An Environment for Literate Smalltalk Programming

Trygve Reenskaug
Anne Lise Skaar

Senter for industriforskning
Forskningsvn. 1
0314 Oslo - Norway

Abstract

The programming environment described in this paper is an adaptation of Donald Knuth's concept of literate programming, applied to Smalltalk programs. The environment provides a multi-media document production system including media for Smalltalk class and method definitions.

There are two outputs from the system. The first output is a document which contains general descriptions and discussions intermixed with precise definitions of program fragments, test inputs and test results. The second output consists of compiled Smalltalk programs installed and ready for execution.

The main idea was to produce program documentation that was just as interesting and fascinating to read as ordinary literature. Our experience showed an added benefit, namely that the literate programming environment was an active aid in the problem solving process. The simultaneous programming and documentation lead to significantly improved quality of both programs and documentation.

1. Introduction

At the Center for Industrial Research (SI) we are a group of about 10 to 15 programmers using Smalltalk-80, a language which is eminently suited for exploratory, incremental programming. Our task has been to develop products, however, and we have therefore faced the challenge to create high quality programs and documents permitting maintenance and extension by people far removed from the original developers.

To the best of our knowledge, object-oriented programs of satisfactory quality had not been produced anywhere when we started our development. We therefore needed to create the necessary software engineering techniques together with a programming environment utilizing these techniques in addition to our specific programming tasks.

This paper is a report on our work. Many terms used in the paper are taken from the Smalltalk programming environment, but we hope it is approachable even for people ignorant of Smalltalk. The work marks the beginning of a major effort at our institute aimed at developing a comprehensive environment for the analysis, design, implementation and maintenance of object-oriented systems.

The idea of *literate programming* is to make documents describing implementation code as readable as ordinary literature. The senior author of this paper was greatly inspired by the Programming pearls article [1], where Jon Bentley presented Donald Knuth's concept of literate programming and the WEB system [5]. Reference [9] contains a similar, albeit very informal, description of a tiny Smalltalk program. Ward Cunningham and Kent Beck of Tektronix took the idea a step further as described in [3][2]. Both the group at Tektronix and the group at SI have since refined the methods further and have started using them for serious development work.

We have based our new software engineering techniques upon a variant of WEB, and have developed our own programming environment to make the method suited for our purpose. The environment permits us to mix general descriptions and discussions with precise definitions of program fragments, test inputs and test results. The output is a program definition document that is adapted to the needs of a human reader, while it may also be compiled into a Smalltalk program.

The rest of this paper is divided into four main parts. The next section gives a description of the literate programming environment for Smalltalk programs and an excerpt from an actual program description. Section 3 discusses a typical work process from a software engineering point of view. Section 4 contains a discussion of the experiences gained when using the method. Finally, section 5 contains a tentative conclusion and a suggestion for the direction of further work.

2. The Programming Environment

Our programming environment is a tool for document preparation, supporting a number of media useful for general documentation, such as texts, tables, raster pictures and vector drawings. In addition, the system provides media for Smalltalk class definitions and Smalltalk method definitions for the purpose of implementation. As a supplement, the Smalltalk *doit*-command is available in the text media to facilitate testing by executing Smalltalk code.

A sample program has been documented in [11]. Short excerpts are given in the third subsection as an example of our present programming style.

The last subsection discusses some differences between our system and the WEB system.

