Aspects of Internal Program Documentation— an Elucidative Perspective

Thomas Vestdam and Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
{odin,normark}@cs.auc.dk

Abstract

It is difficult and challenging to comprehend the internal aspects of a program. The internal aspects are seen as contrasts to end user aspects and interface aspects. Internal program documentation is relevant for almost any kind of software. The internal program documentation represents the original as well as the accumulated understanding of the program, which is very difficult to extract from the source program and its modifications over time. Elucidative Programming is a documentation technique that originally is inspired by Literate Programming. As an important difference between the two, Elucidative Programming does not call for any reorganization of the source programs, as required by Literate Programming tools. Elucidative Programming provides for mutual navigation in between program source files and sections of the documentation. The navigation takes place in an Internet browser applying a two-framed layout. In this paper we investigate the applicability of Elucidative Programming in a number of areas related to internal program documentation. It is concluded that Elucidative Programming can solve a number of concrete problems in the areas of program tutorials, frameworks, and program reviews. In addition we see positive impacts of Elucidative Programming in the area of programming education.

1 Introduction

There is a broad need for documentation of programs. Overall, program documentation is concerned with the description of structure, purpose, operation, maintenance, and data requirements for a computer program (according to The American Heritage Dictionary of the English Language). All of these concerns are oriented towards representing and keeping an understanding of the program such that various processes that deal with the program can be carried out in a way that fits the needs and qualifications of the involved people.

There are many kinds of programs. For our purposes we will distinguish between applications, library modules, and frameworks. Applications are programs that serve a purpose for end users, library modules are software building blocks, and frameworks are general applications that need further elaboration and specialization to serve as applications.

At an overall level we will distinguish between user documentation, interface documentation, and internal documentation of programs. User documentation explains how end users are intended to operate the program. Interface documentation is concerned with use of library modules. Internal documentation is a broad category of descriptions that represent understandings of selected aspects of the program, all of which are important to keep the program alive. Figure 1 shows how the different kinds of documentation apply to the different kinds of programs mentioned above.

In this paper we will confine ourselves to discuss and explore internal program documentation. As it appears from the figure, internal program documentation is relevant for all kinds of programs. In our past efforts we have worked with internal documentation using the Elucidative Programming paradigm, and we have mainly focused on internal documentation of applications. As a consequence of the observation of the broad relevancy of internal documentation, we will in this paper turn our attention to different aspects of internal program documentation and their relation to application programs, frameworks, and library modules. In other words, we will explore the shaded area of figure 1 in order to understand the roles of internal documentation across different kinds of programs.

Internal program documentation may appear in several different ways. We limit ourselves to persistent documentation, in contrast to more transient documentation that tends

	Applications	Frameworks	Libraries
End user documentation	✓	✓	÷
Interface documentation	÷	✓	✓
Internal documentation	✓	✓	✓

Figure 1: *A rating of the relevance of different kinds of documentation seen in relation to different kinds of programs.*

to disappear as time passes (e.g. informal design diagrams, notes on scraps of paper, or drawings on a black board). Persistent documentation may be represented in several forms, and on different media. Of these, the textual documentation in natural language is bound to be of special importance. However, diagrammatic means of expressions also play a significant role in documentation of computer programs. In addition, audio and video representations of computer documentation cannot be ignored.

At this level of the discussion, we will dwell on the role and the importance of program understanding in relation to the lifetime of a program. As an overall observation, any program is likely to be modified in various ways during its lifetime. Huge resources are devoted to such modification processes in the software industry. Any kind of program modification requires some degree of program understanding in order to carry it out safely. For example, consider maintenance activities carried out by other programmers than the original authors of a program. Internal documentation can be seen as an investment that is intended to minimize the total lifetime cost of the software.

In the early phases of the development process, many efforts are used to establish a firm understanding of the problem to be solved by the program, including an understanding of the context in which the program is intended to operate. The description of such an understanding is often called analysis documentation. When the program takes shape, the structure of the program together with overall relations between the program building blocks becomes important. A given program structure facilitates certain software qualities in the remaining part of the program's lifetime. The understanding of these structures is often referred to as design or architectural documentation.

The implementation of the source program calls for an exact and detailed understanding of the problems and their solutions, as formulated in a programming language. Notice in this context that the source programs are likely to carry and represent most of the program understanding during the remaining lifetime of the software; The documentation of the early development efforts (analysis and design documentation) tend to degenerate during the lifetime of the program, because this documentation is only rarely kept up to date in relation to the performed program modifications.

It is worth a consideration how much of the understanding from the implementation process to turn into internal program documentation. If too little internal program documentation is produced, time and efforts will be used in the future to re-establish the understanding from the raw source programs (using a detective's working style and reverse engineering methods [30]). If too much internal documentation is produced, chances are that the investment will never pay back again, and that future program maintainers will be disoriented facing too much information. In addition, it is not just a question of documentation volume; The real challenge is what to describe and explain, and not least how to maintain the internal documentation during the program's lifetime.

