A Tool for the Maintenance of C++ Programs

Johannes Sametinger

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

Abstract

This paper describes a tool that helps programmers understand object-oriented software systems written in C++, a language that is expected to gain widespread use in industry. This task is accomplished by providing information about the set of classes and files comprising the system and the relationships among them. The tool described enables its users to easily browse through the system based on the relations among its classes, files and even identifiers. In addition, the flexible use of global text styles enhances the readability of the source code.

The second part of the paper describes some details about the implementation of the tool. In particular, problems are mentioned that arise when performing static analysis of C++ programs. This analysis is necessary for obtaining information needed about the program system.

The primary goal of developing the tool has been to support software maintenance, but its use is in no way limited to that process.

Introduction

Many facts emphasize the importance of maintenance in the software life-cycle. For example:

- Programmers spend more than half of their time on maintenance [6].
- Most programmers spend 50% of their time on maintenance, and some spend up to 80% of their time on the task [13].

The most difficult problem in changing a software system is understanding the original programmer's intent (see [11], [12]):

- The comprehension process takes more than 50% of the time spent on the maintenance task.

The most obvious way to support program comprehension is to produce and maintain adequate documentation. A tool that supports program comprehension on source code level is of great help (e.g., [2], [3], [15]), especially if documentation is not available or is incomplete and/or inconsistent, or if a tool for its automatic production (e.g., [5], [10]) is not available.

Most software systems are beyond a single person’s comprehension capacity. A tool that supports program understanding can prevent (maintenance) programmers from getting lost in a large system and it can help him/her to attain desired and needed information about it in an easy manner.

Our research group has been working with object-oriented languages and systems for three years. We have had good experiences, but also found that object-oriented systems tend to be more complex than conventional ones. What we needed were tools that help us master the increasing complexity both in the development and in the maintenance process.

Because the search for suitable tools had not been fruitful, we decided to realize a tool for this purpose. We did not consider the implementation to be too difficult when utilizing the concepts of object-oriented programming and, in particular, when using a powerful application framework [18]. As in our research group we mostly deal with software development tools, this would greatly enrich the work we had done so far (e.g., [1], [14]).

The Structure of Software Systems written in C++

Before we describe the tool, some words should be said about the architectural structure of software systems written in the object-oriented programming language C++ [17].

A C++ program system consists of a set of files that contain class definitions, method implementations and global declarations. The global declarations can be used in more than one class and their corresponding methods. There is no restriction on what has to be written in a single file. A file can contain more than one class definition and a class definition together with its method implementations can be spread over several files. We use the extension '.h' for files containing class definitions and the extension '.C' for files containing the implementation of a class, i.e., the implementation of their methods; they are called the *h-files* and *C-files*, respectively.

In order to use a specific class, its h-file has to be *included*. (This is done with a special preprocessor statement.) So there exists a relation among the files of a program system written in C++, the include relation. There exists another relation among the classes of an object-oriented software system, the inheritance relation. A class inherits the properties, i.e., the instance variables (these are variables local to a class) and the methods of its superclass. This means that one has to inspect different files in order to find out the whole story about a class.

The Maintenance Tool

Our maintenance tool eases the process of navigating through the files and classes and helps the user to get any needed 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, e.g., preprocessor statements like the include statement mentioned above or global type declarations.)

These little chunks of information are managed together with their relations among them. 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 overwritten 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, method, or 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. Addi-

tionally, useful information is displayed to protect the user from getting lost in the complex information web.

User Interface

The user interface concept is based on modern application frameworks and the supported concepts thereof (see [16], [19]). It provides two selection lists, an editor window, an icon bar containing several browsing tools, a menu bar, and two information bars (see Fig. 1). The first list displays either the classes or the files of a software system (¿). The second one displays the methods, the subclasses, or the superclass(es) of a class, or those files included by a file selected in the first list (¡). The editor window (¬) displays the code part depending on the selections made in the two lists on the left side. Seven icons (√) are used to browse through the system (see below). The menu bar (ƒ) is used for the activation of more specific commands, and two information bars display the filename of the code currently shown (≈) and the inheritance path (i.e., all the superclasses) of the class or the method currently shown (Δ).

Browsing Facilities

The tool offers several possibilities to browse from one piece of code to another. This can be done by selecting an item from the two displayed lists. In this case the appropriate class definition, method implementation or file description is displayed.

Another and even more useful way to reach other parts of the system is to follow one of the relations associated with the displayed code. This enables the user to browse with a single

mouse click

- from a class to its superclass or any of its subclasses
- from a method implementation to the same method implementation in its superclass
- from a class or method description to the file in which the description is contained
- from a file to any of its included files

It is also possible to select an identifier of the displayed code and to browse

- to the identifier declaration
- to the next or previous use of this identifier

This again can be accomplished by a single mouse click.

