Reverse Literate Programming

Markus Knasmüller

Johannes Kepler University Linz
Altenbergerstraße 39
Linz, 4040, Austria
Tel. +43 732 2468 7133
Fax +43 732 2468 7138
Internet knasmueller@ssw.uni-linz.ac.at

Abstract. Knuth's Literate Programming system allows an author to design and describe a program hierarchically according to the method of stepwise refinement. The result is source code, which can be read sequentially like a book, section after section. This helps when reading printed source code, but on screen source code is read rather selectively like an encyclopedia. There the programmer wants a system which allows, possibly even encourages, selective browsing; zoom in at interesting points; jump to other locations according to control flow or other semantic relationships. This is the approach chosen by hypertext systems. In this paper, we demonstrate a solution, called Reverse Literate Programming, which combines the advantages of Knuth's method and of the hypertext approach. We implemented active text elements making it possible to have the source code as a hypertext screen document. A special command prints this document as a Literate Program, i.e. an essay, including documentation, pictures, and program code. The implementation was done in the Oberon system which offers powerful mechanisms for extending software in an object-oriented way.

1. Introduction

When developing or reusing software components, one is often forced to study source code. In many cases this code contains only little or even no design information, which makes it hard to read and understand it. This is the main reason, why software maintenance costs dominate over all other life-cycle costs. Standish (1984), for example, discovered, that the time needed to understand source code, causes 50 to 90 percent of the maintenance costs. Therefore high software quality, in form of readable source code, is the best way to save time and money for new developments.

During the last years, different tools for constructing readable and maintainable code were suggested. One of these approaches, Literate Programming by Donald E. Knuth (1984), allows an author to design and describe a program hierarchically according to the method of stepwise refinement. The result is source code, which can be read sequentially like a book, section after section. This helps while reading the printed source code of a program. On the other hand, on screen programs are usually read rather selectively like an encyclopedia. The programmer does not want to have the source code in book form, but rather the opposite: a system which allows, possibly even encourages, selective browsing. Zoom in at interesting points or jump to other locations according to control flow or other semantic relationships. This is the approach chosen by hypertext editors. Unfortunately, hypertext editors lack the possibility to generate good and sequentially readable printed source.

Our solution combines the advantages of Knuth's method and of the hypertext approach. We implemented active text elements for hypertext features like folding, linking and bookmarking in a framework of an extensible editor (Szyperski, 1992; Mössenböck and Koskimies, in press). Starting with traditional source code, these features can be used to write Literate Programs. A Literate Programming Section is marked with fold elements in order to allow on demand expanding and collapsing of sections. Comments and pictures, helpful for explaining the source code of a section, can be associated with these fold elements. These annotations are shown (on demand) wherever the name of the corresponding section appears. Relations between sections are set through inserted link elements. Links are used in a similar way as references within an encyclopedia. In addition, graphical elements, timestamp elements, and elements for command execution can be inserted into the source code.

A special print command prints the source code (the hypertext screen document) as a Literate Program, i.e. an essay, including documentation, pictures, and program code.

A proof-of-concept prototype implementation has been done in the Oberon system (Wirth and Gutknecht, 1989), which offers powerful mechanisms for extending software in an object-oriented way. However, the solution is language independent, and can be used for all kinds of programming languages.

2. Literate Programming

Literate Programming is a construction and documentation technique for computer programs. The traditional comments are replaced and enlarged to full program documents. The aim is to write programs, which are not only understandable for the computer, but also for the human reader. A program should be designed as a book.

2.1. The structure of a program document

A program document is divided into sections. This dividing into sections is equivalent to the stepwise refinement of the program. Thereby the following aims should be reached:

- Each section should be short and simple to understand.
- There should only be few and simple relations between a set of sections .

Each section consists of a title, a comment about the section, a definition part, and a source code part. All these parts are optional.

Figure 1 shows an example for such a program document.

