A HYPERTEXT SYSTEM FOR LITERATE C++ PROGRAMMING

Johannes Sametinger and Gustav Pomberger

Institut für Wirtschaftsinformatik
Johannes Kepler University of Linz
A-4040 Linz, Austria

Abstract

Programs are written to be executed by computers rather than to be read by humans. This complicates program comprehension, which plays a major role in software maintenance. Literate programming is an approach to improve program understanding by regarding programs as works of literature (see [Knu84]).

In this paper we present a tool that supports literate programming with the object-oriented programming language C++. The tool is based on a hypertext system which has been developed to support program comprehension and, thus, software maintenance.

An introduction of the basic hypertext system is followed by a presentation of the global structure of the documentation. Afterwards, the user interface of the tool, its use and a sample documentation are given. Some implementation aspects conclude the presentation.

Introduction

Development programmers hate documentation which, therefore, almost never is either complete or consistent. Maintenance programmers need documentation to understand the software system they are responsible for. But they have to be satisfied with the listing, the only accurate and complete representation of the system. Understanding programs is one of the most time-consuming activities in software maintenance. By improving the availability of complete and up-to-date documentation, we can minimize software costs considerably.

When writing programs, we should not try to instruct the computer what to do, but rather we should try to tell humans what we want the computer to do. This is the main idea of literate programming (see [Knu84], [Ben86], [Lin89], [Lind89], [Wyk90]) which leads to significantly better documentation. But as humans and computers do not speak the same language, there is a need for systems that support the idea of literate programming.

The WEB System [Knu84] — the original literate programming system — offers a combination of a document formatting language for the documentation text and a programming language for the source code (Pascal). With WEB the source code and its documentation can be written in parallel and in a single document. The system then produces both a clearly readable program description, intended for human readers, and pure source code, intended for compilation and execution on the computer.

This work was supported by Siemens AG, Munich, Germany.

Several tools for literate programming have been developed so far. They support different languages like C ([Tun89], [Thi86]), Fortran [Ave90], and Smalltalk [Ree89]. Even a generator for WEB-systems is available [Ram89], which can be used to construct instances of WEB from a language description. But these systems, like the original one, are clumsy to use and assume that a program and its documentation are written straightforward.

We have based our tool on hypertext technology [Con87] to ease the navigation through a software system, both the source code and the documentation. The tool not only supports documenting systems from scratch but also offers the opportunity to document already existing systems.

The Documentation Problem

Programmers always have an abstract model in their mind of what they are coding. It is important to write down these thoughts in order to make understanding easier.

Writing down these thoughts is hardly ever done, for one reason because there is no place to write it. If you write it in a separate file, you do not have any connection between the source code and the corresponding documentation file. But this connection is essential to guarantee consistent documentation. Besides, you can hardly refer to specific parts of the code. The only way is to copy it, but this brings an unresolvable update problem.

Another possiblity is to write the documentation straight into the source code (with comments). But this again has its disadvantages:

- The source code becomes illegible because of the numerous comments.
- Programmers familiar with the code do not want to be annoyed by this bulky information.
- The code needed to solve a problem usually is spread over several places in the source code. (This is especially true in object-oriented systems.)

Literate programming tools attempt to solve these problems. However, most existing tools support the documentation of new software. They are not very well suited to support the documentation of already existing software. This would require the modification and restructuring of the existing source code.

The literate programming tool we present in this paper supports both the documentation of new and already existing software systems written in the object-oriented language C++. Furthermore, it offers hypertext features to comfortably navigate through the system, both the source code and the documentation text.

The Hypertext System

Our documentation tool is based on a hypertext system that was developed to support software maintenance [Sam90]. We briefly describe this maintenance tool because it is fundamental to the documentation tool.

The hypertext editor eases the process of navigating through the files and classes of a C++ software system and helps the user to get information in a fast and easy way. To accomplish this, the files of a C++ program are divided into little pieces of information, i.e., class definitions, method implementations and global declarations. (By global declaration we mean anything that does not belong to a class definition or to a method implementation, like preprocessor statements or global type declarations.)

These little chunks of information are managed together with their interrelations. The following relations are used:

- A class is contained in a file.
- A class inherits from another class.
- A method is contained in a file.
- A method belongs to a specific class.
- A method is overridden in a subclass.
- A file is included by other files.

Still other relations exist based on identifiers used in a software system:

- An identifier is defined in a class or method, or is global to a file.
- The use of an identifier is related to a specific definition of this identifier and to other uses of the same identifier.
- A comment possibly contains a short description of an identifier, e.g., the description of a class, a method, or an instance variable.

