Literacy for the Masses: Integrating Software and Knowledge Reuse for End-User Developers Through Literate Programming

Matthew Dinmore and Anthony F. Norcio
Department of Information Systems, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250
E-mail: {mdinmo1,norcio} @ umbc.edu

Abstract

We examine the potential for applying the literate programming approach to end-user development problems in which explicit knowledge capture and sharing is an objective. Research results from the literate programming literature are presented to support this concept, and recent work addressing previously recognized shortcomings of literate programming technology is highlighted. A prototype system and early experiences with an end-user literate programming are briefly described.

1. Introduction

Knuth's Literate Programming (LP) concept [15], introduced almost two decades ago, initially drew a great deal of attention, but ultimately never found widespread acceptance. The paradigm promises improved understandability and transferability of program code resulting in better software engineering, and was considered a potential mechanism for teaching good programming practice to novice programmers. Interestingly, it has never been considered in relation to end-user programmers, defined as people who develop software incidental to some primary task they are performing. This paper adapts the ideas of literate programming as an approach to end-user development problems for which explicit knowledge capture and sharing is an objective.

The paper is organized as follows: first, the problem domain relating to end-user development is described. Then, the key concepts of literate programming are reviewed and relevant research results from the literature are examined. A case is made for applying literate programming to end-user development, and our initial experiences with an early prototype end-user literate programming environment are briefly described.

2. End-user programming in expert problem-solving domains

End-user programming (also known as end-user development) spans a range of activities from simple customization to the creation of complex applications [28]. Of particular interest to this work are experts in some domain who are not computer scientists or trained programmers, but who nonetheless need to write software in the performance of their work. An example would be a scientist who needs to create problem-specific data processing and computation software to support a particular experiment, but is not trained as a computer scientist and has little formal programming experience. Studies of such users in the field of biology [16] illuminate the tensions between performing science and the practice of programming to support it, and the resulting natural emergence of various forms of end-user programming expertise. This has been observed in similar disciplines [26], including geospatial analysis [30] and computer-aided design [11]. These domains all rely on the application of expert analytical knowledge to problem solving, and that problem-solving process, because of its computational demands, requires computer support, for which the end users are often placed in the role of programmers.

In as much as the "developers" of these programs are not programmers, let alone software engineers, it is not surprising that their software suffers from poor software engineering practice. Recent research begins to address the application of software engineering principles to end-user programming (e.g. [4]), specifically in the area of software quality assurance.

Another desirable outcome of good software engineering practice is reuse. It has been observed that end users often begin programming by tailoring (thereby reusing) artifacts created by others, through which they learn how to accomplish tasks and gradually increase their programming knowledge [11]. This is a relatively informal process, and the artifacts
resulting from it are generally not prepared for reuse, that is, structured and stored with the intent they be reused; providing this functionality would likely encourage more sophisticated end-user developers to formally share their work with novices and enable those novices to make better use of it.

Software reuse by end users can be extended to professional-grade code, as well. Artifacts created by professional developers, if properly prepared for end users, may be introduced to the end-user domain and reused there. The idea of applying component-based development to end-user programming is a mechanism for achieving this form of reuse [20].

Finally, there is a dimension of reuse that has not been previously addressed in relation to end-user programming, and may be perhaps the most important to the end users themselves. Integral to an end-user-developed piece of software is the expert knowledge of the work process encoded by that software. In these artifacts, the “how” is, if not explicit, at least mostly recoverable; this can be reverse engineered from the code itself. But hidden — and potentially lost — is the “why,” the “when” and the experience gained in creating and applying the software to real problems. This implicit knowledge is invaluable to the organization and key to reusing the software; it sets the context for reuse and provides the guidance for interpreting results. This knowledge should be recognized in its own right as a reusable artifact. The argument presented here is that this knowledge must be made an explicit part of the software to form a unified, reusable software/knowledge artifact.