Figure 1 - A Literate Programming document

1. Hello World program.
This is a small demonstration of the use of Reverse Literate Programming in a program that prints the famous “Hello World” greeting.

```
MODULE HelloWorld;
   <Import Statement>
   <Output Procedure>
END HelloWorld.
```

See also Sections 2, 3.

2. Import Statement.
The only imported module required is Out. Module Out provides a set of basic routines for formatted output of characters, numbers, and strings. It assumes a standard output stream to which the symbols are written.

```
<Import Statement> =
IMPORT Out;
```

This code is used in Section 1.

3. Output Procedure.
This procedure, called Do, uses the operation Out.String to write the string "Hello World" to the standard output stream.

```
<Output Procedure> =
PROCEDURE Do*;
BEGIN
   Out.String ("Hello World")
END Do;
```

This code is used in Section 1.

See also Section 2.

Example for a Literate Programming document

2.2. The title

The title should sum up the job of the section in a few words like it is done in a title of a literal work.

2.3. The comment part

The comment part can be created optionally and should explain the contents of the section. It is the job of the programmer to write down all relevant information, making the program understandable for another programmer.

2.4. The definition part

The definition part consists of macro definitions. Such a macro can be used to sum up a piece of source code under one name, which can be used at other places in the source code. During the compilation these names are replaced with the original source code. Such macros support a clear design.

2.5. The source code part

The program code part contains standard programming language instructions, but also text macros. Such macros (enclosed by the characters “<“ and “>”) represent source code, which is specified in another section. Macros help to structure a program. Footnotes show the section, in which a used macro is specified, as well as other sections also using this macro.

2.6. Working with the Literate Programming system Web

In Knuth’s Literate Programming system Web (1984) the programmer has to write a program using TeX macros. This leads to a rather unreadable on screen source code. Two tools are offered by Web:

- Tangle: A tool creating a source code file from the TeX input. This source code file can be translated by a compiler.
- Weave: A tool creating a full program document from the TeX input.

3. Active Text For Structuring Source Code

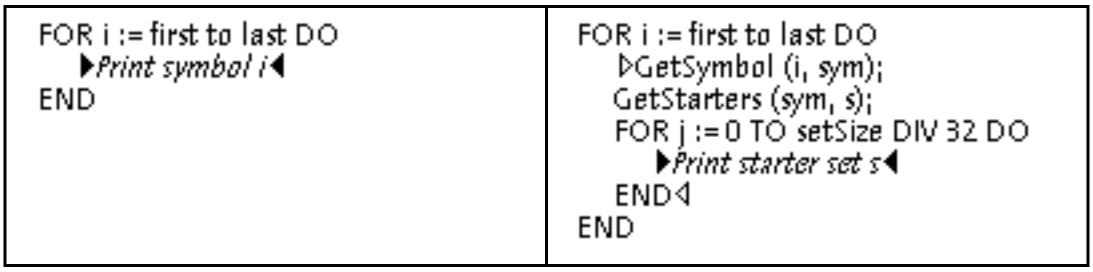
This section shows active text elements for structuring and understanding source code. Elements can be pictures, tables, formulas, popup buttons, hypertext objects, or anything else. They are treated as special characters with a certain width and height, but with unspecified contents. Such elements are a feature of Oberon’s text framework (Szyperski, 92), but they are similar to Microsoft’s OLE (object linking and embedding) architecture (Brockschmidt, 94). Mössenböck and Koskimies (in print) show additional information about active text elements.

3.1. Folding

The idea of folding is, that a piece of text can be collapsed and, possibly, replaced with some other (usually shorter) text. The user can switch between the two texts using the mouse. Folds can also be nested. This can be used for abstracting, commenting, and structuring source code. For our case, the two functionalities, abstracting and commenting, are of interest.

Figure 2 shows how to use fold elements for abstraction. The left part of the figure shows a code fragment, where certain details are folded away, and replaced by a simpler pseudocode statement. If the user wants to see the details, he can “zoom in” and get the expanded view (right part of the figure). He can switch between the two views with a simple mouse click.

