Maintaining Program Understanding - Issues, Tools, and Future Directions

Thomas Vestdam and Kurt Nørmark

Department of Computer Science,
Aalborg University,
Denmark
{odin,normark}@cs.aau.dk
http://dopu.cs.aau.dk

Abstract. The understanding of a program is a key aspect of software development. The understanding is a prerequisite for the initial development efforts. This paper is concerned with the challenge of maintaining the program understanding with the purpose of supporting later phases in the program life time.
One approach to maintaining program understanding is to document decision and rationales behind a program as informal textual explanations—internal documentation. The starting point of this paper is a particular paradigm for program documentation program called Elucidative Programming. As the first contribution of this paper, three key documentation issues are identified on the basis of the experience with Elucidative Programming. Documentation motifs represent thematic elements of software, which typically transverse the structure of the source program files. Documentation proximity characterizes the distance between the documentation and the program. Documentation occasions are temporally situations for capturing and formulating the understanding of the program.
During the years a large number of documentation tools have been developed. As the second contribution of the paper, a number of contemporary documentation tools are reviewed. The tools are selected on basis of relevance for the key documentation issues, and relative to the common attention and interest of the particular tool in the documentation communities.
As the final contribution, and as a conclusion of the paper, a number of future directions and challenges are outlined.

1 Introduction

It is widely acknowledged that any kind of software development is based on a deep and comprehensive understanding of both the problem and the programmed solution. Kristen Nygaard, the founder of object-oriented programming, is often quoted for stating this briefly as "To program is to understand".

For newly developed software, many analysis and design activities can be seen as means to establish the understanding which is necessary for an initial implementation of the solution. For the purpose of maintaining existing software, the understanding is often regained by costly studies of the source code. Typically, no written program understanding has been maintained, and the initial analysis and design documents are inconsistent with the current state of the software. It may, in fact be the case that the initial program understanding already has been forgotten and regained several times.

Our work the last few years has been oriented towards a documentation approach called Elucidative Programming (EP) [1,2,3,4,5]. Using EP, textual documentation can be connected to existing source programs and presented side-by-side with the source programs in a web browser. EP takes the stand of maintaining the program understanding as written documentation which is associated with normal program source files. EP is in several respects inspired by Literate Programming (LP) [6]. Using LP it is necessary to organizes pieces of program source files as constituents of the documentation. In contrast, EP can connect documentation to existing source files. In today's landscape of programming languages and development tools it seems unrealistic to ask for major reorganization of the main software artifacts due to documentation concerns.

Written program documentation can be seen as an investment, which is intended to be paid back during the life time of the program. In case the investment is very large, due to creation of large amounts of program documentation, chances are that the investment never pays back. If no documentation investment is done at all, the cost of regaining the program understanding is most likely to increase dramatically. Little is known about the most optimal investment strategy.

The maintaining of the program understanding is complicated by the fact that changes of the program leads to changes in the understanding of the program. Thus, the program understanding is by no means static during the life time of the program. It may be costly to keep the program understanding up-to-date with the most recent version of the program. It is not clear if it is worthwhile to invest in keeping the program understanding up-to-date. According to [7] the original document seems to be valuable, even under circumstances where it does not reflect the exact status of the program.

It may be the case that the understanding of the program development history is as important as the understanding of a given version of the program. In the initial phase of program development, it is natural to focus of the understanding of the first version of the program. Later on, the focus may be shifted towards and understanding of the way the program is changing in contrast to an understanding of the most recent version of the program. The design decisions made during program development help programmers to understand the current state of a program. Even abandoned design ideas can be helpful for programmers who attempt to gain an understanding of a program [8].

The dream of researchers in the area of program comprehension and program documentation is of course to invent a technique that once and for all solves the "maintenance of program understanding problem". We are well aware that such a breakthrough may be a "fata morgana", but we still find it worthwhile to do work in this area. Due to the large amount of money involved in software maintenance, even modest contributions and improvements will be well worth the efforts.

In section 2 of the paper we will first present some key issues regarding maintenance of program understanding. The issues have been important in our work on Elucidative Programming. We will argue that the issues are central in any work on program documentation.

