A Case for Contemporary Literate Programming

VREDA PIETERSE, DERRICK G. KOURIE AND ANDREW BOAKE
University of Pretoria

In this paper we discuss the characteristics of Literate Programming and the development of programming environments to support Literate Programming in the past two decades. We argue that recent technological developments allow Literate Programming to be re-introduced as a viable approach to improve the quality and consistency of software artefacts.

Categories and Subject Descriptors:
D.2.0 [**Software**]: Software Engineering, General - *Standards*;
D.2.6 [**Software**]: Software Engineering, Programming environments – *Integrated Environments;*
D.2.7 [**Software**]: Software Engineering, Distribution Maintenance and Enhancement – *Documentation*;
General Terms:
Software Engineering
Additional Key Words and Phrases:
Literate Programming
Literate Programming Environment
Elucidative Programming
Theme Based Literate Programming

1. INTRODUCTION

Knuth [1984] coined the term 'Literate Programming' (LP) to describe his approach to program design. According to this paradigm programs should be written in a style that is not merely readable, but actually enjoyable to read. During the past two decades a number of scholars have recognised that LP can be a powerful aid in the program development process. However it was never widely accepted. We think that the main reason is because it was announced at a time that technology could not adequately support it, and at a time that there was not yet a sufficient need for it.

The essentials of LP are presented. We describe the structure and intent of some literate programming environments (LPE's) that have been developed. We contend that it is the emergence of recent LPE's that make the re-introduction of LP into the software development process a viable and desirable undertaking. The implications of incorporating an LP approach into the development process in the light of various current trends are discussed. We suggest that Literate Programming is not only viable, but is likely to add a number of benefits if it were to be integrated into several representation approaches to software development.

2. LITERATE PROGRAMMING ESSENTIALS

LP is a programming style for developing programs and their documentation. In this style a program is primarily seen as a document that explains a problem solution to a human reader. This view is radically different from the widely accepted view of a program as a list of instructions to a computer to solve a problem. In the latter view, human understanding is, to a lesser or greater extent, treated as a sort of secondary issue. Knuth [1984] maintains that, if applied correctly, LP leads to the development of better programs which are more elegant, effective, transportable and understandable. The maintenance and amendment of such programs will also be eased. The essentials of LP may be summarised as follows:

2.1 Literate quality

The completed program is a literate work of art that explains to the reader what the computer is supposed to do. A program, as seen by the computer, is a set of components combined in some structure. The literate programmer can be regarded as an essayist that explains the solution to a human by crisply defining the components and delicately weaving them together into a complete artistic creation [Knuth, 1984]. Lee [1994] goes so far as to describe a literate program as a publishable-quality document that argues mathematically for its own correctness. However most practitioners would not insist on such a level of formality.

Author Addresses:
Lead Author:
V. Pieterse, Department of Computer Science, University of Pretoria, Lynnwood Road, Pretoria, 0001, South-Africa; vpieterse@cs.up.ac.za
Co-authors:
D.G. Kourie, Department of Computer Science, University of Pretoria, Lynnwood Road, Pretoria, 0001, South-Africa; dkourie@cs.up.ac.za
Andrew Boake, CERA@UP, University of Pretoria, Lynnwood Road, Pretoria, 0001, South-Africa; andrew.boake@up.ac.za

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, that the copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than SAICSIT or the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2004 SAICSIT

2.2 Psychological order

The structure and order of the program document is psychologically based. This means that the different modules that comprise the program are arranged into a logical order that will enhance its understanding. Brown and Childs [1990] stress that this structure may be significantly different from the physical order required by the compiler. This attribute distinguishes a literate program from a heavily documented program.

2.3 Integrated documentation

Documentation of the program is not seen as a separate entity that needs to be developed beforehand to plan the development of the program, nor is it an appendix that needs to be added to a program as an afterthought to record some of the thoughts behind the code thus providing some aid for future maintenance or extension of the code. On the contrary it is an integral part of the literate program that is developed alongside the code. Rather than seeing the program as instructions to a computer that includes comments to the reader, the program should be seen as an explanation to a human that includes comments between 'code delimiters' so that delimited code can be extracted to the language system by the LP tools [Williams, 2000].