Figure 2 - Folding



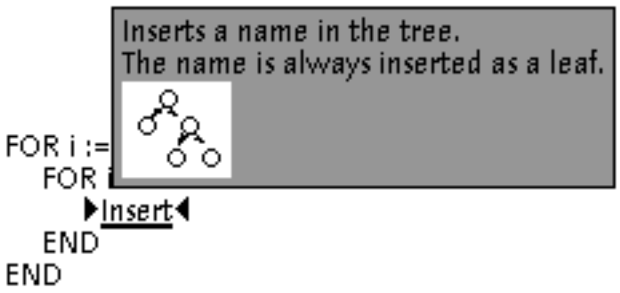
```
FOR i := first to last DO
    ▶Print symbol i◀
END
```

```
FOR i := first to last DO
    ▷GetSymbol (i, sym);
    GetStarters (sym, s);
    FOR j := 0 TO setSize DIV 32 DO
        ▶Print starter set s◀
    END◁
END
```

Abstracting from details

Folding can also be used for commenting. Instead of writing a comment next to the code the user may also wrap the comment into fold elements. A special mouse click (left and middle button) on the fold element, and it shows the comment in a popup window. This process can be seen in Figure 3.

Figure 3 - Commenting



Inserts a name in the tree.
The name is always inserted as a leaf.

```
FOR i :=
   FOR i
      ▶Insert◀
   END
END
```

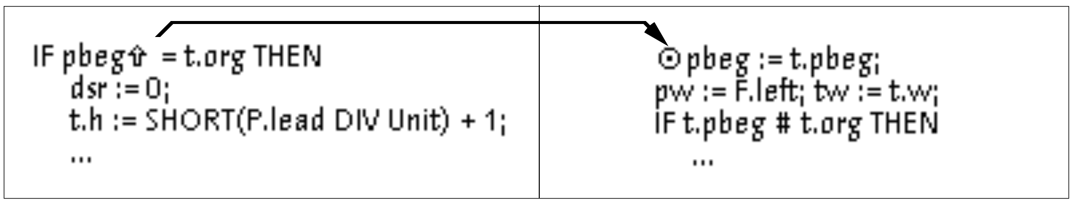
A popup comment for a collapsed program code

3.2. Linking

The original idea of hypertext is to link related pieces of information. This is in analogy to an encyclopedia, where a user can look up a term and find links to other terms, which he may follow.

In our system, the starting point of a link is represented by a link element (⇧) and the end point is represented by a mark element (⊙). When the user clicks on a link element, the corresponding mark element, and its surrounding text is shown in a new window. Figure 4 shows an example where the use of a variable is linked to the statement where the value of this particular use was assigned. Other examples include the linking of code to its documentation, to its specification, or to test data.

Figure 4 - Linking



```
IF pbeg⇧ = t.org THEN
    dsr := 0;
    t.h := SHORT(P.lead DIV Unit) + 1;
    ...
```

```
⊙pbeg := t.pbeg;
pw := F.left; tw := t.w;
IF t.pbeg # t.org THEN
    ...
```

A link between two related pieces of code

4. Reverse Literate Programming

This solution combines the advantages of Knuth's method and of the hypertext approach. Table 1 shows how the elements, introduced in Section 3, can be used to write Literate Programming documents.