Our maintenance tool offers the possibility to easily browse through the system by means of the above relations. Additionally, useful information is displayed to protect the user from getting lost in the complex information web.

In order to enhance the readability of the source code, the user can define global styles for different syntactic constructs, e.g., comments, keywords. Additionally, it is possible to highlight single identifiers or identifiers defined in a certain scope. This helps the user to easily answer questions like *Which global variables does this method use, and where are they used?*

The Documentation System

The maintenance tool has been enhanced with documentation features to a literate programming environment. The system enables its users to write documentation during the design and coding process and offers easy access to written documentation during the maintenance process.

In the following we first describe the general structure of the documentation built with the presented tool. Then we present the user interface and provide sample documentation.

General Structure of the Documentation

We define our documentation structure to consist of a set of documentation chapters. These chapters are organized very similiar to classes; i.e. they are spread over several files and a hierarchical structure exists among them. A chapter itself consists of a title, documentation text (that is pure ASCII text) and program text (actually a link to the source code).

There are two possiblities to use program text in the documentation:

1) single identifiers
2) any text part, i.e. any number of lines of code in succession within a class definition, a method implementation and anything else in a file

Single identifiers can be used right within the documentation text, e.g.:

…
With these constants we add two entries in the menu **identMenu** in the method **CreateMenuBar** of the class **HyperTextDocument**.
…

Source code identifiers are shown boldfaced. The user can choose any font or style for these identifiers. One or more code lines are clearly separated from the documentation text, for example:

…
These menu items to copy or paste an identifier are enabled only when the source code has been parsed (i.e. ,**alreadyParsed** is set to TRUE) and when the current mark (**cm**) or the last mark copied is an identifier. The method **CopiesIdent** provides this information.

```
if (cm && cm->CopiesIdent())
   identMenu->EnableItem(cIdentCopy);
if (GetClipMark() && GetClipMark()->CopiesIdent())
   identMenu->EnableItem(cIdentPaste);
```

Finally, whenever one of these menu items (copy or paste of identifiers) is selected by the user, the method **DoMenuCommand** is executed.
…

Unlike the WEB system and other literate programming tools, the source code is not really intermixed with the documentation text. This means that the source files do not contain documentation and the documentation files do not contain source code. There are only links between them and the tool merges the source code into the documentation for browsing.

User Interface

The user interface concept is based on modern application frameworks and the supported concepts thereof (see [Shn86], [Wei89]). It provides a menu bar, two selection lists, an information box, and an editor window (see Fig. 1).

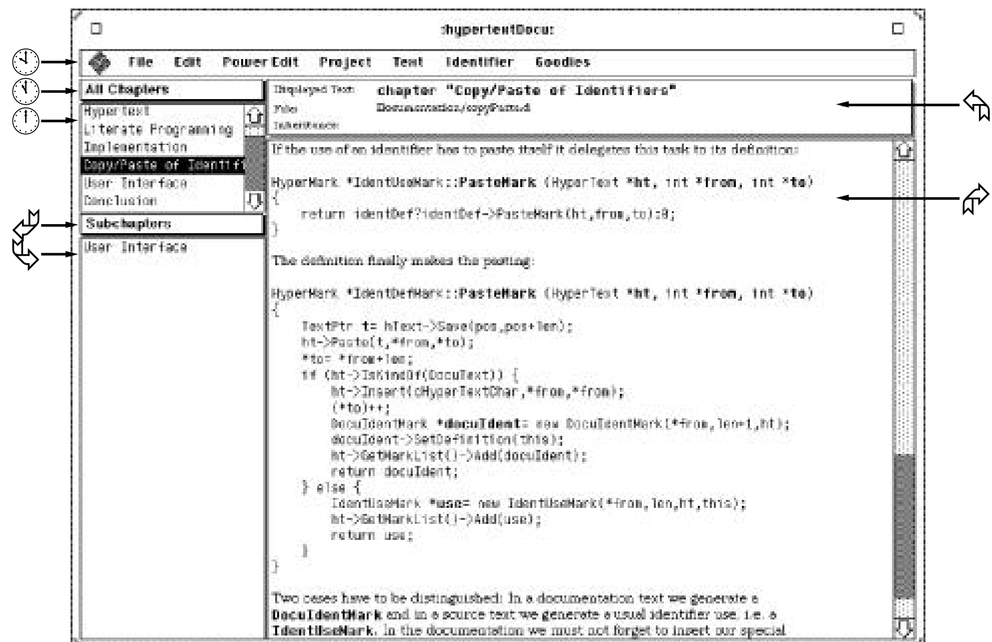

Fig. 1: User Interface

The Menu Bar

The menu bar (¿) is used to perform different commands. It is clearly separated in several groups for commands concerning files, editing, project attributes, and so on.