2.4 Table of contents, index and cross references

The document must have a table of contents, an index as well as cross references between related modules within the program. This information is seen as essential, mainly because the web-structure of a literate program should be conveyed. This additional information can be used by the reader of the program to enhance understanding, as it reveals a global concept of the program and highlights the internal relationships between modules. The automatic generation of this information is important [Denning, 1987]. In modern literate programs it is assumed that all references are hyperlinks. The document must also be searchable using various techniques such as keywords, filtering related concepts, answering specified queries, etc.

2.5 Pretty printing

The term pretty printing is generally used to refer to the automatic application of indentation, font styles and text colours and other typographic techniques to improve the readability and ease of understanding of code. This idea was inspired by Oppen [1980]. Petre [1995] says that algorithms to implement pretty printing are reliable and easy to implement. However, when LP was first proposed the idea of pretty printing in editors was not yet a matter of course as it is today.

2.6 Verisimilitude

Thimbleby coined the term verisimilitude to describe what he considered to be the most critical attribute of literate programs [Cited by van Wyk, 1990]. It refers to the requirement that the generation of executable code and the production of the human readable literate version of the program should not require two different versions of the document, but rather be automatically extracted from the same source document. Van Wyk emphasises the fact that this attribute distinguishes literate programs from programs that have merely been polished for publishing. Verisimilitude is easily violated. For example, Literate Program Browser [Beck and Cunningham, 1986] included tools to generate diagrams to visualise dynamic behavior of a running program from the executable code. These diagrams had to be pasted into the literate version of the program. When the code was changed these diagrams then had to be regenerated and replaced manually.

3. LITERATE PROGRAMMING ENVIRONMENTS

3.1 Pioneers

The idea of LP was proposed at a time when technology could not yet fully support it. Early literate programmers had to build their own programming environments to be able to practice LP [van Wyk, 1990]. The first Literate Programming Environment (LPE) called WEB, was designed by Knuth [1984] as an advance on structured programming. Figure 1 shows the structure and dataflow of WEB.

It uses two processors called WEAVE and TANGLE to convert the original source document into a publishable, human readable program and into an executable program. The program prepared for the human audience is printed using TEX while the program prepared for execution by a computer is compiled and executed by a standard Pascal compiler.

Soon a number of similar LPE's (such as those shown in table 1) evolved to support other programming languages or to produce documents using other formatters.

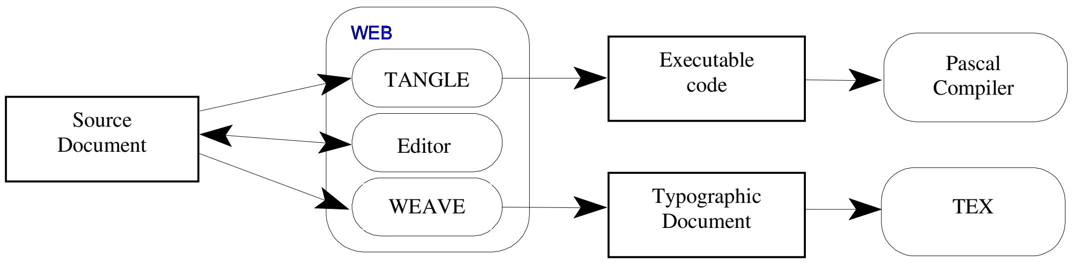

Figure 1. *The structure and dataflow of WEB (the first LPE)*

Table 1. Some Language Specific LPE's

LPE	Language	Format
WEB [Knuth, 1984]	Pascal	TEX
CWEB [Thimbleby,1986]	C	troff/nroff
Literate Program Browser[Beck and Cunningham, 1987]	SmallTalk-80	troff
Galley Editor and System Organizer [Reenskaug and Skaar, 1989]	SmallTalk	*internal to Smalltalk IDE*
FWEB [Avenarius and Oppermann, 1990]	FORTRAN8X	TEX
APLWEB [Dickey, 1993]	APL	TEX

3.2 Language Independency

Van Wyk [1990] commented that the general acceptance of LP would not be possible before a universal LPE could be marketed. In the light of the technology of the time, most developers accepted this as the death knell of LP. Some valiant supporters of LP however continued to build adaptable LPE's (such as those shown in table 2) that were able to support a variety of programming languages, and to produce documents in various specified formats.

