Evaluating Elucidative Programming in an Industrial Setting

Max Rydahl Andersen  
Department of Computer Science  
Aalborg University  
Fredrik Bajers vej 7E  
DK-9220 Aalborg Øst  
max@cs.auc.dk

Claus Nyhus Christensen  
Department of Computer Science  
Aalborg University  
Fredrik Bajers vej 7E  
DK-9220 Aalborg Øst  
santa@cs.auc.dk

December 22, 2000

Abstract

In this paper we present an evaluation of Elucidative Programming in an industrial setting. Elucidative Programming is a modern variant of Literate Programming, which seeks to provide a practical method and environment for maintaining program understanding.

The evaluation is based on a field experiment carried out in a software company, where three developers were to do Elucidative Programming in their daily work. The purpose of the experiment was to evaluate whether Elucidative Programming is applicable in an industrial setting.

While no decisive conclusion for the industry as a whole could be drawn, we have found that both Elucidative Programming in general, and a specific Elucidator tool in particular, were applicable in the conducted experiment. Furthermore, the experiment indicates that implementation of Elucidative Programming in an industrial setting is realistic.

1 Introduction

Today software development typically occurs at a fast pace and in an ever changing environment. Thus the implementation of software can be intrinsic complex and the constant change in the environment implies a need for constant improvement and maintenance.

Furthermore, the need for documenting the software in such a way that it can be understood and maintained by current and future developers becomes evident. This is acknowledged, and even viewed as vital, by developers and managers alike. Still, documentation of implementation details in software are often viewed as a tedious task. It tends to produce a large amount of uninteresting documents with no direct connection to the source code. Furthermore the documentation is often not kept up to date with the current implementation and typically never read. Thus the documentation becomes of no value to neither the developers nor the company.

In an effort to try to remedy this situation, we have been working on a paradigm called Elucidative Programming[2, 3, 14, 17]. Elucidative Programming is inspired by the work on Literate Programming initiated by Donald Knuth[10]. A Literate program is structured according to the needs of the program reader, and not according to rules of the programming language. Furthermore, Literate Programming depend on the source code residing inside the documentation, rather than the other way around. Elucidative Programming is a modern variant of Literate Programming, which seeks to provides a practical method and environment for documenting the internal structure of software.

This paper reports on the result obtained by evaluating Elucidative Programming in an industrial setting. The evaluation is based on a field experiment, carried out in a software company. Here three developers were introduced to Elucidative Programming and was asked to use it in their daily work. The purpose of this evaluation is to examine whether Elucidative Programming is practical applicable and at the same time to get a glimpse of which parameters influences the usage and success of the Elucidative paradigm. Such parameters were expected to be, e.g., the level of motiva-
tion from the developers and the company, their acceptance of the paradigm and the needed degree of tool support.

The rest of this paper is organized as follows. Section 2 starts by describing the basic concepts of Elucidative Programming. Next, in Section 3 and 4, the case of the field experiment and analysis approach, is described. Section 5 presents the gathered key evaluations. Based on these evaluations, Section 6 lists the lessons learned through the experiment. In section 7 we relate our work to that of others, and finally, in Section 8 we conclude on our work.

2 Elucidative Programming

Elucidative Programming, as coined by Kurt Nørmark[14, 16], is a documentation paradigm for maintaining the understanding of software programs for current and future developers.

Elucidative programming is inspired by Literate Programming[10]. It furthermore holds resemblances to documentation produced by extraction of information from comments placed in the source code. This type of documentation is often referred to as interface documentation, and includes tools such as Doc++[1] and JavaDoc[6, 12].

In the following the main characteristics of Elucidative Programming will be described.

A textual viewpoint on documentation:

Elucidative programming relies on formatted text as its main means for documenting the understanding of software programs. The text can freely be formatted using HTML and additional graphics, e.g., UML diagrams, can be used in connection with the text.

The source code and documentation are kept in separate files:

In interface documentation the documentation is inserted as comments to the code. In Literate Programming the source has to be embedded inside the documentation. Both ways, the documentation and source code becomes physically dependent on one another.

This is not the case with Elucidative programming, since the documentation and source code are stored in separate files. This provides two advantages: 1) In contrast to a literate document, other tools (such as UML design tools) can operate on the source code. 2) the documentation is not tied to just one specific source code entity, as it is the case with Literate Programming and interface documentation. It is therefore possible to document transverse issues, e.g., dynamic aspects concerning multiple classes.

The source code and documentation are linked together by hyper-links:

Since the documentation and source code are stored in separate files the physical proximity found in interface documentation and Literate Programming are lost. Instead the documentation and source code are tied together by hyper-links. These have to be followed in order to get from the documentation to the source code or visa versa, thus creating navigational proximity.

The documentation is oriented towards online presentation:

Like interface documentation, but unlike Literate Programming, documentation produced using Elucidative Programming is oriented towards an online presentation. This means that the documentation will be read through an Internet browser, and will contain links for navigation in the documentation and source code. This characteristic is a natural evolution from Literate Programming, seen in the light of the technological changes which have taken place since the introduction of Literate Programming.

The source code and documentation are presented side-by-side:

Normally interface documentation tools such as JavaDoc, are used to produce a set of HTML-pages, or other formats such as PDF or \LaTeX. These contains the interface documentation and relevant declarations from the source code, but no source code is directly visible nor accessible. In contrast Literate Programming tools[4, 9, 11, 23] creates a nice book-like presentation which embeds source code in the documentation.

