Introducing Elucidative Programming in Student Projects

Thomas Vestdam
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg, Denmark
odin@cs.auc.dk

Abstract

In this paper, we present the first evaluation of Elucidative Programming, in an educational context. Elucidative Programming, is a practical variation of Literate programming, and is aimed at maintenance of documentation of program understanding. By maintaining program understanding in documentation, we can retain thoughts and rationales behind a program from disappearing over time. Maintenance of these thoughts and rationales is important if the program is to be maintained or reused.

We have followed a group of 7 students working on a software project at Aalborg University. The result is a documented software system and interviews with the students that helped us answer a number of questions concerning the usefulness of Elucidative Programming. We have found that it is difficult to persuade students to not only use Elucidative Programming but also use Elucidative Programming consequently. However, we have found that Elucidative Programming is useful, for example, in review situations and in order to give the students confidence in the software under production. In general we have found that Elucidative Programming can be useful in student projects.

1. Introduction

Throughout the life cycle of a program, understating of the program is needed. Different kinds of understanding of the program is present in the minds of the different people involved in various phases in this lifecycle, for example programmers, domain specialist, designers, testers. In this paper, we address the understanding of a program at "code level" (e.g. maintenance documentation or internal documentation). This kind of understanding of software will disappear if it is not preserved in some way. Unfortunately, the program understanding at "code level" is often not preserved, making for example maintenance activates difficult. We therefore advocate for preserving this knowledge of the software in documentation. We term this: *documentation of program understanding*.

The focus of this paper is documentation of program understanding in an educational context. As argued by Sametinger [5] we need to train students in writing good quality documentation. Furthermore, Sametinger argues that if education is to profit from the documentation, we need concepts like Literate programming, hypertext, object oriented documentation and corresponding tools, in order to provide examples of well-documented systems.

Program documentation in form of documents written in natural writing, for example like Literate programming, is often considered an idealistic approach to documentation of program understanding. However, we envision that if the rationales and thoughts behind a program are maintained the quality of the software will be enhanced [6]. Several tools supporting the developer when producing such documentation has been build. One of the newest tools is from a family of tools supporting *Elucidative Programming* [10,11,12] and is called *The Java Elucidator* [14].

In Elucidative Programming, documentation and program exist as separate entities. Documentation and program are presented as hypertext and typed links are used to provide navigation between documentation and program. Hence, explanations in the documentation can address relevant pieces of the program. The documentation consists of nodes addressing specific topics. For example, a node can explain how a task is implemented, explain the rationales behind some decision or explain how some decision is realised in the source code. Elucidative Programming is, in contrast to Literate programming [6], better suited for modern software development because it allows programmers to work collaborative on the fragmented documentation and program. Elucidative Programming and The Java Elucidator is discussed in further detail in section 2.

Elucidative Programming embraces two of the concepts mentioned before, namely Literate programming and hypertext (see section 2 and 5.1). The concept of object-oriented documentation is not directly embraced in Elucidative Programming, but similar properties can be achieved through typed links (see section 2.1) and document templates (see section 2.2). Hence, we believe that Elucidative Programming is promising as a documentation concept for use in education.

To this date, no realistic experiments have been performed in order to support the usefulness of

Elucidative Programming (and the Elucidator) during software development in an educational context. As mentioned in section 5.3, Elucidative Programming has been evaluated in an industrial context. In this paper, we present the results of our first experiments with the use of Elucidative Programming in education. In section 3, a number of questions about the usefulness of Elucidative Programming that were raised before the evaluation are discussed. In section 4, we present the software project that the students have developed and documented, and how we have observed and interacted with the students in order to get some answers to the questions listed in section 3. Section 5 presents our findings and in section 6 we relate our work to that of others. Finally, in section 7 we conclude on the work done and outline future work.

2. Elucidative Programming

Elucidative Programming is a practical variation of Literate programming [6] [9] [18], and is aimed at maintenance of documentation of program understanding. A tool supporting Elucidative Programming and the programming language Java has been build and is named The Java Elucidator [14]. Java is often used in education in introductory object oriented programming courses. This is also the case at the Computer Science Department at Aalborg University.

We here present The Java Elucidator as we did for the students. Details on the practical workings of the tool can be found in [13] and [14]. First, we give a general overview of the Java Elucidator (section 2.1) and then we describe the editor support (section 2.2).