Table 2. Some Adaptable LPE's

LPE	Languages used	Formats used
SPIDER [Ramsey, 1989]	Most Algol-like languages, including C, Ada, Pascal, Awk, and many others	TEX, LaTeX
LIPED [Bishop and Gregson, 1992]	Assembler, Pascal, Clipper	IBM PC/AT printer TEX
VAMP [Van Ammers and Kramer, 1992]	Pascal, Fortran, C, Simula	RUNOFF, troff, nroff, TEX, LaTeX
CWEB [Levy, 1993]	C, C++, ANSI C, Java	TEX, LaTeX
Noweb [Ramsey, 1994]	awk, C, C++, Haskell, Icon, Modula-3, Objective Caml, PAL, perl, Promela, Turing, and Standard ML	TeX, latex, HTML, troff
LEO [Ream, 2002a]	Java, C, C++, Pascal, Fortran, Perl, Icon, Python, Smalltalk, Cobol, ...	*internal to LEO*

Figure 2 shows the structure of LIPED [Bishop and Gregson, 1992] as representative of language independent LPE's. It has the same basic structure and dataflow as WEB but requires more input data (in the form of language and typographic specifications) to support a variety of programming languages and typesetters. Theoretically it can support any language and produce output in any format specified.

3.3 Object Oriented Programming

The introduction of object-oriented programming (OOP) uncovered problems associated with LP, such as locating methods that may reside in one of many places within the class hierarchy. This leads to the resulting yoyo problem identified by Taenzer et al [1989], referring to the need for multiple scans up and down the inheritance hierarchy to understand the control flow of a program. To alleviate these problems the basic structure of LPE's were adapted to include tools to create a hypertext version of the program to ease navigation and increase intelligibility of object oriented programs. Figure 3 shows the structure of AOPS [Shum and Cook, 1993] as representative of OOP LPE's.

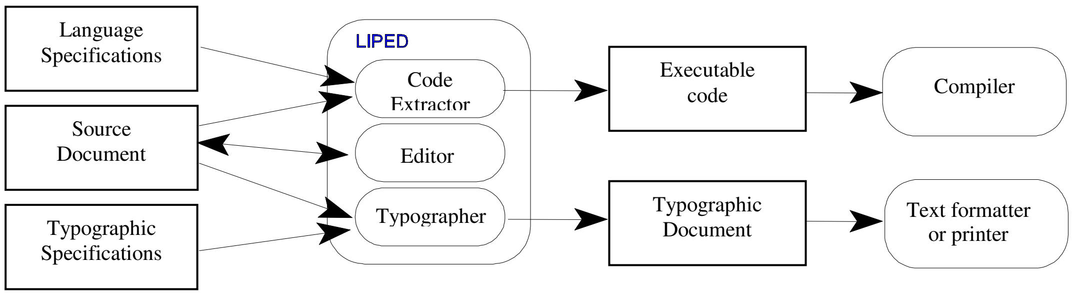

Figure 2. *The structure and dataflow of LIPED (a language independent LPE)*

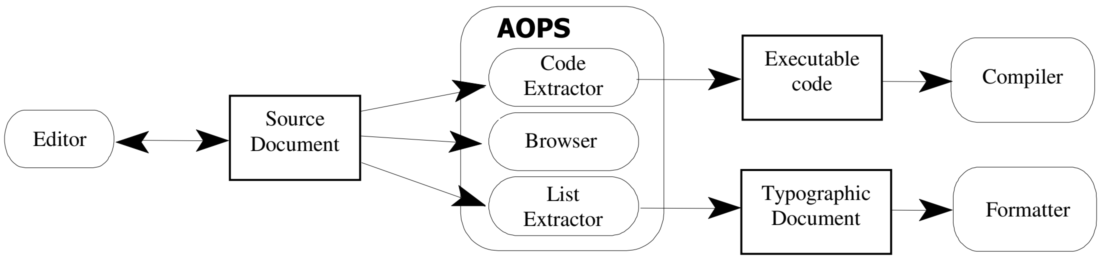

Figure 3. *The structure and dataflow of AOPS (an OOP LPE that includes a hypertext browser)*