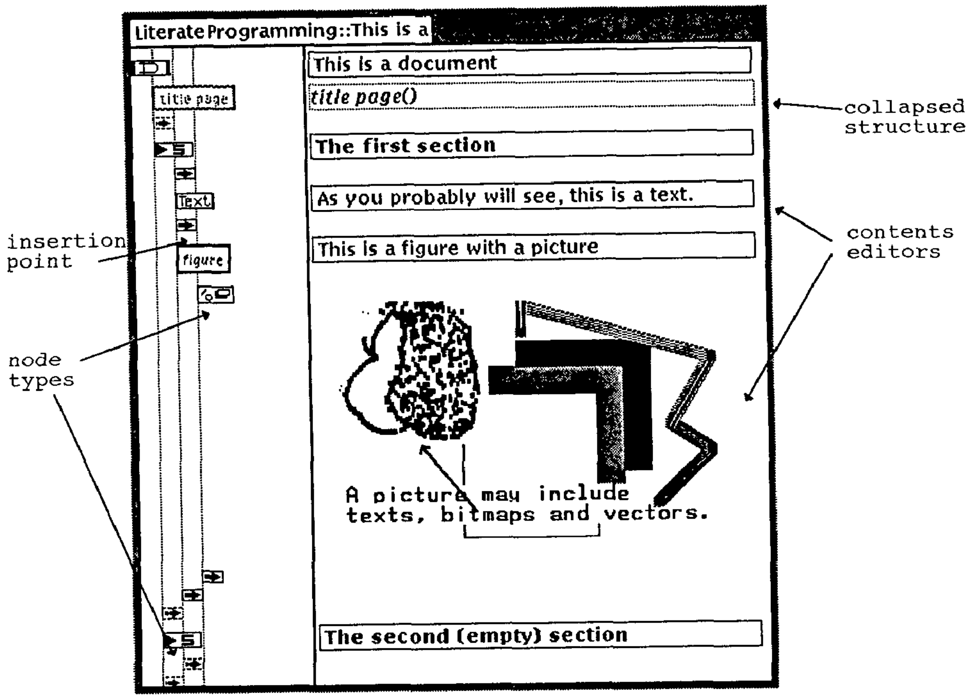
2.1. General User Interface

The user interface of the environment is basically organized according to the Smalltalk metaphors [12]. The central idea is to provide each user with a simple, dynamic information medium tailored to support that particular user in performing all his or her information-related tasks.

The model of the system is implemented in Smalltalk and consists of persistent, shared node objects interconnected by attributed relations into a directed graph structure. An editor for editing the semantic model, called the *Galley Editor*, emphasizes a linear traversal of the graph.

The Galley Editor is divided into two parts: The leftmost part is called the *margin* part and the rightmost is called the *contents* part, see figure 1.

The margin contains an iconic representation of the types of the document nodes. Examples of node types in the figure above are *document(D), title page, section, text, figure, picture.* The document structure may be edited in this margin through *cut* and *paste* as well as using the mouse to *move* the node icons from one position to another. The arrow symbols indicate insertion points. Activating one of these points permits the user to insert a node of any type that is legal at this point in the structure.


LiterateProgramming::This is a

This is a document

title page()

The first section

As you probably will see, this is a text.

This is a figure with a picture

A picture may include texts, bitmaps and vectors.

The second (empty) section

Figure 1. The Galley Editor

The right part of the window shows the contents of the nodes. The appropriate editor is started automatically when the user points to a displayed node. An explicit command from the user is therefore not needed.

Zooming facilities permit the user to collapse any substructure into a single line in the Galley, and to expand the presentation to any depth.