A number of contemporary documentation tools have emerged on the scene in the recent years. Some of these tools have not yet been described and discussed academically. In section 3 we will characterize some of these tools, and we will relate them to the key issues raised in section 2. The tools to be addressed are Web/PAS, Doxygen, a Theme-based Literate Programming tool, and LEO. We will, in addition, compare these tools with Literate and Elucidative Programming tools.

Finally, based on both the discussions of key issues and contemporary tools, we will attempt to set the scene for future work in the area of program documentation.

2 Key Documentation Issues

The area of program documentation is broad and multifaceted. With the purpose of contracting rather than spreading the discussion we will present three key documentation issues in this section.

The issues are conceptualized as *documentation motifs*, *documentation proximity*, and *documentation occasions*. In the subsections below we will explain and discuss these issues. In addition we will argue why we find that these issues are of particular importance for the area.

2.1 Documentation Motifs

We define a *documentation motif*[1] as the formulated understanding of a thematic element of a piece of software. This meaning and phrasing of the term clearly emphasizes the metaphoric impact from music and art. A documentation motif may or may not be manifest in the source program, but typically it is not.

A documentation motif may represent a holistic understanding of scattered parts of the program which add and contribute to a single concept in the program. As an example, an abstract datatype represented as scattered fragments of a C program may be represented as a documentation motif.

A documentation motif may also represent an understanding of certain relationships among parts of a program. A chain of procedure calls, which is crucial for some algorithmic aspect of a program, may serve as an example of such a documentation motif. Instances of design pattern represent another such example of documentation motifs. It has been argued that there is a need to explicitly

[1] From Merriam-Webster dictionary: *motif* 1: a usually recurring salient thematic element (as in the arts); especially : a dominant idea or central theme 2: a single or repeated design or color.

maintain design decisions by recording and documenting instantiations of design patterns [9]. Traces may also be seen as documentation motifs. Traces, or traceability links, connect elements of requirements, design, other subsequent models, and source code [10,11].

A documentation motif may also represent the understanding, observations and arguments behind certain qualities of a program, which are affected by several fragments throughout several modules. For instance, the understanding of an important or critical efficiency aspect can be elevated to a documentation motif. Also, a program often adheres to a specific set of requirements. Each of these requirements can be considered as documentation motifs that most likely have a direct impact on the program at several locations. Similarly, the understanding of a re-usability concern which is scattered on several different pieces of a large program can be seen as a documentation motif.

Our notion of a documentation motif mainly refers to non-local thematic elements that involves program code that is spread across of a piece of software. As such, documentation motifs can be seen as an informal way of representing an aspect-oriented program (AOP) [12]. The use of documentation motifs for such purposes may in particular be useful when programs are written in ordinary, non-AOP languages.

A documentation motif can be used to maintain an understanding of a specific application domain concept. Rajlich and Wilde have presented an overview of the role of such concepts in program comprehension [13]. We see documentation motifs as a more general notation than Rajlich and Wilde's documentation concepts. Rajlich and Wilde's vision of the role of concepts is mainly oriented towards mapping domain concepts onto relevant fragments of the program. Documentation motifs may include other kinds of knowledge about a program than domain concepts, and documentation motifs are oriented towards explaining a program or parts of it.

The overall ideal of preserving the understanding of a program is commonly accepted, but it is frequently unclear which parts of the understanding should actually be documented. In other words, the question is often which kind of insight to incorporate in the documentation. It is not realistic, nor desirable, that the documentation should explain every aspect or detail of a program. We recommend that a relatively small number of documentation motifs are selected, and that these documentation motifs make up the bulk of the total program documentation. Documentation in terms of a few, well-selected documentation motifs may represent essential insights about the program from a number of different and maybe overlapping perspectives. By boosting documentation motifs, the documentation will by and large be oriented towards transverse program understanding that cannot be directly deducted from the program code.

A literate program can be seen as one particular sequence of documentation motifs, where each motif represents knowledge about a named group of program statements. The original ideas behind Literate Programming do not allow for multi themed understandings which are mutually overlapping. More recently, Theme based Literate Programming has been proposed to remedy this limitation

[14]. A theme-based Literate Programming tool is discussed in section 3.2 of this paper.

Elucidative Programming is—almost by conception—well-suited to support documentation motifs. The programmer is free to structure the documentation according to selected motifs. A documentation motif is represented by one of more paragraphs, which typically—as a matter of writing style—includes a number of links as natural constituents of the individual sentences. Seen from the program side, an elucidative program also holds information about the documentation motifs to which the abstractions (such as classes and methods) are contributing.