Finally, the program testing efforts call for special purpose internal documentation. The understanding of the testing process is relevant for regression testing, as the step following program modifications. Hence, explanations such as what has been tested, how and why, can contribute to the understanding of a program.

As just argued, internal program documentation plays important roles in the program's maintenance phase. But internal program documentation may serve other purposes as well. In the slipstream of Literate Programming [8] it has been observed that literate programs are better than 'normal programs', in the sense that they contain fewer errors. The reason is believed to be that careful argumentation and description of the program, done during the implementation process, eliminates a number of errors and misunderstandings. In other words, we hypothesize that internal program documentation done while programming keeps the process on a better track [29]. Also, well-written internal program documentation adds to the immediate value of the software in less tangible ways. The internal program documentation keeps parts of the software together, provides starting points when re-approaching the software, enumerates well-known loose ends, etc.

Adherence to established software engineering techniques (such as modularization and information hiding) are only seldom questioned by today's software developers. As a contrast, there is much doubt and disbelief when it comes to writing internal program documentation. As a typical myth, programmers find it boring and non-interesting to write internal program documentation of high quality, although the same people often spend much time writing reports about other, or perhaps even related topics.

The state of affairs described above can in part be blamed schools and universities that educate the software developers. Too few efforts are used to emphasize and reward good internal program documentation. In the same vein, too little time in the educational process is used on program modification tasks in contrast to implementation from scratch. In turn, the program documentation research community can be blamed for not helping students and the practical software developers to deal with the problems and challenges described above.

In the rest of this paper we will discuss how programmers can use Elucidative Programming for a broader range of internal documentation purposes than previously realized. In section 2 we will present the Elucidative Programming paradigm and the status of the two tools currently supporting the paradigm. In Section 3, which is the main contribution of this paper, we investigate a broader applicability of Elucidative Programming. We will discuss some applicability in relation to the shaded area of figure 1, but also discuss more general applications of Elucidative Programming in the area of computer science education, software reviews, and programming environments. Finally we conclude on our findings so far.

2 Elucidative Programming

Elucidative Programming emerged because Literate Programming was found to be unfit for modern software development [14]. Elucidative Programming can be seen as a practical variant of Literate Programming, building on both inspiration and experiences gained from Literate Programming. Details on the background of Elucidative Programming can be found in [14].

The basic idea in Elucidative Programming is to link explanations and program fragments together instead of using physical embedding. An elucidative program is presented and explored in a WEB browser, which exploits the linking in a two-framed setup with mutual navigation between the explanations and relevant fragments of the program. See figure 2 for an example. With this setup we can associate explanation and documentation without forcing the programmers to reorganize or re-chunk their programs. Recently, we have refined this support to tutorials, which in some sense brings Elucidative Programming closer to the starting point of Literate Programming, without compromising the idea of separate documentation and source programs. The common elucidative setup is illustrated in Figure 2. The top frame contains various navigational features, such as an index of the program abstractions and a documentation index. The left frame presents the explanation and the right frame presents program files (i.e. source code).

The association between explanation and documentation is provided through use of typed hyperlinks. These typed hyperlinks are anchored both at the source end and the destination end, and they can effectively be navigated in both directions. This provides navigation between a piece of explanation in the left frame and the involved program fragment in the right frame. In addition, navigation is also provided between applied and defining name occurrences in the source program.

As seen in figure 2, both the documentation and the program are presented as on-line documents in an ordinary WEB browser. Documentation and program are not in *physical proximity* as in Literate Programming, but the typed links are used to provide *navigational proximity* [15] between explanations and program fragments. As a major challenge, we wish to support navigational proximity in both a browser context and in the context of the program development environment.

Currently, two different tools supporting Elucidative Programming exist [13, 15], one for the programming language Scheme [7] and one for Java [24]. Both of the *elucidators* are supported by Emacs [22, 23] at the editor level. The main purpose of the editor support is to assist the programmer when creating links from the documentation to source code. In order to do this, the elucidator must *abstract* both source code and documentation. An abstraction gathers information about program units such as functions, methods, and classes.

Fundamental for both elucidators is that documentation and program exist as separate entities, and typed links provide navigation between explanations in the documentation and relevant places in the program. The program parts that can be addressed are the abstractions in the program (syntactical elements such as functions, methods, classes, variables, etc.) or special comments in the program serving as anchors. We term these *source markers*.

In the following two sections, we will present the two tools. We will focus on their individual differences and their current status.