Elucidative Programming creates a hybrid of the two, but unlike Literate Programming, Elucidative Programming does not go for an embedded presentation of the program inside its documentation. Rather, Elucidative Programming provide for mutual navigation between documentation and source code in a two-frame layout.
Besides the main characteristics of Elucidative Programming it should be noted that Elucidative Programming is considered a complementary method to other commonly used documentation tools/methodologies, such as interface documentation and UML. In fact, due to the separation of the source code and documentation, Elucidative Programming can successfully be used to tie other types of documentation together. As an example, the current Java implementation of the Elucidative paradigm has direct support for linking to JavaDoc documentation.

2.1 The Elucidator tools

When performing Elucidative programming three tools are involved:

**An Elucidator** which converts the documentation and the source code to an online viewable format. The Elucidator tool does this in two steps:

**Abstraction:** The abstraction process parses the documentation and source code. It thereby extracts information about the entities of the documentation and source code, as well as the relationships between these.

**Synthesis:** Based on the information obtained in the abstraction process, the synthesis process generates the representation of the documentation and source code which are shown in the browser.

**An Internet browser** is used for the presentation of the Elucidative program. The browser allows for mutual and easy navigation between the documentation and the source code. A screen capture of the internet browser can be seen in Figure 1.

**An editor** for producing both the source code and the documentation. The editor applies a side-by-side layout in the same way as the browser, and it supplies functionality for Elucidative Programming tasks such as the creation of hyper-links between the documentation and the source code. A screen capture of the editor can be seen in Figure 2.

Doing Elucidative Programming with the three tools mentioned above can be viewed as a three step cycle. First you use the editor to produce some documentation and source code. Next you apply the Elucidator tool to the written documentation and source code. Finally you view your documentation and source code in the browser. From here you either modify or create new documentation and source code, thereby forming a cycle.

It is possible to implement the three tools in various ways and for various programming languages. At the moment two distinct implementations exists.

The **Scheme Elucidator** is the initial implementation of the Elucidative paradigm, done by Kurt Nørmark[15]. Some of the main characteristics of this implementation includes: The documentation is stored in one single file, the Elucidator synthesis produces static HTML documents and the navigational functionalities of the editor closely mimics that of the browser representation.

The **Java Elucidator** was developed as part of our earlier work on Elucidative Programming[2, 3, 17]. It was developed for two purposes: 1) To implement an Elucidator for a more commonly used language than Scheme. 2) To be used as the foundation for further development of the Elucidative paradigm.

A number of differences exists between the two implementations. Besides the choice of language, the biggest difference is that the Elucidator for Java presents the documentation and source code in the browser is done dynamically and on demand. In practice this has two advantages: 1) The synthesis process is shortened as it is not necessary to produce every single HTML-page. 2) It is easier to provide more interactive documentation, such as searching and different context-dependent views/presentations.

The rest of this paper only deals with the Elucidator tool for Java.

2.2 Evolution of the Java Elucidator

Different versions of the Elucidator tool for Java has been developed. Most of the versions only add extra functionality to the tool, while the concepts of Elucidative Programming, explained above, are kept untouched. However, one conceptual change has been made. In an evolutionary step away from using just one long essay to document the program
### Figure 1: Screen capture of the Internet browser used by the Elucidator tool to display Elucidative programs.

The screen capture is not from the Elucidative programs produced during the experiment.

### Figure 2: Screen capture of the editor used when making Elucidative programs. The screen capture is not from the Elucidative programs produced during the experiment.
as used in [2, 17], the documentation is split up in several documentation nodes[3]. As a result of this each node is typically used by the developer to address one distinct aspect of his program. Furthermore, the documentation nodes can be hyperlinked, just like it is the case with the documentation and source code, and thereby creating a hypertext.

A further evolutionary step was taken by dividing the documentation nodes into three categories: Motivations, Rationales and Solution description. Thereby they were elevated to something more than just simple nodes. This division of the documentation nodes are expressed in the MRS-model for Elucidative programs[3]. This model furthermore has typed hyper-links. In order to unify the documentation effort, also a set of predefined templates for the structure of the documentation nodes.

3 Case

The research presented in this paper is based on a field experiment within a Danish software house. At the time of writing the company has 16 full-time developers and a few free-lance student developers. The company is specialized in object-oriented development, especially Java, and does both actual software development as well as more research minded activities. The company furthermore uses the experiences of their developers to provide counseling and courses to other companies. Finally, the company has previous experience with environments for software documentation, and has ideas for developing their own tool for documenting software.

The purpose of the experiment was to evaluate the applicability of Elucidative Programming in an industrial setting. This includes a study of the usability of the Elucidator tool described previously (see Section 2.1). Given these observations, it should be evaluated if the implementation of Elucidative Programming in the industry is realistic. Furthermore, pointers to how this could be done should be provided.

Three developers (Developer A, B and C) participated in the field experiment.

Developer A and B both have masters degrees in Computer Science and six to eight years of experience with software development in the industry. The two developers work together on the same project - an application for construction of electronic medical case records via templates. It is part of a bigger system for electronic medical case record in public hospitals. The system is build on Enterprise Java Beans (EJB) technology and the Swing GUI toolkit\(^1\). It has been in production for a year with 200 users.

The third developer (Developer C) is the company’s Chief Research Officer (CRO). Developer C completed his Ph.D during the experiment and has six years of development experience. He is working, together with developers who did not participate in the experiment, on the EJB application server in which the aforementioned electronic medical case record application is running.

Common for the three developers are that they describe the development method applied in the two projects as an adhocracy[13]. They adapt their work to the current situation and work in small teams which allow them to communicate their ideas and understanding verbally (mutual adjustment[13]). For Developer A and B this form of development works well. Developer C, which has a more explorative type of work, also seems to find this satisfactory. Still, all three developers wish they had more documentation available to them, describing the thoughts and ideas behind the actual implementation of the systems they maintain and develop. This goes both for their own work and the work of the other developers in the teams.

