AALBORG UNIVERSITY

Literate Smalltalk Programming using Hypertext

by

Kasper Østerbye

R 93-2025 ISSN 0908-1216 August 1993

Reference information:

This paper was obtaind by ftp from iesd.auc.dk
file pub/reports/tech-reports/R93-2025.ps.Z.
Author can be contacted by email at kasper@iesd.auc.dk

INSTITUTE FOR ELECTRONIC SYSTEMS
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
Fredrik Bajers Vej 7 — DK 9220 Aalborg Ø — Denmark
Tel.: +45 98 15 85 22 — Fax.: +45 98 15 81 29

Abstract

The problem examined in this paper is: How and to what extent can hypertext support literate Smalltalk programming? In order to examine this question we have created a hypertext system which allows us to write Smalltalk code and documentation, and to link code and documentation freely. We have used the system for different tasks, and based on an examination of the resulting hypertext documents we reflect on our experience, which is that hypertext can be used for literate programming. Our experience is, however, that the literate programs created in hypertext differ from those created with traditional literate programming tools.

Keywords: Programming environments, literate programming, hypertext, Smalltalk.

1 Introduction

This paper will investigate hypertext in the domain of software engineering. Specifically, we will examine how, and to what extent, hypertext can be used to support literate Smalltalk programming, and how Smalltalk programming in a hypertext based system differs from Smalltalk programming in the traditional Smalltalk-80 environment.

In [10] Knuth presents literate programming as a special way of programming, where the creation of the program is intertwined with its documentation and rationale. To support literate programming, Knuth designed a system called WEB; in WEB the program and its documentation are written in a single file, which can then be processed to create either a document intended for human reading (which is why it is called *literate* programming) or a program to be handed to a compiler. The document created for human reading provides a rich set of cross-references and indexes, to help the reader find his way around the document.

Hypertext has at several occasions [1, 7, 20] been proposed as a technique for managing all life-cycle documents in a software development project. The common theme is that hypertext has something to offer both at the modeling level and at the user interface level. Hypertext will allow us to create a connected structure which makes explicit the relations between various documents created throughout the lifetime of a software project.

To fully examine the use of hypertext to support all aspects of software development would be an enormous task. Instead we will take a detailed look at a much smaller area, that of literate programming, where there is already a proposed solution, the WEB system, for maintaining relationships between code and documentation; a solution to which we can compare our findings. In this context, the applicability of hypertext can be stated as in the abstract: *How and to what extent can hypertext support literate Smalltalk programming?*

Behind this question lies a belief that hypertext can provide the same proximity between code and documentation and provide similar index facilities for the author as those found in WEB style literate programming. Furthermore, we believe that hypertext will allow for more flexible ways of relating documentation and code than the WEB system.

When we need to distinguish between literate programming using WEB and using hypertext, we will refer to the former as WEB-programming, and the latter as hypertext-programming.

In order to examine the question, we have built a hypertext system that allows us to write Smalltalk code and documentation, and to relate code and documentation freely.

The rest of this paper is organized as follows: In section 2 the idea of literate programming will be examined, and we will especially look at how hypertext can be used to support literate programming. Our experimental system will be described in section 3, and we will give an example of a hypertext literate program in section 4. In section 5 we will examine some other systems that have been built to support literate programming. We will end the paper with a summary and conclusion, and some directions for future work.

2 Background

This section will give a short introduction to hypertext and literate programming.

2.1 Hypertext

Hypertext is a way of organizing information, with particularly convenient means to relate different pieces of information to each other. The information to be handled by the hypertext system is divided into *nodes* (see figure 1). Each node typically stores about a page of information, though most hypertext systems do not place any restriction on the size of its nodes. The relationships between different pieces of information are represented using *links*, which tie together two (or more) nodes. Often the links are *anchored* to specific points or regions within a node; this makes it possible to relate not only nodes, but also individual words or sentences to each other. Both nodes and links are typed to allow for different semantic interpretations of both node contents and link-relations.

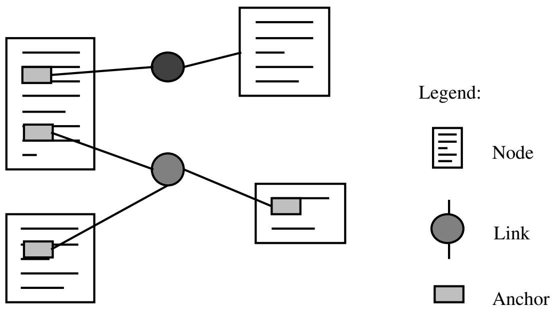


Figure 1: *A simple hypertext with four nodes and two links, one link relating three nodes through anchors, and one link relating two nodes though an anchor and the whole node.*

Most hypertext systems are to various degrees multi media systems (hypermedia systems), allowing nodes to contain graphics, sound, and video in addition to text. From a software engineering perspective, the emphasis will typically be on text and various forms of structured graphics.

The typical way of accessing information in hypertext is through navigation. At the interface level, a node appears in a view of its own, and anchors for the links relating the presented node to others are clearly visible. By clicking the mouse (or some other simple gesture), at the anchor, the other nodes of the link relation are brought up, each in their own view; this is know as *following* the link. If a link is not anchored to the internal of a node, there is no visible presentation of the link. Such a link must be followed by selecting it from a menu or by similar means.

Hypertext systems commonly allow the attachment of attributes to both nodes and links. Such attributes are simple name-value bindings, and are often used to contain author names, last update date etc.