It is beyond the scope of this paper to address the organizational challenges of knowledge reuse. We contend that, in a socio-technical environment conducive to end-user development, such as that represented by the emerging meta-design paradigm [9], the conditions for knowledge sharing exist. Based on the knowledge reuse typology by Markus [19], we find two classes that apply here: users documenting for similar others (their work peers, effectively), and users documenting for dissimilar others (in this case, most likely novices in their field, or customers of their work). As Markus illustrates, documentation of rationales and procedures, even for similar others, is still a difficult task, and additional considerations come into play when documenting for dissimilar others. In this paper, the principal focus is on an enabling mechanism that lowers the cost of producing reusable artifacts, but work toward realizing this must also address the end-to-end system to ensure a complete solution is presented in practice; this is a basic requirement for successful knowledge reuse [8].

In other words, the technical solution cannot stand in isolation from the organizational one; the technical focus of this paper should in no way be taken as minimizing that requirement.

3. Literate programming concepts

Literate programming was proposed by Knuth with the goal of ultimately viewing programs as works of literary art [15]. The paradigm attempts to shift the focus on a program as something that tells a computer what to do, to an explanation to humans of what the intent of the program is. In practice, this is accomplished by breaking the program into chunks, each of which contains marked-up narrative and the associated code and variable definitions. Chunks are ordered in a way that best suits the goal of explaining the program; the ordering of the chunks does not have to be related in any way to the order of program execution. For this reason, the resulting program can be thought of as a web of interrelated pieces linked in the conceptual space of the user; in fact, Knuth refers to the language – actually composed of programming and formatting languages – as WEB [15].

A key concept to be recognized here is verisimilitude: the description of the program and its code are all kept in one source file – they are a unified artifact. When the program is to be executed, or the documentation prepared for reading, the source it processed through a program (originally TANGLE for the executable portion, or WEAVE for the documentary portion) to generate the necessary intermediate file for additional processing.

3.1. Literate programming research

The central claim of literate programming, that it improves program understanding and communication between programmers, is the most appealing aspect of the paradigm. Notably, there are few published empirical studies addressing the effectiveness of this in practice. In one study that did examine this question, Bertholf and Scholz [3] tested comprehensibility of literate programs versus traditional structured programs (written in FORTRAN, in this case), and found that novice programmers were more successful in software maintenance tasks – modifying an existing program to add new functionality – with the literate program than the traditional one.

Childs and Sametinger [5] discuss applying literate programming techniques to documentation reuse. They identify several literate programming features which serve to enable reuse, among them: the use of common, defined documentation structures; extraction of common information from similar parts; and alternate views of the document to serve different
reader populations. A goal of their work is to allow the documentation for component-based systems to be assembled from a foundation built on the incorporated components' own documentation, and then extended to describe the composite system. This, in effect, enables knowledge about the components to be placed in an aggregate context that describes the function of the composed system, and therefore represents a form of knowledge reuse.

Knuth makes the claim, based on his own experience with WEB, that the time invested in developing a literate program is essentially the same as using a traditional programming language, but the results are better [15, p. 129]. The benefit is more formally established with respect to maintenance: it has been found that programmers, both novice and expert, are more effective at comprehension and maintenance tasks with book-style representations of the code than traditional listings [21]. Also supporting the claim of better quality for the same effort is a study by Shum and Cook [27] in which comments in traditional and literate programs were compared. The authors found a significant difference in the ratio of comments to source lines between methods, though not more comments lines themselves, suggesting greater information content. On further examination, it was found that the literate programs contained comments both about what the code did and how, while traditional programs only contained what comments.

Research has also examined whether literate programming is beneficial to novice programmers. Based on teaching concepts suggested by Wittenberg [32] around step-wise refinement and progressive revelation of detail in program development, Cockburn and Churcher [6] enumerate design criteria for a novice literate programming environment aimed at introductory programming courses. These criteria include the need to explicitly associate code and documentation, which can consist of multimedia elements, and provide an overview of the literate program's structure. It should be noted that the previously-described study by Shum and Cook had, as its goal, to teach good documentation skills to undergraduate computer science students.