The three developers were asked to perform Elucidative Programming in their respective projects for a period of two months. An agreement between the three developers, the management and the authors were forged, allowing the developers to spend approximately five hours each week on Elucidative Programming. However, the developers had to perform other tasks besides their development project. This resulted in an effective experimentation period of approximately two weeks for each developer, spread throughout the two months.

4 Approach

As described in Section 3, a field experiment was conducted in order to evaluate the usability

\(^{1}\)For further information on these technologies see: http://java.sun.com/products.ejb/ and http://java.sun.com/products/jfc/
of Elucidative Programming in an industrial setting. This section describes the methodology used throughout the experiment.

Due to our active involvement with the Elucidative paradigm, and our roles in the experiment as observers, our objectivity towards the collected data is limited. To accommodate this, data were collected from different sources to provide for data and method triangulation. Thus the following three data sources were used to ensure a credible qualitative evaluation of the experiment:

1. Open ended interviews with the participants
2. Field observations of the actions of the participants
3. Measurements on the produced Elucidative programs

The method is inspired by Patton[18] and provides a mixture of both subjective and objective data. In the following a more thorough description of the method used for each of these data sources is presented.

**Interviews with the participants**

In order to get the participants subjective opinions and thoughts on Elucidative Programming, each participant was individually interviewed four times during the experiment. The interviews were evenly spread throughout the experiment period, thus possible variations in their opinions and thoughts could be tracked.

Each interview was conducted as a standard open ended interview. This means that the interviewer asks the questions according to a list of questions, where the exact wording of each question is written down in advance. If the interviewer sees a need to probe or ask a follow-up question he is still free to do so, as long as the general structure and wording of the prepared interview is followed. This ensured that all participants were asked the same questions, thus increasing comparability of responses.

At the end of the experiment, the three developers participated in a group interview. The goal of the interview was to have an open and free discussion about Elucidative Programming and the experiment they had been participating in. At the beginning of this interview, preliminary observations and experiences gathered from the four preceding interviews were presented to the participants. Thus they were allowed to reflect upon the observations and through the later discussion provide us with valuable feedback and corrections.

All the interviews were recorded on tape and transcribed for further investigation.

**Field observations**

Throughout the experiment period we were present at the company’s location two to three days a week. By being present, the daily work of the participants could be monitored. This allowed us to make field observations on what happened, but also what did not happen. To keep track of the field observations they were written down in a diary[8].

**Measurements on the produced Elucidative programs**

In order to produce objective quantitative measurements, the usage of the Elucidator tool was logged. Furthermore, the actual produced Elucidative programs were measured with respect to the amount of text, links, documentation nodes etc. This provided statistical material from the experiment, which was used to track the documentation productivity over time.

Finally a subjective evaluation of the Elucidative programs was performed, in order to investigate the quality of the contents.

**5 Evaluations**

As described in Section 4, three sources have been applied to evaluate Elucidative Programming. These evaluations are expressions of viewpoints held by the three developers participating in the experiment, field observations done by the authors and measurements on the documentation produced throughout the experiment.

In this section we report on the data gathered from these three sources. The data is structured into eight issues. Whenever feasible quotes from the interviews are used. These are freely translated from Danish. Typographically the quotes are written in a serif font, and surrounded by “ and ” (quote/end-quote) signs.
These issues will later form the basis for both a number of lesson learned through the experiment, and the conclusion.

There are eight issues. Each issue is presented as a section. Each of these sections may contain additional sub-sections, in order to report on sub-issues.

5.1 Developer characteristics

As the actions and viewpoints held by participants in the experiment may be influenced by their characteristics as developers, these are introduced.

Developer A and B

Developer A and B sees the software implementation as the primary product of their work. This is expressed by Developer A: "... in the projects I am working on, the source code is the primary product". Likewise their view of the world centers around the source code. This is symbolized by Developer B stating: "Once a hacker, always a hacker".

Both developers express that they have some experience with writing documentation for their design and implementation. However, not in a systematic form. Their view on the importance of documenting the internal parts of the design and implementation (also known as internal documentation) are expressed by Developer B: "[internal documentation is] very important to get a hunch on how stuff works ... it gives a tremendous kick-start to be able to get an overall coherent description".

Finally it was observed that both developers showed a practical approach to the usage of the Elucidator tool, by seamlessly adapting the tool in their development process.

Developer C

Like Developer A and B, Developer C expresses the need for internal documentation: "It [internal parts of the design and implementation] is often my concern in my daily work - it is on this level I am concerned the most". When asked how he rates internal documentation he answers: "It is the most important".

Developer C has recently completed his Ph.D. and his project and work can be viewed as more exploratory and research based, than the work of Developer A and B.

Opposite to the two other developers, Developer C did not accept the Elucidative paradigm and tool right away. Instead he needed to have a thorough understanding of the concepts involved before he could be motivated to use the Elucidator tool.

5.2 Documentation evaluations and measurements

Statistical information concerning the amount of documentation produced throughout the experiment is presented in Table 1 on the following page.

The documentation produced by Developer A can be characterized as being an attempt on making a common structure on the documentation for the project. Evidence of this includes the relatively high number of created nodes and links to documentation nodes, and the absent of links to the source code. It was observed that a noticeable portion of the documentation nodes where outlines, containing only the intention of the developer expressed by an abstract.

The documentation written by Developer A falls into two categories: project management, expressed by to-do lists etc., and design considerations for sub-parts of the project. No real internal documentation was produced.