The Selection Lists

The first list (¬) shows all the nodes of the information web of the software system (all chapters, all classes, all files) or a subset thereof (depending on various categories). In the second list (*ƒ*) nodes are shown that have a certain relation to the one selected in the first list (e.g., subclasses, subchapters, included files). The text bar above the first list (¡) indicates the category of nodes that can be seen in this list. The text bar above the second list (v) indicates the relation among the nodes of this list and the node selected in the first list.

The Information Box

The information box (˜) is used to display useful information about the text part currently under inspection, i.e., the name of the text currently shown (file, class method or chapter), its filename, and the inheritance path (i.e., all the superclasses or superchapters).

The Editor

The editor window (?) displays the text part depending on the selections made in the *Selection Lists* on the left side. It offers usual editing capabilities (like cut, copy, and paste) and text processing facilities.

Writing Documentation

Documentation should be written at the time of coding. Unfortunately, this is seldom done, either because of time constraints or because of the fact that code changes, which necessitates a time-consuming change of the documentation, too.

Therefore our tool supports both cases, writing documentation together with the source code and writing documentation for already existing source code.

The source code for fulfilling a task is usually spread over several locations — even several files. This is especially true for object-oriented systems, where many methods are overridden in subclasses.

To write documentation for already existing source code we create a new chapter, write English text and simply include identifiers and text parts from the source. This is done by simply activating the (menu) commands for copying and pasting identifiers and texts.

Modifying Documentation

There are several possiblities to make (implicit or explicit) changes in the documentation:

- Common text editing capabilities (cut, copy, paste) allow any changes in the documentation text to be made.
- An identifier has been changed in the source code:
  The system automatically changes this identifier at all occurrences both in the documentation text and in the source code.
- Any other changes have been made in the source code:

Due to links between source code and its documentation text, these changes are made automatically in the documentation, too.

- An identifier or a text part of the source code that is used in the documentation has been removed, let's say, with another text editor:
  In the documentation we can see the name and the location of the identifier or the text part instead of the actual source code.

Sample Documentation

We introduce a new feature into our software system and describe its implementation in a new chapter. Introducing the new feature usually necessitates the insertion of code at several places in the existing code.

We might start writing the documentation text and the source code in parallel. But this is possible only when we know exactly what to do. We suggest first inserting and testing the source code (supposing it is not too extensive) and afterwards writing down the documentation text and integrating it with the source code.

Copy/Paste of Identifiers

If the user wants to copy an identifier, we just remember the identifier (i.e., the current mark) in a global variable by calling the method **SetClipMark** .

```
case cIdentCopy:
   if (textView->GetCurrentMark())
      project->SetClipMark(textView->GetCurrentMark());
   break;
case cIdentPaste:
   ...
   HyperMark *newMark;
   textView->GetSelection (&from, &to);
   textView->SetSelection(from,to);
   if (project->GetClipMark())
      newMark= text->PasteMark(project->GetClipMark(),&from,&to);
   textView->SetText(text);
   textView->SetCurrentMark(newMark);
   textView->SetSelection (from, to);
   break;
```

If the identifier is to be pasted ,we send the text the message **PasteMark** and provide the mark of the identifier as first parameter. (We get this global variable again by calling **GetClipMark**).
Afterwards we call **SetText**. This guarantees that the styles are set correctly (especially for the one identifier or text we just pasted). Finally, the identifier is selected.

```
HyperMark *HyperText::PasteMark (HyperMark *m,int *from,int *to)
{
   DoDelayChanges dc(this);
   HyperMark *newMark;
   newMark= m->PasteMark(this,from,to);
   //marks are already updated correctly in PasteMark
   SetChanged();
   return newMark;
}
```

We just call the method **PasteMark** of the **HyperMark** that we get as our first parameter. **PasteMark** is a virtual method of the class **HyperMark** which is overwritten in the marks for identifiers (i.e., in **IdentDefMark** and **IdentUseMark**). In **HyperMark** we do not have to do anything:

```
virtual HyperMark *PasteMark  (class HyperText *, int *, int *)
{ return 0; }
```