2.1 The Scheme Elucidator

An elucidative Scheme program [13, 11] is organized as a *documentation bundle* consisting of a number of programs, a documentation unit and a setup file. A simple markup language is used when writing the documentation,

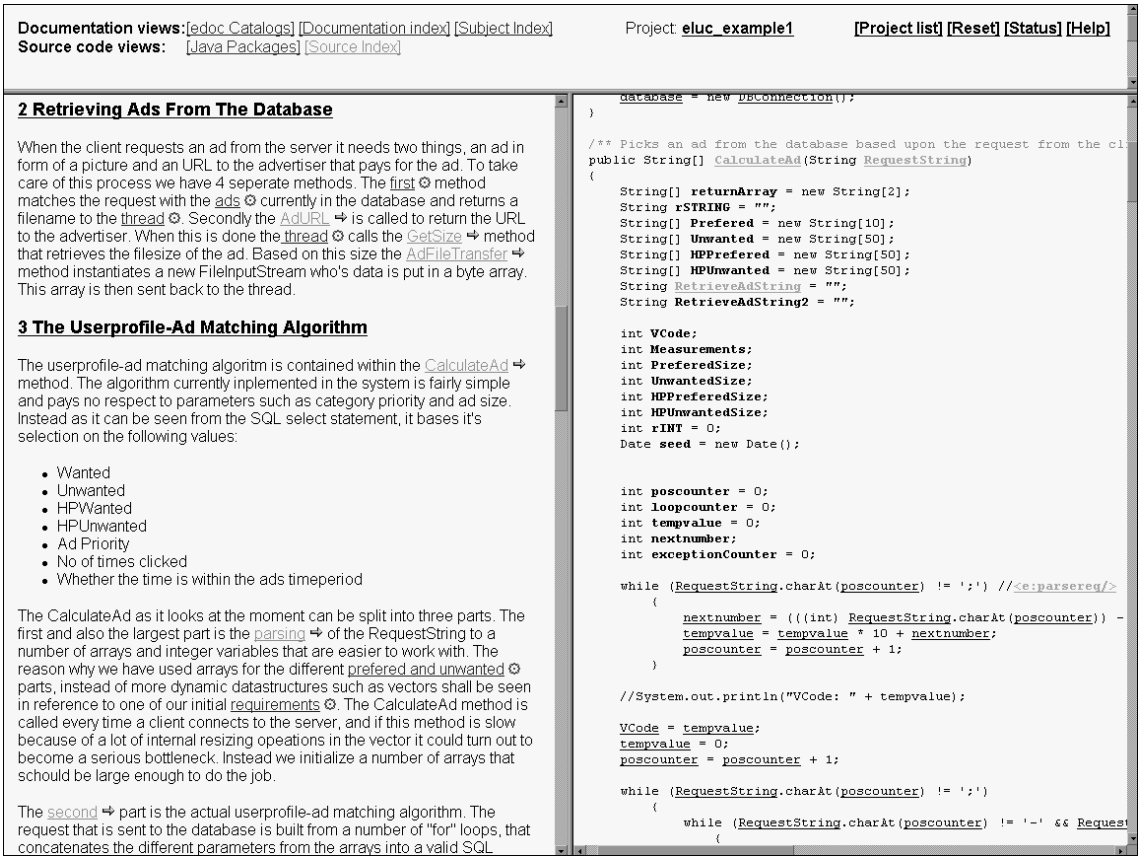

Figure 2: *An example of the presentation of an elucidative program in a WEB browser.*

in order to specify structure such as chapters and sections, and in order to specify links to source code.

The markup language is primarily designed with terseness in mind, such that the footprint of the markup does not disturb the textual contents too much. The Scheme Elucidator operates with weak and strong link types. Weak links are used in documentation that just mentions a program part, whereas strong links are used in documentation that discusses a given program part in more detail.

In addition, links are provided from standard Scheme functions to explanation in the Scheme language report [7]. The Scheme Elucidator is fully operational, but has not yet been used in software projects outside our research unit.

2.2 The Java Elucidator

In the Java Elucidator [15] the documentation bundle contains more than one documentation unit. Each documentation unit is an XML document explaining a specific aspect of the program, and the markup used is essentially very similar to that of the Scheme Elucidator, although it is more verbose.

The presentation of an elucidative program is produced by a web-server utilizing a database containing the information from the last abstraction of the documentation bundle. This setup makes collaborative Elucidative Programming more attractive. As both documentation and program source code are available as an on-line resource, a group of geographically distributed developers can easily share their collective understanding of the program [15]. In addition, the setup provides *context views* for all program abstractions (for example, a list of all documentation units linked to a specific program abstraction, or a list of all parts of the source code using specific variable, class, or method).

The work on the Java Elucidator is ongoing. Future work will involve an improvement of the elucidative program development environment (using TogtherJ [26]).

2.3 Experiments

An experiment has been conducted in an educational context as a preliminary evaluation of Elucidative programming, and as a first attempt to introduce Elucidative programming in education [28]. The experiment involved 7

students (at Aalborg University, Denmark) working on a student project for 4 months. The students made a software system (20 classes of varying complexity), and they used Elucidative Programming throughout the project in order to document the source code. The students used the Java Elucidator to document the structure of their software as well as essential methods and algorithms. Empirical data was collected through qualitative interviews with the students, and an evaluation of the produced documentation.