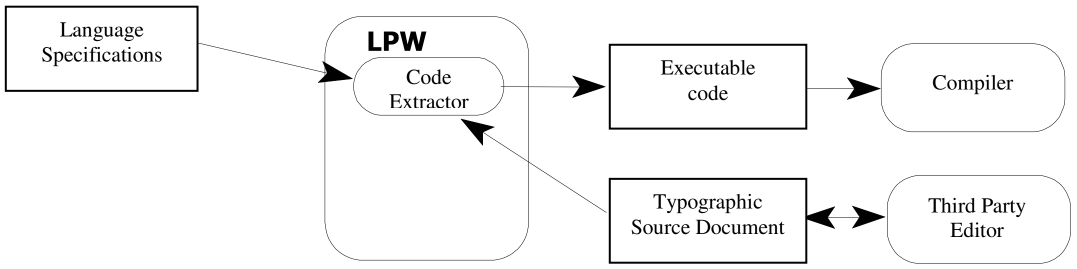

Figure 4. *The structure and dataflow of LPW (a LPE that integrates a third party editor)*

It allows the user to use an external editor of choice to create and edit the source document. It includes a browser that shows the system as a hierarchical tree with code, documentation and graphical nodes. The nodes can contain hyperlinks to related information. In contrast to other language independent LPE's, its code extractor and extractor that creates the typographic document do not use additional information to create documents for the compiler and formatter. Instead it merely extracts and organises the text using special instructions that the programmer has to embed in the source document.

3.4 Wysiwig

Not all LPEs that were created kept to the original structure. Instead of creating custom made editors some LPEs integrated powerful existing editors (such as a commercial word processor with WYSIWYG capabilities) for the editing and preparation of the human readable version of the program.

This led to a simpler structure in which the source document is no longer created and maintained in the LPE. Instead the main source document is created and maintained in the format of a third party editor. The third party editor is used to replace the role of both the formatter and the browser of OOP LPEs. The need for a formatter is delegated. The processor that creates the human readable document is replaced by a human that uses built-in capabilities of the given editor, or other tools to prepare the human-readable document. The preparation of the human-readable document is therefore less automatated than in LPEs complying to the original structure. Figure 4 shows the structure and dataflow of Literate Programming Workshop (LPW) [Lindenberg, 1991] as a representative of such LPE's.

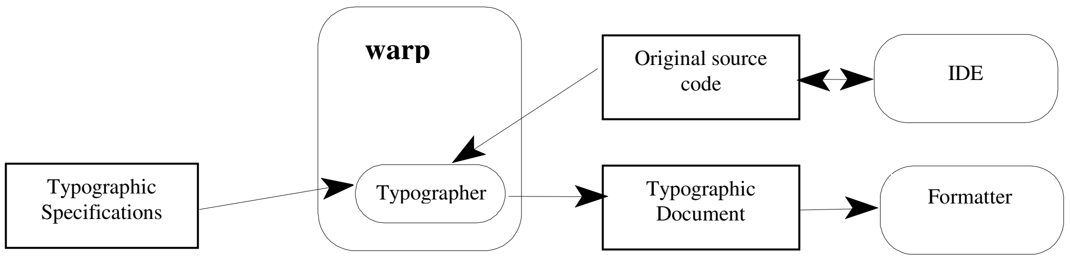

Figure 5. *The structure and dataflow of warp (a LPE that integrates third party programming environment)*

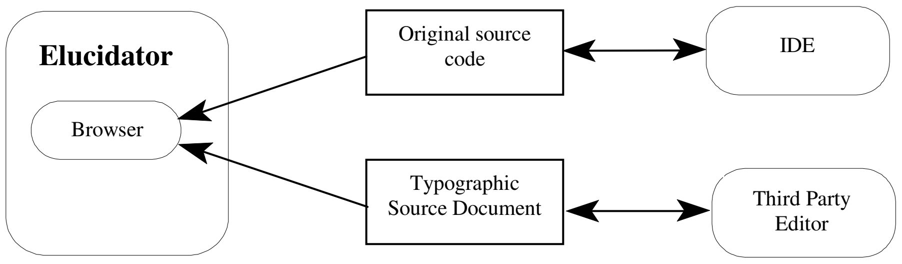

Figure 6. *The structure and dataflow of Elucidator (a tool to enhance a programming environment for elucidative programming)*

