
  may explain the function {file$function}.
.END
```

```
<chapter label="intro-section">
 <title Introduction </title>
  Introductory text which may refer to
  section <dlink href = "attack-plan">.

<section label="attack-plan">
 <title The plan of attack </title>
  Documentation describing the plan of attack which
  may explain the method <slink role = "strong"
  href ="package.class@method()>.
 </section>
</chapter>
```

**Fig. 2.3**: *An illustration of the documentation markup used in the Scheme environment (left) and in the Java environment (right).*

- **The *weak doc-prog* relation**:
  An element (D,P) in the *weak doc-prog* relation represents that the program entity P is mentioned, without being explained, in the documentation entity D.
- **The *prog-prog* relation**:
  An element (P1,P2) in the *prog-prog* relation represents that P1 uses the entity P2.
- **The *doc-doc* relation**:
  An element (D1,D2) in the *doc-doc* relation represents that the documentation in D1 relies on the documentation in D2 seen from an elucidative point of view.

In addition, there is a relation which we could call *prog-lang*, which relates an instance of language construct or standard library facility in a program to its description in a hypertext version of the language report or library manuals.

In an elucidator tool each element in one of the relations mentioned above are represented by one or more hypertext links. An element (D,P) in one of the *doc-prog* relations gives rise to two links:

- a link source anchored in a position of the documentation entity D, and destination anchored in the program entity P.
- a link somehow source anchored in the program entity P, and destination anchored in the documentation entity D.

An element in the *prog-prog* relation relates an applied name occurrence to its defining name occurrence. In order to be more precise, let us assume that (P1,P2) is an element in the *prog-prog* relation, that P2 is named N, and that (P1,P2) gives rise to a link L. L is source anchored at an applied occurrence of N in P1. The destination anchor of L is a presentation of P2, which defines N.

An element (D1,D2) in the *doc-doc* relation gives rise to a cross reference link from one place in the documentation to another section.

### *2.4 Sectional Comments*

In imperative Scheme programs it is meaningful to have expressions and commands outside any top-level procedure or function. In Java programs this is not possible. Thus, in elucidative Scheme programming there may be a need for discussing and explaining program fragments located outside of named abstractions. For this purpose we use *sectional comments* via which we can identify an arbitrary top-level locations. (More localized positions can be identified via use of source markers, see section 2.5). A sectional comment name is enclosed in double colons within a Scheme comment line. The Scheme form succeeding the sectional comment can be referred to by the qualified name like `file:sectional-comment`.

### *2.5 Source markers*

The links derived from the relations described in section 2.3 can be used to connect sections in the documentation with named abstractions in a program source file. However, in some explanations it is desirable to address finer details in the program. Of that reason we have introduced the concept of source markers. A *source marker* denotes a particular point or region in a program entity. Source markers must appear in a comment positions in order not to interfere with the syntactic rules of the programming language. In order to minimize clutter in the program comments we use a minimal two character notation `'@c'` to denote the source marker `c` in a Scheme program. In Java, XML markup like `<e:maker/>` and `<e:marker> ...</e:marker>` is used to denote a point or a region in a source program.

At the documentation side in the Scheme environment, source markers may be used when we explain the program details next to a source marker in the program. A source marker in the documentation is associated with the anchor of the link corresponding to closest preceding strong `doc-prog` relation element. A source maker in the documentation is the source anchor of a link which goes to the corresponding sourcer marker in the program, and vice versa.

### *2.6 Organizational aspects*

The concept of a *documentation bundle* is central seen from an organizational point of view. A documentation bundle keeps the documentation part and the program parts of an elucidative program together as a unit.

The editor that supports Elucidative Programming is supposed to be aware of all parts (files) in a documentation bundle. The editor awareness is used to open, save, process, and close all such files with single operations in the editor. We will describe the editing tool of the elucidative Scheme environment in more details in section 4.2.