Opposite Developer A, both Developer B and C produced real internal documentation. This is shown by the relative high ratio between the number of links to the source code and number of documentation nodes. Furthermore the contents of the documentation nodes are of a more technical and detailed nature than the nodes written by Developer A.

5.3 The Elucidative paradigm

Based on the views expressed in the interviews, four sub-issues concerning the Elucidative paradigm have been identified.

Applicability of hypertext

The realization of the Elucidative paradigm is based on the usage of hypertext. All of the developers voiced that they found this usage of hyper-
Developer A and B participated in the same project. Table 1 lists the measurements of the Elucidative programs written during the experiment. Developer A and B particularly pointed out by Developer A: “What this tool can be used for, is to create what correspond to these mind maps ... where one more or less disregards the structure and is more concerned with what is actually written”. He later continues: “One can just write down what comes to mind, and then later be able to search it - maybe apply a structure on it later”.

Furthermore both Developer A and Developer B said that they felt that the Elucidative paradigm fitted the ad-hoc development process used in their project: “It is a paradigm which fits the lightweight process we use. It is very good to write down some notes on how this class works. So I believe that if we did not have such an Elucidator tool, then in the best case this would be scribbled down on a piece of paper, which would eventually be crumpled up”.

Proximity between source code and documentation

All participants in the experiment expressed that the navigational proximity between the source code and the documentation, is one of the primary strengths of the Elucidative paradigm. This is illustrated in the following statements: “It is a good way of doing things. Trying not to sound XP [Extreme Programming] self-righteous, the primary product we produce is our software, our code, therefore it is fine that all this stuff with the documentation of our thoughts about stuff we have done, are hooked up to it [the source code], rather than maybe relating your source code to some kind of design document. It is better to link the source code to your stuff” and “If I was to write the documentation completely isolated from the source code - well then it would have been completely useless - so it has to be done on the source code in some way”.

When it comes to reading the documentation Developer B observed the following about the proximity: “It is rather cool to be able to jump back and forth in it [the documentation and source code]”. He continues to observe that it is useful to be able to navigate from the source code, to where the source code is documented, and back again.

### Maintaining program understanding

Given the span of the experiment, the issue of maintaining the program understanding over time, has not been investigated thoroughly. Still Developer A provided the following thoughts: “I think - this is just a thought - I think that I will be truly glad that I have made these notes about these assumptions, when I has to make corrections to it in three months. This I can already predict”.

It should be noted that both Developer A and B plans to keep using the Elucidator tool in the future. They thereby get a chance to obtain experience on maintaining program understanding over time.

### Documenting transverse issues

Finally, Developer C noted an important property of the Elucidative paradigm: “I believe that it [Elucidative Programming] is well suited for exactly what we are working with. Because it involves some relatively complicated programs. They contain very few lines of code, and they have consequences across a high number of modules. They are all modules that intervene with each other one way or another, they provide service for each other. And all these crossing dependencies can be very hard to figure out, so here I strongly believe that the paradigm will be able to contribute”.

<table>
<thead>
<tr>
<th>Documentation nodes</th>
<th>Developer A</th>
<th>Developer B</th>
<th>Developer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of documentation</td>
<td>1600</td>
<td>346</td>
<td>829</td>
</tr>
<tr>
<td>Links to documentation</td>
<td>18</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Links to source code</td>
<td>0</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 1: Measurements on the Elucidative programs written during the experiment. Developer A and B participated in the same project.
5.4 The Elucidator tool

When evaluating the Elucidator tool the following three sub-issues were identified.

Documentation with two representations

One of the main characteristics of the Elucidator tool is that the documentation and source code have two representations. One in the editor, and one in the browser. When the user of the tool wishes to navigate the documentation and source code he invokes a program called an abstractor, which takes care of the transformation from the editor to the browser representation. This characteristic has lead to a number of comments from the three developers.

Developer C stated: “The development method used here is done in modes. First you program your documentation, then you compile your documentation, and then you are able to view it. It would be an essential feature for the usability [of the tool] to completely eliminate the fact that you have documentation [in two representations]. Imagine that I write some documentation, then while I write this documentation, it is possible to shift-click on some of the links I have created, and then follow these links, and then continue documenting over there etc. Thereby we circumvent this cycle which you really don’t need because I got it all live, I got all the data, all the information anyway. So if you had an intelligent editor, then it could validate those links in the instance you write them”.

Similarly, the following comment was voiced by Developer A: “The reason for my position on not using the browser much, but instead just used Emacs, was that I did not bother to shift [between the two] - this was probably the reason for just using the editor . . . there was first the extra step of having to do an abstraction, and then switch to another tool - so I could maybe imagine some sort of functionality in Emacs, by which I could just press a key, and then the browser would come forth presenting the documentation”.

Furthermore, the fact that the abstraction process required the source code to be compilable, created some problems for Developer C: “Because you actually sit and write some program and simultaneously as I write this program . . . I would like to document it, and this program piece I have written may not even compile. I believe that people differ in their development style. I like to write a whole lot of code and then I make it compile later”.

It was observed that the problems with the two representations were not noted by Developer A and B during the experiment, but first when they heard Developer C opinion at the final group interview. An explanation for this was expressed by Developer C: “I believe, as far as it goes, that we are not in that much of a disagreement - we have just had different points of attack on the same problem - Developer B who did not know that much about Emacs eventually got used to it, Developer A just stayed in Emacs and I allowed myself to be frustrated by having to switch back and forth. So we do not really have that different opinions”.

Both Developer A and B agreed on this.

Server architecture of the Elucidator tool

The Elucidator tool used for the experiment was configured as a client-server setup. The clients were running on the developers laptops, while the server was running on a separate stationary machine. This setup proved to give some problems for the developers.