2.2. Interface to Program Fragments (Classes and Methods)

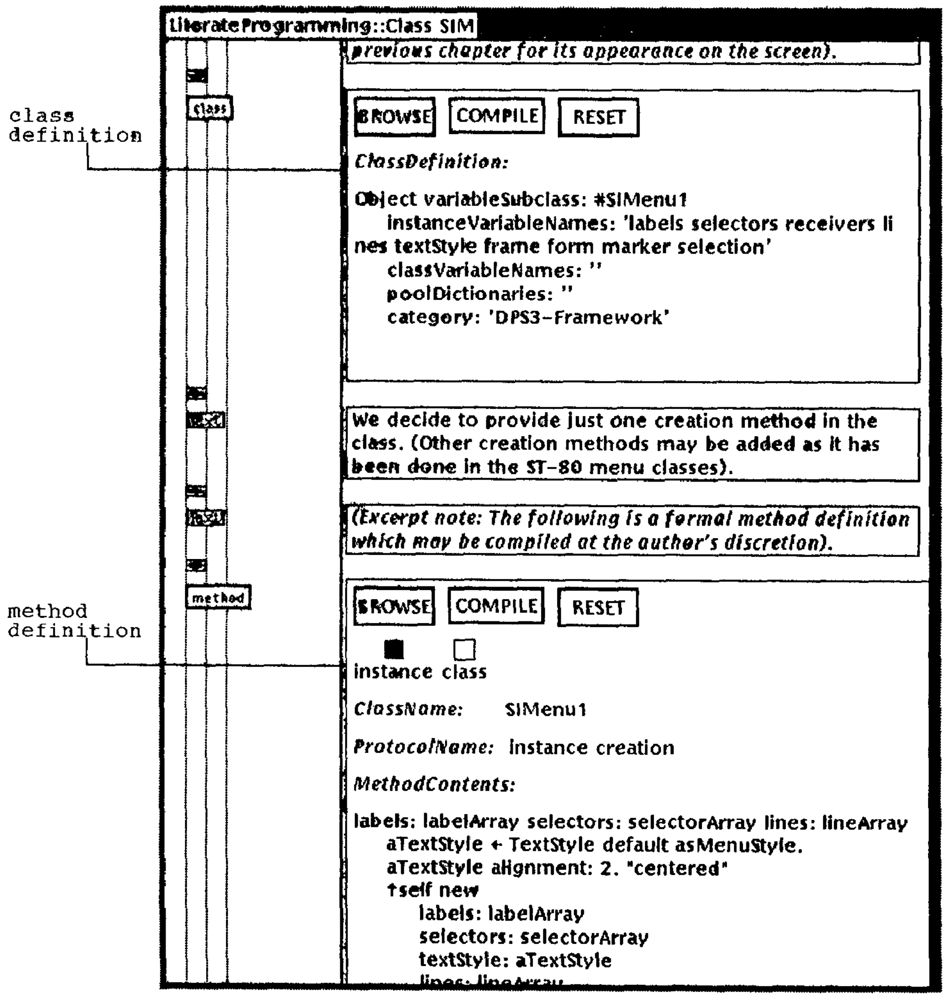
The special contents editors for the Smalltalk class and method definitions are shown in the figure 2 and are described below.

In both the class and the method editors, there are three push-buttons and a main text editor. In addition, the method definition has auxiliary text editors for the class name and the protocol name.

The *browse*-button gives access to the Smalltalk program library, called the *system organizer*, permitting the user to select any existing class or method definition. When a selection is made, the source code is automatically inserted into the editor. The *compile*-button activates the Smalltalk compiler, compiles the source code in the editor and inserts the result into the system organizer ready for execution. The *reset*-button resets the contents of the editor to the current state of the system organizer. Both the *compile* and the *reset* buttons are inversed when the source code corresponds to the current version of the code in the system organizer.

As a supplement to the individual *compile*-buttons, a general menu command in the margin can be used to cause all classes and methods in a selected section or document to be compiled.

Classes and methods may be redefined, so that several versions of the same piece of code may exist



previous chapter for its appearance on the screen).

ClassDefinition:

```
Object variableSubclass: #SIMenu1
    instanceVariableNames: 'labels selectors receivers li
nes textStyle frame form marker selection'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DPS3-Framework'
```

We decide to provide just one creation method in the class. (Other creation methods may be added as it has been done in the ST-80 menu classes).

(Excerpt note: The following is a formal method definition which may be compiled at the author's discretion).

■ instance ☐ class

ClassName: SIMenu1

ProtocolName: instance creation

MethodContents:

```
labels: labelArray selectors: selectorArray lines: lineArray
    aTextStyle ← TextStyle default asMenuStyle.
    aTextStyle alignment: 2. "centered"
    ↑self new
        labels: labelArray
        selectors: selectorArray
        textStyle: aTextStyle
        lines: lineArray
```

Figure 2. A class definition and method definition as it appears on the screen.

in the same program document. Old versions may be activated for execution by recompiling the proper part of the document.

The environment provides a general multi-step *undo* and *redo* mechanism so that the document can revert to any of its previous states.

2.3. Extract from an Example

In this chapter, we reproduce a few, abbreviated fragments of a document describing a new menu class in Smalltalk. (The fragments have actually been inserted into this paper by copying some chapters from [11] and editing the result).

2.3.1. Motivation and specification

No excerpt.