2.2 Documentation Proximity

The value of documented program understanding critically depends on the availability of the documentation in relation to the program source files. The issue of *documentation proximity*[2] characterizes the distance between the documentation and the documented entities. The documentation proximity affects

- the ease of finding relevant documentation of a given program part,
- the updating of the documentation if the program is changed, and
- the qualities of both documentation and program development tools.

The distance between the documentation and the program fragments is a metric which can be measured in several different ways. In order to enhance the value and the availability of the documentation, we usually seek an arrangement where the documentation and the program source files stay in *close proximity*.

Physical documentation proximity denotes the situation where the documentation is weaved together with, and kept close to the relevant parts of the program source files. With physical proximity the documentation can be represented as comments in the source code. This may lead to the style of *self-documenting code*, as described by Brooks [15]. The main rationale behind physical documentation proximity is the expectation that documentation is available whenever needed by the programmer, simply because it is in plain view. Evenly important, it is usually hypothesized that documentation nearby the relevant program fragments is more likely to be updated whenever the program is updated.

Large amounts of textual comments in physical proximity with the source program can easily lead to *illegible* programs [16]. This problem may deteriorate if diagrams and graphical elements are allowed as annotations of the source program [16,17]. More importantly, as stated in section 2.1, the documentation motifs that are important to programmers often pertain to relationships among the different program units. Code comments are local to a specific point in the program and are not suitable to explain relationships across a program.

Literate Programming relies on physical documentation proximity. Compared with the notion of self-documenting code, Literate Programming reverses the

[2] From Merriam-Webster dictionary: *proximity* the quality or state of being proximate *proximate* 1: immediately preceding or following 2: very near.

roles of comments and programs. This leads to an arrangement where program fragments are located as constituents of the program explanations. Due to use of a programming language, a documentation language, and some notation for separating the two of these, the illegibility problem in Literate Programming is profound. This particular aspect of Literate Programming is often referred to as the *three syntax problem* [14].

Documentation of program libraries—interface documentation—in the style of JavaDoc [18,19] also relies on physical documentation proximity. Elements of interface documentation are in close proximity with the documented abstractions. The interface documentation is usually arranged in program comments, and it is typically written with use of a special-purpose markup vocabulary.

It is difficult to combine physical documentation proximity with multiple documentation motifs. Literate Programming, as coined by Knuth [6] and supported by different web systems, is strictly oriented toward a sequence of overall documentation motifs (together forming the *essay* about the program). There is only one psychological ordering [14] of source code fragments and textual explanations, chosen by the author, and this order can only be changed by rearranging and rewriting large portions of the literate program.

Navigational proximity represents a situation where the documentation and the program are located in different documents, connected by navigational measures. With navigational proximity, the distance between documentation and program units is measured in terms of the interaction steps involved in the navigation between the units. Navigational proximity calls for special support from both development tools and browsing tools. As a consequence, the updating of the documented program understanding is a particular concern.

Navigational proximity can more easily coexist with multiple documentation motifs than physical proximity. Each documentation motif can form a documentation unit, which is linked together with the relevant program fragments. The mentioned links are used as the navigational basis that forms the proximity. The good match between multiple documentation motifs and navigational proximity has been a driving force in our promotion of Elucidative Programming [2,5].

With use of non-physical proximity it is therefore important to augment the tools in the programming environment. This makes it realistic to locate relevant documentation and possible to remind programmers of the existence of relevant documentation.

2.3 Documentation Occasions

A *documentation occasion*[3] is defined as a situation at which a programmer has gained some understanding of the program that potentially is relevant to maintain. For every programming task there are a number of occasions for writing

[3] From Merriam-Webster dictionary: *occasion* 1 : a favorable opportunity or circumstance 2 a : a state of affairs that provides a ground or reason b : an occurrence or condition that brings something about 3 : a need arising from a particular circumstance.

documentation, or for updating existing documentation. Stated roughly, these occasions are before, during or after a given programming task has been performed.

Ideally, writing documentation should be considered as a cycle starting before code is written and concluded after the code has been written [8]. This is obviously not the common approach. Studies also show that documentation is seldom changed when the program is changed [7,20,21].