Almost all modern hypertext system supports yet another entity type *composites*, which can contain other entities, much like a directory in a file system. The main purpose of composites is to provide a way of partitioning the document, but it can be used for a variety of purposes, including results of queries [19] and versioning [13, 8].

The paper by Conklin [3] provides an excellent introduction to hypertext.

2.2 Literate programming

The term literate programming was coined by Knuth in [10]. The idea behind literate programming is that the programmer should consider the program as a piece of literature intended for humans to read, rather than a set of instructions for a computer to follow. Instead of only concentrating the programming effort at getting the program to work, the programmer must also realize that the program is to be read by others, and pay due attention to the explanation of how the program works.

There are several claimed benefits of literate programming. By explicitly stating how the program works, and why it is organized the way it is, the quality of the program should be improved, because it forces the programmer to state the solution both in plain text and as a formalized program. Literate programs are easier to maintain, in that there is a detailed description of what the program does, and most likely some textual description of how one piece fits together with other pieces.

The rest of this section will be organized as follows: First we will describe the WEB system that Knuth designed in order to realize his ideas. We will then look at its strong and weak sides, and finally we will examine some experiences from using WEB.

The WEB system

The first WEB system was designed for the programming language Pascal, and the typesetting language TeX. A WEB program consists of a series of numbered paragraphs. Each paragraph may consist of three parts: documentation, macro definitions, and program text; all three parts being optional.

There are two things one can do with a WEB file. One can use the WEAVE program to produce a typeset presentation of the program, or one can extract the program for compilation using the program TANGLE.

Part of the typeset presentation is an extensive set of cross-references indicating which paragraphs depend on each other, and a table of contents with the name and number of each paragraph. Finally an index is generated in which each identifier is listed with references to where it was declared and where it was used.

The program text is written using a combination of the programming language proper, and references to other paragraphs. This means that the order of the paragraphs does not need to be (and rarely is) the same as one would find in the program, but rather the order the author finds most appropriate for the exposition of the program.

Strong sides of WEB

The idea behind literate programming is that documentation and program are seen as one product, and not as two separate artifacts. This is supported in WEB by having the two things appear in the same file. In the actual file, the different paragraphs that makes up the literate program are coherent units, where each paragraph divided into the same three parts, documentation, macros, and code.

When a programmer need to change something, he will find the code together with the documentation in the text file he is editing.

Seen from a program development perspective, one of the real advantages of WEB is that it allows the programmer to present the program in a rational top-down fashion, rather than having to stick to the rules of programming language. Aspects of a procedure that are not relevant in the current paragraph can be deferred to another paragraph.

Weak sides of WEB

A weakness of WEB is that it adds yet another layer of complexity to the programming environment. Where a programmer previously only had to know the programming language, he must now master both the programming language, TeX, and WEB.

Another drawback is that we are back to the times when we were doing program development on paper rather than using programming tools. Cross-references and indexes produced by the weave program are not available online, and are not updated incrementally, which means that they are not of any immediate help to the author of the literate program.

Finally, WEB only addresses the need for a clear explanation of the program *per se*, and provides no means for relating that exposition to other documents from earlier phases of the system development. And it is not clear that it can be extended to do so. First, the WEB system is built around the idea of paragraphs, where the documentation part mainly concentrates on program exposition and not rationale; too much discussion of alternatives will clutter the exposition of what actually takes place in the program; second, the automatic index facilities of WEB is based on a single document approach, and it is not feasible to store all project documentation in a single document.

Experiences from using WEB

In [15] Ramsey and Marceau report on their experience from using WEB for a non-trivial project (33.000 lines of code). Their overall impression is that literate programming did provide good support for development and maintenance.

They do, however, report on some shortcomings of WEB[1]

1. There is a need to include materials other than plain text, especially figures and diagrams (It is very complex to produce figures in TEX).
2. Programmers complained more about pretty-printing than about anything else. (WEB ignores the indentation done by the programmer when typesetting program fragments)
3. No WYSIWYG. When one is programming, a lot of time is spent reading the WEB-source of the screen. WEB-source is *very* different from WEB-output. Especially in this project where a lot of mathematical notations were involved.

3 The system

To test the idea of literate programming with hypertext, we have created a prototype system based on the Hyperbase and Smalltalk kernel used in HyperPro [14], with a simple Smalltalk based user interface.

This interface and its integration with Smalltalk is described in this section. First, we will describe the general hypertext part, then the Smalltalk part, and finally, how the two are interrelated.

3.1 The underlying datamodel

In the datamodel used in this system the entities are organized in an inheritance hierarchy, so properties defined for a supertype also hold for a subtype. The types are as follows:

Entity. The general entity provides attributes, which are key/value pairs attached to the objects. Entities must be members of at least one Context, see below. In our datamodel, it is up to the user to specialize the built-in types of the system to suit the specific needs of concrete applications.

Node. The node is the basic entity for storing contents. We require all nodes to have a name.

Link. The link is an entity that relates a source node to a destination node (or subtypes of nodes). Links can be anchored to points within its source/destination node. However, the interface of the Smalltalk system discussed in this paper supports only source anchoring.

Context. The contents of a context node is a sequence of entities. The main purpose of a context is to provide the notion of hierarchical structure. All entities are members of at least one context[2].

3.2 A simple interface

The user interface to our hypertext system is quite simple. We provide two basic types of presentations. Context nodes are presented as lists of the entities they contain, and textual nodes are presented in windows where the text can be edited using the simple standard Smalltalk editor. Following usual Smalltalk conventions, the left mouse button is used to position the text cursor and select items, the middle button brings up a menu of operations that can be performed in the view with the selected items, and the right mouse button is for window management.