We found that Elucidative Programming gave 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 dealing with parts of the software written by others. In addition, Elucidative Programming was found to be a suitable means for presenting and evaluating software during reviews.

The students found that the Java Elucidator provides good support making maintenance of elucidative programs manageable during development. Furthermore, the students used the documentation actively, which required that they were careful about keeping the documentation and program consistent.

The students mostly used Elucidative Programming to document the structure of their software and some essential methods and algorithms. The idea of Elucidative Programming is to document the actual source code, but the students had a tendency to omit this and resort to interface documentation instead.

A similar experiment has been conducted in a software company [1]. This experiment involved three programmers using Elucidative Programming in their respective projects for six months.

In general, Elucidative programming was found to be very useful in an industrial context. The programmers spoke positively about their experiences with Elucidative Programming. Elucidative programming allowed them to document the individual parts of a program as well as to document the complex communication in one of their systems. This is in accordance with the experiences of the students, as they also found it easy to explain how different parts of their program were working together in order to perform certain tasks.

A major concern was discovered in the industrial application of elucidative programming. Due to ”bad” habits, as discussed in section 1, the programmers seldom got around to writing internal documentation addressing actual source code. Two of the programmers used Elucidative Programming to produce a number of documents addressing different aspects of the program, but the documents were often not completed.

Based on the two experiments and literature study a number of guidelines in form of documentation patterns has been produced [29].

3 An Elucidative Perspective on Internal Program Documentation

In this section we will review the area of internal program documentation in relation to software tutorials, program frameworks, reviews, teaching habits, and support from integrated development environments. We will, in particular, discuss these areas seen from the perspectives and the potentials of Elucidative Programming.

Reviewing these five areas serve the common purpose of broadening the applicability of Elucidative Programming. In addition, these five areas cover our recent and current research efforts. Hence, it is our aim to discuss the area of internal program documentation more carefully than done before [14, 12, 15] in the light of the Elucidative Programming ideas.

3.1 Program tutorials

We define a *program tutorial* as a document that informs the user about the internal properties of a program, especially with the goal of using the program for construction or composition of another program. Program tutorials, seen as internal documentation, are therefore primarily relevant for frameworks and libraries, cf. the overview in figure 1.

The intended usage of a framework, a library, or a program may have an impact on both design and implementation. Intended usage is often an important concern during the design phase, for example as UML use-cases [18] or user stories in Extreme programming [2]. When developing a framework or a library, *example programs* are often written in order to test the software. These programs may serve as examples of how to use the software (or can be modified to become examples) and can prove useful in upcoming tutorials. Hence, the developers can from the beginning start to document how to use the software, extend it, or reuse it. The tutorials are a useful means for developers to maintain an understanding of how the software is going to be used. In addition, by writing tutorials from the beginning, potential problems connected with usage of the software may be detected at an early stage. It can be advocated to apply user scenarios to guide the design process [5], but making them part of the documentation makes them even more valuable.

Tutorial documentation maintained in separate documents introduces a mental gab between the program and the documentation. Even if precise references between documentation and source code exist, the user must do manual work to “carry out a trip” from a place in the documentation to a relevant place in source code, or vice versa.

Literate programs may serve as program tutorials, in which documentation and source code are interleaved in arbitrary sized chucks chosen by the developer [19]. This brings documentation and source code in close proximity. But as argued in section 2, the Literate Programming tools achieve this by a physical embedding of program fragments in the documentation text. In our opinion, this is not a realistic alternative in many software development projects today.

Using the ideas of Elucidative Programming, a program tutorial can be written in two different styles: *linked style* and *in-lined style* respectively. Using the linked style, a tutorial text is separate from the program, but connected to the program via relations that are rendered as bi-directional links by the elucidator tool. The linked tutorial style comes as a straightforward exploitation of the key ideas of Elucidative Programming and the principle of navigational proximity. Because of the two-framed presentation of an elucidative program, the reader keeps the context of the explanation when navigating to a program fragment, and similarly the other way around. Still, however, it calls for explicit action from the reader to exploit the connection between tutorial explanations and program fragments. Moreover the mutual navigations within the two frames may disturb the reading rhythm, and the underlying understanding of the matters.

To remedy these problems we support the in-lined tutorial style in the Java elucidator. The main idea behind the in-lined tutorial style is to extract specific parts of the source code, and to present these as part of the tutorial text. Hence, in-lined tutorials are a different way of addressing the actual source code, beyond referring to specific program entities from the documentation. The extraction is done by the elucidator tool, based on a specification of what to extract. In this specification the documentation author addresses the abstractions of the program to be extracted, as described in section 2.2. It is possible to extract a class, a method, a block, or individual lines of the program. In addition, a level of detail can be specified, for example, methods can be presented in full detail or only as an outline (i.e. method signature). We do not believe that manual extraction of source code fragments is a realistic option in a software development process where both the documentation and the programs are frequently changed.