As mentioned in section 1, documentation and program exist as separate entities in an Elucidative Program. The documentation consists of different nodes explaining different aspects of the software. Typed links are used to provide navigation between explanations in the documentation and relevant places in the program. Furthermore, navigation is provided from the program to places in the documentation that address a specific part of the program.

As note, the Java Elucidator can be categorised as a software exploration tool, aimed at reducing the documentation reader's (a programmer or a reviewer) cognitive overhead when studying the program and documentation [19].

2.1. Working with the Java Elucidator

Both the documentation and the program are presented as online documents in an ordinary web-browser. The documentation and the program are not in physical proximity as in Literate programming, but typed links are used to provide *navigational proximity* [14]. The typed links are presented as anchored hyperlinks based on the abstractions in the program and the documentation. Program abstractions are classes, methods, attributes, parameters, local variables and source markers. Source markers are placed in the source code when programmers wish to address other parts of the source code than the mentioned abstractions. Documentation abstractions are documents, sections, figures, and paragraphs.

As mentioned, hyperlinks also provide navigation from anchors in program abstractions to places in the documentation that directly address a specific program abstraction. Activation of a program abstraction link results in a list of links addressing that specific program abstraction. The list includes the types of the links, which is useful when selecting a link to follow. The types used by the students in our experiment were "mentions" and "describes". The first type was used when a program abstraction was just mentioned, and the latter was used when the program abstraction was actually described in detail.

Furthermore, in the presentation of the source code navigation is also provided between occurrences of identifiers and their declaration. For example, when a method call is encountered in the source code it is linked to its corresponding declaration.

Documents in the Java Elucidator are written in an XML based language, called *Edoc*. The Edoc language provides a number of elements (or tags), including source links, documentation links and external links. Edoc also allows inclusion of ordinary HTML elements.

In Figure 1, an example of the online presentation can be seen. The left frame contains some documentation addressing the source code shown in the right frame. The colour of a link indicates the link type. For example, links from the documentation to the source code are red (e.g. CalculateAd and AdShown in the left frame), links from documentation to documentation are blue (e.g. active and requirements in the left frame). Links from the source code to the documentation are black if the particular program abstraction is not documented (e.g. poscounter and RequestString in the right frame), and orange if it is documented (e.g. CalculateAd in the right frame). Links from identifiers to their declaration are underlined.

2.2. Editor support

Editor support for link- and node- creation is provided for the Emacs text editor. Through the editor, the programmer can initiate an *abstraction* of both source code and documentation. An abstraction extracts the program- and documentation- abstraction mentioned in section 2.1 and store information about these in a database. This information is used to provide navigation in the online presentation and when creating links between documentation and program in the editor.