## 3. An Example of an Elucidative Program

Before the discussion of the tools in the elucidative programming environment we will present a concrete example of an elucidative program, and we will explain the process of its development. In the paper we will illustrate an elucidative Scheme program, but as an on-line accompanying resource both a Scheme and a Java version is available.

The example is intended to illustrate the concepts introduced in the previous section. However, the reader should be aware that the example is too small to illustrate the real needs and challenges of documented program understanding "the elucidative way". Furthermore, we should be aware that Elucidative Programming is not targeted at program publication in the same way as Literate Programming, cf. the first requirement in figure 1.1. Thus, it is not really the intention to polish an elucidative program. In that respect, the example given below may be somewhat misleading.

The Elucidator of the example is available at the Internet address

```
http://www.cs.auc.dk/~normark/elucidative-programming/njc/
```

The Scheme version refers to the Java version, and vice versa. The reader is encouraged to bring the example up in a browser while reading this section of the paper.

The example is concerned with the development of a program that can decode the number of seconds elapsed since January 1, 1970, 00:00:00 to a year, month, day, hour, minute, and second. Most computers can deliver an integer representing this measure of time, and therefore the conversion forms a very useful basis for a convenient and regular handling of time in terms of a number of seconds.

Using the editor tool of the elucidative programming environment we create the documentation bundle and the underlying directory structure, which hosts an empty source file, an initial template of the documentation file, the setup file, a directory for internal files, and a directory for HTML files. This is done by the editor command `make-elucidator` (see also section 4.2). The editor prompts the user for all necessary information and creates these files and directories automatically. Next the user issues the command `setup-elucidator` which reads the documentation bundle into editor buffers, and establishes the characteristic split-view window on the documentation and the program (still empty, of course). Now the elucidative programming process can start.

First we establish a little context around the problem. We discuss how to possibly attack the problem. Two approaches are identified, and we happen to go for a mixture of them in our solution. We shift between writing a piece of documentation, and writing pieces of programs. In case a name exists in the program it can be smoothly transferred to the documentation buffer. This makes the writing about the program relatively easy and "secure". We run the Elucidator regularly and refresh the editor in order to get access to a list of known identifiers. We introduce concepts in order to write about

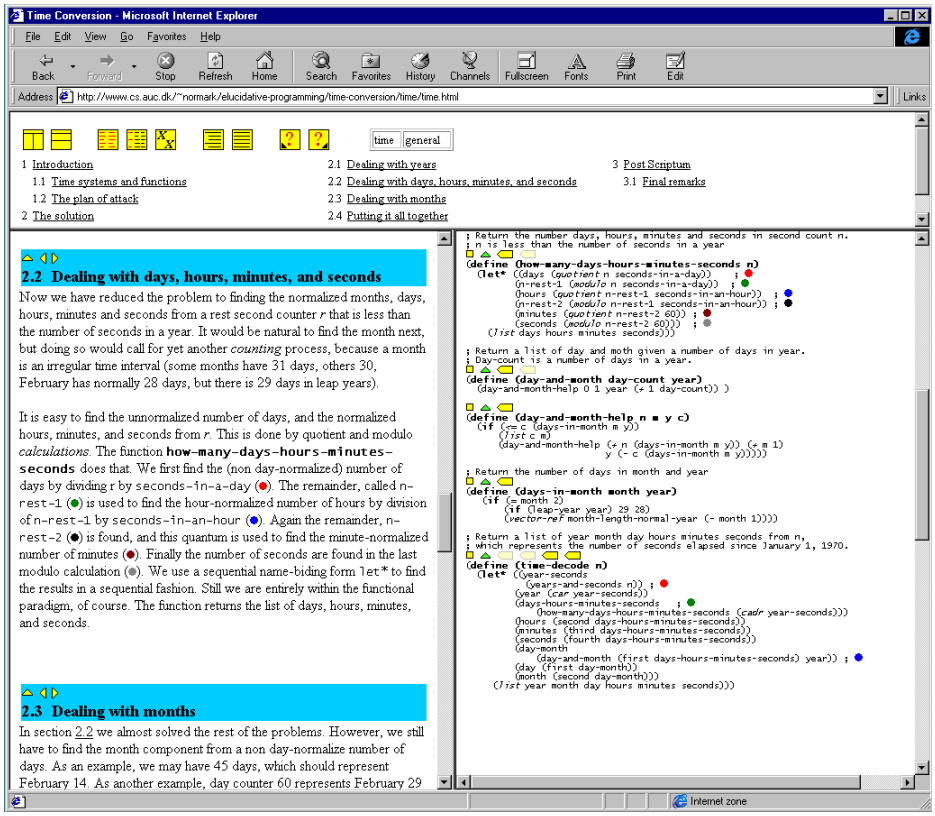


**Fig. 3.4**: *A screen shot of the browser in an elucidative Scheme environment.*

the program in a concise and precise way. This sharpens our understanding of the problem, and makes the solution easier to understand, hereby easing the development of the program. We introduce source markers for program details which we want to address in details in the explanations.