Developer A said: “If I should have used the browser more often, then I think you should consider to also have a web-server running locally on the machine - my machine is a laptop and I am often working at home. In this situation you would like to browse your own documentation even though you do not have a connection to [the stationary] web-server”.

This was further elaborated by Developer C: “The fact that we all have laptops which we carry around, and connect and disconnect to the net . . . this requires some different configurations. Some different problems concerning this exists. This could be more streamlined and the solution could be better adapted to the environment”.

Integration to existing development environment

The Elucidator tool is integrated into Emacs which was used by all three developers before the experiment. Developer A had the following comment to this integration: “It [the Elucidator tool] is a nice tool. It is nevertheless, at least the editing part of it [the Elucidator tool], Emacs I use when I develop software, and that makes it a nice tool [for writing Elucidative Programs]”.

9
Developer B and C agreed that the integration of the documentation tool into their existing development environment was useful, but they would both have liked a more What-You-See-Is-What-You-Get (WYSIWYG) like documentation tool. Developer C pointed out that such a tool should still be integrated with Emacs.

5.5 Writing Elucidative programs

The following present four sub-issues concerning the act of writing Elucidative programs.

Benefit of writing

All the developers felt that it was advantageous to write down their thoughts on design and implementation issues. This is illustrated by the following story told by Developer B: “While documenting the tree-model, I experienced that some part of the implementation sounded cumbersome. Why could I not just do it this way? At this point I could not grasp this question. Therefore I tried to make another implementation, but while doing this I discovered: oh, that’s why! I was not able to grasp this up-front. This [observation] I wrote down in the documentation: well, that’s why it has to be this way. . . . One could say, I tried this, but if you look here in the code they you can easily see that there is a problem, because it returns another object each time, and it is very important that it is the same object every time”.

Developer C has had similar but not as pronounced experiences: “In the situations I have been in during this experiment, it [writing Elucidative programs] have helped me uphold, confirm me in, the understanding I had already obtained”.

He continues to express the feeling that the time spent writing Elucidative programs probably is worth the effort. This is illustrated in the following quote: “It [writing Elucidative programs] clearly help to structure other things, so yes. At first it feels like: this takes time. But I do not think this is necessarily the case”.

Developer A shares these views, but also points out that this advantage is not unique for Elucidative Programming, but can be obtained through other types of writing as well.

Structuring documentation

Even though writing documentation seems to be beneficial, it is not necessary easy. One of the problems are to structure your thoughts and ideas. This was experienced by Developer B: “Yes, it is also hard some time, that is, to find the right level of granularity, how much detail should you include?”.

Later he elaborates on this: “You have an enormous overhead by administrating this information - one thing is to write it . . . but another is to get a structure on the ideas. This is extremely demanding on your [mental] resources. This is the worst for me, since I many times has got some sort of a writers block. I would like to document it, but how am I going to structure it?”.

To deal with the problem of structuring the documentation Developer B suggest the usages of “Writers Workshops”, which he describes as: “A joint activity. That you in a group decides that we want to document this thing or that. It is actually also a method of exchanging information - that you somehow agree on how we structure our information”.

Verbal communication versus written documentation

As the three participants are doing software development in teams, they often use verbal communication to express and exchange design and implementation ideas. Comparing this to written documentation, Developer C has the following comment: “I use a lot of time and energy standing by the blackboard, telling my colleagues my intentions, why I intent to do so, how it is structured and so forth. Then, when you get to it [writing it down] and spends time on it, then it is just another way of externalizing your ideas, making it possible to reflect upon them. . . . I believe that if you have the time to do so, it is a more efficient form, to interact with some other people. But of course, this is demanding on the resources. There has to be someone present to spend time on it”.

Developer B expresses a wish to integrate the verbal communication into the written documentation: “If it was possible to include [into the written documentation] what is written on the blackboard and discussed in the team, it would be really cool. Because this is often where the design is created. . . . And it would probably be a good idea to follow up
on the decisions which are being made there”.

Intimate documentation

While discussing the act of writing Elucidative documentation, Developer C has the following observation: “It actually is a very intimate tool, Elucidative documentation. Actually, you reveal yourself and expose the thoughts you have made, the half finished solutions, the problems there exists etc. Right? This is not end user documentation, and it is not polished analysis and design. It is actually very intimate, so on that set it reminds me a lot of this diary I have written from time to time”.

Furthermore Developer C states that the level of intimacy is especially high for internal documentation.

One could be lead to believe that this intimacy could be a problem. To this Developer A says: “Of course you have to be honest and write down the things that you actually mean. If you is afraid of doing this, then you have an assumption that you is employed in a supervising organization, which monitors what you write, how much you write and so on. This is fortunately not the case here. This could be a problem elsewhere”.

This viewpoint was supported by both Developer B and C.

5.6 Reading Elucidative programs

Due to the setup of the experiment the experiences with reading Elucidative Programs differs.

As Developer A and B were working together on the same project, they had the opportunity to read each others documentation. Both developers expressed that by reading the other persons documentation, they gained an insight into his work.

Opposed to that, Developer C was the only one from his project who was writing Elucidative programs. This meant that he had no Elucidative programs to read other than his own, and nobody but himself read his. He expressed a wish for reading Elucidative programs written by developers from his team, and he felt his written documentation lost some of its value as it was not read by team members either.

Group communication

Another aspect of reading Elucidative programs are exploring the possibilities for using it to enhance the group communication.

On this aspect Developer A states: “I believe that the strength of this paradigm is first really shown when more people are involved, which do not necessarily know what each other are doing. What if you need to look into what some one else has created, and you has not priorly been involved with the making of this? Then I truly believe it will show its strength”.