2.3.2. Class SIMenu1

We started with the study of the standard ST-80 menu classes *PopUpMenu* and *ActionMenu*, but we decided that our new class should be a subclass of Object. We gave it the following attributes:

1 *labels* - an Array of Strings giving the labels of each item. (We prefer this to the traditional *withCRs*, or even worse, carriage returns in the source code).

2 *selectors* - an Array of Symbols, each giving the command selector to perform when an item is selected by the user.

3 *receivers* - an Array of Objects (or *nil*) which are the potential receivers of the command message if the user selects the corresponding menu item. A *nil* value indicates that the item may not be selected by the user. This is new, and combines the function of dispatching a command selector to its final receiver with a command passivation mechanism.

(Excerpt note: The following is a formal class definition which may be compiled at the author's discretion, see the previous chapter for its appearance on the screen).

```
Class definition: SIMenu1
Object variableSubclass: #SIMenu1
    instanceVariableNames: 'labels selectors receivers lines
    textStyle frame form marker selection'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DPS3-Framework'
```

We decide to provide just one creation method in the class. (Other creation methods may be added as it has been done in the ST-80 menu classes).

(Excerpt note: The following is a formal method definition which may be compiled at the author's discretion).

```
Method definition: SIMenu1 |labels:selectors:lines:
Protocol: instance creation
labels: labelArray selectors: selectorArray lines: lineArray
    | aTextStyle |
    aTextStyle ← TextStyle default asMenuStyle.
    aTextStyle alignment: 2. "centered"
    ↑self new
        labels: labelArray
        selectors: selectorArray
        textStyle: aTextStyle
        lines: lineArray
```

2.3.3. Testbed for SIMenu1

We write a pair of dummy classes to enable the testing of the new Menu class.

All we need is a View that can react to a small number of different commands, and a Controller that can offer a suitable menu to the user.

A simple solution seems to be to let the View be a subclass of StandardSystemView, and let the commands effectuate changes to its inside color.

```
Class definition: TestView3
StandardSystemView subclass: #TestView3
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Interface-Support'
```

(Excerpt note: Much material removed here).

We start the Menu when the yellowbug is pressed.

```
Method definition: TestController3 |controlActivity
Protocol: control defaults
controlActivity
    Sensor yellowButtonPressed
    ifTrue:
        [YellowMenu
            commandOn: #yellowButton
            withHeading: nil
            owner: self].
```

The Menu will first ask its owner (the Controller) about each Menu item in turn. In this test, we could have let the View be the owner of the Menu, but since the test is also going to double as a demo example, we let the Controller relay the questions to the View.

```
Method definition: TestController3 |receiverForSelector:
Protocol: menu messages
receiverForSelector: aSymbol
    ↑view receiverForSelector: aSymbol
```

To test the dynamic passivation of commands, we decide on a logic that permits colors to be selected freely if the color is gray, otherwise all color commands but gray are blocked.

```
Method definition: TestView3 |receiverForSelector:
Protocol: menu commands
receiverForSelector: aSymbol
    (insideColor = Form gray) | (aSymbol = #gray)
        ifTrue: [↑self]
        ifFalse: [↑nil]
```

2.3.4. Test of the new programs

The program should now be ready for test, but we must remember to initialize the Controller class:

TestController3 initialize.
((TestView3 new) label: 'test'; borderWidth: 3) controller open

(Excerpt note: The above statements are executed in the document presentation on the screen by a doit-*command).*

The test works nicely, but it is impossible to close or reframe the window. We correct this omission by adding suitable action for the blue (right) mouse button and try again. (*Excerpt note: We typically write things like this during testing, and clean up the code (and the document) when everything works to our satisfaction*).

```
Method definition: TestController3 |controlActivity
Protocol: control defaults
controlActivity
    Sensor yellowButtonPressed
    ifTrue:
        [YellowMenu
            commandOn: #yellowButton
            withHeading: nil
            owner: self].
    Sensor blueButtonPressed
    ifTrue:
        [self blueButtonActivity].
```

Both tests were interrupted with the menu open, and their appearance copied into this document. In the first test, the inside color is gray. All commands are available and we select the command veryLightGray. In the second test, the inside color is veryLightGray. We notice that gray is the only command available, all the others being passivated.