3.5 Interactive Coding and Debugging

Several problems arise if the programmer is unable to interact with the extracted code. To eliminate this problem, the original source programs should be intact at all times. This approach requires the creation of a tool which allows the original source programs to remain in their standard format. Such code remains editable in the existing integrated programming environment (IDE) which includes a compiler and can include numerous other tools such as debuggers and code generators. The need for a code extractor is therefore eliminated and the IDE fulfils the role of the browser of the OOP LPE. Figure 5 shows the structure and dataflow of warp [Thimbleby, 2003] as a representative of such LPE's.

It is important to note that warp departs from LP in the sense that it allows the programmer to create a typographic document which is not necessarily the whole program as implied by traditional LP. This is mainly motivated by the idea of having a tool to support journal publications that include reliable code [Thimbleby, 2003].

3.6 Elucidative Programming

Nørmark proposed a new variation of literate programming which he calls elucidative programming. The main difference between literate programming and elucidative programming is the position and nature of the target document. In literate programming the goal is to create a version of the program which is ultimately a printed technical document describing the whole program, while the goal of elucidative programming is to create a program that includes explanations that can be experienced in the programming environment to support the programmers who are responsible for the maintenance thereof [Vestdam and Nørmark, 2000].

Instead of using physical embedding of program fragments in the documentation text, or vice versa, Elucidative Programming Environments (EPEs) maintain code and documentation in separate documents while achieving proximity between code and documentation using bi-directional links. This approach eliminates the need for both code extractor and formatter. The browser of an Elucidative Programming Environment permits selected fragments to be viewed or printed.

Table 3. Elucidative Programming Environments

EPE	Language
Scheme Elucidator, Nørmark [2000]	Scheme
Java Elucidator, Nørmark et al [2000]	Java

3.7 Theme Based Literate Programming

Kacofegitis and Churcher [2002] point out that existing LPEs enforce a single psychological order for a program or system while developments in the use of hypertext and XML enables us to create documents that can be presented to different audiences in different orders. They propose an enhancement of LP to support this idea called Theme-Based Literate Programming (TBLP). In their prototype TBLPE called CBDE (Context Based Development Environment) the atomic units that are assembled to create a system are called chucks. A chuck can be a code segment, a piece of documentation, a diagram, a unit test, etc. CBDE supports the creation of multiple themes for a single system. The original processes of WEB to generate executable code or a publishable document respectively, are represented by themes in CBDE. Likewise the subset of a system that is prepared for publication in a journal as proposed by

Thimbleby in paragraph 3.5 can be created as a theme in CBDE. Other themes can be added, for example to create a document that describes the temporal evolution of components at a finer granularity than conventional file-based version management tools.

4. TRENDS

The following trends in modern software development are indicators that the time is ripe to re-introduce contemporary LP as a norm to be adopted widely as a coding standard.

4.1 Documentation

Much has been said about the advantages of having proper documentation for programs [Heyman, 1990, and Kotula, 2000] and about the disadvantages of having documentation that does not match the system [Kotula, 2000, Thimbleby, 1986]. LP emphasizes the advantages of documentation and provides a solution to the problem of incorrect documentation by insisting on tools to create and synchronise the documentation. Shum and Cook [1993] experimentally found that LP leads to better consistency between code and documentation. When using LP, programmers are given the support needed to improve their documentation standard.

4.2 javadoc

The javadoc tool to create documentation for API's is widely accepted and used by java programmers. This is an indicator that the resistance of programmers to put enough emphasis on the documentation aspect of programming is no longer as severe as it was experienced when LP was first introduced. The introduction of similar tools to enable and support LP will not require programmers to change their current style and habits to the extent that was required earlier to move to LP. Current programmers are already halfway there.