Developer C elaborates on this: “When a group of developers uses it [the Elucidator tool] to exchange information about the program they have written together, then this is like an extension of our cognitive relationships inside it [the Elucidator tool]. In the true spirit of hypertext”.

Another aspect of using Elucidative programs in group communication, is to provide the developer with a purpose for writing the documentation. This is expressed by Developer C: “I sit on an island with my documentation, and that is not much fun, since I already has it in my head anyway. [So therefore it must be] put in a context where the documentation you has produced can be beneficial”.

Besides the paradigm facets, Developer B noticed that the web-server part of the Elucidator tool encouraged group communication.

5.7 Motivation and habits

A common perception is that one has to be motivated in order to write documentation. Therefore, a number of questions were asked with the purpose of revealing whether or not the developers were motivated to produce Elucidative programs before and during the experiment.

Initially, Developer A stated: “I do not miss documentation, I just derive the information from the source code”.

Halfway through the experiment Developer A explained: “I have been looking at Developer B’s documentation... he has started documenting some of the things we have made previously and also some of the assumptions behind these. In this I see a clear benefit. Because in such an organization where we would never ever write these huge documents - we have no use for them - but these notes about our
assumptions you have made in your brain when developing, these can be used for something. So, for this I am highly motivated." He continues: "Yes, I can see the usefulness in this and I know it is something that can assist me sometime, and save some time for me, this is what is motivating me. It surely is not because I think it is fun to sit and write these nodes, it is because I see its usefulness".

When it comes to Developer B he initially said: "Right now my head is about to explode since I lack documentation which explains the connections of the program I am working on right now. . . . It should be something which describes these connections - the big picture, and preferably something which relates the overview to the code".

When questioned in the middle of the experiment Developer B expressed his motivation: "I think it is really great to get some of the thoughts you make when developing systems down on paper".

Finally, Developer C initially stated that internal documentation was for him the most important type of documentation. Furthermore he stated: "The issue of explaining relationships, context demands between classes, multi threaded programs and implicit assumptions can be very hard to communicate - I am not really sure where to put them".

Through most of the experiment Developer C did not document his work. When confronted with this, Developer C explained it with a lack of motivation because he did not feel he had the full understanding of the concepts involved in Elucidative Programming. To remedy this, it was arranged that one of the authors would be coaching him by developing and documenting together with him for the rest of the experiment. The result of this was that Developer C actually produced documentation.

Through the questions asked about motivation, another factor was identified: habits. This was expressed by Developer B: "I can answer with a figure of speech which says that the road to hell is paved with good intentions. It also includes habits. For example, I think it would be a really really good idea if I stopped smoking, but . . . What I am trying to say is that the motivation may be there, but you may not do it anyway. You could then argue that you are not motivated enough, because if you where motivated enough, then you would do it. It concerns habits. Since our last talk I has started to use it more and more, and has gotten past the hurdles I had in the beginning. You also get more acquainted with the tool you are dealing with".

To overcome the habits Developer B suggests: "It probably is something you has to integrate in the process . . . to get it on your mind, as a daily activity, as a natural part, right ?".

5.8 Implementation of Elucidative Programming in an organization

In the last individual interview, Developer C expressed that he did not think he would continue writing Elucidative programs when the experiment ended. However, both Developer A and B stated that they intended to continue using Elucidative Programming. Both in their current and future projects.

Based on this declaration the developers were asked how Elucidative Programming could be successfully implemented in their organization. This resulted in two points.

Creating awareness

The three developers expressed that one of the major obstacles was to increase the developers awareness of Elucidative Programming. This is expressed by the following quote from Developer C: "You need some sort of critical mass in order for it to really get under your skin. So yes, I believe in the paradigm, but the problem is surely to establish an awareness about it".

Developer A suggested the following idea for how awareness could be established: "Maybe we would have used it [the Elucidator tool] more if you had kind of forced us to do it [write Elucidative programs]. So if you saw me write some code, then you could have said to me: well, why don’t you just create a node and explain why you are doing this and that".

Choosing the project

As mentioned earlier all the developers felt that Elucidative Programming has its strength when it is used by a group of developers.

With respect to the implementation of Elucidative Programming in an organization, Developer A extends: "It is not crucial that improvements are
made on the tool. What I think would be important is that you chose the right project to do it in. You got to have a project with five or six members. And a project which is in a production state, where you can imagine some things you would like to change, and some things you do not dare to change: phew, I do not dare to mess with this, but I can always make a note about it - if I one day want to change something. It has to be a real project, it should not be a prototype project where you play around and explore technologies. Because, this is not where it [the Elucidative paradigm] has its strength. It has to be a real project, which is being delivered to customers, and there will be multiple releases, and it is an important project. Also to signal that this is something we take seriously”.

6 Lessons

The previous section presented the evaluation of the field experiment. The lessons learned from this, our field observations and experiences, has been categorized into five groups. These groups summarize the knowledge gained through the experiment.

6.1 Applicability of Elucidative Programming in the industry

In our opinion, three properties captures the essence of Elucidative Programming: separation of documentation and source code, documentation of transverse issues, and navigational proximity.

First, in an Elucidative program the documentation and source code are separated. An effect of this is that the documentation is accentuated. This is contrary to comments in, the code, since this type of documentation tend to vanish from the developers awareness. On the other hand, the accentuation does not make the documentation as dominant as it is seen in Literate Programming, which documentation model can be hard to adapt[16].