Documentation written before, during, and after programming contribute to the maintained program understanding in different ways. Documentation written *before* coding includes sketches of code design and descriptions of the context of the part of a program that is to be written (i.e. the programming task at hand). For example, the problem to solve, requirements or the conditions under witch the program part is to work. When documentation is written on beforehand, the programmer is forced to think more carefully about the code that is about to be written [6]. This helps ensuring the correctness of the chosen solution, and can even affect code quality such as performance and flexibility. This is often claimed by programmers adhering to Literate Programming, although the evidence is still anecdotal [22].

Programmers using eXtreme Programming [23] aim at ensuring the correctness of the chosen solution by writing units tests before any program code is written. Such tests can be seen as a representation of the specification and understanding of a given programming task. In a similar way, design by contract [24] can also be seen as a way to specify and preserve an understanding of the program.

Some programming tasks are best done by doing. However, by writing documentation *during* coding the programmer can ensure that existing documentation reflects the current program and not previous versions. In addition, by switching from programming to writing documentation the programmer is forced to think more carefully about the current programming task, hereby keeping the program on the right track.

Some aspects of a program are first really understood *after* a programming task has been completed. Hence, the real and final understanding cannot be established before or even while performing the programming task. Many solutions to a problem are first well understood after the programming is done. Through an *after rationale* the programmer is able to produce a good explanation of the solution and to present the reasons behind the solution. Writing documentation after the fact requires that the programmer carefully reviews the new code in order to ensure that important implications and decisions are not forgotten. When programmers have lived with a program for a long time they tend to take decisions and rationales for granted [25].

Legacy software often lacks proper documentation. Maintenance of such programs therefore involves a costly and error-prone reestablishment of the program understanding. The process of writing documentation after a program has been completed is often termed *re-documentation* [26]. The goal of re-documenting a program is to recover program understanding and hereby to make future maintenance easier.

It can also be argued that the need for documentation of specific parts of a program can be user-driven [27] and prompted *by needs*. This documentation occasion is the extreme variant of "after documentation". Programmers may, for instance, via the Internet ask questions about some program code. Developers, or other programmers, who have an answer to the question update the documentation according to the question. It has be argued that this approach will work particular well in open-source projects [27]. We conjecture that the user-driven documentation demands will be able to work just as well in larger teams of programmers who collaborate on longterm software projects. In general, documentation by need pushes the occasions for writing documentation until a concrete question about the program is asked, or until programmers start debating the actual source code.

Based on a survey among programmers, Forward and Lethbridge have found that there might be a need in real world software development for tools that support lightweight documentation [20]. *Lightweight documentation* is everyday documentation that should be easy to create rather than easy to maintain. Lightweight technologies should enable quick and efficient means to communicate ideas (e.g. pictures of a whiteboard session taken by digital camera), enable reader feedback, and should be simple to use. Reader feedback is also an important issue in [27].

As mentioned in the introduction, another temporal aspect of documentation of program understanding is the history of the program. Neither Literate- nor Elucidative Programming tools directly provide means for maintaining the program history. It may be possible to maintain a kind of *differential documentation* that describes the changes made to the program, and the rationales behind the changes. It is, however, difficult to re-establish a particular version of both the program and the documentation without support from a version control system.

3 Contemporary Documentation Tools

Documentation tools support and assist programmers with the handling of documentation. Some documentation tools are integrated with the program development environment, and others are separated from the program development tools. A number of documentation tools produce documentation which is supposed to be accessed as a World-Wide Web resource.

The support of documentation motifs put specific requirements on documentation tools. It must be possible to annotate specific parts of a program or somehow group explanations and relevant fragments of a program together. Documentation approaches based on navigational proximity call for special support of the documentation tools. In particular, the updating of the documentation following some program changes is only realistic if it is supported by the documentation tool. Furthermore, documentation tools should not restrain when documentation should be written. Hence, documentation tools must support writing documentation at different occasions as needed.

Literate Programming tools in the web family are in almost all cases non-interactive (batch-oriented) and separate from other program development tools (non-integrated). However, there are exceptions such as the Literate Programming environment for introductory programming made by Cockbrun and Churcher [28]. The first generation of Elucidative Programming tools for Scheme [29,3] and Java [2] has characteristics similar to the original Literate Programming tools. More recently, the elements of Elucidative Programming have been implemented and integrated in TogetherJ [30], which is an integrated development environment for Java.