Table 1. - Representation of Active text as a Literate Program

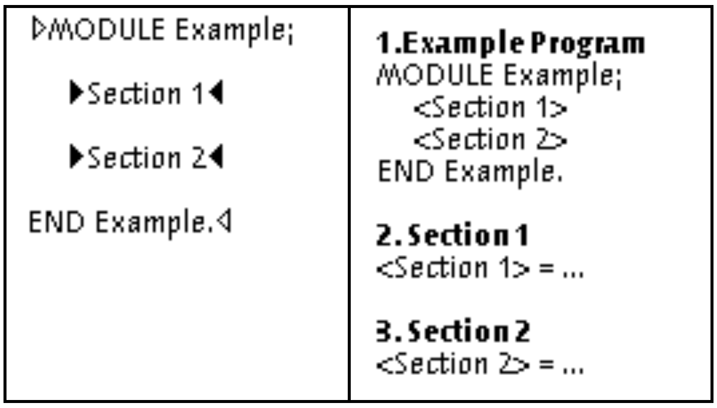
Active text	**Literate Program**
Fold element	Section with a macro definition
Collapsed source of fold element	Source code part
Comment of Fold element	Documentation part
Link element	Relation between sections

A special print command prints the source code (the hypertext screen document) as a Literate Program, as an essay, which includes documentation, pictures, and program code. While printing, all fold elements are logically collapsed. The Main Section is printed first. It introduces the program skeleton. In the source code part of a section each occurrencing fold element is replaced by a macro. Each macro is later specified in a section of its own. Figure 5 shows how text elements are transformed into Literate Programs. The left part of the figure shows a hypertext document, the right part the corresponding Literate Program.

Link elements are represented by footnotes, which indicate inter-section relationships to the reader.

The remaining source code is taken as it is, i.e., font styles, and additional text elements are copied unchanged into the printed document.

Figure 5 - Short example



```
▷MODULE Example;

    ▶Section 1◀

    ▶Section 2◀

END Example.◁
```

1.Example Program

```
MODULE Example;
    <Section 1>
    <Section 2>
END Example.
```

2. Section 1

```
<Section 1> = ...
```

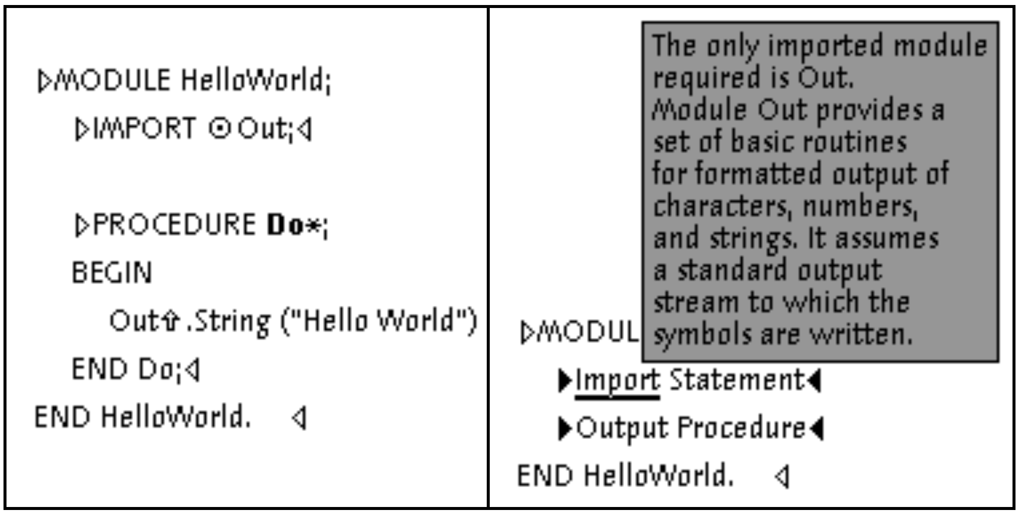
3. Section 2

```
<Section 2> = ...
```

Representation of Fold elements in a Literate Program

Figure 6 shows a sample source code, structured with fold elements. It is a small demonstration of the use of Reverse Literate Programming in a program, that prints the famous “Hello World” greeting.