Using the in-lined tutorial style, the reader can concentrate on a fixed thread [27] of understanding, which is prepared by the author of the tutorial. The price paid for this is more voluminous documentation, but as long as the information is on-line only (as opposed to being printed on paper) this is not seen as a problem. As a common need, the reader often wishes to understand the context of the in-lined program fragment. Therefore we always render links to the full program from the presented program excerpt. With this, it is possible from the tutorial to navigate to the program fragments in their full and original program context. In this way we combine the power of the in-lined tutorial style with tutorials using the linked style.

3.2 Frameworks and design patterns

Documentation of an object oriented framework mainly emphasizes the architecture, class responsibilities, interaction between classes, and static dependencies between classes. Programmers who use the framework need in-depth information such as information on how to extend the framework (e.g. overriding methods), information on relationships and interaction between classes, as well as general information on external interfaces.

Design patterns represent elements of mini architectures, which represent good and well-proven solutions to recurring problems. Design patterns are seen as a valuable resource for programmers evolving or mining a framework (looking for reusable parts, for instance) [4]. As such, design patterns may play an important role in the area of internal framework documentation.

Framework documentation consists of a diversity of documents intended for specific audiences. Butler and Dénommée introduce guidelines to what kind of documentation an application developer needs when working with a framework [3]: (1) An overview of the framework is needed, (2) a set of example applications are needed as well, and (3) cookbook recipes should be available (e.g. tutorials). However, these types of documentation do not quite fit the needs of framework developers and maintainers. Internal documentation of frameworks can be produced by recording instantiations of design patterns during development [17]. Hence, framework developers can produce documentation by recording instantiations of design patterns ('documenting by designing' [17]).

The main contribution of Elucidative Programming to the area of framework development and programming based on design patterns is related to description of *transverse issues* [29, 15]. A transverse issue is a description that involves details from separate program units. As noticed above, the relations between classes and methods play a key role in the understanding of frameworks and many design patterns. As such, it is not sufficient to describe a single program fragment. The crucial points of understanding stems from observations that relate to more than one program fragment. The ideas of Literate Programming and in-lined tutorials, as discussed in section 3.1, fall short in meeting these demands. Elucidative Programming, which may support several program references from each paragraph in the document, is well suited for this kind of documentation. The descriptions, as represented in the documentation, keep the related program fragments together - not by formally defined relations - but in the natural-language paragraphs of the written documentation. Figure 3 shows an example

of documentation of a design pattern instance, where a few paragraphs of documentation relates a number of classes in the way described above.

3.3 Reviews

Elucidative Programming may play a role in software reviews in two different ways. First, a software review should be oriented towards the wholeness of programs and documentation, and not just the programs. In other words, the quality of the documentation is to be reviewed side by side with the program as such. Second, Elucidative Programming may play an interesting role in the review process itself, because it allows the reviewers to address details in the code in ways, which are difficult to approach without tool support. We will now, in turn, discuss both of these observations.

During the implementation of software, quality assurance of the actual code can be achieved through periodical code reviews. We believe that reviews can also be applied to assure the quality of the internal documentation. Such reviews can ensure documentation coverage, comprehensibility, correctness, and in general keep documentation and source code in accord. This concurs with the findings of Siy and Votta [21], where data from 130 modern code inspection sessions is analyzed. In these sessions 60% of the issues raised were soft maintenance issues (and 18% were true code defects). Of the soft maintenance issues almost 50% of them where concerned with the documentation, such as calls for clarification, correction, and documentation of future work. Furthermore, roughly 50% of all source line changes involved code comments. Besides strengthening the quality of the documentation we envision that reviews can bring documentation-awareness into a development team and make documentation an important artifact, on equal footing to the source code.

It is well known that it is very difficult to keep comprehensive, internal documentation up-to-date while modifying a program. The main challenge is to identify all the relevant places in the documentation that are affected by the documentation. As a consequence the programmer will often switch to writing *differential documentation*, i.e. documentation which describes the differences between 'an original program' and 'the modified program'. If a software development project is to benefit from documentation reviews, well-structured internal documentation must be submitted to a review board. This requires that the development team take the time needed to rewrite and restructure the documentation on some occasions. As part of such a restructuring process, differential documentation must be transformed into ordinarily structured internal documentation.

As a second point in relation to software reviews, we see the need for written program reviews. Such reviews address details in the software. In general, it is difficult to *write about a program*, and in particular to address details in the program from written statements. Often, the best solution is to refer to details using file names and line numbers of program printouts. Elucidative Programming contributes with a solution, were the reviewers are able to address program details from a piece of text, using the means of identification which are used to connect documentation and fragments of an elucidative program. With such written, 'elucidative reviews' it is possible for all the involved readers to navigate back and forth between the reviews and the involved programs. During the reading of a source program it is particularly attractive to be able to spot fragments, which have been commented during the review. Elucidative programming supports this via the marking of program units to which there exists associated pieces of reviews.