In this paper we do not cover the overall landscape of documentation tools. Tools for documentation of the early software development phases, such as UML and other modeling related tools, are not considered in this work.

In the forthcoming sections we focus on documentation tools in two different categories: (1) Tools which seem to be popular and in widespread use, but not yet addressed in an academic context. (2) Tools which are of particular interest relative to the three documentation issues, which we raised in section 2. We structure the discussions in tools for interface documentation and tools for internal documentation.

3.1 Interface Documentation Tools

Interface documentation aims at describing the conditions and restrictions for interaction with a program unit. Beyond documentation of a general explanatory nature of a program unit, interface documentation fulfills the need for very precise documentation which spells out the way that the program unit may be used by clients [31].

Interface documentation is essentially a separate resource, either maintained manually or generated automatically from the source program code. The developers of the Eiffel libraries [32,31] made use of automatic extraction of interface documentation from the Eiffel classes [32]. The interface documentation of the Eiffel libraries was intended as printable resources. With the advent of Java, interface documentation made by the JavaDoc tool [19,18,33] became an on-line resource that allows easy global sharing [27]. Today, there are numerous variants of documentation tools—all with their own unique features for automatically generating documentation.

Doxygen. Doxygen [34] is one of the many interface documentation tools that have appeared in the programming community since the advent of JavaDoc [18,19]. Doxygen differs from JavaDoc as it is a more general documentation tool that offers many highly configurable features. Doxygen also supports many different programming languages such as C++, C, Java, Objective-C, IDL, and there exits add-ons for a range of additional languages. Furthermore, Doxygen can produce both web-documents and printable documents. The printable documents that can be generated are latex and RTF documents as well as man-pages.

This implies that Doxygen allows embedded LaTeX formulas in code comments (documentation blocks). Doxygen can also produce XML documents which enable programmers to define their own resulting output by using XSL-stylesheets. The web-documents are HTML files that are structured in a similar fashion as JavaDoc (i.e. a page is generated for each class which includes documentation of class members).

The *components* of a program that can be documented are the major abstractions, such as fields (and global variables in C), methods/functions, and classes (and enumerations and structs in C). Documentation is embedded in the program as code comments and structure using special documentation tags. Code comments that are internal to a given component body can also be selected for inclusion in the documentation of the component.

Doxygen allows grouping components together to create "module" structures in the documentation, even though these structures are not present in the source code. It is also possible to group members of a class or file, such as fields and methods. As an example, it is possible to group together all get and set methods in a class. Both kinds of groups can be associated with a name. As Doxygen provides the option of linking to any defined name in either program or documentation, grouping can be used in order to form documentation motifs, and even form documentation motifs that transverse the program structure.

As a speciality, Doxygen can extract, pretty print and present source code in the resulting documentation. Defined names in the source code are marked and linked to the relevant documentation of the particular component. Furthermore, Doxygen can even inter-operate with a graph drawing tool in order to generate inheritance diagrams.

The main occasion for writing interface documentation is at the time a class or a member (e.g. function, field or method) is defined. However, besides from producing interface documentation with rich features Doxygen can also be used to extract the code-structure from source code that has not been documented with Doxygen. This means that Doxygen can be used as an outline tool that can help programmers to get an overview of a program. Following, Doxygen can be used to re-document a program in order to maintain a reestablished understanding of a program. Hence, Doxygen can be used to perform re-documentation in the same manner as Web/PAS (see below).

As the documentation is embedded in the source code the two entities are in physically proximity of each other. However, there is a price to pay if using the grouping feature mentioned above. The syntax for grouping can become too voluminous and render the source code illegible.

Re-documentation using Web/PAS. Partitioned Annotations of Software (PAS) [26] is a hypertext based notebook in which programmers record their understanding of a program. This understanding stems from the programmers work with and observations of an existing program. Hence, PAS supports programmers in incremental re-documenting programs. The resulting documentation is studied in a web-browser.

PAS divides a program into components (i.e. classes, functions, dependencies, and function arguments). Each component is associated with an *annotation* that explains that particular component. Each, annotation is divided into a number *partitions*. The partitions for classes are for example, domain annotation, class dependencies, dependency annotation, authors comments, and member functions. Other partitions can be created as needed. The division of a program into components and partitions is automatically generated. The job of the programmers is then to fill the partitions and annotations with explaining text.