4.3 IDE Development

Modern IDE's support some of the LP essentials such as pretty printing in editors, easy navigation in the system through the use of indexes, a table of contents and hyperlinks, verisimilitude between modelling tools and the code that is presented by the models, etc. We are investigating the effort required to adapt such IDE's so that they can support all the essentials of LP. The addition of LP support should enable programmers to include valuable information such as the design rationale in the normal documentation. In current use of modern IDE's the recording of such information is not supported in the IDE and can easily be lost.

4.4 Event Driven Programming

The character of programs has moved from batch programs with little or no interaction during execution to programs that are highly interactive, and that spend most of their time waiting for the next event. This has become practical, thanks to faster processors and operating systems capable of supporting GUIs and threads. This shift spawned the need for elaborate and comprehensive user documentation in the form of On-line help and user manuals. Documentation within the code in an intelligible format of what events to expect and how to react to them becomes increasingly complex as systems grow. Theme Based LP tools can assist in the automatic creation of user manuals and on-line help documents, by defining a theme for each. Relevant information for the user can be extracted and generated from the source. This should ensure that user documentation is synchronized with the actual implementation in the same manner that it is used to create the literate version of the code as technical documentation.

4.5 Design Patterns

There is currently a lively interest in the standardisation of descriptions of design patterns. An objective of such standardisation is to enable the categorization and classification of design patterns in huge data stores where programmers can easily find them using general queries. Without having to deviate from the agreed upon way to document design patterns, the use of LP tools for the description of design patterns will not only make the design patterns more understandable and accessible to readers, but will also provide standardised information of higher quality to enhance the searchability of a design pattern repository.

4.6 Portability

Technologies such as XML and XMI define global standards to interchange data of complex structure between diverse applications. Recent development and advances enable sharing of data in a single repository among various tools that can be used by developers of software systems. The existence of these technologies opens up the possibility of assembling a powerful integrated LPE using existing state-of-the-art development tools and applications to support and promote LP in a familiar environment. An example of a recent LPE implementing XML technology to integrate different tools is described by Ream [2002a].

4.7 Agile Methods
Extreme Programming (XP) and other agile methods have proven successful in enabling development teams to complete projects that seemed impossible using other established methods. For example the C3 project that was transformed from a system that 'most of us knew would never go into production' to a system that was ready to begin performance tuning and parallel testing 33 weeks later [Hendricksen, 1999]. Scalability and outsourcing have been identified as areas where XP needs to be adapted to be applicable. We feel that it will be possible to alleviate concerns about team size, project size and project character that are often associated with XP, by adding aspects of LP to it since the use of LP tools and application of LP principles will enable better communication in larger groups and produce documentation that encapsulates knowledge and intent in a way that endures beyond the current team.

4.8 Open Source
The movement towards greater acceptance and implementation of open source is a reality. One of the major concerns that might debar its success lies within the fact that the comprehensibility of code is a key factor in its usability. The adoption of LP can play a significant role in advancing the growth and success of the open source movement.

4.9 Product line based software Engineering
Product lines embody a strategic reuse of both intellectual effort and existing artefacts, such as software architectures and components. Taulavuori et al [2004] mention that third-party components are increasingly being used in product line based software engineering. They point out that this raises a problem that needs to be addressed, namely that software integrators have difficulties in finding out the capabilities of components, because components are not documented in a standard way. Application of LP to support better documentation of components in a prescribed structure can help to alleviate such problems.

4.10 Aspect Oriented Programming
Aspects are the issues that are addressed by design decisions that cross-cut the system's basic functionality, such as power consumption, information combination, failure handling, security issues, communication strategy, etc. Aspect Oriented Programming (AOP) is a technique for improving separation of concerns in software design and implementation [Kiczales et al., 1997]. AOP allows the programmer to define both functional units called components, (objects/ procedures/ functions) and non-functional units called aspects, using high level languages. An Aspect Oriented Programming Environment (AOPE) includes an aspect weaver that accepts a component program and one or more aspect programs as input and emits a complete program in a high level language such as C. The nature of aspects and aspect weavers constitutes highly reusable code both within the system and across applications. Application of LP to support structured documentation of aspects can help programmers to locate existing aspects and support programmers to understand the intention of the aspects and the methods they implement to achieve their goals and hence simplifying their reuse.