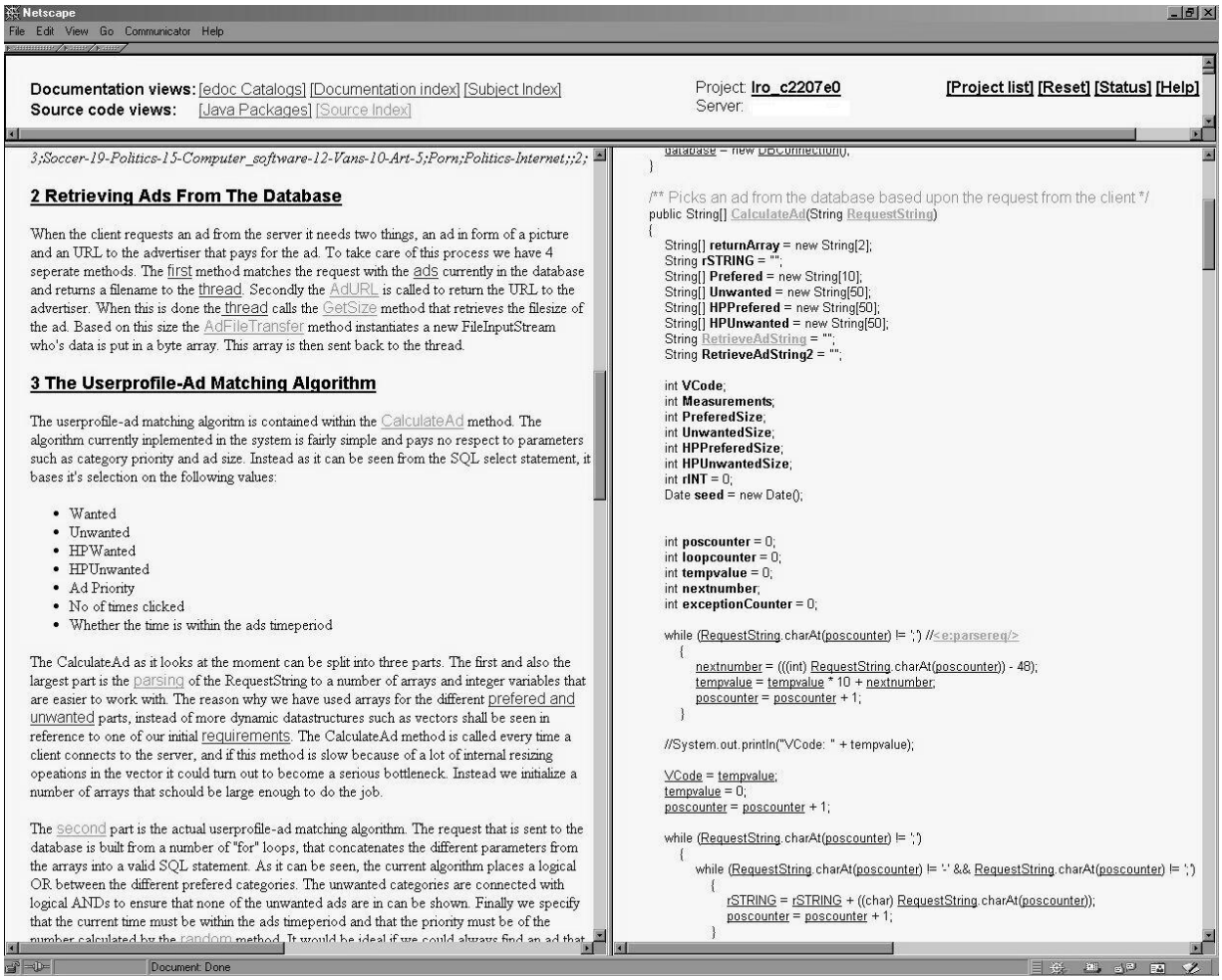

Figure 1, an example of the online presentation from the experiment.

Links are created in two ways. The programmer can mark where the source-anchor should be in the documentation and then select one of the program abstractions in the database. The selected program abstraction then becomes the destination-anchor. Alternatively, the programmer can mark where the destination-anchor should be in the source code or the documentation and afterwards indicate in the documentation where the source-anchor should be inserted.

If changes are made to the source code or files are moved some links may become invalid. Invalid links are reported after an abstraction, and the programmer can easily navigate to the locating of the links in the documentation by mouse clicking or using Emacs shout-cuts.

The editor also supports a variety of node templates. All templates include a standard header with tags defining fields author of the node, status of the node, date of creation, and an abstract of the node. In addition, templates include a number of titled sections depending on the purpose of the template. However, in order to keep things simple the students were advised to only use a template called "essay", which only includes the standard header.

3. Questions about the usefulness of Elucidative Programming

The experiment reported in this paper involves a student project at Aalborg University (explained in further detail in section 4). The project involved 7 students and ran for a period of 4 months, and the actual programming was done in 3 of these months. The result of the student project is a documented and running software system. The students were at beginners' level, meaning that they had no real experience in system development and only little programming experience.

In extreme programming [4] automated tests are considered to give the programmers confidence in the correctness of the program. Whenever the program is changed, the programmer can run prewritten tests in order to ensure that everything still works. We use the term *confidence* in a similarly way about the effect of documenting software. Due to the lack of programming experience, students are often uneasy about the actual programming done in for example a student project. However, if the students write documentation addressing the actual source code, we expect them to gain *confidence* in the source code (and consequently the selected solutions).

Through the process of explaining the source code in natural writing the programmer is prone to consider the source code more thoroughly, as opposed to "just coding away". Through these considerations and possible revisions, the programmer gains confidence in the selected solution. Subsequently, when other programmers read the documentation and study the corresponding code they will quickly gain confidence in the correctness of the source code (or perhaps gain confidence in the incorrectness of the source code). We expect that this confidence can be further enhanced if both tests and their results are included in the documentation.

