Design and Implementation Aspects of an Experimental C++ Programming Environment

J. SAMETINGER AND S. SCHIFFER
Institut für Wirtschaftsinformatik, C. Doppler Laboratory for Software Engineering
Johannes Kepler University of Linz, A-4040 Linz, Austria

SUMMARY

A good programming language alone is not sufficient for economic software production. The programming environment has a significant influence on the productivity of software engineers. Providing a programmer with information about an object-oriented software system requires extracting information from the source code, e.g., class, method, and variable names. We use separate structure files for holding this information and take advantage of proven tools like *make* and the C preprocessor for keeping the structure files up to date and for processing software systems that heavily use macros.

In this paper we describe the concepts used for comfortable processing of C++ software systems, and discuss interesting design and implementation aspects, including structure files, the applied *make* mechanism, and the exploitation of the C preprocessor.

KEYWORDS Programming Environment Design Implementation C++ UNIX *make* C preprocessor Macros

INTRODUCTION

Large software systems comprise many thousands of lines of source code. Sequential reading and writing is inadequate for mastering this complexity. In particular, the structure of object-oriented software systems makes it necessary to inspect many locations in order to get the picture of the class hierarchy, of overridden methods, and so on. Simple file browsers that were acceptable for conventional software systems are insufficient in object-oriented programming.

Presenting structural information about complex object-oriented software systems requires obtaining this information from different source files. Working on very large systems (consisting of hundreds of thousands of lines of source code) calls for a strategy to efficiently get this information in order to provide users with both comfortable and

fast browsing mechanisms. Additionally, obtaining structural information from C++ programs may be hindered by the macro capabilities of the language.

In the following sections we describe concepts and interesting design and implementation details of the programming environment DOgMA (see [Sam91]). Thereby, a possible way to provide a comfortable and efficient (in time and memory) environment for object-oriented software systems written in C++ is depicted. We presume the reader has a basic knowledge of the object-oriented programming paradigm, the C++ programming language, the C preprocessor, and the UNIX *make* mechanism.

GOALS

Typically, object-oriented software systems are not built from scratch, but rather existing components are reused. Furthermore, component descriptions, i.e., classes, are relatively small units, but have many interrelations with other classes. This complicates the comprehension process, makes linear reading inadequate, and raises the need for tools that present complex structures to the user and allow comfortable navigating through a web of components.

Program comprehension is a major problem with software reuse. It is the most time-consuming task. One of the most important ways to improve program comprehension is system documentation. Thus the improvement of both program comprehension and documentation promises a major reduction in software costs. An early version of DOgMA, as described in [Sam91], has been developed primarily to support the comprehension process of object-oriented software systems. We defined the following goals for the achievement of such an improvement :

- *Concentration on the comprehension process*

  Rather than trying to support all activities of the software development process, the main goal was to attain a considerable improvement in the familiarization process with justifiable effort.

- *Synthesis of new concepts*

  The synthesis of useful new concepts like literate programming and hypertext promised to emerge as a much better aid than the combined use of tools for each concept (e.g., the separate use of hypertext tools and literate programming tools).

- *Compatibility*

  An isolated tool does not help very much unless it supports all activities of the software life cycle process. Therefore an easy integration in an existing environment was essential.

- *Documentation access and completeness*

  The user has to be told whether documentation exists for a given part of the source code and where it is. The amount of documentation will vary from project to project.

However, a maintenance tool must be able to tell the user which parts of the systems are documented and where, and which are not.

- *Documentation consistency*

  Documentation is seldom up-to-date and therefore gives incorrect or misleading information. This is certainly worse than no documentation at all. A tool cannot really check consistency of a documentation text, but it should help the programmer to check it.

- *Information access*

  As important as the access to documentation is the (fast) access to the source code itself. Understanding a piece of code hinges on the answers to many questions, e.g.: Where is this variable defined/set/used? An effective maintenance tool must help the user to answer questions like these in an easy and fast manner.

- *Preventing side effects*

  Side effects of a maintenance change are very often triggered by just making a little improvement. The reason for this is incomplete understanding of the system to be maintained and can again be prevented by easy and fast access to relevant information (see above).

- *Dealing with high complexity*

  Being better able to deal with complex software systems is another important goal because new paradigms like object-oriented programming tend to result in even more complex system structures.