The system remembers the browsing paths of the user and thus enables him/her to undo any browsing activities and get back to where he/she came from.

Global Text Styles

In order to enhance the readability of the source code, the user can define global styles for different syntactic constructs, e.g., comments, keywords (see Fig. 2). 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?*

We support three different highlights to allow the user to distinguish even among global identifiers of a class and local identifiers.

Identifier Remarks

If a short description exists for an identifier (usually a short comment after its declaration) then this description together with some other useful information (e.g., point of declaration) can be

shown at any place this identifier is used (see next section). We assume that a comment after the definition of an identifier contains a description of this identifier. Assuming this style for existing programs might — at worst — lead to the display of an identifier and a comment that does not contain a description of this identifier. But on the other hand, the system can provide the user with information that is very useful for program comprehension and can do so with little effort on the part of the user (i.e., by writing a short comment for all or many identifier definitions).

Sample Scenario

This sample scenario illustrates how we might try to understand the function of a method.

For this purpose we first select the method's class (see again ¿ in Fig. 1) and the method itself (¡) in the class and method list on the left side, respectively. We now see the source code of the method (¬), the name of the corresponding file it is contained in (√), and the inheritance path (ƒ) of its class. If we know some of the superclasses already, the inheritance path might give us a feeling of the properties of a class. For example, we might know that subclasses of class *EventHandler* usually handle certain events.

We can now highlight all local identifiers, i.e., identifiers that are defined in this method, including parameters. Furthermore, we might want to know the instance variables of its class that this method uses. For this purpose we browse to the class, highlight its local identifiers and browse back to the method. Identifiers are highlighted wherever they are used, so we can easily locate all global identifiers in our method, too. In order to distinguish local and global identifiers, we can use different text styles (see Fig. 2), e.g., outline for global identifiers of the class, bold for local ones of the method, and additional underlining for identifier definitions (see Fig. 3).

Any identifiers that are not highlighted in our method so far are declared in one of the superclasses or even somewhere else in the software system — except keywords. We can select any identifier and browse to its definition with a single mouse click. By using a third text style, e.g., another text font or even another style or size, we can highlight the identifiers of another class or even of a certain file to see the occurrences of it in our method at a glance.

Concise information about an identifier can be of great help when trying to understand a piece of code. On demand our tool tells the user the name, location, corresponding file, inheritance path and the declaration of a certain identifier. A short description is also displayed if available, i.e., if a comment is written right after the definition of this identifier in the source code (see Fig. 4). This, again, is accomplished with a single mouse click.

Experiences and Planned Enhancements

We have been using object-oriented programming and application frameworks for about three years. The presented tool is of great help in mastering the complexity of both our own devel-

opments, including the presented tool itself, and software written by others, like the application framework ET++ ([18], [19]). As there is no documentation available for ET++ so far, we are forced to obtain any needed information out of the source code on our own. This is not an uncommon situation in practice but is tough work when done without tool support.

Another big benefit of our tool is its use by students of object-oriented programming. Its simple mechanism for browsing allows them to obtain useful information about the system und thus understand faster what is happening.

A tool cannot replace good documentation. At most it can produce more documentation automatically, but this will never be a real substitute for documentation written by hand (e.g., the description of concepts). To support the availability of documentation we plan to extend our system to a Literate Programming Environment [8]. This will be another important step toward a system for better program comprehension.

Activities of software maintenance and software development are very similar. Thus a real maintenance tool must also support

software development. Usually program comprehension alone plays a somewhat minor role in the development process. Therefore, it seems useful to extend our tool in a way that it can be used for software development, too. For example, one should be able to start compilation within the tool, and the presented hypertext features should also be integrated in the debugger.

Some enhancements are needed in providing information about a software system and in browsing through it. The use of categories for classes and methods is being considered. Besides, questions like *Which methods send a particular message?* or *Which classes implement a particular message?* have to be answered by the system (see [7]).

The support of other programming languages, especially a non-object-oriented one, would be of interest to prove the portability of the language-independent part of the system.

Current Implementation Restrictions

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

- Incremental static analysis is not supported.
- Some features for editing of programs are still missing (e.g., comfortable insertion of new classes and methods).
- All information about a software system is kept in main memory. The use of a database is being considered.
- Multiple inheritance is not supported.

Implementation

It is impossible to describe the implementation in detail in this paper. Only a rough description of the basic structure of the implementation is given.

The presented tool was implemented with C++ under UNIX on SUN workstations, using the application framework ET++ [18] [19]. The tool is clearly separated in two parts: a language independent hypertext browser (see [4]) and a language (C++) dependent static analyzer, in order to get needed information about the inspected program.