If an identifier use has to paste itself, it delegates this task to its definition:

```
HyperMark *IdentUseMark::PasteMark  (HyperText *ht, int *from, int *to)
{
    return identDef?identDef->PasteMark(ht,from,to):0;
}
```

The definition finally does the pasting:

```
HyperMark *IdentDefMark::PasteMark  (HyperText *ht, int *from, int *to)
{
    TextPtr t= hText->Save(pos,pos+len);
    ht->Paste(t,*from,*to);
    *to= *from+len;
    . . .
if (ht->IsKindOf(DocuText)) {
    ht->Insert(cHyperTextChar,*from,*from);
    (*to)++;
    DocuIdentMark *docuIdent=new DocuIdentMark(*from,len+1,ht);
    docuIdent->SetDefinition(this);  ht->GetMarkList()->Add(docuIdent);
    return docuIdent;
} else {
    IdentUseMark *use= new IdentUseMark(*from,len,ht,this);
    ht->GetMarkList()->Add(use);
    return use;
}
}
```

Two cases have to be distinguished: In a documentation text we generate a **DocuIdentMark** and in a source text we generate a usual identifier use, i.e., an **IdentUseMark**. In the documentation we must not forget to insert our special character (**cHyperTextChar**), so we can recognize these identifiers when writing the documentation text on a file.

In order to enhance readabilty, it is possible to define global text styles for keywords, identifiers, and comments (see [Sam90]). These global styles are used in the documentation text, too. This is the reason why we can see identifier definitions in boldface. The style for identifiers used within documentation text can be defined similiarly. In the example above these identifiers are also boldface.

Reading Documentation

There are two possible ways to browse through a system. One can either inspect the source code and look at the documentation, where available, or one can read the documentation and look at the source code when the documentation does not provide enough information. (The system facilitates comfortable work with documentation, but it cannot guarantee complete and high quality documentation. This remains the responsibility of the user/developer.)

Similiar to navigating through classes, methods and files the hypertext editor allows easy access to chapters based on their relations among each other. When reading a documentation chapter, one can easily inspect related chapters, i.e., subchapters, the superchapter, and brotherchapters (the subchapters of the superchapter).

There are several possiblities to branch to the source code, because all source code parts used in documentation text offer the same hypertext features as their counterparts in the actual source code:

- One can select an identifier (in the documentation text) and jump to its definition. (The definition is always in the source code.)

- One can jump to the location from which a block of source code is taken/copied from.
- Identifiers are highlighted in the documenation, too. So it is easy not only to find out where an identifer is used in the source code, but also to find its uses in the documentation text.

Whenever documentation text is available for the source code, the menu entry *Show documentation* is enabled. This is the case when a piece of source (i.e., a class, method, file, or part thereof) has been used somewhere in the documentation. Selecting the menu entry is sufficient to branch to the documentation.

Besides, one can highlight one or more identifiers and easily find out whether and where they are used in the documentation.

Implementation

Space limitations prohibit a detailled description of the implementation. Only the basic structure is given.

The presented tool was implemented with C++ [Str86] under UNIX on a SUN work-station using the application framework ET++ [Wei88] [Wei89]. The implementation is clearly separated into three parts: a language-independent hypertext browser (see [Big88], [Con87]), a language-dependent static analyzer (C++) used to get needed information about the inspected program text, and a simple parser for the documentation text.

Language-Independent Hypertext Browser

The language-independent hypertext browser controls the user interface and manages the following information about a software system:

- text pieces (e.g., chapters, class descriptions, method implementations)
- any relations among these text pieces for browsing (e.g., subchapters, inheritance, include relations)
- classification of text parts (e.g., keywords, comments, identifiers)
- relations among identifiers (the definition of an identifier and its uses)
- additional information (e.g., inheritance path, file location)

Based on this (language-independent) information, the tool manages easy browsing through a software system.

Language-Dependent Source Code Parser

The language-dependent parser analyzes the source code, cuts it into small pieces of text (classes and methods), and passes information to the hypertext browser, e.g.:

- the definition of an identifier
- the use of an identifier
- any keyword
- any comment
- any inheritance relation
- the location of files (directory path)

Documentation Text Parser

Documentation text like source code is stored as text. Links to the source code are marked with a special character and consist of the name of the source code part (e.g., ‘class HyperText’, ‘HyperText::Cut’) followed by the name of the identifier or the text part (e.g., ‘class HyperText;marks’, ‘HyperTextDocument::DoMenuCommand;Copy/Paste of Identifiers‘). This external representation is created automatically by the tool. The user does not need to work with these cryptic files.