Despite the many advantages of DOgMA (see [Sam90], [Sam92]), experience revealed some major drawbacks. The reasons for these drawbacks were that a modified compiler had been used in order to get information about the source code, and all the information had been held in main memory. This led to the following problems:

- Dealing with large projects became very inefficient and storage consuming.
- Switching to a new compiler (or even to a new version of a compiler) made it necessary to adapt the new compiler. This, in addition, required the availability of the compiler's source code.
- Only syntactically correct programs could be processed with DOgMA, because the compiler, naturally, did not turn a blind eye on syntax and/or semantic errors.

In order to overcome these drawbacks, we started a new project and defined the following goals for improvement:

- *Efficiency*

  It must be possible to deal with very large software systems without hogging very large amounts of main memory and with acceptable response time.

- *Fuzzy parsing*

  The user must be able to work on incomplete software systems that are not syntactically correct.

- *Compiler independence*

  The programming environment must not be dependent on a particular compiler (version).

- *Documentation*

  The documentation structure must be better harmonized with the structure of object-oriented software systems.

In this paper we describe the essential concepts of the new version of DOgMA and design and implementation aspects of these improvements.

CONCEPTS

Presenting the Logical Model

Typically, an object-oriented program system written in C++ consists of a set of files that contain class definitions, method implementations and global declarations. Global declarations can be used in more than one class and in the 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. There exists a 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. Since usually the definition of classes is kept in different files, it is cumbersome to obtain complete information about a class.

Using simple file browsers to process object-oriented software systems is absolutely inadequate because the physical (file) structure does not necessarily correspond to the logical (class) structure and because simultaneous access to various source-code and documentation locations is required in order to inspect classes and/or methods.

What we need is a programming environment that effectively and efficiently presents the logical model of very large software systems and gives fast and comfortable access to information needed for the comprehension of logical units (e.g., classes and methods).

Graphical User Interface

Our user interface was designed to let the user operate on the logical model of a software system rather than on the physical file structure. It is based on modern application frameworks and their supported concepts (see [Shn86], [Wei89]). It provides a menu bar, an information box, an editor window, and selection lists (see Fig. 1).

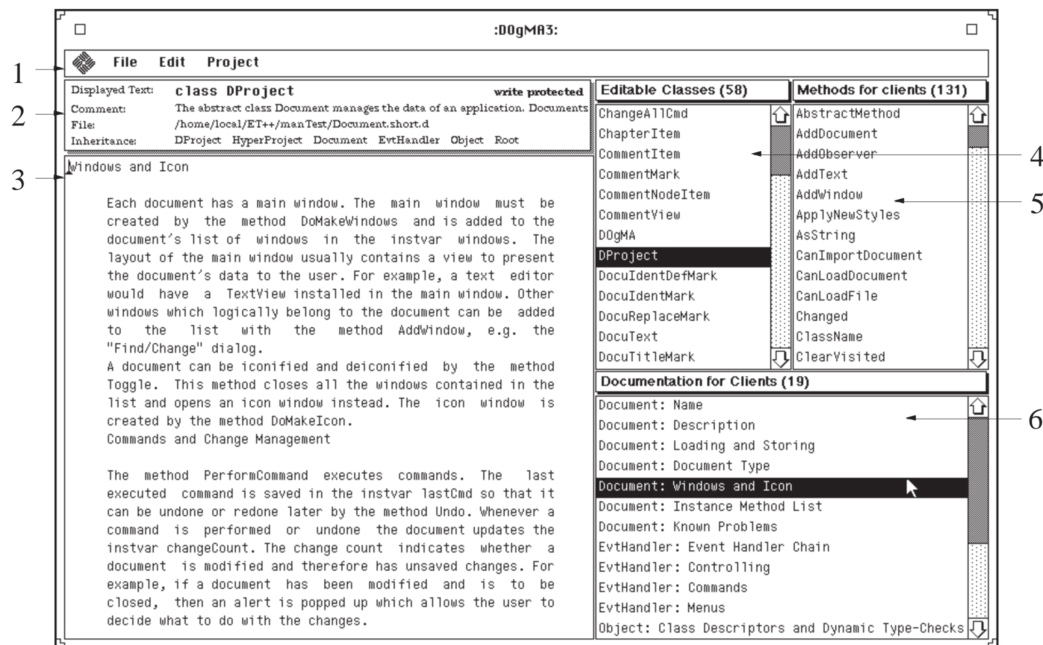

Fig. 1: User Interface of DOgMA

The *menu bar* (1 in Fig. 1) is grouped into commands for files, editing, and projects.