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., class descriptions, method implementations)
- any relations among these text pieces for browsing (e.g., inheritance, include relations, methods of a class)
- 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 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)

Static Analysis of C++ Programs

C++ is an object-oriented superset of the programming language C [17]. We will present some details about the static analysis of C++ programs because the structure and the history of the language burden the development of tools for it.

The compilation of C++ usually consists of three parts: the C preprocessing (*cpp*), the transformation to C (*cfront*), and finally the compilation of the C program (*cc*, see Fig. 5). This 3-part compilation makes C++ programs upward compatible to C.

The *C Preprocessor (cpp)* first reads the source code and processes the preprocessor statements (lines beginning with a '#'). It includes other files, handles the definition of identifiers and replaces these identifiers in the subsequent text with their defined strings (with parameters), and skips parts of the text according to if-then-else-statements, which requires the evaluation of constant expressions.

The *C++ Front End (cfront)* parses the output of the preprocessor and generates a C program. To do this, a full syntactic and semantic analysis is necessary.

Finally the *C Compiler (cc)* reads the output of the C++ front end and generates object code. In some implementations object code is generated directly, i.e., the transformation to C code is left out.

In order to get the information needed for our hypertext browser, the syntactic and semantic analysis of *cfront* has to be carried out. This analysis cannot be done with the output of the preprocessor because the preprocessor generates a new intermediate source file and it would be impossible to determine exactly the definition and use of identifiers of the original source file. Therefore, these two steps have to be integrated; i.e., the functions of *cpp* and *cfront* have to be carried out simultaneously.

Our language-dependent parser has to recognize and perform any preprocessor statements (include files, manage a symbol table of preprocessor-defined symbols, evaluate constant expressions in if-then-else statements and possibly skip lines). Whenever an identifier is read from the regular C++ code it has to be checked whether it is a preprocessor-defined identifier. In this case this identifier, and possibly an argument list have to be replaced with the appropriate string and passed on for further analysis. Otherwise the hypertext browser has to be informed about an identifier definition or use. The same holds for keywords and comments.

Conclusion

A tool was presented that supports the maintenance of C++ program systems by providing a modern user interface, comfortable browsing facilities, global text styles, and the possibility to obtain useful information about identifiers.

The tool is divided into a language-independent hypertext browser and a language-dependent static analyzer. Thus, with little effort it should be possible to support other programming languages, both object-oriented and procedural.

With the help of better maintenance tools we can decrease the time required for the comprehension process and thus reduce the costs of the software maintenance task.

Many things currently done by tools that automatically produce documentation can be done better by interactive tools. Interactive supply of information about software systems shortens the time needed for their understanding because one does not have to pick up desired information from the (possibly, even probably, very extensive) generated documentation.

References

[1] Bischofberger W., Pomberger G.: SCT: A Tool for Hybrid Execution of Hybrid Software Systems, First International Modula-2 Conference, Bled, Yugoslavia, Oct. 1989.

[2] Cleveland L.: An Environment for Understanding Programs, Proc. of the 21st Annual Hawaii Int. Conf. on System Sciences, Vol. 2, 1988.

[3] Cleveland L.: A User Interface for an Environment to Support Program Understanding, Proceedings of the Conference on Software Maintenance, pp. 86-91, 1988.

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

[5] Fletton N. T., Munro M.: Redocumenting Software Systems Using Hypertext Technology, Proceedings of the Conference on Software Maintenance, pp. 54-59, 1988.

[6] Gibson V. R., Senn J. A.: System Structure and Software Maintenance Performance, CACM, Vol. 32, No. 3, pp. 347-358, 1989.

[7] Goldberg A.: Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, 1984.

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

[9] Kuhn D. R.: A Source Code Analyzer for Maintenance, Proceedings of the Conference on Software Maintenance, pp. 176-180, 1987.

[10] Landis L. D., et al.: Documentation in a Software Maintenance Environment, Proceedings of the Conference on Software Maintenance, pp. 66-73, 1988.

[11] Letovsky S., Soloway E.: Delocalized Plans and Program Comprehension, IEEE Software, pp. 41-49, May 1986.

[12] Parikh G., Zvegintzov N.: Tutorial on Software Maintenance, IEEE Computer Society, pp. 61-62, 1983.

[13] Parikh G.: Techniques of Program and System Maintenance, Second Edition, QED Information Sciences, Inc. 1988.

[14] Pomberger G., et al.: TOPOS — A Toolset for Prototyping-oriented Software Development, Actes de la 4ème Conférence-Exposition de Génie Logiciel, AFCET, Paris, Oct. 1988.

[15] Rajlich V., et al.: VIFOR: A Tool for Software Maintenance, Software—Practice and Experience, Vol. 20, No. 1, pp. 67-77, January 1990.

[16] Shneiderman B., et al.: Display Strategies for Program Browsing: Concepts and Experiment, IEEE Software, pp. 7-15, May 1986.

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

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

[19] 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.