The following text shows how the documentation chapter *Copy/Paste of Identifiers* presented above is stored in a text file:

```
If the user wants to copy an identifier, we just remember the
identifier (i.e., the current mark) in a global variable by calling
the method @"class HyperProject;SetClipMark".

@"HyperTextDocument::DoMenuCommand&Copy and Paste Identifiers"

If the identifier is to be pasted, we send the text the message
@"class HyperText;PasteMark" and provide the mark of the identifier
as the first parameter. (We get this global variable again by calling
@"class HyperProject;GetClipMark").
Afterwards we call @"class HyperTextView;SetText". This guarantees
that the styles are set correctly (especially for the one identifier
or text we just pasted). Finally, the identifier is selected.

@"HyperText::PasteMark"

We just call the method @"HyperText::PasteMark;PasteMark" of the
@"class HyperMark;HyperMark" that we get as our first parameter.
...
```

The documentation text parser analyzes the documentation files, cuts them into small pieces of text (chapters), and reestablishes the links to the source code.

Current Implementation Restrictions

The tool described has been implemented, but certain details have not yet been completed and are scheduled for inclusion in future improvements.

- All information about a software system is kept in main memory.
  The use of a database is being considered.
- Multiple inheritance is not supported.
- Graphical information representation (graph and tree browsers) is still missing, but should be easy to implement.
- Text processing features have to be improved.

Conclusion

A tool was presented that supports the documentation of C++ program systems by providing a modern user interface and comfortable browsing facilities. The tool supports the concepts of literate programming and hypertext and can also be used for documenting already existing software systems.

We have been using the presented tool in various projects at our institute. Its usefulness has been proven by a drastic reduction of necessary effort to maintain software systems and to avoid error-prone changes while maintaining a software system.

References

[Ave90] Avenarius A., Oppermann S.: FWEB: A Literate Programming System for Fortran8x, SIGPLAN Notices, Vol. 25, No. 1, pp. 52-58, Jan. 1990.

[Ben86] Bentley J.: Programming Pearls: Literate Programming, Communications of the ACM, Vol. 29, No. 5, pp. 364-369, May 1986.

[Big88] Bigelow J.: Hypertext and CASE, IEEE Software, pp. 23-27, March 1988.

[Con87] Conklin J.: Hypertext: An Introduction and Survey, Computer Vol. 20, No. 9, pp 17-41, Sept.87.

[Knu84] Knuth D. E.: Literate Programming, The Computer Journal, Vol. 27 No. 2, pp 97-111, 1984.

[Lind89] Lindsay D.: Literate Programming: A File Difference Program, Communications of the ACM, Vol. 32, No. 5, pp. 740-755, June 1989.

[Lin89a] Lins C.: A First Look at Literate Programming, Structured Programming, Vol. 10, No. 1, pp. 60-62, 1989.

[Lin89b] Lins C.: An Introduction to Literate Programming, Structured Programming, Vol. 10, No. 1, pp. 107-112, 1989.

[Ram89] Ramsey N.: Weaving a Language-Independent WEB, Communications of the ACM, Vol. 32, No. 9, pp. 1051-1055, Sept. 1989.

[Ree89] Reenskaug T., Skaar A.L.: An Environment for Literate Smalltalk Programming, OOPSLA '89 Proceedings, pp. 337-345, October 1-6, 1989.

[Sam90] Sametinger J.: A Tool for the Maintenance of C++ Programs, Proceedings of the Conference on Software Maintenance, 1990.

[Str86] Stroustrup B.: The C++ Programming Language, Addison-Wesley, 1886.

[Thi86] Thimbleby H.: Experience of ‘Literate Programming’ Using CWEB (a variant of Knuth’s WEB), The Computer Journal, Vol. 29, No. 3, pp. 201-211, 1986.

[Tun89] Tung Sho-Huan: A Structured Method for Literate Programming, Structured Programming, Vol. 10, No. 2, pp. 113-120, 1989.

[Wei88] Weinand A., Gamma E., Marty R.: ET++ — An Object Oriented Application Framework in C++, OOPSLA ‘88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, pp. 46-57, 1988.

[Wei89] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No.2, Springer International 1989.

[Wyk90] Van Wyk C.J.: Literate Programming: An Assessment, Communications of the ACM, Vol. 33, No. 2, pp. 361-365, March 1990.