3.4 Educational Coverage

Schools and universities that educate software developers seldom teach how to write internal documentation, or for that matter, why internal documentation is important. Often students are only presented with analysis and design documentation, and as stated before, this kind of documentation tends to degenerate during the lifetime of the software.

Some computer science educators have expressed serious concerns about the teaching and practicing of *intangible program qualities*, such as those reflected by internal program documentation [10]. Without focus on such qualities the authors claim that many students will use a *generate and test method* of programming, which is characterized by ad hoc methods, minimal planning, and a "trial and error" approach to problem solving. Experiments with introducing students to Literate Programming have been performed [20]. One of the experiences gained from these experiments is that Literate Programming encourages the students to write more consistent documentation. In addition, it is believed that the Literate Programming paradigm forces the students to work more systematically, hereby avoiding the worst consequences of the *generate and test method* of programming. Furthermore, chances are that students will learn the value of the documentation when writing about their own programs.

Program textbooks typically include small toy programs. With the advent of Literate Programming it has been demonstrated that large and *real programs* can be published in textbooks of their own [9] on the ground of academic interests in the source programs. We have recommended use of elucidative versions of *real programs* in future teaching materials [16]. Overall, we are convinced that there is more to learn from a single *real program* than from a number of small toy programs. With Literate and Elucidative

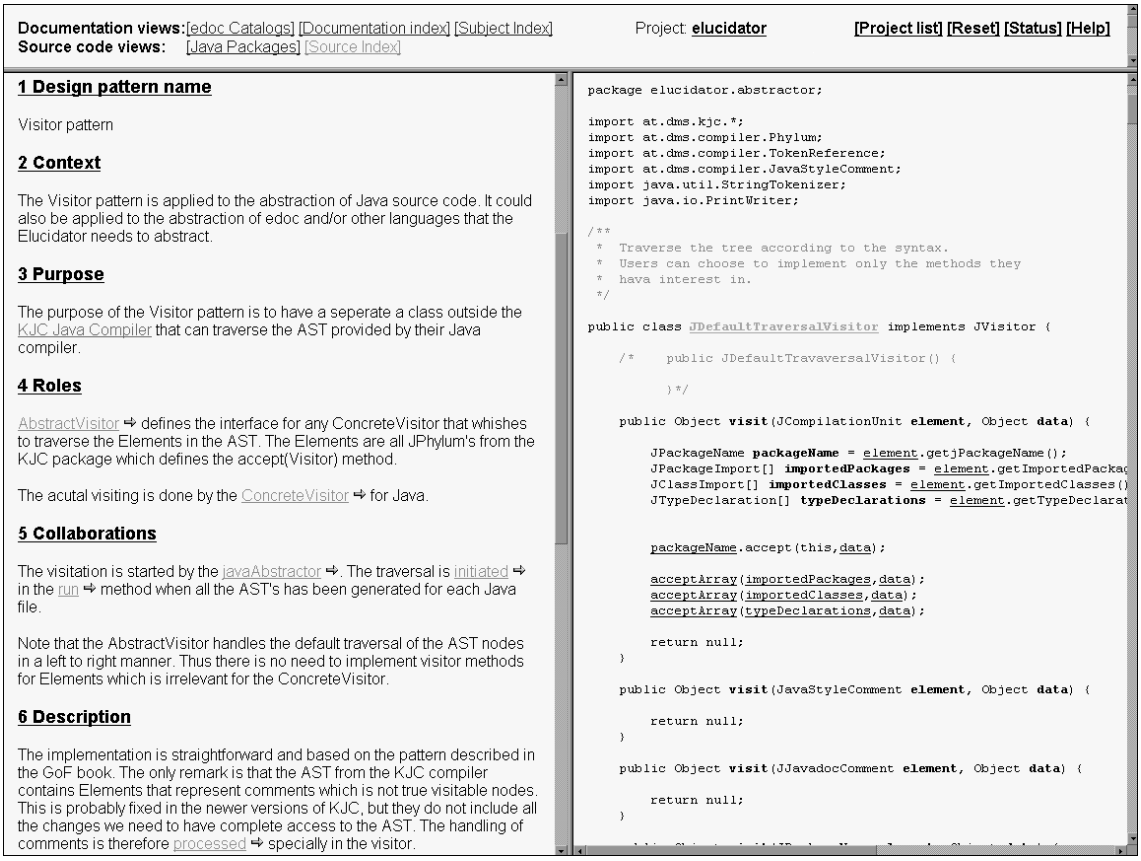

Figure 3: *An example of documentation of a design pattern instance in an elucidative program.*

Programming we have practical techniques to handle the complexity of explanations of real programs. Using Elucidative Programming ideas it is realistic to attach explanations to an existing program without changing the original sources. In addition, the program understanding can be approached from both linear reading of the documentation and from reading the program source code (via the links to relevant documentation from the program entities).