Figure 6 - “Hello World”



```
▷MODULE HelloWorld;
    ▷IMPORT ⊙Out;◁

    ▷PROCEDURE Do*;
    BEGIN
        Out⇧.String ("Hello World")
    END Do;◁
END HelloWorld.    ◁
```

The only imported module required is Out. Module Out provides a set of basic routines for formatted output of characters, numbers, and strings. It assumes a standard output stream to which the symbols are written.

```
▷MODUL
    ▶Import Statement◀
    ▶Output Procedure◀
END HelloWorld.    ◁
```

Example input for Reverse Literate Programming

The left part of the figure shows the hypertext document with all fold elements expanded. There is a link between the call of the operation *Out.String* and the import statement for module *Out*. The right part of the figure shows the wrapped comment of the *Import Statement* Section. Figure 1 shows the corresponding Literate Programming document (with comments added to Sections *Output Procedure* and *Main*).

5. Conclusions

5.1. Comparison

Compared to Knuth’s Web system, we offer better readable source code on the screen. Figure 7 shows the TeX input necessary to construct the programming document shown in Figure 1.

Figure 7 - Standard Web input

```
@* Hello World program.
This is a small demonstration of the use of Reverse Literate Programming in a program that prints
the famous “Hello World” greeting.
@c
MODULE HelloWorld;
@<Import Statement@>
@<Output Procedure@>
END HelloWorld.
@*Import Statement.
The only imported module required is Out. Module Out provides a set of basic routines for
formatted output of characters, numbers, and strings. It assumes a standard output stream to
which the symbols are written.
@c
@<Import Statement@> = IMPORT Out;
@*Output Procedure.
This procedure, called Do, uses the operation Out.String to write the string "Hello World" to the
standard output stream.
@c
@<Output Procedure@> =
PROCEDURE Do*;
BEGIN
  Out.String ("Hello World")
END Do;
```

An example input for the standard Web system

Furthermore, since elements are implemented as special characters with attributes containing a reference to the object, the compiler simply ignores the element character when compiling a text containing elements. This makes it possible, to compile a Literate Program, including graphical elements or links, with the standard compiler. This is another advantage compared to Knuth’s Literate Programming system, where a special 'tangle' phase is necessary, before the compiler can compile the program document.

Although newer Literate Programming systems offer WYSIWYG editors (Brown and Childs, 1990; Knasmüller, 1993), we see, that the on-screen representation of such programming documents, is not as good as in our approach, because programs are not read sequentially like a programming document, but rather selectively like an encyclopedia. This is possible with our approach.

In comparison with other hypertext editors, we can say that these editors offer, like our approach, a very readable document on the screen, but, in contrast to our approach, a rather unreadable printed code.

5.2. Future work

We intend to include further text elements in our Reverse Literate Programming system, e.g. elements including formatting hints. Furthermore, it should be considered, whether a multi-stage title hierarchy would be better than the presently used single-stage title hierarchy. Also, the construction of indices (used idents and used macros) should be implemented.

We use our approach, to implement an object-oriented database system (Knasmüller, 1996). We think, that this is an sufficiently large example to evaluate the usability of our system for Reverse Literate Programming.

The newest information about Reverse Literate Programming can be obtained via WWW from http://www.ssw.uni-linz.ac.at/Projects/RevLitProg.html.

6. References

Brown, M. and Childs, B. 1990. An Interactive Environment for Literate Programming. *Structured Programming*. **11(1)**: 11-25.

Brockschmidt, K. 1994. *Inside OLE 2*. Washington: Microsoft Press.

Knasmüller, M. 1993. *Ein grafischer Editor für Literate Programming*. Diploma Thesis, Linz: Johannes Kepler University.

Knasmüller, M. 1996. *Adding Persistence to the Oberon System*. Linz: Johannes Kepler University.

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

Mössenböck, H. and Koskimies, K. in press. Active Text for Structuring and Understanding Source Code. *Software - Practice and Experience.*

Szyperski, C.A. 1992. Write-ing Applications: Designing an Extensible Text Editor as an Application Framework. In *Tools 1992*. Dortmund: 247-261.

Standish, T.A. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*. **10(5)**: 494-497.

Wirth, N. and Gutknecht, J. 1989. The Oberon System. *Software - Practice and Experience*. **19(9)**: 857-893.