It takes longer time to produce an elucidative program than just to write a conventional source program. However, it is our firm belief that the quality of the program is improved through this process. Several author's of literate programs support this observation [16, 27]. Furthermore it should be evident that the construction of the documentation is an investment which, to some degree, will pay off when we need to modify the program. Notice, however, that future program modifications imply a substantial work on updating the program understanding, as represented by the documentation.

Figure 3.4 shows a snapshot of a browser which presents the result produced by the Elucidator. (For a better presentation, please consult the on-line version of the example). The three frames in the browser correspond to the panes of the basic layout, as illustrated in figure 2.2. The menu and index pane show the detailed table of contents of the documentation.

```
.ENTRY days-hours-minutes-seconds
.TITLE Dealing with days, hours, minutes, and seconds
.BODY
Now we have reduced the problem to finding the normalized months,
days, hours, minutes and seconds from a rest second counter <em>r</em>
that is less than the number of seconds in a year. It would be natural
to find the month next, but doing so would call for yet another
<em>counting</em> process, because a month is an irregular time
interval (some months have 31 days, others 30, February has normally
28 days, but there is 29 days in leap years).<p>

It is easy to find the unnormalized number of days, and the normalized
hours, minutes, and seconds from <em>r</em>. This is done by quotient
and modulo <em>calculations</em>. The function
{*how-many-days-hours-minutes-seconds} does that. We first find the
(non day-normalized) number of days by dividing r by
{seconds-in-a-day} (@a). The remainder, called {-n-rest-1} (@b) is
used to find the hour-normalized number of hours by division of
{-n-rest-1} by {seconds-in-an-hour} (@c).  Again the remainder,
{-n-rest-2} (@d) is found, and this quantum is used to find the
minute-normalized number of minutes (@e). Finally the number of
seconds are found in the last modulo calculation (@f).

We use a sequential name-biding form {-let}* to find the results in a
sequential fashion. Still we are entirely within the functional
paradigm, of course. The function returns the list of days, hours,
minutes, and seconds.
.END
```

**Fig. 3.5**: *An excerpt of the documentation source text.*

Figure 3.5 shows a portion of the documentation source, in order to illustrate the specialized markup introduced for our purposes. The example is relative to the Scheme version of the Elucidator. In the figure we see the mixture of specialized markup (roff-like 'dot notation' at the start of a line) and HTML markup. The excerpt in this figure corresponds to section 2.2, as shown in figure 3.4.

## 4. Tools in an Elucidative Environment

There are three important tools in an elucidative programming environment:

(1) The elucidator.

(2) The editor.

(3) The browser.

The *elucidator* is the most central of these. The Elucidator processes the files of the documentation bundle in order to prepare for a presentation of an elucidative program in the browser. The second tool is the *editor*. It is the qualities of the editor tool that make it realistic and feasible to produce a program and its related documentation. Without specialized editor support, Elucidative Programming is probably out of reach for most programmers. The *browser* is a standard Internet browser, and as such it supports a user friendly exploration and navigation of the documentation bundle.