2.4. Differences from the WEB system.

It should be noted that there are several differences between our tools and Knuth's WEB system. In WEB, program fragments may be defined in any order. Macro- and other facilities makes it simple to subdivide any code fragment in any way the author pleases. The author has to know four languages: English, the WEB specification language, the TeX text formatting language, and the target programming language, e.g. Pascal.

Our program fragments are always compilable Smalltalk code: a class definition or a method definition. (Definitions may be redefined in later chapters). The author has to know two languages: English and Smalltalk. The author does not need to know the formatter language (e.g. TeX or TROFF), since our high-level document media have default for-

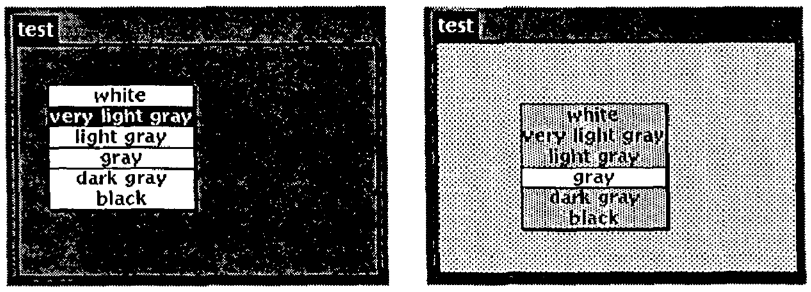


Figure 3. The result of two tests.

matting rules. The use of the document preparation system is sufficiently simple for us to claim that it should not be counted as a separate language.

Knuth's published programs indicate that his documents are always two levels deep, the TeX program itself having 54 chapters and 1377 sequentially numbered sections. In contrast, we freely use the section structure permitted by our multi-media editor to make the document as readable as possible. We have also programmed our text formatter to give us an alphabetically ordered index of class and method definitions with page numbers rather than section numbers.

3. The Work Process

Programming is usually described as a top-down process, proceeding in an orderly manner from specification through analysis, design and coding to final testing and installation. Reality is very different. The problem is attacked from any angle that seems profitable, top-down analysis is mixed with bottom-up synthesis into something that could be called the yo-yo approach.

During this process, various aspects of the problem are analyzed, assumptions are established and decisions are made. In a traditional approach, only the "hard" decisions are recorded in the form of code and formal specifications. The underlying assumptions and considerations are neither recorded nor shared, and exist only as long as remembered by the originators.

The literate programming approach offers a marked improvement on this situation. The main goal is no longer just to get a working program, but to provide a coherent description of a problem and its solution. It now becomes interesting and possible to capture not only the "hard" decisions such as code and interface definitions, but also the "soft" arguments about the writers' understanding of the problem, assumptions made, the considerations that preceded the "hard" decisions, and so on.

This method of working is not just an artifact to help a reader understand the program, it is an active aid in its development. For example, if a particular piece of the program is unduly complicated, slight modifications of the general base of assumptions and decisions often lead to dramatic simplifications. Traditionally, it is very expensive and dangerous to make such changes because it is hard to oversee their consequences. With literate programming, one can find the exact arguments that have to be changed, modify them and then read the documentation to find the consequences of the changes.

There are many different ways one can organize a literate program document, the main consideration should be to make it easy for a reader to appreciate a program, its why's and how's. One should also remember that some of the most important readers are the writers themselves, written arguments tend to be clearer and last longer than mental pictures.

One possible way to organize both the work and the documentation is a kind of stepwise refinement which we have used successfully and will now describe in more detail.

We first establish and document the overall architecture of the new system, defining a small number of main modules and their interdependencies. We then develop and document each module in turn, giving it is own chapter, or even its own volume if the program is a large one.

In the description of each module, the main goal of programming and testing is divided into a number of *sub-goals*, each sub-goal signifying that a step in the development has been completed. The implementation of a goal is done by writing about the problem, possibly discussing alternative solutions, and describing the solution itself including the required source code. A sub-goal has its own chapter containing the description of all work involved for attaining the goal, including test data and test results. The document thus reflects the development process.