Another effect of the separation is that the documentation structure is not restrained by the structure of the source code or visa versa. The second property, which is the possibility of documenting transverse issues, is a consequence of this. Transverse documentation captures dynamic aspects and interaction between entities such as classes and methods, which is not natural to express in neither interface documentation nor Literate programs.

The third property of Elucidative Programming is navigational proximity. It is realized by inserting links from the documentation to the source code. This brings proximity to the Elucidative program, since it allows you to navigate back and forth between the documentation and source code.

The usefulness of these properties were supported by the three developers. We base this on our observations and their viewpoints as presented in Section 5.3. They found it beneficial to write Elucidative programs. It accommodated their needs for describing software, while at the same time preserving the connection to their primary product, the source code.

In summary it was shown that Elucidative Programming was applicable to the two projects in this experiment. Due to the relatively small size of this experiment this lesson cannot be directly applied on to the industry in general.

However, we expect that other software houses also have a need for documenting transverse issues and for proximity between their documentation and source code. This indicate that Elucidative Programming is applicable in the industry. However, further experiments need to be conducted to verify this.

6.2 Usefulness of the tool

A natural part of evaluating Elucidative Programming in the industry is to evaluate the Elucidator tool used by the developers. The main issue was to evaluate if it was useful to the developers while writing Elucidative programs.

As the tool was only a prototype it was not clear from the start whether the developers would adopt the tool. However, during the experiment it was observed that the developers did adopt the tool. This was further stressed during their remarks in the interviews. It is our impression that one of the decisive reasons for the developers adopting the tool was because the prototype integrated with an editor the developers were accustomed to. If they had been forced to change programming editor for programming purposes we suspect that they would have found the tool harder to use.
While they adopted the tool, they still voiced some problems (see Section 5.4). Two were mentioned several times.

First, they found it cumbersome to have two representations of the documentation, one in the editor and one in the browser. They found it laborious to handle two rather different visual representations of the documentation. Furthermore the necessity of compiling the documentation was found unnaturally.

Second, the Elucidative paradigm and the tool encourage the developers to write documentation while developing. The tool requires the source code to be compilable for it to work properly. Since programs are not always in a compilable state while being developed, this constraint was experienced as being too limiting.

It was suggested that the writing of documentation should be done in a more WYSIWYG-based editor. This could replace the current documentation editor and browser by combining the Elucidative Programming functionalities, such as link insertion, navigation and visual presentation, in a single program. We acknowledge the importance’s of integrating into the developers existing programming environment. Therefore this should not be interpreted as a suggestion for the creation of a complete Integrated Development Environment (IDE). Instead the documentation editor should interface to the developers existing programming editor and environment.

6.3 Structuring and writing Elucidative programs

The three developers voiced that by using the Elucidator tool they gained the advantage of being able to focus on creating contents rather than structure. It seemed to fit well with them that the documentation was based on a hypertext model.

On the other hand, they also expressed the actual act of writing the documentation could be hard. They needed guidance on how to start documenting, how much detail should one include and what to document in the first place. This resulted in administrative overhead, and an increased demand for mental resources. This calls for some kind of predefined structure or method for writing the documentation.

This problem is not foreign to us. We have experienced this earlier and even attempted to provide a set of predefined templates for the documentation nodes[3, Section 4.2, pp. 46]. Our experiences from this were twofold. First, creating exhaustive predefined templates are hard, if not impossible. Second, when the number of templates rises it gets increasingly hard to locate the correct template for a specific situation. Especially if you has not participated in the definition of these.

Even though the structuring of Elucidative programs were not the primary focus of this experiment, some suggestions for possible solutions were expressed by the developers (see Section 5.5).

The most interesting of these suggestions were the notion of Writers Workshop. By conducting Writers Workshop’s developers can combine their experiences and insights, thus establishing awareness of the documentation problems at hand. The team can then jointly try to provide solutions to these problems, e.g., through either templates or guidelines. By conducting these workshops continuously and with appropriate intervals, we believe that the team can improve their documentation skills over time. These concepts and findings are supported by [20, 21].

6.4 Thoughts in Elucidative Programming

As mentioned earlier in this section, the developers found it rewarding to write down their thoughts and ideas. Furthermore, they emphasized that Elucidative Programming also seems to be well suited for supporting group communication. The reason was that their thoughts and ideas could easily be shared with team members.

They also expressed that verbal communication between team members, generally was more efficient. However only one of the developers saw this as a reason for not doing Elucidative Programming. The two other developers instead focused on the possibility of combining the two. They expressed a wish for transforming the verbal group communication into writing, see Section 5.5, to be able to access it in the future.

In summary we support their view on combining the verbal and the written communication in the Elucidator tool. However, we acknowledge the difficulty and problems in documenting the verbal communication. To ensure that this difficulty does not become the reason for disregarding Elu-
cidative Programming, we believe further research efforts should be put into solving this problem.

6.5 Implementation of Elucidative Programming in the industry

One of our goals for this experiment was to see if Elucidative Programming could be implemented in the industry and to find indications on how such an implementation could be done.

All developers found the Elucidative paradigm useful and applicable for their projects. Furthermore, two of the three developers stated that they would continue to use Elucidative Programming in the future.

Throughout the interviews the developers gave some indications to what affected their decision. As seen in Section 5.7 these included motivation, habits, stereotype etc. While we think that these issues play a role, the material gathered through the experiment does not allow us to make a decisive conclusion on how these issues influence a developers decision regarding Elucidative Programming. Instead, further investigations have to be conducted in order to elaborate on this.

Focusing on the two developers who will continue to use Elucidative Programming, some pointers were given on how to implement Elucidative Programming.