In the following two sections we will discuss the Elucidator tool and the editing support in an elucidative programming environment.

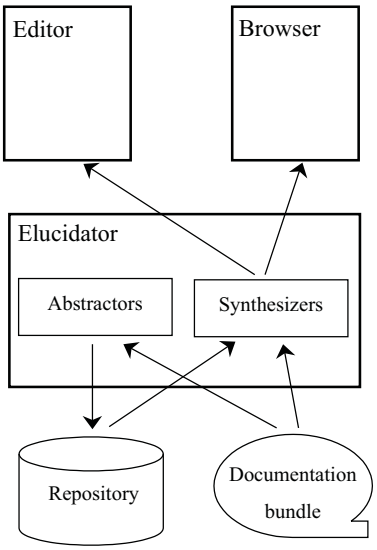


**Fig. 4.6**: *The architecture of an elucidative programming environment.*

*4.1 The elucidator tool*

An Elucidator is composed by two major components: an abstractor and a synthesizer. In turn, both the abstractor and the synthesizer has a documentation part and a program part.

The *program abstractor* parses a source program in order to identify the relevant program entities. The *documentation abstractor* similarly parses the documentation with the purpose of identifying the documentation entities. The program abstractor makes up the only programming language dependent component of an Elucidator. Consequently, another programming language can be supported solely by providing an abstractor for it.

The result of the abstraction processes needs to be organized as a program model in a *repository* such that the synthesizer component of the Elucidator as well as the editor tool (see section 4.2 below) can make use of the information. The interplay between these components are illustrated in figure 4.6. It is worth emphasizing that the repository is confined to hold a structural extract of the files in the documentation bundle; The full, but less structured information is available in the documentation and the source program files.

The information in program comments cannot be entirely ignored by the Elucidator tools. The sectional comments of the Scheme Elucidator (see section 2.4) and the source markers (described in section 2.5) both require that the program comments are parsed. This causes a problem because program comments usually are considered as lexical elements, and as such they are not available during the syntax analysis phase. The solution applied in the Scheme Elucidator is to apply a pre-processor which transforms

lexical Scheme comment to syntactical Lisp forms. In the Java Elucidator, the parser has been modified to deal properly with the information in the comments.

An *incremental abstraction process* becomes a need when large programs are to be dealt with. In such situations we cannot afford to throw away the entire repository upon a few modifications of either the documentation or the program. In the Scheme Elucidator we can manually arrange that only a subset of the documentation bundle is abstracted and further processed. A better and more automatic solution would be major improvement for the environment.

The *program synthesizer* renders the source programs of the documentation bundle in HTML such that the programs can be shown in a browser. Similarly, the *documentation synthesizer* decorates the documentation text. The handling of anchoring and linking is the most important concern of the synthesizer components. As discussed in section 2.3, an element in the *doc-prog* relation gives rise to a link from the documentation to the program as well as a link in the opposite direction. The first of these makes it possible to navigate from a discussion in the documentation to the involved program entity. The opposite link is helpful in a situation where the program reader is located in a program entitity and wants an explanation of it. In the Scheme environment this navigation is carried out by following one of the links that are anchored in the tiny yellow 'left arrow symbols'. These are the symbol that are rendered in front of the Scheme definitions (see figure 3.4). More than one such symbol may be present for a given definition in case that the definition is explained in several sections of the documentation. In the Java environment a so-called *navigation window* supports this possibility. The navigation window is brought up by the browser when the name of an abstraction (class, method, or variable) is selected.

Based on the result of the abstraction process the Scheme Elucidator makes a number of useful indexes:

- An index of the definitions in the program
- A cross reference index of the names in the program (only names that are bound at top-level).
- A table of contents of the documentation (in two different depths).