The *information box* (2) displays useful information about the text part currently under inspection, i.e., the name of the text currently shown (file, class, or method), its filename, and the inheritance path (i.e., all the superclasses). When multiple inheritance is used, all the paths are shown in the box.

The *editor window* (3) displays the text part depending on the selections made in the *selection lists* on the right side (see below). It offers the usual editing capabilities (like cut, copy, and paste) and text processing facilities.

The first *selection list* (4) shows all the nodes of the information web of the software system (all classes, all files) or a subset thereof (depending on various pre-defined categories). In the second and third lists (5 and 6 in Fig. 1), nodes are shown that have a certain relation to the one selected in the first list (e.g., methods, documentation sections). The text bars above the various lists indicate the category of nodes that can be seen in the lists, and can be used as pop-up menus to select other categories.

Browsing Facilities

When a programmer is inspecting a software system, information in related code pieces is of utmost importance. In object-oriented software systems these related code pieces are the superclasses and the overridden methods because they contain most of the information that is needed to understand a subclass or a(n overriding) method. What we need is easy access to this related information, e.g., to all direct and indirect superclasses and subclasses of a class, to all overriding and overridden versions of a method, and to the files that contain these classes and methods.

Displayed Text:	**HyperFindDialog::Control**
File:	HyperFindDialog.c
Inheritance:	**HyperFindDialog** **Dialog** **DialogView** View VObject **EvtHandler** Object Root

Fig. 2: Information Box

We provide most of this information in the information box, which displays the name of the class or method currently being inspected, the name of the file containing the class or method, and information about the inheritance path (see Fig. 2). In this example we see that the method Control of the class HyperFindDialog is currently displayed (the method's name being shown in C++ notation). This method is contained in the file HyperFindDialog.c. The superclasses of HyperFindDialog are Dialog, DialogView, View, etc.

The class's inheritance path in the information box enables the user to instantly inspect any of the superclasses of the class and to get back again by simply selecting an item in it. Additionally, if the browser displays a method, then all the classes in the current inheritance path that implement the same method are highlighted. Thus the user can see at a glance which classes implement a certain method for a given inheritance path. In Fig. 2 the class names HyperFindDialog, Dialog, DialogView and EvtHandler are highlighted (shown in boldface) because they implement the method Control. The classes View, VObject and Root do not implement a method Control. Selecting a highlighted superclass causes the respective method of the superclass to be displayed in the editor window.

In programming languages like C++, not only the classes and methods under consideration are of interest, but also the files where they are defined, because files can contain global declarations that are used in classes and methods. Therefore, the file name is necessary in the information box and—as with superclasses—provides easy access to these global declarations by simply clicking on the file name. Using this possibility together with the features of the inheritance path allows easy inspection of all the files related to the superclasses and their method implementations, even if they are spread over several files and directories.

Notion of Program

One of the shortcomings of current browsing tools is that they do not support the *notion of program* (see [Wu90]). Browsers usually help to view the methods of a selected class. But what programmers need is to view the methods intended for the reuse of a class, i.e., part of the methods of the selected class and part of the methods of its superclasses (inherited methods). C++ distinguishes among private, protected and public methods. The access of clients of a class is restricted to public variables and methods, whereas builders of subclasses (i.e., heirs) may also use any protected data. Anything private to a class must not be used outside (except friends, see [Str91]). Besides viewing private, protected and/or public methods of a class, DOgMA offers the possibility to view *methods for clients* (i.e., the public methods of a class and all its public superclas-

ses), *methods for heirs* (i.e., the public and protected methods of a class and all of its superclasses), and the *methods for the implementor and friends* (i.e., the public, protected and private methods of a class and the public and protected methods of all of its superclasses). Figure 3 shows the menu that allows the selection of any of these method categories. When methods are overridden in a subclass, only the downmost methods are displayed. (With the information box mentioned above, the overridden methods can easily be accessed, too).

The current version of DOgMA displays only the methods of a class for browsing. For future versions the inclusion of variables, constants, types, and macros is planned as well.

Fig. 3: Menu for method selections

Find Mechanism

Object-oriented programming supports the reuse of existing source code. This enhances the need for comfortable searching in order to speed up the process of understanding. It is necessary to find both the classes and methods to be reused as well as examples where the use of these classes and methods can be inspected.