The components of a program are presented as nodes in a tree. From the root all annotations of classes span as child nodes. The actual source code is represented as child nodes of these annotations. When browsing, programmers can move up and down in this hierarchy which follows the hierarchical structures of the program.

There is no option for documenting motifs that transverse the program structure. The structure of the documentation strictly follows the program structure. In addition, the browsing facilities do not include source code or the option of navigating from source code to relevant documentation. Hence, documentation and program are physically separated. As a consequence the documentation becomes a secondary product in danger of being forgotten after some time.

The intended occasion for writing documentation is potentially a long time after the program has been completed. PAS is not well suited for writing documentation before or during programming. However, the parsing tool of PAS (HMS) offers support for adding and deleting annotations and partitions as the program changes, but program components must be defined before any documentation can be written.

3.2 Internal Documentation Tools

Internal Documentation is a commonly used term for documentation aimed at maintaining the understanding of the actual program code. Usually, internal documentation involves writing *about* the program with the intent to explain the details of the program in order to maintain an understanding of why and how the program works.

There are many different internal documentation tools around today. Literate Programming, represented by the web family of tools, is still the most radical approach to internal documentation. Literate Programming and the set of supporting tools has inspired many tool developers. Many programmers consider the original ideas of Literate Programming as unfit for modern software development [2,14]. However, the idea of writing about the program is still fundamental to many modern tools for internal documentation. A few selected tools has been mentioned in the previous sections, and in the next couple of section we will describe two additional tools, each of which represents a novel approach to internal documentation.

Theme-Based Literate Programming. Theme-based Literate Programming (TBLP) [14] has been introduced as a means for creating different paths through the chunks of a literate program. These paths through the chunks are called *themes.* Themes allow creating several sequences of documentation motifs, for example for different audiences or different purposes such as transverse program understanding (e.g. explaining patterns, aspects, or concepts). As such, each path corresponds to a weave operation, and the current TBLP tool provides the theme author with several options for configuring the resulting presentation. In addition, the tangled program is considered a theme.

In contrast to traditional Literate Programming [6], chunks are not limited to code or documentation. New chunk types can be defined as needed (e.g. chunks for figures, diagrams, or unit tests). As an additional contrast, all chunk types can be nested in TBLP.

Theme-based literate programs are explored and created in a special tool. The tool provides a facility for browsing themes and for editing the contents of the individual chunks of a given theme. All available chunks are stored in a common repository, and from the repository chunks can be assigned to relevant themes. Hence, an occurrence of a given chunk in any theme is a reference to the stored chunk in the repository. Furthermore, the tool contains an integrated version control system allowing themes to refer to specific versions of chunks.

Program and documentation are in close physical proximity, although the chunks that make up a theme are actually references to chunks in the repository. The presentation of a theme contains physically interleaved documentation and source code, just like a traditional literate program. The explanations and source code that make up a chunk must be able to fit into different contexts (themes). Hence, the chunk must be "self-contained" and "self-explanatory". However, this kind of explanations can be difficult to produce, and the situation becomes worse the smaller the chunks are [35].

As TBLP essentially supports literate programming, the occasions at which programmers ideally should write documentation is before coding. At least chunks must be created before coding, as chunks will hold the actual source code.

Leo: Literate Editor with Outlines. Leo is a tool that supports Literate Programming in a pragmatic fashion [36]. Leo is in part operated a graphical user interface. In Leo a literate program is *outlined* as a tree of nodes. Each node represents a section (i.e. a chunk) and displays the chunk name. Sub-nodes contain the body of chunk names which are defined in a parent node. In this way Leo outlines provides an overview of the literate program.

Leo allows *cloning* nodes. A cloned node is a copy of the node and its entire sub-tree of nodes. If any clone is changed (e.g. node contents are edited, nodes are added, deleted or moved) this change is mirrored in every clone. Hence, it is for example possible to create a documentation motif that transverse the program structure by adding relevant clones to a node containing an explanation of the documentation motif.

Leo provides a vast range of convenience features for structuring and managing outlines. The GUI allows moving nodes around in outlines in a natural way using drag-and-drop, and Leo also provides a number of commands for restructuring outlines. In addition, Leo provides support for many programming languages in form of syntax highlighting.