3.2. Recent and related work in literate programming

Since 2000, there has been a renewed interest in literate programming in the literature. Pieterse, Kourie and Boske [24] make a case for "contemporary" literate programming, arguing that user interfaces and other technological support have matured to a point where broader implementation and adoption are possible. Among the recent developments they highlight are literate programming-like documentation generators, such as javadoc, which extracts marked-up documentation from Java source files, and widespread use of integrated development environments (IDEs). These IDEs go far beyond the basic text editors in use when Knuth first proposed literate programming; developers are increasingly distanced from the raw source files, which are managed by the IDE and are almost universally interpreted by the editor to provide display markup to highlight keywords, variables and programming structures such as loops and conditionals. Another trend the authors cite is the evolution toward event-driven software that is inherently interactive and nonlinear. Recall a feature of literate programming is that the written flow of the program does not need to mirror the execution flow; it can be written in a way that is sensible to explain. In an event-driven program, where execution order is essentially unknown, this becomes a very powerful feature.

Holmes [13] takes this a step further and closer to the end user in his piece calling for "perspicuous programming." Under this paradigm, the documentary effort is focused on what the user wants the program to do, rather than just documenting the algorithms. Here the algorithms simply make it work, and the program and incorporated documentation as a whole becomes a kind of user manual. It is structured around the users' needs; multiple levels of documentation may be provided and revealed as necessary to different levels of users, similar to the concept of views discussed earlier [5]. Holmes notes that the programmer must think differently to be successful as a perspicuous programmer because of the required user focus. We argue here this is not a substantial step if the programmer is the user.

The concept of presenting multiple views is furthered by Kacofegitis and Churcher [14] who introduce the idea of theme-based literate programming. They call for an expanded number of chunk types with their own attributes that can be navigated along different paths. These paths represent themes, each of which may be a more useful way of traversing the program to particular subsets of readers. As before, the order of the chunks in the source or along any path is completely independent of the execution sequence.

4. Literate programming for end users

An initial burst of interest and activity around literate programming followed its introduction, though as a whole, it was never widely adopted. As noted by Cordes and Brown [7], the suggestion that
literate programming was most appropriate for computer scientists to describe their work, while mainstream programmers would have little use for it, potentially relegating its use to a small elite (McIlroy bluntly describes literate programming as a “Fabergé egg... a museum piece from the start.” [15, p. 174]). Further widening this gap was the lack of natural integration with the software life cycle in which programmers work.

This is one area where we clearly wish to depart from the original vision. The argument for literate programming for end-user developers, especially in knowledge-intensive problem-solving domains, is that beyond simply solving the problem, the domain expert wants to share the solution with others. In effect, the expert is writing a description of the solution and to the extent that this is also an executable (perhaps even tailorable) representation of the solution, it becomes a compelling vehicle for sharing and reusing the artifact. So, we adopt the same reasoning as Knuth applies to computer scientists, though we are now more interested in the literate conveyance of the domain knowledge rather than the software solution itself, as called for in the perspicuous programming paradigm; as is the case in end-user programming, the software is simply a means to an end.

Notably absent in end-user programming research is an integrated approach to explicit knowledge capture in the development process. For example, while the European End-User Development Network of Excellence research agenda [22] repeatedly recognizes knowledge workers as principal end-user developers, it fails to recognize that the programs they develop are implicit knowledge artifacts, effectively making them one of the users' knowledge products. Given the potential knowledge content of literate programs developed by end users [27], more attention needs to be paid to this area.

A key requirement for introducing literate programming to end users is integrated tool support. In the original incarnation, any text editor could be employed, using a literate-programming specific markup language such as WEB, to create literate programs. Inherent limitations in this, including lack of support for multiple programming languages and the complexity of the markup language [77] led to a number of improved literate programming systems (e.g. NOWEB [25]). Some also introduced graphical interfaces (e.g. [6,7]). In end-user programming, integrated tool support is a necessity. In particular, the need for a user-employed markup syntax must be eliminated, as must the requirement for certain explicit structures. These must become entirely transparent to the end user. Achieving this should not be difficult, though. Just as a modern word processor hides the underlying implementation structure of a document in its presentation to the user (for example, the ability to store logically separate text, such as the body, the footnotes and the headers, in the same file), an end-user literate programming tool would perform a similar translation between the storage artifact and its presentation (In fact, the idea of using a commercial word processor or web editor as a literate programming tool is appealing and has already been demonstrated by Wittenberg [29]).