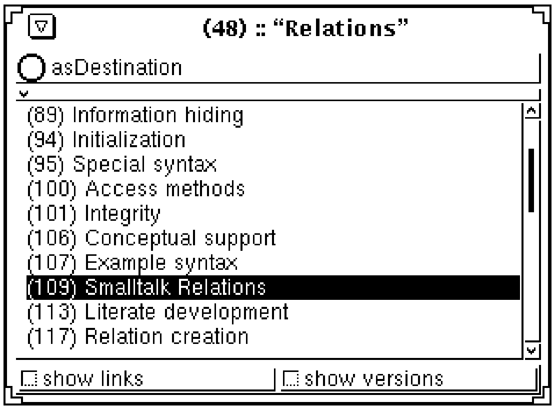
The main purpose of the interface has been to provide a means for testing our thesis regarding hyperstructured programming; the interface should therefore not be seen as our proposal for a hypertext interface. In the paper [14] we outline our design of a generic hypertext interface.

When we start the system, a presentation of

[1] Other topics are also discussed, but they are not important for our discussion.

[2] The only exception is the outermost composite in a hyperdocument.

the outermost context is brought up. While the datamodel allows for contexts to form cycles, the interface can currently only create strict hierarchical nesting of contexts. A context view is illustrated in view 1.



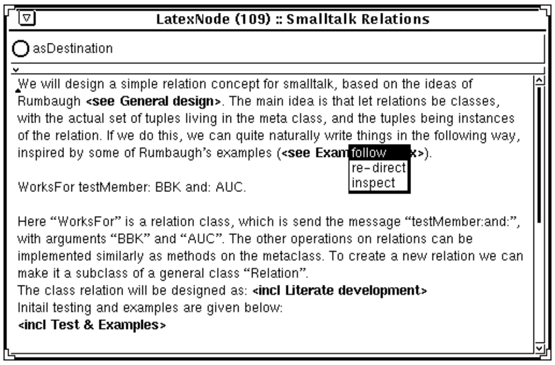
View 1.

The basic hypertext operations one can perform in a context view is to select entities and have them presented in a window of their own, and to create new nodes and delete entities. Contexts can be made targets for links using an "asDestination" button (the large bar across the top of view 1). Link creation will be described shortly.

The interface primarily supports textual nodes. When a node is selected in the context view, an editor view with the selected entity can be spawned (see view 2). The text of the node can be manipulated using standard editing commands.

Embedded links will appear as "<link anchor label>", that is some name between angle brackets. In view 2, the link anchors are shown in bold, but on the screen they appear in a dark purple font as well as in bold, which makes them clearly distinguishable from ordinary bold text. When the mouse is over a link marker, using the middle mouse button will bring up a menu, with the default selection being to follow the link.

Since links do not have a destination anchor in our interface, the incoming links are not shown directly in the text. One of the middle mouse commands will bring up a list of "back-links", so



View 2.

one can follow incoming links to their source. If the incoming link has a source anchor, that anchor will be highlighted, when the link is backfollowed.

While our datamodel do not place any restriction on which types of links it will support, the interface is only aware of three kinds: hierarchical, cross-reference, and "other" links (it is possible to create other types of links, but that is cumbersome in the present interface). Hierarchical links are shown as "< incl name of dest. node>", and cross-reference links are shown as "< see name of dest. node>".

The usual way to create new nodes is to place the cursor in an existing text, and select "new section" from the menu; This creates both a new node, and a hierarchical link to that node, anchored at the cursor position.

To create a cross-reference, one first locates the destination node, and presses a button marked "asDestination" (The large bar across and above the text in view 2). Then the source node is located, the cursor is placed where the anchor is wanted, and finally one selects "addXreference" from the menu, and the link is created.

To redirect an existing link, one will need to locate the new destination, and press "asDestination". Then the anchor of the link to be redirected is located, and "re-direct" is selected from the menu.

Besides the text node and context, the interface supports a limited sort of graphics intended merely for experimentation. A special node type has been defined, and a viewer has been implemented, that allows for the presentation of bitmaps. These bitmaps cannot be edited from within the system, but can be targets of links, and it is possible to specify rectangular regions to be used for source anchors. Though this is a very limited support for graphics, even this has proven to be useful for documentation purposes. If attributes “filename” and “tool” are defined, we can spawn off the actual drawing tool for manipulation of the actual drawing rather than the bitmap. The bitmap must then be recaptured from the screen, and the link anchors must be updated to sit in the right places.

3.3 Embedding Smalltalk in hypertext

The programming language chosen for our experiment is Smalltalk-80; Smalltalk programs are particularly well suited for hypertext because they are quite naturally divided into node sized chunks. And as the datamodel was already implemented in Smalltalk it was also relatively easy to provide integration with the Smalltalk system.

Several node and linktypes have been specified to cater for Smalltalk code. A Smalltalk class consists of a class definition and a number of methods. All methods are organized into protocols, and likewise classes are organized into categories.

The hypertext representation of this is illustrated in figure 2. We have created two node types, one to hold class definitions, and the other to hold methods. Methods are related to their class through a “method” link. An attribute at the link stores the protocol name of the method[3]. Class definition nodes are connected to their superclass through a “super” link, and to their meta class through a “meta” link.

[3]The protocol name is stored with the link so that the very same method text can actually be used in two different classes under different protocols. We have not used this feature, however.