5. CONCLUSION

In the forgoing, we have provided the essentials of literate programming and surveyed, roughly in chronological order, how literate programming environments have evolved over the past two decades. We then enumerated a number of trends that argue that this may be the καιροσ[1] for re-introducing literate programming as a viable option for closing the semantic gap between raw code and human understanding.

That there is such a gap seems self-evident. It is also generally acknowledged that the gap should be closed. Where there is currently a lack of consensus is in regard to the question of whether external documentation is the correct way in which to address the problem.

The devotees of agile methodologies tend to believe that the path to closing the gap is through so-called 'self-documenting' code. This is to be achieved by adopting coding standards which emphasise good internal documentation, appropriate variable name choices and the like. This, taken together with strategies such as pair programming and collective code ownership are seen as sufficient. However, while such strategies may ameliorate the semantic gap problem and be good enough for small-scale ephemeral products, they can hardly be seen as a panacea or a truly professional approach for applications of significant size and duration.

In such cases, there is a clear need for a higher level view of the overall purpose and structure of the code, of literate explanations guiding the reader through complicated code paths and of explanations of various critical features of the application that are not self-evidently reflected in the mere perusal of internal documentation – albeit neatly presented in hypertext style such as provided by Javadoc. In agile communities, such arguments have been countered with a view that external documentation is then a separate concern: if management requests it, then it should be seen as a separate budgeted deliverable. However, such a Cartesian cleavage between code and documentation dooms the latter into being

[1] This is an allusion to the term used in the so-called 'Kairos Document' issued by the South African Council of Churches in the late eighties, in which it was argued that the due time – i.e. the καιροσ – had finally arrived for fully resolving the country's longstanding and seemingly intractable political problems.

perpetually out of date. Other professions would certainly not tolerate the notion that the responsibility for insisting on documentation should lie with the client: certainly not the medical, nor the legal, nor traditional engineering professions.

We are cognizant of the commonly observed fact that programmers are not enthusiastic about documentation [see, for example, Parnas, 2001: 558] and we do not underestimate the resistance to documentation from those who produce code. Neither are we ignorant of the fact that in practice – probably because of this resistance – documentation is frequently out of sync with evolving code. However, it is precisely for this reason that literate programming, especially in its theme-based incarnation, seems well poised to at least start to address the problem. Of course, it is not a silver bullet, and its wider acceptance would appear to require a shift in the value system of software engineers.

This, in turn, points to the need for introspection on the part of those in the educational system from which software professionals emerge. Nørmark [2000] attributes the underrating of the importance of documentation to negligence on the part of educators. It is at this level that both good internal and external documentation should be emphasized and rewarded.

Given the cited evidence that LP improves software quality, our agenda for future research is to identify an IDE that supports as many as possible of the identified facets of LP, to integrate other facets into such a product as necessary, and thereafter, to pursue an empirical study in an educational context. The purpose will be to measure the extent to which LP can be adopted and used to promote stronger student commitment to professional software documentation.

6. REFERENCES

AVENARIUS, A., AND OPPERMANN, S. 1990. FWEB: a literate programming system for Fortran8x. *SIGPAN Notices*, 25 (1), 52-58.

BECK, K., AND CUNNINGHAM, W. 1987. Expanding the Role of Tools in a Literate Programming Environment. *CASE'87. Boston Mass.* http://c2.com/doc/case87.html last visited 2003-12-19.

BISHOP, J.M., AND GREGSON, K.M. 1992. Literate Programming and the LIPED Environment. *Structured Programming*, 13 (1), 23-34.

BROWN M., AND CHILDS, B. 1990. An Interactive Environment for Literate Programming. *Structured Programming*, 11 (1), 11-25.

DENNING, P.J. 1987. Announcing Literate Programming. *Communications of the ACM*, 30 (7), 593.

DICKEY, L.J. 1993. Literate programming in APL and APLWEB. *APL Quote Quad*, 23 (4), 11.