Navigational possibilities in the same vein are made possible via use of the navigation window of the Java Elucidator. All indexes are presented in the 'Menu and index pane', cf. figure 2.2. The entries in the indexes are anchors for links to the appropriate entities in the documentation or the program. The cross reference index maps names to all the definitions, in which they are applied. As a convenient shortcut in the Scheme Elucidator, a definition of (say `N`) in the synthesized program is prefixed with an icon which allows navigation to the name `N` in the cross reference index. Using these shortcuts it is relative easy to follow a selected chain of function calls, from the details towards the overall program structures (upwards in a possible procedure calling chain).

The Scheme Elucidator creates a fixed number of HTML files, which statically present the files from a documentation bundle. All the bindings are done at *elucidation time*. The Java Elucidator creates the HTML files on demand, at at *browse time*. The WWW server behind the Java Elucidator produces the HTML file via Java servlets, based on the documentation and the program files in the documentation bundle and the structural knowledge in the repository (see figure 4.6).

The static solution, which produces the fixed number of HTML files, is independent of WWW server technology. Once the HTML mirror of the documentation bundle is produced the result can be accessed from an arbitrary WWW browser (via a server, or locally on a lab top, for instance). The dynamic solution is more flexible in the sense that it can provide a great variety of different presentations if or when they are requested. But the dynamic solution is also more vulnerable, because it depends of a WWW server which supports a particular technology. If such a server is not available, the documentation bundle cannot be browsed.

### *4.2 The editor tool*

In this section we will discuss the editor tool of the elucidative Scheme environment. In a future development we want to develop a similar editor support for the Java environment. The editor tool of the elucidative Scheme programming environment presents the documentation and a selected program file in a split-view window, in a similar way as the Elucidator. We use a customized version of the Emacs editor. The customization is programmed in Emacs Lisp.

The editor offers navigation possibilities which are similar to the facilities in the browser discussed above. More specifically, the following kinds of navigation are supported via a generic `elucidator-goto` command in the editor:

- navigation from a program name N in the documentation to the definition of N in a program.
- navigation from a defining, top-level name occurrence N in a program to a section in the documentation that explains N.
- navigation from an applied name occurrence N in a program to the corresponding defining name occurrence.
- navigation from one section to another in the documentation via a *doc-doc* cross reference link.

Navigation steps are stacked in order to provide for convenient backing up to previous locations (using the `elucidator-back` edit command). The navigation made possible by `elucidator-goto` and `elucidator-back` is more powerful than plain text searching, because it may move the focus from one Emacs buffer to another. Currently the editor does not support direct navigation between pairs of source markers.

As it can be seen in figure 3.5 anchored links a represented by specialized markup in the documentation. As an example, {`multiplum-of`} refers to the place of the definition of the function `multiplum-of`. The editor supports the creation of this markup, in particular the entering of names such as '`multiplum-of`'. The name may either be taken and transferred from a program window, or it may be entered by means of Emacs completion (just type the first few letters of a name and Emacs will finalize it). Both of these creations are supported by the editor command `prog-ref`. In a similar way, `doc-ref` supports the creation of cross reference links between sections in the documentation.

The editor shows the documentation in raw and undecorated form, without any special rendering of the elucidator-specific markup nor the HTML markup. Therefore, it is much more pleasant to explore an elucidative program in an Internet browser than in Emacs. In the development situation, it is attractive to use both an editor and a browser. The core navigational functionalities are overlapping. The editor is more flexible with respect to searching than both Netscape and the Internet Explorer. The browser provides more elaborate and more user friendly navigation.

The editor offers a number of other convenient commands specific to the Scheme Elucidator. The editor knows the files of a documentation bundle. At any given point in time, one of the program files is in focus. The command `show-program` brings another program in focus. The command `reset-elucidator` establishes a split-view window, with documentation in the upper part and a program in the lower part. The `reset-elucidator` command is very useful if Emacs has been used for other and perhaps non-related purposes, such as mail reading or plain file editing.