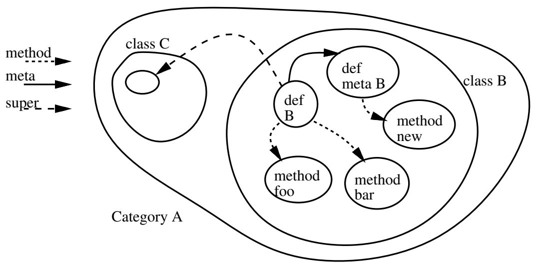


Figure 2: *Hypertext representation of Smalltalk code. Categories and classes are represented as contexts, and class definitions and methods are represented as text nodes.*

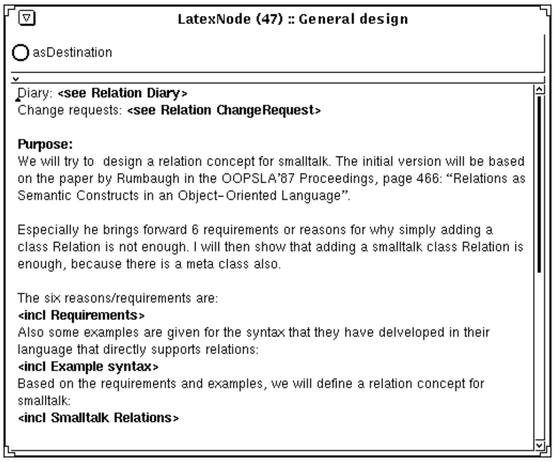
Besides the basic network representation, some hierarchy is provided to prevent the user of the system from having to deal with too large a hypertext at a time, which is particularly nice in the context browsers which are often used to access nodes directly rather than through link following. All nodes and links concerning a single class, its meta class and their methods are grouped in a class context. All classes of the same category are then again grouped in a context of their own.

The different types of nodes (class definitions and methods) are shown using the text editor described earlier, but the available menu commands differ. For both types of nodes it is possible to import and export definitions to and from the Smalltalk image. For methods, the menu gives easy access to the class node, and for class definitions it is possible to bring up a menu of all methods, as well as to create a new method node which will be linked to the class.

Besides these special facilities for handling Smalltalk, both types of Smalltalk nodes provide the usual hypertext linking facilities, so that source code can be linked to other source code or plain text.

4 An example

To gain some experience with hypertext based literate Smalltalk programming we have carried out a number of small programming tasks (about twice the size of those literate programs that originally were published in the Communications of the ACM [21, 22]). The next pages will be a tour of of one of these hypertexts, where we highlight our observations regarding the use of hypertext for literate programming, and illustrate these with examples from the hypertext document that was created. We conclude this section with a summary of our findings; regarding both hypertext for literate programming, and some observations with respect to Smalltalk programming in a hypertext environment as opposed to the traditional Smalltalk-80 programming environment.

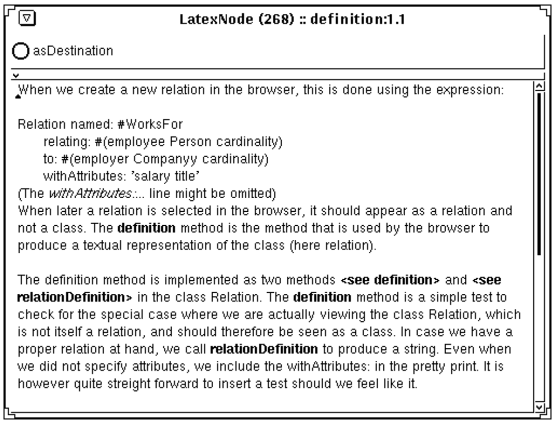


View 3.

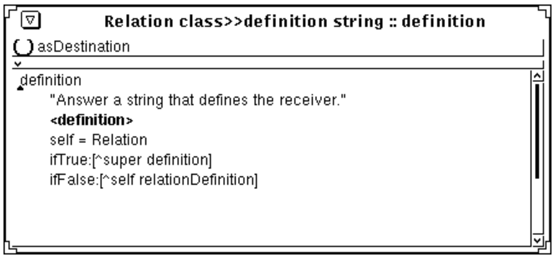
Explicit background material
The overall purpose of the program is stated in view 3. Here we start with an explanation of what we want to do. We have retyped important passages from the paper [17] (not shown in this node, but available by following the "requirements" and "example syntax" links). This material has proved quite valuable throughout the programming task, as a guideline and goal for what we wanted to do, and it is frequently referenced from the nodes where more detailed design is discussed.

In view 2 we outline the general approach to relations in Smalltalk.

- *For documentation purposes, we frequently found it useful to enter background material into the hypertext.*



View 4.

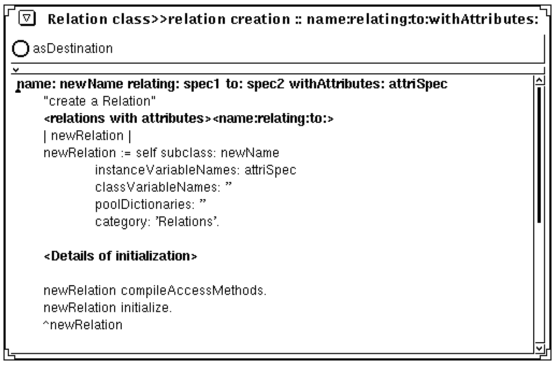


View 5.