In courses where we have asked students to write minor literate programs we have experienced that it is hard to convince students to work "the literate way". We hypothesize that part of the reason is the physical splitting of the program and the embedding of the program in the textual documentation. If this is true, Elucidative Programming will be better suited than Literate Programming in educational contexts. However, it is still too early to make final conclusions on these matters.

3.5 Documentation enabling IDEs

Integrated Development Environments (IDEs) seldom support documentation writing beyond "simple" text editing of lexical program comments. This is sufficient when writing interface documentation such as JavaDoc [6]. However, it is undesirable to embed internal documentation in the source code. If attempted, an interested reader would have a hard time to locate the program within the documentation, or the other way round. Furthermore, the explanation structures (as intended for humans to understand) are often detached from the program structure (as intended for automatic processing tools).

Internal documentation beyond lexical program comments is also used. Let us here mention UML diagrams and traditional, monolithic documentation. Both of these, however, suffer from the drawback that the documentation and the program are not formally interrelated.

When a programmer has to work with different tools there is a high degree of risk that switching between these will disorient the programmer. A programming environment and a documentation environment reflect two different views on the software. If the cooperation between the two is not effective the user needs to do manual adjustments. As an example, a programmer who needs to consult the documentation while writing source code, must switch to

the documentation tool and start navigating to the relevant place in the documentation. This should be avoided as argued in [25], which addresses a hierarchy of cognitive issues to consider during design of software exploration tools. A solution to these problems is to integrate documentation support in the programming environments.

Good IDE support for documentation can provide several convenient features reliving the programmer of some of the tiresome labor associated with documentation maintenance. Navigational features can help the programmer navigate within the documentation, and navigate from documentation to relevant source code. Templates can suggest what to document. Error checking features can detect "dead" relations between program and documentation. When the program is changed, such features can suggest where to change the documentation accordingly. Filters can hide the details of the underlying documentation structure/language.

Elucidative Programming depends on the features outlined above. Without good IDE support of Elucidative Programming, it would be difficult to maintain relations between program and documentation during the program development process. The current implementations of the Java and Scheme elucidators provide easy link creation facilities in the Emacs editor [22, 23]. A link to a program abstraction is created by selecting the abstraction in the source code. In addition, the Java Elucidator provides "dead link" detection in terms of a table of links that do not address existing source code. Most of the navigation features provided by the current elucidators are present in the documentation browser (an ordinary web-browser). This navigation is facilitated by the two-framed setup (see figure 2) which allows the user to see both the documentation and the source code at the same time.

To avoid switching between the programming environment and the documentation environment the two must be merged. We recommend that the navigation facilities of the documentation browser are present in the programming environment as well. Modern IDE's often include a variety of functionality that makes the life of a programmer easier. Much of this functionality is also relevant when dealing with documentation. The link creation support, as discussed above, is analogous to *code completion*. Navigation from links in the documentation to their destination can be seen as *symbol browsing* (i.e. navigation between occurrences of identifiers and their declaration). The listing of available documentation for a given syntactical element is analog to *object browsing* (e.g. lists of identifiers, such as available methods and fields in an given class).

As our next project we will implement documentation support in a modern and popular Java IDE (such as TogtherJ [26]). We intend to implement the two-framed elucidative setup with the functionality discussed in this section. Hence, in this project we go for *documentation enabling* of a commercial and professional development environment.

4 Status and Conclusions

In the introduction to this paper we started by realizing that internal documentation plays an important role with respect to applications, frameworks and libraries (the shaded area of figure 1). Until now our work on Elucidative Programming has been focused on internal documentation of applications. By broadening the perspective we have identified five areas that transcend our previous focus. We have in this paper argued why internal documentation is important within these areas. In addition, we have discussed the potential role of Elucidative Programming relative to the areas.

The documentation of frameworks has been thoroughly explored in the literature. Most of this is concerned with documentation suited for programmers using frameworks. However, internal documentation of frameworks is also important, and a common solution is documentation using design patterns. A design pattern can be seen as an issue traversing a program. An elucidative program can keep these related program fragments together within "nice" bodies of natural writing.

Software tutorials are an important supplement for a programmer using a library or a framework. Software tutorials contain well-explained example programs that can be studied in order to understand a library or a framework. By introducing tutorial support in one of our Elucidative Programming tools we have brought some of the tutorial advantages, as known from Literate Programming, into Elucidative Programming.

In general, the ability of being able to refer to syntactically elements makes Elucidative Programming a powerful tool in the hands of programmers, who either wish or need to write about their programs. The possibility of mixing the linked and the in-line tutorial styles provides a powerful stylistic blend of Elucidative and Literate Programming.