They stated that it was not the tool that needed to change in order to implement Elucidative Programming. Instead they focused on the importance of creating awareness on Elucidative Programming. As seen in Section 5.8 they concretely suggested that Elucidative Programming should initially be implemented in a pilot project. The project chosen should have some importance, in order to signal the seriousness of the company’s commitment to Elucidative Programming. If this project succeeds, Elucidative Programming should be able to evolve to other projects.

To elaborate on this, we suggest to choose a respected person inside the organization, which should be responsible for the progress of the implementation. Such a person is often designated a Change Agent[7].

A change agent can perform a number of actions. Two such actions were identified through the experiment. First, the benefit of using a coach to make one of the developers produce Elucidative programs was observed and coaching were furthermore requested by another. This coaching could be performed by the change agent. Second, the change agent could arrange Writers Workshops throughout the project period to increase awareness.

7 Related work

In this section we relate our work to that of others. First, our work is related to experiments concerning the usage of Literate Programming in the industry. Second, an experiment with Elucidative Programming in an educational setting is examined.

7.1 Literate Programming in the industry

Elucidative Programming can be viewed as a practical variant of Literate programming. Therefore, in the context of this paper, studies on how Literate Programming has been used in the industry is interesting.

Such evaluations of Literate Programming in the industry exists. Fischer and Jensen[5] as well as Ramsey and Marceau[19] have all used Literate Programming for a considerable duration and in projects of significant size.

Both emphasize the advantages of doing Literate Programming while developing. Two main reasons are mentioned. First, Literate programs were experienced to ensure the quality of the software. Second, Literate programs were found easier to maintain over time. Ramsey and Marceau provide an example of this: “… an editor for constructing proofs was implemented by a programmer who then left the project. The programmer who took over the job of maintaining the proof constructor read the program in two hours and found herself well prepared to change the code.”[19].

Furthermore, the proximity between source code and documentation in Literate Programming is praised. Ramsey and Marceau states, like the developers in our experiment, that if this proximity was not present the documentation would neither be used nor kept up-to-date.

We also find proximity to be essential. Opposite Literate Programming, Elucidative Programming
do not require the source code and documentation to be stored in the same physical file. Instead documentation is linked to source code and the environment provides navigational proximity. This also makes it possible to document transverse issues.

In conclusion, we find that Elucidative Programming provides the same properties as the authors of the two papers praises. Therefore we believe that their experience with Literate Programming in the industry may apply to Elucidative Programming as well.

7.2 Elucidative Programming and education

The Elucidator tool used to evaluate Elucidative Programming in this paper has been used in an additional experiment [22]. Contrary to our experiment, this paper reports on the usage of Elucidative Programming in student projects. The experiment involved a group of seven Computer Science students which through a period of three months used the Elucidator tool to develop a personalized web advertising system.

Through the experiment, it was found that the Elucidator provided good tool support for Elucidative Programming, and made maintenance of the Elucidative program manageable during development. It was furthermore concluded that Elucidative Programming gave students confidence in their knowledge about the software being developed. The primary reason for this was that the student could find support in the documentation, both when continuing work started by another and when using other parts of the written software. This encouraged them to keep the documentation consistent with the source code.

Interestingly we find that these results generally converge with the result obtained through our experiment.

8 Conclusion and further work

The work presented in this paper reports on the findings of an evaluation of Elucidative Programming in an industrial setting. Three developers, working in two different projects, were asked to use Elucidative Programming in their daily work. During the experiment period, five interviews were held with the developers, gathering information on their views and experiences with Elucidative Programming. Furthermore field observations and quantitative measurements on the produced Elucidative programs were used in the evaluation.

Based on the collected data material it can be concluded that Elucidative Programming is practical usable in the conducted experiment. Similarities in the documentation needs of the experiment case and the software development industry in general, indicates that this result may apply to the industry too. However, due to the limited size of the experiment, no decisive conclusion can be made on this.

The Elucidator tool provided to the developers were generally found useful for writing Elucidative programs. However, some enhancement to the editor part of the tool, would have been desirable. Further research and development should be initiated to deal with this.

The developers participating in the experiment, found that Elucidative Programming was useful and applicable for their projects. Two of the three developers even indicated that they intend to continue using Elucidative Programming in the future. They furthermore stated that the decisive parameter for successful implementation of Elucidative Programming in a development project, is to create awareness on Elucidative Programming in the project. The usage of change agents is suggested by the authors for this purpose.

All though the experiment cannot decisively conclude whether the implementation of Elucidative Programming in the industry is realistic, the experiment indicated that it might be.

We see multiple options for further work. In order to weaken or affirm the conclusions drawn in this paper, it would be interesting to evaluate Elucidative Programming in a bigger experiment. In order to maximize the reliability of such an experiment, it should include multiple projects in multiple companies.

It would furthermore be interesting to conduct an experiment with a significantly longer duration. The purpose of such an experiment would be to explore whether Elucidative Programming in fact eases the maintenance of programs.
Finally we find the question of how to enhance the Elucidator tool interesting. We see two main possibilities for enhancement. First, it could be interesting to bring the two representations of the documentation closer together. We expect that a WYSIWYG-based documentation editor would have to be developed for this purpose. Second, we believe that it will be profitable to include more tasks from the software development process into the Elucidator tool, in order to provide traceability from these to the actual source code. Examples of such tasks could be analysis and design documentation or bug reports.

9 Acknowledgements

The company participating in this experiment is acknowledged for allowing us to conduct the experiment. Especially, the three developers who agreed to spend time on doing Elucidative Programming and participating in interviews throughout the experiment, are gratefully acknowledged.

Finally, Kurt Nørmark and Thomas Vestdam are also gratefully acknowledged for reviewing this paper.

References