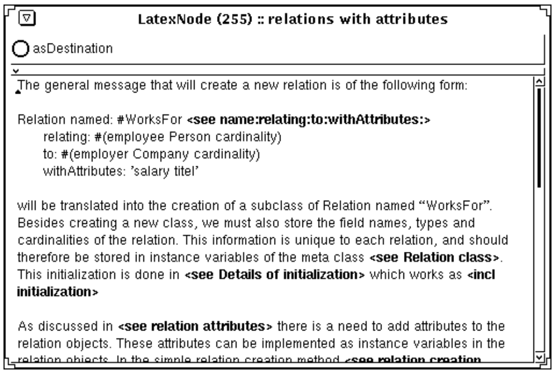
Design rationale and literate commentary
View 4 explains the rationale behind the definition method. This kind of rationale and explanation is often much more useful than more detailed code descriptions. The actual code of the definition method (see view 5) is, as stated in the commentary, quite simple and would not deserve much comment in itself. Both this node and the node in view 9 explains some common aspects of the program which is implemented by several co-operating methods.

- *This is a recurring theme throughout the entire hypertext: the things we found worth documenting were aspects relating several methods and variables.*

This is different from WEB-style literate programming, which does not in any direct way support commentary of several code fragments. The problem is that WEB is intended to produce a linear text, where each fragment is associated with a single commentary. One can, of course, refer to other paragraphs in the commentary; but one cannot have several code fragments associated with the same commentary. In our hypertext system we can have several node views on the screen at the same time, and can link to the same code fragment from several commentary nodes.



View 6.



View 7.

Hyperstructured source code

When working in the traditional Smalltalk browser, one must deal with the source code one method at a time. We have sometimes found it useful to divide larger methods into several nodes, rather than to split the method into several smaller methods (and thereby create a new scope where there is no lexical access to parameters and locals). This is illustrated in view 6, which implements the relation creation method discussed in view 7. Here we see that some details regarding initialization have been moved to a seperate node. This is similar to the way fragmentation takes place in WEB style literate programming. As with the code for the definition method (view 5), we have created an explicit documentation link, that will bring one to the commentary that deals with this code. Several documentation links could, of course, be created if it was seen as appropriate (more often than not, however, a single method does one thing, and therefore gets one documentation link).

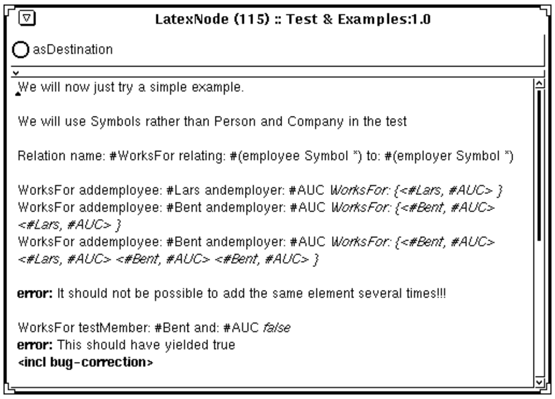
Besides the documentation link and the link to further details of initialization, we have a link to some other code (next to the documentation link) which was considered to be interesting as well (In this case it is a method for creating relations *without* attributes).

As can be seen at the end of the method, we call two other methods "compileAccessMethods", and "initialize". It would be useful to have links to these too. If such links were to be created manually the program would soon become impractical to maintain; such cross-references should be created and maintained automatically. The present system does not automatically create links. Here the compiled approach of WEB has an advantage, in that it automatically creates such cross-references.

Testing the program

Besides being useful for providing background material, design rationale, and literate commentary, the plain text nodes can also be used as Smalltalk workspaces; that is one can enter and evaluate Smalltalk expressions in ordinary text nodes. This can be used to link test code and data into the hypertext quite naturally (cf. view 2).

View 8 is an example of such a simple test. Here we try to create a simple "WorksFor" relation and some expressions, to check if the relations behave as expected (the slanted text is the result of evaluating the preceding expression). As



View 8.

can be seen, the code did not work at first attempt. A node is then created that explains what the problem is and how to address it.

In this specific case, it turned out to be a rather technical problem relating to the semantics of sets and hashing. The test and bug-correction nodes were used as explanation and rationale for the changes needed.

- *While we cannot always expect the programs to be of such a nature that we can test them in a simple text node, we have found it useful to be able to provide example expressions in the hypertext itself.*

Versions and commentary development
The 1.0 number after the title of view 8 indicates that this is version 1.0 of this node. After the bug correction was carried out, a new version of the test node was created (and the test passed).

It is not quite clear to what extent versions are useful in explaining programs. On the one hand, it gives a very true story of the development, and it tends to separate the explanations into smaller sized chunks, with the old nodes concentrating on the basics, and the new ones addressing various complicating factors. On the other hand, a reader of the hypertext is most likely interested in a concise explanation of a working program rather than in its whole evolution.

One can imagine that both the process-oriented exposition style and the concise exposition can be useful. The concise exposition will be important for someone who is trying to understand the program, in order to use it or adapt for some other purpose. The process-oriented style will be useful to a person who is trying to track down a bug; a bug that may have been introduced as a consequence of the new version. In that case it is necessary to understand both versions and their relationship, as well as why the new version was created.

4.1 Findings

In this section we will report on our findings from using a hypertextual interface for Smalltalk programming. The first subsection will highlight our findings regarding hypertext for literate programming, and then some aspects of using hypertext for Smalltalk programming will be addressed.

4.1.1 Hypertext for literate programming

- WEB-style literate programming concentrates on providing literate commentary. Hyperstructured programming extends this to include test and rationale as well.