The considerations above have been summarised in six questions about the usefulness of Elucidative Programming. We have followed the student project, mentioned earlier in this section, in order to get some answers to these questions:

Is Elucidative Programming a suitable means for presenting and evaluating software during reviews? Instead of just reviewing the actual code we hypothesise that program development can benefit from reviews of the documentation as well. The documentation includes the rationales behind the written code and direct access is provided to program details from these explanations. Hence, reviews allow a kind of quality control of the rationales and consequently the source code. For example, experts can review the work of novices in order to suggest improvements, evaluating the soundness of rationales, or simply to educate [16].

Can Elucidative Programming give the developers confidence in their knowledge about the software? If rationales and thoughts behind the program are maintained in the documentation, the documentation can provide the programmers with confidence in their understanding of the software.

Is maintenance of Elucidative Programs manageable during development? As software often evolves rapidly, the documentation needs constant maintenance. We wish to investigate the feasibility of this maintenance when using Elucidative Programming and the Elucidator.

Can documentation of tests included in Elucidative Programs provide further confidence in the software? By consulting the documentation, the developers can gain confidence in the "correctness" of the program or find information on what has been tested and what has not.

Do programs become less error prone when the developer is forced to write Elucidative Programs? As Knuth [6] argues, we can expect an improvement in the quality of the programs by using Literate programming. Elucidative Programming is closely related to Literate programming and we would expect a similar experience of quality improvement.

Does Elucidative Programming ease maintenance? The value of the documentation during maintenance is investigated although the software projects that we are observing are not likely to be maintained. We believe that if program understanding is captured in the documentation, other programmers (such as newcomers or re-users) can gain understanding of the program by reading the documentation.

In general, we have observed the students in order to identify how the Elucidator is (and can be) used during their work. As presented in section 2, Elucidative Programming is quite flexible and can be used in many ways and for many purposes.

4. The experiment

Aalborg University has 25 years of experience in running project-organised studies, where all students are involved in a new project each semester. The approach to problem based learning through project organisation used at Aalborg University has proven to be an effective educational system. The approach is even considered a novel project centred higher education method [15]. The 7 students that took part in our experiment formed a project group in this tradition.

The semester that the 7 students attended consisted of 15 project groups. All these project groups were asked if they would participate in our experiments, but only one group was interested. This is interesting, as we did advocate/advertise for Elucidative Programming and most of the students actually thought that they could benefit from using Elucidative Programming. The reason may have been that the students were already introduced to a lot of new subjects, such as object oriented analysis and design (and system development), object oriented programming in Java, algorithms and data structures and

so on. Elucidative Programming was simply another overhead.

The project members, of the project group that did use Elucidative Programming, were all beginners, as both programmers and system developers. The development method used was the object oriented analysis- and design-method presented in [8].

The project resulted in a personalised web advertising system. The system is a client-server application. The server will be run at the individual advertising company and its main task is to provide the client with ads based on requests coming from the individual client. The client's job is to monitor the browser and extract information about the homepages that the web surfer visits and the ads he/she clicks. Furthermore, based on the user profile the client can request ads from the advertising server whenever the web surfer visits a homepage that contains ads that use the system. Hence, it is possible to present personalized advertising to the web surfer without any form of central registration. This is because all information concerning the web surfer is only stored on the web surfers own computer. This only leaves the advertising server with the task of selecting what ads to display on a site. The entire system consists of about 37 classes of varying complexity.

Each project member participated in the programming- and test- activities. The software was broken into different logical parts and the responsibility of the different parts was distributed among the students. Each student was typically responsible for his/her part(s) of the software throughout the entire project. However, the responsibility of a part of the software was sometimes shared among two or more project members, and sometimes the responsibility was reassigned.

At the beginning of the project, the students were introduced to Elucidative Programming. Emphasis was put on using Elucidative Programming to maintaining thoughts and rationales behind the software in order to share this information amongst the project members. Furthermore, the usage of Elucidative Programming in forthcoming code-reviews was encouraged.