We find that the use of Emacs is a better alternative than proposing a new and special editing tools, targeted exclusively at the creation of elucidative programs. It seems to be a general experience that programmers are reluctant to use brand new and special purpose program construction tools.

In the ideal situation, however, the elucidative editing tool should be part of an existing integrated development environment. In that situation, support of Elucidative Programming would be implanted into an existing and more complete environment, which hereby will be *documented enabled*.

### *4.3 Tool integration*

Integration of tools is an issue in every programming environment. In an environment with an Elucidator, an editor and a browser, as outlined above, the following integrations are attractive:

- Editor-elucidator integration
- Editor-browser integration.

In the current Scheme environment, it has been made very flexible to activate the Elucidator tool on the current documentation bundle from the

editor (single keystroke activation). In the Java environment, the Elucidator is a more implicit tool which is activated by the WWW server. However, the activation of the abstractors from the editor is still an issue in the Java environment.

The editor depends on information from the Elucidator tool. As explained in section 4.1 the Elucidator saves the information, which is extracted by the abstractors, in the repository, cf. figure 4.6. This information is used by the editor to support both navigation and flexible creation of links. When the Elucidator finishes its processing, the editor command `refresh-elucidator` updates the editor's knowledge about the documentation bundle, using the information from the repository.

The editor and the browser are often used side by side, because of their complementing support of useful functionalities. A simple integration of the editor and the browser would make it possible to activate or re-position the browser from the editor, and vice versa. By such an integration it would be easier to perform more frequent switching between the two tools.

A total integration of a development environment (including editor functionality), an Elucidator, and the browser can be seen as a final goal. The development environment should provide a seamless integration of the editor, Elucidator, and browser functionalities. In addition, such an environment should make it possible to export a documentation bundle as HTML. However, in our point of view, such an integration is a natural concern for the leading vendors of professional, integrated development environments. In the longer perspective we can hope for *elucidative documentation enabling* of the major professional development environments.

## 5. Related Work

The field of Literate Programming is the foundation of our work on Elucidative Programming. Literate Programming was coined by Knuth in 1984 [16] as a result of major software undertakings with the TeX text formatting system [17]. Knuth's research group implemented the WEB system [15, 18] as a set of tools (weave and tangle) which supports Literate Programming. After that a number of similar systems appeared [5, 33, 14]. The main variations stem from the programming and documentation languages supported. Some systems, such as Noweb and Nuweb, are programming language independent. There exists a published annotated bibliography of Literate Programming [30]. However, the most complete and up-to-date bibliography is available via a WWW page provided by Nelson H. F. Beebe [3]. In our earlier paper on Elucidative Programming [22] we refer to a number of small examples of literate programs (mainly from *The Communications of the ACM* in the late eighties) and to other literate programming approaches than WEB-systems.

Sametinger and colleagues at Johannes Kepler University of Linz, Austria, have developed the DOGMA programming environment for C++ [29].

The DOGMA project has in the same way as the elucidative programming project gained inspiration from Literate Programming. Both systems are based on relations between documentation units and program units. The relations are used to make relevant documentation appear when documented program units are in focus. It is worth mentioning that Sametinger et al. have developed a notion of *object-oriented documentation* for DOGMA [28]. Using these ideas, object-oriented concepts (such as inheritance) is used on sections of documentation, which are attached to classes in C++. DOGMA is a complete and integrated programming environment for C++, including a residential text editor. As a contrast, the elucidative programming environments that we have developed consist of two separate and partly overlapping tools (an editor and a browser, which is powered by the Elucidator) with a relatively simple integration in between them. In particular, we rely on an external editor (Emacs) which is customized to support an elucidative programming process. This may seem to be of minor importance; Nevertheless, we believe that most programmers are conservative in adopting new editing environments, and as such our less integrated environment may turn out to be a good, pragmatic alternative to a system like DOGMA.