HENDRICKSON, C. 1999. DaymlerChrysler: The best team in the world. *Computer*, 32 (10), 75

HYMAN, M. 1990. Literate C++, *Computer Language*. 7 (7), 67-69.

KACOFEGITIS, A., AND CHURCHER, N. 2002. Theme-Based Literate Programming. *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC'02)*

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C.V., LOINGTIER, J-M., IRWIN, J. 1997. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, 121-145.

KNUTH, D.E. 1984. Literate Programming. *The Computer Journal*, 27 (2), 97-111.

KOTULA, J. 2000. Source Code Documentation: An Engineering Deliverable. http://csdl.computer.org/comp/proceedings/tools/2000/0774/00/07740505abs.htm visited 2004/01/08

LEE, C. 1994. Literate Programming – Propaganda and Tools. http://www.cs.cmu.edu/~vaschelp/Programming/Literate/literate.html visited 2004/05/17

LEVY, S. 1993. Literate Programming and CWEB. *Computer Language*, 10 (1), 67-70.

LINDENBERG, N. 1991. The 'Literate Programming Workshop' for the Mac. http://www.desy.de/user/projects/LitProg/LPW.html visited 2004/05/20

NØRMARK, K., ANDERSON M.R., CHRISTENSEN C.N., KUMARAND S. STAUN-PEDERSEN S., AND SØRENSEN K.L. 2000. Elucidative Programming in Java. *Proceedings on the 18th annual international conference on Computer documentation (SIGDOC).* ACM,

OPPEN, D. 1980. Prettyprinting *ACM Transactions Programming Language Systems,* 2 (4), 465-483.

PARNAS, D. 2001. *Software Aging in Software Fundamentals.* Addison-Wesley.

PETRE, M. 1995. Why looking Isn't Always Seeing : Readership Skills and Graphical Programming. *Communications of the ACM* 38 (6), 33-44

RAMSEY, N. 1989. Weaving a Language Independent WEB. *Communications of the ACM*, 32 (9), 1051 - 1055.

RAMSEY, N. 1994. Literate programming simplified. *IEEE Software*, 11(5):97-105.

REAM E.K. 2002a. Leo Literate Editor with Outlines. http://personalpages.tds.net/%7eedream/intro.html visited 2004/03/28.

REAM E.K. 2002b. Leo and Literate Programming. http://personalpages.tds.net/%7eedream/design.html visited 2004/05/22

REENSKAUG, T., AND SKAAR, A.L. 1989. An environment for literate Smalltalk programming. *OOPSLA 1989 Proceedings*, New Orleans, 337-345.

SHUM, S., AND COOK, C. 1993. AOPS: an abstraction-oriented programming system for literate programming. *Software Engineering Journal*, 8 (3), 113-120.

TAULAVUORI, A., NIEMELÄ E., AND KALLIO P. 2004. Component documentation—a key issue in software product lines. *Information and Software Technology,* 46(8), 535-546.

THIMBLEBY, H.W. 1986. Experiences of 'Literate Programming' using CWEB. *Computer Journal*, 29 (3) 201-211.

THIMBLEBY, H. 2003. Explaining code for publication. *Software Practice and Experience*, 33 (3), 975-1001.

TAENZER, D., GANTI, M. AND PODAR S. 1989. Object-oriented software reuse: the yoyo problem. *Journal Object-Oriented Program.* 2 (3), 30-35

VAN AMMERS, E.W., AND KRAMER, M.R. 1992. VAMP: A tool for literate programming independent of programming language and formatter. *CompEuro 1992 Proceedings. Computer Systems and Software engineering*. Dewilde P; VandeWalle J., 371-376.

VAN WYK, C.J. 1990. Literate Programming : An Assessment In Literate Programming. *Communications of the ACM*, 33 (3), 361-365.

VESTDAM, T., AND NØRMARK K. 2002. Aspects of Internal Program Documentation – an Elucidative Perspective. *Proceedings of the 10th International Workshop on Program Comprehension.*

WILLIAMS, R. 2000. FunnelWeb Tutorial Manual. http://www.ross.net/funnelweb/tutorial/intro_what.html visited 2003/01/05