The students were not given any specific documentation model to follow, as we expected the introduction of further complexity beyond the absolute essential would scare of the students. Instead, the students were provided with a *documentation consultant* (the author of this paper). The students were given free hands on how to organise and write documentation, but whenever they were stuck, they could call for the consultant, who would then supply advise on how to attack the problem at hand. We anticipated that it would be difficult getting the students to use the Java Elucidator tool because they would find it hard to figure out how to document. The students were therefore strongly encouraged to contact the consultant whenever they thought they had a problem. This was to ensure that the students would not give up on using Elucidative Programming.

The frequency of consultations was steady throughout the project, at lest once a week. Furthermore, the consultant kept prompting the students in order to ensure that they really did not have any problems, and were still using the Elucidator.

Semi-structured interviews were performed every time the consultant was talking with the students about their experiences with using the Elucidator (In general, inspiration on how to do and plan the interviews was found in [3]). The interviews consisted of simple guidelines for the consultant, such as questions on:

- What the students where doing, and how they used elucidative programming.
- How the documentation was used.
- The amount of time used on writing documentation.
- How the students felt about using elucidative programming and the elucidator (also compared to the last time they were asked).
- How the students felt about writing and reading the documentation.

After each interview, the consultant reflected on the interview and the result of the interview, and if necessary revised the guidelines. The questions listed in section 1 were kept in mind before, during and after each interview.

In order to collect experiences from all project members the consultant made an effort to talk to each student on several occasions throughout the project.

5. Findings

In this section, the observations of the student project are presented. The observations are based on the interviews, described in section 3, and a study of the documentation produced during the project. The produced documentation consists of 37 nodes of various lengths, 151 documentation-to-documentation links, and 137 documentation-to-source-code links. The average word count in the 37 nodes is 674 words, which is a very high word count compared to the traditional amount of program documentation written in this kind of projects at Aalborg University.

Below we present our findings concerning each of the questions presented in section 1:

Is Elucidative Programming a suitable means for presenting and evaluating software during reviews? The students chose to structure the documentation very carefully from the beginning. Hence, first a document containing an overall description of the software, then documents describing the main parts of the system (e.g.

the client and server part, see section 3) and then several documents explaining even smaller parts of the software (e.g. classes). In addition, links where used to "bind" the documentation together and relate the documentation to actual source code. This structure and the actual substance of the documentation give a good overview of the software, and make it possible to go further and further into detail. In a review situation, this can be very useful. The reviewers can become acquainted with the entire structure of the software by first reading the overall descriptions, and then go further and further into the details of the software by studying the more detailed documents. A disadvantage with the current Java Elucidator is that a reviewer cannot make annotations in the online presentation of an Elucidative Program, or create bookmarks in order to return to a specific point of interest.

However, a more serious problem was encountered during reviews. The reviewers of the students software had problems with finding their way through the Elucidative Program. A kind of introduction on paper along with a map of the available documents was requested.

Can Elucidative Programming give the developers confidence in their knowledge about the software? The students were at the beginning of the implementation phase very confident in the way they had structured the software because of thorough analysis- and design-phases. However, as they started documenting the structure of the software, they became even more confident in the structure of software. Two things contributed to this: First, the students found that the structure of the documentation actually reflected the structure of the design. Hence, as source code and documentation was written, the students could see the same overall structure of the system as they did when they designed it. Second, they found confidence in the software because they could consult the documentation in order to be reminded of both details as well as the overall structure of the system. The students could study documentation explaining source code written by another student, for example, in order to make use of or continue work on that particular part of the program. Hence, the Elucidative Program made the students confident that they were doing things correctly when communicating using parts of the software written by others or continuing work started by others.

Furthermore, a few parts of the software were more complicated than the rest of the software. The students found it useful to document these parts of the system before the actual code was written. By documenting ahead, the students felt more confident when writing the code because they knew exactly what had to be done and in what order.

Is maintenance of Elucidative Programs manageable during development? The student project consisted of 37 classes and counting 9000 lines of Java code. Hence, the project is relatively small and the students have not experienced real problems with management and maintenance of the documentation. The documentation in the Java Elucidator is placed in different catalogues (i.e. directories). For example, in the student project documentation of the client- and server-side (see section 3) is placed in two different catalogues. In each of these catalogues, there are documents concerning the client- and server-side respectively along with other catalogues containing documentation of different aspects or details of the particular side. The catalogues helped the students in physically organising the different documents. Furthermore, the students found it useful that the Java Elucidator notified them of invalid links (see section 2.2) when things were moved around or the program was changed. All in all the students found it easy to logically organise their documents as well as reorganise their documents.