In our current work on Elucidative Programming we use the World Wide Web and the Internet as a medium for program documentation. JavaDoc [6] is the main inspiration with respect Internet mediated program documentation. JavaDoc makes it possible to extract interface documentation from comments in Java programs that are marked in a special way and follow special conventions. The extracted interface documentation is organized as a set of interrelated WWW pages. JavaDoc documentation is particularly useful for documentation of class libraries, and as such it is oriented towards program reusability. Elucidative program documentation is targeted at the team of programmers who are responsible for further development and maintenance of particular applications.

Work on program understanding can be categorized in at least two different groups: *Prevenient* and *posterior* approaches. Our work represents the prevenient approach. As discussed in this paper the idea is to document the program understanding before, or side by side with the development of the program. We hypothesize that interleaved documentation and program develop processes improve the quality of our programs. Furthermore, we see the documented program understanding as an investment that pays off during the maintenance phase. However, we are aware that the prevenient approach is somewhat idealistic, and that mainstream program development takes place without much emphasis on pro-active documentation of the program understanding. The posterior approach deals with extraction of program understanding from existing programs. A variety of different reverse engineering tools have been proposed for such endeavors (for an overview, see [32]). Because of the relative dominance of the posterior approach there is much literature on this branch of program understanding. The IEEE *International Workshop on Program Comprehension* is one of the main forums for reverse engineering papers [7, 8, 9, 10, 11, 12, 13].

A few years ago our work was focussed on *hyper structured programming environments*. We carried out a number experiments with a system called HyperPro. The main contribution of this work was the notion of rich hypertext, and in particular the possibilities of defining flexible interaction schemes on rich hypertexts [24, 26]. The work on Elucidative Programming steps away from a fine grained hypertextual representation of the underlying programs and documentation. In our current tools we deal with traditional and coarse grained program source files. As such, our current work is much more pragmatic than the work on HyperPro.

Kasper Østerbye's work on literate Smalltalk programming [25] was an important part of the HyperPro project. Like in HyperPro, Østerbye's Smalltalk environment was based on a fine grained representation of classes, methods, and textual documentation. The work by Reenskaug and Skaar [27] is also about literate programming support for Smalltalk.

It is natural to study the needs for program documentation in relation to both the analysis and design phases of the software development process. UML is the dominating 'language' for description of analysis and design artifacts. The work by Vestdam et al. (from our group) describes a contribution to a CASE system which introduces the idea of documentation threads [31]. Documentation threads may involve elements from UML diagrams, program fragments, as well as pieces of documentation.

## 6. Status and conclusions

In this paper we have described the ideas behind Elucidative Programming. The paper has addressed both the Scheme Elucidator and the Java Elucidator.

The idea of dividing a window (or screen) in a documentation pane and a program pane, in between which mutual navigation takes place, is central to Elucidative Programming. A complicated programming situation is often characterized by juggling with many aspects of the program at the same time. Often it is difficult and demanding to mobilize sufficient concentration on all these aspects (pieces of the program). In this situation the documentation pane may be used to keep a number of program parts together (by means of links) in a way, which makes it easier and safer to handle a complex programming task.

Both the Scheme Elucidator and the Java Elucidator are in local but limited use at Aalborg University. We plan to use the Java Elucidator in the introductory object-oriented programming course at Aalborg University in the fall 2000. A number of elucidative programs are available from the *Elucidative Programming home page* [19].

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 which divides the documentation into three interrelated deliberative categories: Motivations, Rationales and Solution descriptions. Finally, this new model is utilized in the Java elucidative programming environment by the implementation of a coloring scheme and extensive navigation facilities.

The work on the Scheme and Java Elucidators raises several interesting questions with respect to documentation of program understanding. First, is it possible to convince and discipline programmers to document their program understanding? Second, is it possible to convince the managers of program development projects to invest in an improved program quality, by means of documented program understanding "the elucidative way". Third, can the documented program understanding of an elucidative program be maintained with reasonable means? And finally, can we develop practical documentation patterns that will allow average programmers to write good elucidative documentation of their programs? In the next couple of years we hope to find good answers to these questions.