- WEB-style literate programming facilitates and favours an exposition of single program fragments whereas hyperstructured exposition promotes and supports cross-structural aspects.

- Versioning plays a role in evolutionary exposition, but it is not clear how to utilize this properly.

- Hypertext based literate programs are intended to be read off the screen, and we therefore directly address critique points 2 and 3 of Ramsey and Marceau (see section 2.2). Our simple way of including graphics addresses point 1, though better support is certainly needed.

- When browsing the literary commentary in hyperstructured programming, one can get lost in the hypertext. This is a well known problem with hypertext, and we have two things that help us navigate. First the commentary is organized in a strict hierarchical , which enables us to move up from a detailed paragraph to a chapter; second, we provide a graphical browser of the hierarchical structure.

4.1.2 Hypertext based Smalltalk environment

Besides the benefits of being able to link Smalltalk code with other types of information, several aspects of the hypertext representation of Smalltalk are interesting in their own right.

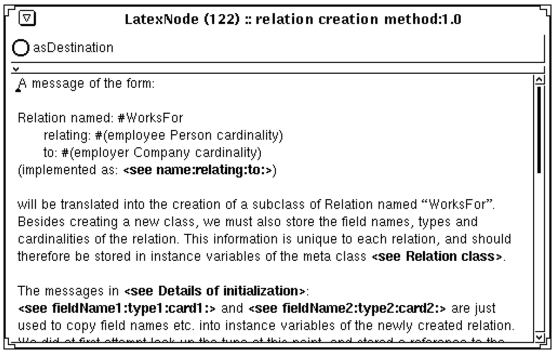
1. It is very easy to move methods around between classes, since all we need to do is to change a link.

2. The class definition is now stored as text rather than decompiled from its internal representation, as is done in the Smalltalk-80 system browser. The problem arose when one used a non-standard expression to create a class. In that case, the generating expression is lost using the browser. The problem and a possible solution is discussed in view 3. If the browser stored the class definition text as it stores methods, and as we do in our hypertext system, this would not be an issue.

3. It has proven useful to be able to break large methods up into smaller nodes. Normally methods are not large, but every now and then one gets a method that will not reasonably fit into a single view, and it can then be broken up into smaller pieces. Only when it is installed into the Smalltalk image do we linearize it into a single text. A typical example of a large method is a method that composes a window from a number of subviews.

4. A side effect of the interface is that it is simple to have the class definition in sight in its own window while one is writing a method for that class. It is very annoying not to be able to do this in the standard Smalltalk browser.

5. As we are storing everything outside the Smalltalk image (in a shared database), we can leave a method only half finished, another thing that the standard browser does not support.



View 9.

6. The documentation has proved to be a useful “mini browser” as illustrated in view 9, where the commentary refers to four code fragments which co-operate to perform a single task. The commentary thus provides an annotated mini browser that provides easy access (by link following) to *related* methods and class definitions.

7. Our hypertext system brings multiuser support in sight. All source code is stored in a common repository, and one could imagine that we provided further integration with the Smalltalk kernel to allow several images to share source code. However, this is not an aspect we have investigated in this project.

5 Relation to other work

Other projects have developed environments for literate programming in Smalltalk, or hypertext environments to support literate programming.

One system is developed by a group headed by Reenskaug at Senter for Industriforsking

[16] (Hereafter called the SI-system). The SI-system organizes the literate program into a hierarchical structure of sections and paragraphs, where each paragraph is either text, graphics or Smalltalk source code. An advanced browser allows for viewing the literate document at both an outline format, and at a detailed level. The source code is manipulated at the level of individual methods and class definitions.

The system is intended to produce paper output for human reading, and to allow easy installation of the source into the Smalltalk image. The paper output is provided with an index of class and method definitions. However, considerable effort has been put into the literate programming browser/editor to make it convenient and pleasant to work with the literate program on-line as well.

The SI-system and their approach resembles our in a number of ways. Like us, they do not use any specific formatting language. We depend on the simple facilities provided by Smalltalk, which seems also to be the case for the SI-system. They are, however, able to produce a printed version of their text. We have not yet found a satisfactory way to print our hyperstructured programs. The main problem is where to place the code fragments, as each fragment is typically referenced from several places in the commentary part.

Though phrased differently, they have experienced the versioning problem too. When a program undergoes initial experimentation or testing, the program changes rapidly, it is cumbersome to update the commentary, and it feels more natural to document the rationale behind the changes than to rephrase the entire documentation to fit the new understanding. Based on this they conclude:

> "We envisage the material of a hypermedia structure with facilities for browsing and printing this material anyway appropriate for any purpose. We call this "the technical documentation" metaphor to emphasize its high utility and low artistic value[4]." [16, page 344]

Another system for literate programming in Smalltalk was developed at Tektronix by Cunningham and Beck [6][5]. Their system is somewhat simpler than ours and the SI-system, but has at least one interesting new aspect. Neither our system nor the SI-system provides any facilities that can aid in the *exposition* of Smalltalk programs. We both rely only on text and general graphics. The Tektronix system includes a special notation invented by Cunningham and Beck [5], which is particularly well suited to explain the dynamics of Smalltalk programs. Language specific exposition notations is a very interesting aspect of literate programming. So far, tailoring literate programming (both WEB-style and hyperstructured) to a specific language has only been a question of pretty-printing or fragmenting the program, and to collect appropriate cross-reference information.

An aspect that is common for all three systems is that they are all written in Smalltalk to support Smalltalk development, and as such all systems extend the WEB-style paradigm to include facilities for interacting with the underlying image in the form of "do-it" expressions for testing, and import and export of source code to and from the standard image.