Any software project can benefit from code reviews. We have argued that internal documentation should be reviewed as well. Through such reviews the maintainability of the software can be improved. In addition we have found that Elucidative Programming can provide good support during reviews. For example by facilitating navigation from source code to relevant documentation, or by serving as an annotation instrument.

However, writing documentation is difficult for the inexperienced. Consequently, documentation is often not written at all. Furthermore, writing documentation actually demands a considerable discipline. In our opinion we must educate students in the area of internal documentation. We find that Elucidative Programming is a good vehicle for ed-

ucational use, because it does not interfere nor conflict with the recommended forms and structures of source programs.

In the future, we wish to explore integration of Elucidative Programming in commercial IDEs. It is difficult to automatically create documentation, but we can integrate good documentation support in a modern IDEs, and we can bring documentation closer to the production of code. We hypothesize that this will make the programmers more aware of the documentation.

Finally, as mentioned in section 2.3, we have conducted two small experiments in order to start evaluating Elucidative Programming in various situations. In the future we wish to perform additional and larger experiments that will add to the empirical base of the Elucidative Programming project.

References

[1] M. R. Andersen and C. N. Christensen. Evaluating elucidative programming in an industrial setting. Unpublished paper available via http://dopu.cs.auc.dk, December 2000.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley Publishing Company, 1999.

[3] G. Butler and P. Dénommée. Documenting frameworks. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, pages 495–504, September 1999.

[4] G. Butler, R. Keller, and H. Mili. A framework for framework documentation. *ACM Computing Surveys, electronic symposium*, 32(1), March 2000.

[5] J. Carroll. Making use a design representation. *Communications of the ACM*, 37(12):29–35, December 1994.

[6] L. Friendly. The design of distributed hyperlinked programming documentation. In S. Frass, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France*, 1995.

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

[8] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[9] D. E. Knuth. *Tex: The Program*. Computers and Typesetting. Addison Wesley, 1986.

[10] 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):284–295, Fall 1998.

[11] K. Nørmark. The Elucidative Programming Home Page, 1999. Available via http://www.cs.auc.dk/ normark/elucidative-programming/.

[12] K. Nørmark. Elucidative programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.

[13] K. Nørmark. An elucidative programming environment for Scheme. In *In Mughal and Opdahl (editors), Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, pages 109–126, May 2000.

[14] K. Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*, pages 119–128. IEEE, June 2000. Also available via [11].

[15] K. Nørmark, M. R. Andersen, C. N. Christensen, V. Kumarand, S. Staun-Pedersen, and K. L. Sørensen. Elucidative programming in java. In *Proceedings on the eighteenth annual international conference on Computer documentation (SIGDOC)*. ACM, September 2000.

[16] K. Nørmark and T. Vestdam. Elucidative programming in computer science education. Submitted for publication, November 2001. Also available via http://dopu.cs.auc.dk/.

[17] G. Odenthal and K. Quibeldey-Cirkel. Using patterns for design and documentation. In *In Proceedings of ECOOP'97, LNCS 1241*, pages 511–529. Springer-Verlag, June 1997.

[18] OMG. UML specification 1.4, September 2001. Available via http://www.uml.org/.

[19] K. Østerbye. Minimalist documentation of frameworks. In *Presented at 3rd ECCOP'99 Workshop on Experiences in Object-Oriented Reengineering*, 1999.

[20] S. Shum and C. Cook. Using literate programming to teach good programming practices. In *SIGSE Bulletin: The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, volume 26, pages 66–70, March 1994.

[21] H. Siy and L. Votta. Does the modern code inspection have value? In *Proceedings on International Conference on Software Maintenance*, pages 281 – 289. IEEE, November 2001.

[22] R. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In D. Barstow, H. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill, 1984.

[23] R. Stallman. *GNU Emacs manual*. The Free Software Fundation Inc, June 1985.

[24] G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, second edition edition, 2000.

[25] M.-A. Storey, F. Fracchia, and H. Muller. Cognitive design elements to support the construction of a mental model during software visualization. In *In Proceedings of the Fifth Iternational Workshop on Program Comprehension*, pages 17–28, March 1997.

[26] TogetherSoft. TogetherJ. Available via http://www.togethersoft.com/.

[27] T. Vestdam. Documentation threads - presentation of fragmented documentation. *Nordic Journal of Computing*, 7(2):106–125, 2000.

[28] T. Vestdam. Introducing elucidative programming in student projects. Unpublished paper available via http://dopu.cs.auc.dk, January 2001.

[29] T. Vestdam. Writing internal documentation. In *Proceedings of the 6th European Conference on Pattern Languages of Programs (EuroPLoP 2001)*. Universittsverlag Konstanz, 2001.

[30] R. C. Waters and E. Chikofsky. Reverse engineering: Progress along many dimensions. *Communications of the ACM*, 37(5):22–25, May 1994.