The Scheme Elucidator is available as free software from the LAML home page on the Internet [20].

**Acknowledgements.** The Java Elucidator was implemented by Max Rydahl Andersen, Claus Nyhus Christensen, Vathanan Kumar, Søren Staun-Pedersen, and Kristian Lykkegaard Sørensen in the first part of a Master thesis project.

## 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] Nelson H. F. Beebe. A bibliography of literate programming. http://www.math.-utah.edu/pub/tex/bib/litprog.htm, 2000.

[4] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. ACM, May 1993.

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

[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] IEEE. *Proceedings of the Second Workshop on Program Comprehension*, July 1993.

[8] IEEE. *Proceedings of the third Workshop on Program Comprehension - WPC'94*. IEEE Computer Society Press, November 1994.

[9] IEEE. *Proceedings of the fourth Workshop on Program Comprehension*. IEEE Computer Society Press, March 1996.

[10] IEEE. *Fifth International Workshop on Program Comprehension*. IEEE Computer

Society Press, March 1997.
[11] IEEE. *Sixth International Workshop on Program Comprehension*. IEEE Computer Society Press, June 1998.
[12] IEEE. *Seventh International Workshop on Program Comprehension*. IEEE Computer Society Press, May 1999.
[13] IEEE. *Eight International Workshop on Program Comprehension*. IEEE Computer Society Press, June 2000.
[14] Andrew L. Johnson and Brad C. Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.
[15] Donald E. Knuth. The WEB system of structured documentation. Technical Report STAN-CS-83-980, Department of Computer Science, Stanford University, September 1983.
[16] Donald E. Knuth. Literate programming. *The Computer Journal*, May 1984.
[17] Donald E. Knuth. *The Texbook*. Addison-Wesley Publishing Company, 1984.
[18] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison Wesley, 1993.
[19] Kurt Nørmark. The elucidative programming home page. http://www.cs.auc.dk/-∼normark/elucidative-programming/, 1999.
[20] Kurt Nørmark. The LAML home page. http://www.cs.auc.dk/∼normark/laml/, 1999.
[21] Kurt Nørmark. An elucidative programming environment for Scheme. In *Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, May 2000. Available via [19].
[22] Kurt Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*. IEEE, June 2000. Available via [19].
[23] Kurt Nørmark, Max Rydahl Andersen, Claus Nyhus Christensen, Vathanan Kumar, Søren Staun-Pedersen, and Kristian Lykkegaard Sørensen. Elucidative programming in Java. In *The Proceedings on the eighteenth annual international conference on Computer documentation*, September 2000. Accepted but not yet published.
[24] Kurt Nørmark and Kasper Østerbye. Rich hypertext: A foundation for improved interaction techniques. *International Journal of Human-Computer Studies*, (43):301–321, 1995.
[25] K. Østerbye. Literate Smalltalk programming using hypertext. *IEEE Transactions on Software Engineering*, 21(2):138–145, February 1995.
[26] 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.
[27] Trygve Reenskaug and Anne Lise Skaar. An environment for literate Smalltalk programming. *Sigplan Notices*, 24(10):337–345, October 1989.
[28] J. Sametinger. Object-oriented documentation. *Journal of Computer Documentation*, 18(1):3–14, January 1994.
[29] J. Sametinger and S. Schiffer. Design and implementation aspects of an experimental C++ programming environment. *Software Practice and Experience*, 25(2):111–128, February 1995.
[30] L. M. C. Smith and M. H. Samadzadeh. An annotated bibliography of literate programming. *Sigplan Notices*, 26(1):14–20, January 1991.
[31] Thomas Vestdam. Pulling threads through documentation. In Mughal and Opdahl, editors, *Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, May 2000.
[32] Richard C. Waters and Elliot Chikofsky. Reverse engineering: Progress along many dimensions. *Communications of the ACM*, 37(5):22–25, May 1994.
[33] Ross Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, 1992.