We have found that if programs become intolerably complex or show many bugs that are hard to track down, there is usually one of two possible causes for the problem. One possibility is that the foundations we have laid in the previous steps are lacking in functionality or quality. Another possibility is that we do not really understand the computing process we are trying to code.

When we discover that the foundation we have built in previous steps is lacking in functionality or quality, we go through the document to find the best place to introduce the required modification or addition. Sometimes we find that we have overlooked something, making us insert a new subgoal at the appropriate place in the document. It is a highly satisfactory experience to make relatively small changes to an early chapter, and notice how a large and complex program fragment collapses into a few lines of simple code.

After having made such changes, we recompile the document step by step, performing the old tests after each step and compare them with the old test results to confirm that the system is still working. We may also introduce more sophisticated tests if experience has told us that the original tests were insufficient.

When we backtrack to redo old work in this fashion, we find our own documentation invaluable. It helps us remember not only what we did, but also why we did it the way we did.

If we do not really understand the computing process we are trying to specify, we zoom out, and write a discussion of the computing process itself rather than its implementation. We often find that figures and tables are useful for clarity. The motivation for the written discussion is two-fold. Firstly, it helps us better understand our ideas. Secondly, we know that the description will become part of the final program documentation. This inspires us to write (and think) just a bit clearer than we otherwise might have done.

In some cases we find that we are not even able to write a description of the computing process. We then zoom further out and describe what we are trying to achieve, i.e. we write a detailed description of the current subgoal. We usually find that if we succeed in this, we will also succeed in writing the process description and the precise program code.

It does happen that even the specifications are outside our reach because we do not really understand what we are trying to achieve. This could be caused by our original specification either being self-contradictory, or because there were important cases that we had not considered. We may then have to go through extensive experiments and discussions before we are able to continue the main line of work.

4. Experiences

Several programs have been written using our literate programming environment. A new version of the environment itself, consisting of some 75,000 lines of Smalltalk code, has actually been designed and implemented by the authors using an early version of the environment. Pål Stenslet and Anne Hurlen has used it to write a storage service, Else Nordhagen [6] has used it to write a new text editor.

Our experience indicates that the method requires a fairly good initial idea about what we want to do and how to do it. The authors knew the functionality of the new environment because it is very similar to its predecessor. Else Nordhagen performed extensive experiments with the text module before she felt it natural to write intelligently about its design and implementation. But in all cases, the documentation of the implementation had profound effects upon the final design and implementation of the module.

For some kinds of work, we have found that two persons working together on one workstation are very productive since they challenge each other's clear thinking and immediately document the results of this thinking. In other cases, it may be better for a person to work in isolation for a while, and then submit his document to one of his peers for review. Whatever way we work, it is always a goal to achieve egoless programming and a situation where one programmer may complete what another has started.

We do know that our productivity as measured in statements per working day is low, we have not measured how low it is. But this is not surprising since we spend a fair proportion of our time massaging the design and the code, making it simpler, more powerful, more general and incidentally also smaller and easier to understand. The number of statements produced on a good day is therefore often negative. We do believe, without being able to prove it, that the total cost of the programs developed is reasonable considering their functionality and quality.

The environment has proved to be of very good help during the design and development stages. The environment seems less convenient in the testing stage, because it is somewhat cumbersome in our present environment to find a specific code part in a document. Because of this, we have often just appended new sections at the end of the document describing the corrections.

Inexperienced programmers trying to adapt to our current programming style have found much help and support in our literate documents. But they tend to need some additional information. The overview of the programs found through the Smalltalk Browser is of great help.

No maintenance or extensions have been done by persons not knowing the system or the original programming culture. We therefore have no experience with how suitable the documents are for this purpose. But we hope that the fact that the whole programming process is documented, including assumptions and decisions made during the development, will help the future maintainers.

5. Conclusions

Our general conclusion is that Knuth's idea of literate programming is a very powerful aid in the program development process, and we are very enthusiastic about this method of programming. We find that the mixing of general descriptions (text, pictures and tables) with precise program fragments is very powerful. For a reader, the general description supply the background information needed for appreciating the code. For the authors, describing the problem and its solution is an active aid in the problem-solving process. We have also found that recording our ideas and assumptions in this way helps us correct errors and introduce extensions when later work indicates that this is desirable. The visibility of all the material provides a tremendous support for working in teams.