Program and documentation is in close and physical proximity relative to each other. In addition, Leo provides an “tangle” and “untangle” command which exports all defined files and vice versa. Exported files contain redundant information (as code comments) that enables Leo to import files again even if they have been modified in another tool. This means that programmers are not restricted to the limited programming environment that Leo provides.

Leo does not put any specific temporal requirements on writing documentation. At any occasion documentation can be written, but code chunks, in form of nodes, must still be created prior to coding.

4 Future Directions and Challenges

Our own efforts the last few years have been oriented toward the evolution and investigation of Elucidative Programming. As an overall conclusion of our work so far we have identified three key issues, which we find of particular importance for the maintenance of program understanding. In the first part of this paper we have characterized and discussed these issues.

As argued in section 2.2, navigational proximity can more easily coexist with multiple documentation motifs than physical proximity. Due to this observation, we continue to advocate Elucidative Programming as an alternative to the approaches where documentation and program are physically intertwined.

As it appears from the discussion in section 3, language independent tools play an important role in the area of program documentation. Language independence is important because many programmers do not limit their work to a single programming language, but also because many software projects involve more than one programming language. Due to the nature of Elucidative Programming we are required to have some knowledge about the programming language. A future challenge in the area of Elucidative Programming is therefore to develop an approach that enables an addressing of programs in a fashion that is uniform across programming languages. As a first step, it will be attractive to provide a generalized markup of programming languages, in the style of srcML [37].

Only little work has been done to integrate version control mechanisms in documentation tools. As argued in section 1 and 2.3 knowledge about the program evolution is an important documentation aspect. As a future challenge, we will like to investigate how to support maintenance of the understanding of both the current program and previous versions of the program. The major goal of involving older program versions is to enable a documentation of the program evolution. In Elucidative Programming this may call for versioning of the relations between the documentation and the program source code. It is known from the area of hypertext that versioning of links and relationships is a particular tricky and difficult area [38]. As part a versioning support, tools are needed for management a given version scheme and for appropriate presentations of the versioning information to help programmers comprehend this level of information.

Documentation tools should in general support different documentation occasions. We should definitively provide tools that help programmers in writing documentation during or after programming. As pointed out in section 2.3, documentation created much latter, by need or for re-documentation purposes, should also within the scope of our documentation tools. However, documentation written late in the development process is likely to lack important information because programmers, with time, forget decisions and rationales. Even if documentation is written during the programming process, programmers may have problems with identifying aspects that are relevant for maintaining the program understanding. As a remedy, and as a future challenge, more effort should be put into identifying and describing *abstract documentation motifs*, much in line of documentation patterns [8]. To go even further, more efforts could be put into investigating options for automatic generation of templates based upon knowledge of abstract documentation motifs. This involves an identification process that most likely is very difficult to realize as a fully automatic process. By use of semi-automatic approaches, involving some input from the programmer, there might be some unexplored options for future documentation support.

References

1. Nørmark, K.: Requirements for an Elucidative Programming environment. In: Eight International Workshop on Program Comprehension, IEEE (2000)
2. Nørmark, K., Andersen, M., Christensen, C., Kumar, V., Staun-Pedersen, S., Sørensen, K.: Elucidative Programming in Java. In: Proceedings on the eighteenth annual international conference on Computer documentation (SIGDOC), IEEE Educational Activities Department (2000) 483–495
3. Nørmark, K.: Elucidative Programming. Nordic Journal of Computing **7** (2000) 87–105
4. Nørmark, K.: The Elucidative Programming Home Page (1999) Available via http://www.cs.auc.dk/∼normark/elucidative-programming/.
5. Vestdam, T., Nørmark, K.: Aspects of internal program documentation - an elucidative perspective. In: 10th International Workshop on Program Comprehension, IEEE (2002) 289 – 292
6. Knuth, D.E.: Literate programming. The Computer Journal **27** (1984) 97–111
7. Lethbridge, T., Singer, J., Forward, A.: How software engineers use documentation: The state of the practice. IEEE Software **20** (2003) 35–39
8. Vestdam, T.: Writing internal documentation. In: EuroPLoP 2001 - Proceedings of the 6th European Conference on Pattern Languages of Programs 2001, Universittsverlag Konstanz (2001)
9. Odenthal, G., Quibeldey-Cirkel, K.: Using patterns for design and documentation. In: Proceedings of ECOOP'97, LNCS 1241, Springer-Verlag (1997) 511–529
10. Riebisch, M., Philippow, I.: Evolution of product lines using traceability. In: Proceedings of Workshop on Engineering Complex Object-Oriented Systems for Evolution at OOPSLA 2001. (2001) Published online at: http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml.