In [2] Brown describes a hypertext system to manage WEB source code. The main objective is to provide index facilities similar to those available in the paper index. The WEB source is divided into a number of nodes corresponding to WEB paragraphs. Each node therefore contains a documentation section, a macro section, and a code section. A special node editor that is aware of the internal structure of WEB modules is provided. To support the different types of indexing, five different link types are supported, corresponding to different kinds of indexes.

This is a quite straightforward translation of WEB into hypertext. However, we believe that it will be beneficial to be less stringent than

[4]Ramsey and Marceau also refrain from claiming their literate programs to be works of literature; they see them more like "car repair manuals" [15, pages 678–679].

[5]We have only had access to a draft version of their paper, so we may occasionally have misinterpreted their work.

WEB, allowing for a more flexible relation between code and documentation.

Another interesting system, NelleN, is described by Leonard and Prince in [11]. Here the literate programming paradigm is transferred to the domain of data modeling in relational database systems. The main focus of the system is to combine literate programming with the framework for deliberation schemas as known from the hypertext system gIBIS [4]. Compared to our system, NelleN is much more restricted in its intended usage, and does not contain any possibilities for arbitrary linkage. The idea of combining literate programming with deliberation schemas is interesting though, as it might point to a way of separating program exposition from rationale, while maintaining both as important aspects.

6 Future directions

The experiences with our simple hypertext system for literate Smalltalk programming point to a number of new areas to be explored. In this section we will examine some of these.

Support for other languages

The current system is restricted to handling Smalltalk code only. But besides being able to evaluate Smalltalk expressions directly in the text nodes, there are actually very little of the system which is language specific. It would therefore be interesting to create a system than can support other languages in a generic way. To support other languages we need to be able to tailor the system in a number of ways:

- We must be able to specialize the datamodel to be able to accommodate the structures of the language.
- We must be able to linearize the program part of the hypertext so we can handle it to a compiler; this is the hypertext equivalent to TANGLE.
- We must be able to provide means for supporting the creation of the programs at a more intelligent way than using a plain hypertext. This can be seen as a parallel of the functionality in the Smalltalk experiment where we can create a method and link it to its class in a single interface action.

We are currently finishing our first prototype of a generic hyperstructure programming environment which addresses these needs [12, 14]

Automatic incremental index generation

As mentioned on page 8, our system does not automatically create and maintain links. It might not be that important to be able to do incremental generation of cross-references and indexes. It depends on the usage of the hypertext literate program. As far as we can see, there are at least the following types of usage

- Browsing through an existing program
- Authoring a new program
- Modifying an existing program

When one is browsing through an existing program, it is important that the index and cross-references are up to date and complete; otherwise one could misinterpret the structure.

Assuming that the author of a new program has an interpretation of the structure he is creating, it should not be of vital importance that the secondary link structure is up to date at all times.

And assuming that one who has to modify the program has been through a browsing phase first, he will have learned the overall interpretation beforehand. In this case, however, we are not certain whether the secondary link structure can be left outdated.

The practical conclusion is that one may be able to use some kind of batch approach to maintain the secondary link structure. Further investigation is needed to see how such links should be maintained. Another aspect of this is how to provide a generic way to support links that depend on the semantics of the language. To create a whole parser and semantic analyzer for each language we want to support seems to be a high a price to pay. However, our experience shows that there is a need for such a secondary link structure.

Hyperstructured programming

We believe that representing programs as hypertext and being able to relate programs using hypertext links has a lot of potential beyond what is offered by the WEB-style literate programming. We call this new way of programming *hyperstructured programming*. Some of the potentials and directions of hyperstructured programming are:

- Literate review. Hypertext has been used as a way to annotate and review electronically stored documents. The paper by Johnson and Tjahjono [9] describes how they use hypertext for code review. An interesting path to be explored is how to combine their review mechanisms with the idea of reviewing literate programs as done in [22].

- One problem with WEB-style literate programming is that one cannot include too much discussion about alternatives, since that will have a tendency to blur the decisions that were actually made. In hypertext we can keep exposition and rationale separated, so only when necessary do we consult the rationale. Furthermore, there is already some work in the area of hypertext which discusses how to capture design rationale, and even how to structure a discussion of design alternatives e.g. the gIBIS system by Conklin [4] and the Author's Argumentation Assistant by Schuler and Smith [18].

- The work by Leonard and Prince [11] indicates that the idea of literate programming can be transplanted into the area of databases. Likewise, it would be interesting to explore how to do literate design and literate analysis, and to rethink the boarders between analysis, design and programming in the context of literate action; hypertext is interesting in this respect, in that it allows for a close coupling between the documents produced throughout the life cycle of a software project, while the contents of the individual nodes can be kept concise.

However, much research needs to be carried out before we can say if it is possible to fulfil those visions, and maintain hyperstructured programming as a new programming paradigm.

7 Summary and Conclusion

We see hyperstructured programming as a way of doing literate programming that differs from using the WEB system; our way of intertwining code and documentation is at the same time both looser and tighter than in WEB style literate programming. It is looser in the sense that code and documentation do not appear together in the same window. When one opens a code view, one does not see any documentation right away. On the other hand, one can say that code and documentation is tighter coupled, in that cross-references from a discussion of a single global effect can be linked to all the pieces of code that are affected; such cross-references are not available to the author in WEB, and only available to the reader of the weaved document in the form of cross-references. When one compares hyperstructure literate programming to WEB-style literate programming, it is important to notice that there is no WEAVED output! *The primary outcome of hyperstructure style literate programming is the hypertext itself.*

Besides the findings regarding hypertext as a medium for doing literate programming, we have found several interesting aspects of using a hypertext based interface for Smalltalk programming as opposed to the traditional Smalltalk-80 interface.