As previously noted, Cordes and Brown [7] comment on the lack of support in literate programming for the traditional software life cycle, particularly in the specification and design phases. However, in end-user programming, ideas such as meta-design [9] and its supporting seeding, evolutionary growth, reseeding process [11], explicitly move formal systems design for the underlying infrastructure to software developers, while engaging end users in iterative, domain-centric design activities throughout the life cycle. These design and documentation activities are well supported by literate programming.

4.1. Design of a LP environment for end-users

What would a literate programming environment for end users look like? Text input is certainly a key requirement. WYSIWYG editing is a necessity for end-users; burdening them with even a simple markup language (TeX was used in Knuth’s WEB) distracts from the purpose. As relatively powerful rich text editing is built into the frameworks of modern operating systems, this is not difficult to implement. Beyond this, there is a need to incorporate multimedia to enable users to be expressive and allow for multiple modalities of knowledge representation. The other primary requirement relating to text is that we must be able to chunk sections of the program into logical units, which can then be manipulated as chunks (reordered, themed, etc.)

To implement the program itself, instead of end users entering programming code, they would insert ready-made components that provide a proxy interface within the editing environment [20]. A basic software component is defined by its inputs, outputs and function. This interface would represent this essentially as a form the user could fill in. As is done in visual environments, the components could be wired together and their runtime parameters set for a specific application. Given the integral importance of text and readability, and a desire to avoid secondary notations found in 2D visual programming environments [23], a linear arrangement of the components is recommended; recent commercial and
research end-user development tools have adopted a similar design [2, 33].

To assist end users in component selection, it should be possible to make suggestions for appropriate components based on an analysis of the narrative. A variety of text analysis techniques exist, and their application to software reuse is not novel (e.g. [12]). Work has also been done in converting text-based stories into program shells [18]. Little and Miller [17] have recently shown how keywords entered by end-user developers could be converted into executable code. In a specialized domain, these approaches may be even more effective due to the level of domain-specific terminology in use.

To support archiving, sharing and interoperability, Extensible Markup Language (XML) is an obvious choice. Aguier and David's [1] work relies on XML as its common storage format. Others have noted the power of XML as a basis for modern literate programming [14, 24, 31]. XML's near ubiquity and web-centric design ensures broad accessibility and understandability. Further, its ability to encapsulate heterogeneous objects, be transformed in a standard manner to support different applications (through XSLT), and be rendered in various views through stylesheets addresses many of the requirements for a representation medium identified here.

4.2. End-user literate programming prototype

The authors have developed and are currently performing usability studies with a prototype end-user literate programming environment (Figure 1). It provides a simple interface for assembling reusable "building blocks" into a linear worksheet to solve a problem. Our objective in this research is to develop a collection of meta-design [9] principles to inform the creation of domain-specific literate end-user development environments. In this first step, we experimented with methods for text annotation, finding a user preference for predefined, structured text annotation of their programs. A weakness of the prototype, and requirement derived from it, was the lack of annotation of and assistance in selecting building blocks, thereby improving reuse. Without support for this, scalability to a large number of components, including user-contributed building blocks, will be impossible.

Looking forward, we will further develop the prototype around the principles we have begun to enumerate. As we move toward experimenting in specific domains, it is critical that the software effectively implement these findings while minimizing usability problems so that we can determine the overall efficacy of the approach. A generic wrapper component will be available so that domain-specific components can be rapidly added to the environment by professional developers. The ability to nest worksheets as components (thereby enabling user-level reuse) will provide the flexibility required for real-world applications.

5. Conclusion

We have proposed the novel application of literate programming concepts to end-user development to support explicit knowledge reuse in expert problem-solving domains. Results drawn from the extant literate programming and end-user development literature, as well as our ongoing research, will lead to the development of design principles for implementing end-user literate programming environments in specific domains.
6. References


[26] Segal, J. Two principles of end-user software engineering research. First workshop on End-user software engineering, (2005), 1-5.