Furthermore, the students found navigational proximity quite useful. When changes were made to documentation or source code, navigational proximity was used in order to navigate to all fragments containing links to the part of documentation or source code being changed. Hence, documentation addressing the part of the source code or the documentation being changed was found and correspondingly changed.

Can documentation of tests included in Elucidative Programs provide further confidence in the software? The students have included descriptions of the results of different tests of software in the documentation. However, the tests were not thoroughly described. The descriptions lack explanations addressing both the test code and only a few places the parts of the software that is affected by the test in pointed out. Furthermore, the descriptions lack information on when the tests were performed, what input was used and the resulting output.

Although the students did not describe their test thoroughly, we found that even sparse documentation of tests does give the reader more confidence in the correctness of the software. Hence, if tests are more thoroughly documented we expect them to be quite useful in an Elucidative Program.

Do programs become less error prone when the developer is forced to write Elucidative Programs? Our observation of the students did not give any concrete answers to this question. However, the process of writing about a program, in human writing, before actually writing the program is likely to improve the quality of the program. Through the documentation, the programmer must essentially explain why things were done as they

were to another programmer. Another programmer, or the original programmer, will challenge the explanations in the documentation and consequently challenge the selected solution. Explanations in documentation are often (ideally) easier to understand compared to understanding actual source code. Hence, the process of writing about the selected solutions will force the programmer to think more thoroughly about a selected solution.

The students we observed did, at times, write thorough documentation as described above, but it is difficult to conclude anything as we have no way of comparing or measuring the "correctness" of the program. This would demand more carefully planned tests, if at all practically possible. However, the students felt that in the situations where they wrote the documentation before the source code they got a more clear perspective of the problem at hand.

Does Elucidative Programming ease maintenance? As both structure and details of the software are documented, we believe that the documentation can ease maintenance. Future programmers can go through the documentation accompanied by the source code in order to gain an understanding of the program. However, the student project is never to be maintained so we have no real evidence of the ease of maintenance, but we expect that the documentation produced in the project is quite useful in maintenance situations.

It could be interesting to evaluate the usefulness of documentation in maintenance situations. For example, students could be handed a documented software system and be asked to make some changes to the software, or student projects could collaborate on building a larger software system. However, it will be difficult to run such experiments in student projects because of current structure and themes of the different semesters at Aalborg University.

General use of the tool: The students quickly learned how to use the tool and were from the beginning producing documentation. However, although they structured their documentation well, it seamed as if they could have used more help in doing this. The documentation consultant could have helped the students more, but the students were reluctant to either contact or tell the consultant when they experienced problems. The students suggested that written guidelines or examples of a "good" Elucidative Program could have been useful. Another suggestion from the students was having more templates. Such as, templates for classes or test cases, containing a number of suggested topics to be covered.

The students mainly used Elucidative Programming to document the structure of the software, but some details of the software were also documented. A number of methods and algorithms (including the essential algorithm in the software) were documented. However, all methods were documented using JavaDoc [17]. The students found interface documentation useful, as they often needed interface descriptions of the different parts of the software. The students knew that the software was not going to be maintained, and therefore found no reason to consequently make thorough maintenance documentation of all methods.

In general, the students found Elucidative Programming useful. Their main argument is that Elucidative Programming provides freedom to structure the documentation as one likes. In addition, the students found it appealing that when working on a part of the software the respective documentation could be written without having to “fit” the documentation in a specific context. Instead, links to relevant places in the documentation can be created and a reader demanding more detail can follow these links.

6. Related work

In this section, we will relate our work to that of others. First, our work is related to Literate programming and some experiments with the use of Literate programming in education. Second, we look at an experiment with the introduction of elucidative programming in an industrial context.

6.1. Literate programming and education

Shum and Cook have performed an experiment that shows that students like using Literate programming and that the use of Literate programming results in more thorough documentation of programs in student assignments [2]. The experiment included two types of student assignments: assignments solved using traditional programming and assignments solved using Literate programming. In the traditional programming assignments, the program (including code comments) was supplemented by external documentation such as design diagrams and pseudo code. Shum and Cook found that the documentation produced in assignments that used Literate programming had fewer inconsistencies than the assignments using traditional programming. This is due to the proximity property of Literate programming; it is simply difficult to ignore the documentation when it is present right beside the program code. However, the assignments used in [2] are assumed smaller than the student project in our experiment, and the assignments are not collaborative. The students in our experiment were carefully about keeping documentation and source code consistent, because they often used the documentation in order to be reminded of details. Furthermore, the navigational proximity property of Elucidative Programming makes it easy to quickly check whether the

documentation addressing some part of the program is up to date.