Have we been able to support our thesis as stated in the introduction? We believe that one can do a kind of literate programming using hypertext; the literate programs are different from those made using WEB, however. In this sense we believe that the thesis is supported. The problems with using a simple hypertext system like ours for literate programming are:

1. The cross-references and indexes that are generated automatically by the WEB system are missing, or must be maintained by hand. On the other hand, those that are set up manually, are available to the authors when they are working at the screen.

2. The notion of getting lost in hyperspace, that is, one looses the feeling of where the information presently available at the screen belongs in the overall document, is also present here. The same problem does arise in larger WEB-documents too, though.

3. The possibility to easily link the program exposition to other parts of the system documentation does not ease the problem of keeping the documentation consistent. In traditional development, one produces several documents; as users and developers of such documents we are used to live with a certain inconsistency, and we know which documents are old and which are new. In hypertext based systems we do not know when we get hold of an old piece of information.

4. Finally, hypertext programming does not produce any tangible product; this is actually one of the very fine virtues of WEB-style literate programming. The WEB-programmer can take the weaved typeset output and hold it in his hand and say "See how good this thing looks". One should not underestimate that aspect of WEB programming.

The prototype system has reached a point of development where we use it on a daily basis for Smalltalk programming and writing in general. Currently a few students are beginning to use the system and we hope to be gaining further experiences from their usage.

References

[1] James Bigelow. Hypertext and CASE. *IEEE Software*, pages 23–27, March 1988.

[2] M. Brown. A hypertext for literate programming. In S. G. Akl, F. Fiala, and W. W. Koczkodaj, editors, *Advances in Computing and Information, ICCI'90. LNCS 468*, pages 250–259, 1990.

[3] Jeff Conklin. Hypertext: An introduction and survey. *IEEE Computer*, September 1987.

[4] Jeff Conklin and Michael L. Begeman. gIBIS, a hypertext tool for exploratory policy discussion. *ACM Transaction on Office Information Systems*, 6(4):303–331, October 1988.

[5] Ward Cunningham and Kent Beck. A diagram for object-oriented programs. *Sigplan Notices*, 21(11):361–367, November 1986. OOPSLA'86 Proceedings.

[6] Ward Cunningham and Kent Beck. Scrollcontroller explained, an example of literate programming in smalltalk. Technical Report CR-86-53, Tektronix, Inc., 1986.

[7] P. K. Garg and W. Scacchi. Ishys - designing an intelligent software hypertext system. *IEEE Expert*, 4(3):52–63, Fall 1989.

[8] Anja Haake. CoVer: A contextual version server for hypertext applications. In D. Lucarella, J. Nanard, N. Nanard, and P. Paolini, editors, *ECHT'92, Proceedings of the ACM Conference on hypertext*, pages 43–52, 1992.

[9] Philip Johnson and Danu Tjahjono. Improving software quality through computer supported collaborative review. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work*, September 1993.

[10] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[11] Michel Leonard and Ian Prince. NelleN: a framework for literate data modelling. In *Advanced Information System Engineering, CAiSE'92*, pages 239–256. Springer-Verlag, 1992. LNCS, vol 593.

[12] Kurt Nørmark and Kasper Østerbye. A hypertext interaction engine for a generic programming environment. In Preparation.

[13] Kasper Østerbye. Structural and cognitive problems in providing version control for hypertext. In D. Lucarella, J. Nanard, N. Nanard, and P. Paolini, editors, *ECHT'92, Proceedings of the ACM Conference on hypertext*, pages 33–42, 1992.

[14] Kasper Østerbye and Kurt Nørmark. The vision and the work in the HyperPro project. Technical Report R-93-2012, Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, April 1993.

[15] Norman Ramsey and Carla Marceau. Literate programming on a team project. *Software-Practice and Experience*, 21(7):677–683, July 1991.

[16] Trygve Reenskaug and Anna Lise Skaar. An environment for literate smalltalk programming. *Sigplan Notices*, 24(10):337–345, October 1989. OOPSLA'89 Proceedings.

[17] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In Norman Meyowitz, editor, *OOPSLA'87 Proceedings*, pages 466–481, 1987.

[18] Wolfgang Schuler and John B. Smith. Author's argumentation assistant AAA: A hypertext-based authoring tool for argumentative texts. In A. Rizk, N. Streitz, and J. André, editors, *Hypertext: Concepts, Systems and Applications*, pages 137–151. INRIA, Cambridge University Press, November 1990.

[19] Helge A. Schütt and Norbert A Streitz. Hyperbase: A hypermedia engine based on a relational database management system. In A. Rizk, N. Streitz, and J. André, editors, *Hypertext: Concepts, Systems and Applications*, pages 95–108. INRIA, Cambridge University Press, November 1990.

[20] John B. Smith and F. Donelson Smith. ABC: A hypertext system for artifact-based collaboration. Technical Report TR-91-021, University of North Carolina at Chapel Hill, April 1991.

[21] Jon Bentley with Donald E. Knuth. Programming pearls: Literate programming. *CACM*, 29(5):364–369, May 1986.

[22] Jon Bentley with Donald E. Knuth and M. D. McIlroy. Programming pearls: A literate program. *CACM*, 29(5):471–483, June 1986.