The technique of literate programming is no panacea, of course. The quality of the documents is very person-dependent, and the writing of high-quality pedagogic material is hard work at the best of times.

The method does not seem appropriate in turbulent times, such as during initial experimentation or final testing. The reason may be that the analogy between a program and a textbook is not appropriate for real programs. The knowledge presented in a textbook is usually well known and stable, much work can therefore be put into its polished presentation. In contrast, many real world programs are undergoing continuous modifications and extensions, rapidly making the original "textbook" obsolete.

We seem to have two contradicting conclusions: The method of literate programming is extremely useful during some stages of the development process, but the analogy it is built on is not entirely appropriate. We believe that the style of presentation should be sacrificed for enriched structure. We envisage the material of a literate program document organized in a general hypermedia structure with facilities for browsing and printing this material in any way appropriate for any purpose. We call this "the technical documentation" metaphor to emphasize its high utility and low artistic value.

6. Acknowledgements

The Literate Programming Environment was implemented as an extension of an existing system called the *DPS*[7][8], an extensible work environment for professionals now being marketed by Taskon AS. Its first application was to the preparation of bids for engineering projects [10]. Our particular extension was implemented in a few man-weeks of work.

The results reported in this paper have been produced by a team working closely together. The team members are Arne Jørgen Berre, Tor Bothner, Anders Bremer, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Næss-Ulseth, Pål Stenslet and the authors.

The computer technology that forms the basis of our work was developed at the Xerox Palo Alto Research Center (PARC) in the seventies[4]. Our information environments gain their power, user-friendliness and extensibility from the use of Smalltalk, the most advanced system for the development of user interfaces in existence today. We will therefore extend special thanks to Alan Kay, Adele Goldberg and their team in the Xerox Palo Alto Research Center (PARC) for the creation of the wonderful world of Smalltalk, and for the support given to the authors and their colleagues.

7. References

1 Jon Bentley: Programming Pearls. *Comm. ACM 29, 5 (*May 1986), 364-369

2 Ward Cunningham, Kent Beck: A Diagram for Object-Oriented Programs. *Sigplan Notices, 21,11* (November 1986), 361-367.

3 Ward Cunningham, Kent Beck: *ScrollController Explained, an Example of Literate Programming in Smalltalk.* CR-86-53, Computer Research Laboratory, Tektronix, Inc., 1986.

4 Adele Goldberg, David Robson: *Smalltalk-80. The language and its implementation.* Addison Wesley, Palo Alto, California, 1983.

5 Donald E. Knuth: *TEX: The Program.* Addison Wesley, Boston, Massachusetts, 1986.

6 Else Nordhagen: A New Text Editor Implementation for Smalltalk-80. *SI-report (830303) EKI-N-42.* Center for Industrial Research, Oslo, Norway, December 15, 1987.

7 Gro Oftedal: Taskon EndUser Support. *Taskon AS, PB. 6 Blindern, N-0314 OSLO 3.* Oslo, Norway 1988.

8 Gro Oftedal, Carl Petter Swensson: User Guide for Document Production. *Taskon AS, PB. 6 Blindern, N-0314 OSLO 3.* Oslo, Norway 1988.

9 Trygve Reenskaug: Literate Programming. An Early Experiment with Knuth's Method for Programming Smalltalk. *Memo 830303-ree-24.* Center for Industrial Research, Oslo, Norway, June 1986.

10 Trygve Reenskaug, Eirik Næss-Ulseth: Tender One, An Environment for Tender Preparation. *Ninth International Cost Engineering Congress, Oslo, Norway.* 1986. Pages B-4 ff.

11 Trygve Reenskaug, Anne Lise Skaar: A Literate Programming Environment Written in Smalltalk. *EKI Technical Note 830303-44.* Center for Industrial Research, Oslo, February 1987.

12 Larry Tesler: The Smalltalk Environment. *Byte 6, 8* (Aug. 1981) page 90 ff.