Cockbrun and Churcher have proposed and designed a prototype of a Literate programming environment for introductory programming [1]. The tool supports students when making assignments and assessors (e.g. the teacher or teaching assistants) can use the environment when evaluating the students work. An assessor can load a student project into the environment and view the overall structure of the Literate program (i.e. the chunks in the Literate program), the program and the documentation. Hence, poor structure and missing documentation can quickly be identified. We have found that the documentation produced by the Java Elucidator can be used not only by an assessor, but also in review situations. Reviewers can browse through both documentation and program, using the various facilities explained in section 2 and identify poor structure and missing documentation.

6.2. Elucidative Programming in the industry

The experiment described in this paper involves a student project, but similar experiments are also being carried out in an industrial context. Recently, an experiment involving three programmers was performed in a small software company [7]. The experiment ran for two month and based on interview with the programmers it was concluded that Elucidative Programming was useful in the particular experiment. Andersen and Christensen [7] speculate that Elucidative Programming the experiment indicate that Elucidative Programming can be generally useful in industrial contexts, although their experiments do not give direct evidence for this.

The interviews with three programmers also gave rise to five lessons learned. These lessons summarise the usefulness of and how to implement Elucidative Programming in an industrial setting. For example, how *change agents* could be applied in order to create awareness of Elucidative Programming, or Writers Workshop could be held in order to improve the programmers documentation skills.

Surprisingly, the results of the interviews presented in [7], are generally similar to the experiences gained through the experiment presented in this paper.

7. Conclusion and further work

The work presented in this paper can be seen as a preliminary evaluation of Elucidative Programming, and as a first attempt to introduce Elucidative Programming in education. We have followed a student project, consisting of 7 students, at Aalborg University, using Elucidative Programming in their project work. The result is a documented software system and interviews with the students that helped us to answer a number of questions about Elucidative Programming.

In general, we found that it is very difficult to persuade students to use Elucidative Programming. The main reason is that the students felt that Elucidative Programming was just another overhead in their already busy schedule. Furthermore, this kind of documentation is not a traditional part of the education, in the same way, that documentation of object oriented analysis and design is.

The students we did persuade to use Elucidative Programming used it to document the structure of their software and essential methods and algorithms. The Elucidative Programming ideal is to thoroughly document and discus the actual source code, but the students had a tendency to omit this and resort to interface documentation instead.

However, we did find that Elucidative Programming is a suitable means for presenting and evaluating software during reviews if the reviewer is proper introduced to the Elucidative Program.

Elucidative Programming did give the students confidence in their knowledge about the software under development. This confidence was attained mainly because the individual student could find support in the documentation when either continuing work started by another or when communicating with parts of the software written by others.

The Java Elucidator provides good tool support that makes maintenance of Elucidative Programs manageable during development. Furthermore, the students we observed were careful about keeping the documentation and program consistent as they used the documentation actively.

We have found that if documentation of tests is included in the Elucidative Program, there is a potential of providing the programmers with further confidence in the correctness of the software. Furthermore, the tests will become a natural and active part of the documentation.

Our observations and interviews do not give any answer to whether Elucidative Programming eases maintenance or whether programs become less error prone. However, the documentation produced in the experiment does, in our opinion, eases maintenance of the produced software, and the process has lead to more thorough thoughts and rationales behind the program.

Generally, we find Elucidative Programming useful in student projects and the students are satisfied with the Java Elucidator (but they have suggested some improvements). However, further experimentation with Elucidative Programming in student projects is needed to backup the results presented here.

The next step is to run more experiments as the one presented in this paper. We will use the experiences

gained during the experiment when planning and running the new experiments. The Elucidative Program produced by the students in the project presented in this paper can be used to demonstrate the usefulness of Elucidative Programming to other students. A concrete example of the use of Elucidative Programming can persuade other students to start using Elucidative Programming as well.