11. Sametinger, J., Riebisch, M.: Evolution support by homogeneously documenting patterns, aspects and traces. In: Proceedings of the 6th European Conference on Software Maintenance and Reengineering. (2002)
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of European Conference for Object-Oriented Programming, Springer-Verlag (1997) 220–242
13. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: Proceeding of the 10th International Workshop on International Workshop on Program Comprehension, IEEE (2002) 271–278
14. Kacofegitis, A., Churcher, N.: Theme-based literate programming. In: Proceedings of the Ninth Asia-Pacific Software Engineering Conference, IEEE Computer Society (2002) 549– 557
15. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley (1995) Twentieth Anniversary Edition.
16. Sametinger, J., Pomberger, G.: A hypertext system for literate c++ programming. Journal of Object-Oriented Programming **4** (1992) 24–29
17. Tilley, S., Müller, H.: Info: a simple document annotation facility. In: Proceedings of the 9th annual international conference on Systems documentation, ACM Press (1991) 30–36
18. Friendly, L.: The design of distributed hyperlinked programming documentation. In Frass, S., Garzotto, F., Isakowitz, T., Nanard, J., Nanard, M., eds.: Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France. (1995)
19. Sun Microsystems: JavaDoc tool home page (sun microsystems). Available from http://java.sun.com/products/jdk/javadoc/index.html (2004)
20. Forward, A., Lethbridge, T.C.: The relevance of software documentation, tools and technologies: a survey. In: Proceedings of the 2002 ACM symposium on Document engineering, ACM Press (2002) 26–33
21. Kajko-Mattsson, M., et al.: The state of documentation practice within corrective maintenance. In: Proceedings of IEEE International Conference on Software Maintenance, ACM Press (2001) 354–363
22. Hamer, J.: Literate programming: A software engineering perspective. In: Proceedings of the Software Education Conference (SRIG-ET 94), University of Otago, Dunedin, New Zealand, IEEE Computer Society Press (1994) 282–288
23. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley Publishing Company (1999)
24. Meyer, B.: Object-oriented software construction, second edition. Prentice Hall (1997)
25. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. IEEE Transactions On Software Engineering **12** (1986) 251–257
26. Rajlich, V.: Incremental redocumentation using the web. IEEE Software **17** (2000) 102–106
27. Berglund, E.: Library Communication Among Programmers Worldwide. PhD thesis, Linköping University (2002) Linköping Studies in Science and Technology, Dissertation no. 758.
28. Cockburn, A., Churcher, N.: Towards literate tools for novice programmers. In: Proceedings of the second Australasian conference on Computer science education, ACM Press (1996) 107–116
29. Nørmark, K.: An Elucidative Programming environment for Scheme. In: Mughal and Opdahl (editors), Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research. (2000) 109–126

30. Vestdam, T.: Elucidative Programming in open integrated development environments for Java. In: Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java. (2003) 49–54
31. Meyer, B.: Lessons from the design of the Eiffel libraries. Communications of the ACM **33** (1990) 68–88
32. Meyer, B.: Reusable Software. Prentice-Hall (1990)
33. Kramer, D.: API documentation from source code comments: a case study of JavaDoc. In: Proceedings on the seventeenth annual international conference on Computer documentation. (1999) 147–153
34. van Heesch, D.: Doxygen. http://www.doxygen.org (2004)
35. Vestdam, T.: Documentation threads - presentation of fragmented documentation. Nordic Journal of Computing **7** (2000) 106–126
36. Ream, E.K.: Leo - outlining editor. http://webpages.charter.net/edreamleo/front.html (2004)
37. Maletic, J., Collard, M., Marcus, A.: Source code files as structured documents. In: 10th International Workshop on Program Comprehension, IEEE (2002) 289 – 292
38. Østerbye, K.: Structural and cognitive problems in providing version control for hypertext. In: Proceedings of the ACM conference on Hypertext, ACM Press (1992) 33–42