We also intend to design further enhancements of the Java Elucidator: tool support for more automatic generation of documents by using knowledge of the software, integration of interface documentation in Elucidative Programs and tools for annotating documentation while reading. Furthermore, we believe that it is possible to produce documentation guidelines in form of *documentation patterns*. We expect these patterns to be useful for programmers who need some help in getting started on documenting their software, such as students at beginners level.

8. Acknowledgements

The 7 students who agreed to use Elucidative Programming in their project and participate in our experiment are gratefully acknowledged for their effort.

The post-graduate students Max R. Andersen, Claus N. Christensen and Kristian L. Sørensen are acknowledged for the construction of the Java Elucidator [13]. In addition, Kurt Nørmark is gratefully acknowledged for reviewing this paper.

9. References

[1] A. Cockburn and N. Churcher, "Towards literate tools for ovice programmers", In ACM Australasian Computer Science Education Conference '97, ACM Press, Melbourne, Australia. 2-4 July, pp. 107-116, 1997.

[2] S. Shum and C. Cook, "Using literate programming to teach good programming practices", SIGSE Bulletin: The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education, 26(1), pp.66-70, March 1995.

[3] Michael Quinn Patton, "Qualitative Evaluation and Research Methods", Second Edition, 1990, SAGE Publications, ISBN: 0-8039-3779-2

[4] Kent Beck, "Extreme Programming Explained: Embrace Change", Addison Wesley Publishing Company, 1999

[5] J. Sametinger, "The Role of Documentation" in Programmer Training, Programming Languages: Experiences and Practice, Mark Woodman (Ed.), Chapman & Hall, 1994.

[6] Donald E. Knuth, "Literate Programming", The Computer Journal, vol. 27(2), May 1984, pp. 97-111.

[7] Max Rydahl Andersen, Claus Nyhus Christensen, "Evaluating Elucidative Programming in an Industrial Setting", Aalborg University, 2001 (submitted to ICSM 2001).

[8] Lars Mathiassen and Anders Munk-Madsen and Peter Axel Nielsen and Jan Stage, "Objekt orienteret analyse og design", Marko Publishing, 1998.

[9] Donald E. Knuth, "The WEB System of Structured Documentation", STAN-CS-83-980, Department of Computer Science, Stanford University, September 1983.

[10] Kurt Nørmark, "Requirements for an Elucidative Programming Environment", Eight International Workshop on Program Comprehension, June 2000.

[11] Kurt Nørmark, "An Elucidative Programming Environment for Scheme", In Mughal and Opdahl (editors), Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research, May 2000, pp. 109-126.

[12] Kurt Nørmark, "Elucidative Programming", Nordic Journal of Computing, vol. 7(2), 2000, pp. 87-105.

[13] Max Rydahl Andersen, Claus Nyhus Christensen and Kristian Lykkegaard Sørensen, Søren Staun-Pedersen, Vathanan Kumar, "Internal documentation in an Elucidative environment", Aalborg University, June 2000, Available from http://dopu.cs.auc.dk.

[14] Kurt Nørmark, Max Rydahl Andersen, Claus Nyhus Christensen and Kristian Lykkegaard Sørensen, "Elucidative Programming in Java", Available from http://dopu.cs.auc.dk.

[15] Finn Kjersdam and Stig Enemark, "The Aalborg Experiment -- Project Innovation in University Education", Aalborg University Press, Niels Jernesvej, DK-9220 Aalborg, Denmark, 1994.

[16] G. T. Leavens, A. L. Baker, V. Honavar, S. M. LaValle and G. Prabhu, "Programming is Writing: Why Programs need to be Carefully Read", Journal of Mathematics and Computer Education, 32(3), pp. 284-295, Fall 1998.

[17] Lisa Friendly, "The Design of Distributed Hyperlinked Programming Documentation", Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, Sylvain Fraïssé and Franca Garzotto and Tomás Isakowitz and Jocelyne Nanard and Marc Nanard (editors), 1995.

[18] Norman Ramsey and Carla Marceau, "Literate Programming on a Team Project", Software Practice and Experience, vol. 21 (7), July 1991, pp. 677-683.

[19] M.-A.D. Storey and F.D. Fracchia and H.A Muller, "Cognitive design elements to support the construction of a mental model during software visualization", In Proceedings of the Fifth International Workshop on Program Comprehension March 1997, pp.17-28.