We use a very comfortable and powerful—yet simple—find facility (see Fig. 4), where the user can specify where to search (in the currently displayed class or method, in the whole corresponding file, in the inheritance path, or in the whole project) and what to search for (any strings, class names or method names). Any items found are displayed in a list (located in the lower half of the find dialog in Fig. 4). With this list a simple mouse click suffices to jump to any of these items. The user can also specify whether the list should display the text titles (i.e., the names of the classes and methods containing the search string) or the line contexts of the found locations.

The main advantage of this simple tool is the rapid access to all places where a textual piece of information occurs even if it is spread over dozens of files or directories.

Integrated Documentation

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. Integrating the source code and the documentation is the main idea of literate programming [Knu84]. We chose a simple mechanism for this integration by relating documentation sections to

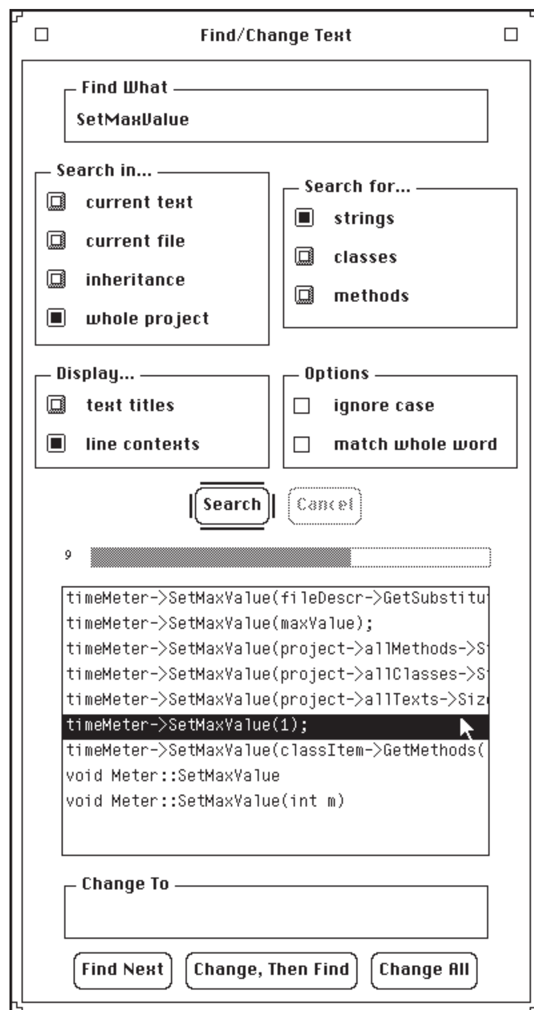

Fig. 4: Find/Change

classes or methods and displaying them whenever a class or method is selected. We also apply object-oriented techniques to the documentation (for more details see [Sam94]). Similar to the source code, a class inherits the documentation of its superclasses. Documentation sections are defined by the user (programmer) and used for inheritance in the same way as methods. Similar to the methods' inheritance mechanism, sections are either left unchanged, removed, replaced, or extended in subclasses (see Fig. 5). Examples of such sections are: short description, conditions for use, instance variables, and instance methods. Depending on the classes, other sections have to be added, for example: event handling or graphical objects.

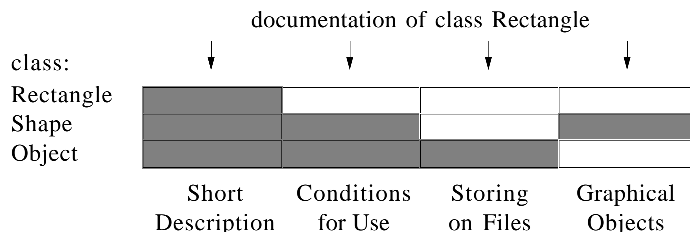

Fig. 5: Inherited and overridden documentation of class Rectangle

Figure 5 shows the documentation of class Rectangle, which is composed of documentation of classes Object, Shape, and Rectangle itself. The relevant sections can be identified by looking from the top vertically into the documentation “building”: "Conditions for Use" and "Graphical Objects" are inherited from class Shape, "Storing on Files" is inherited from class Object, and the "Short Description" is defined in class Rectangle itself. Thus, the documentation of class Rectangle consists of four sections, although only one has been written for this class.

In Fig. 1 we see various documentation sections that have been inherited from the classes Document, EvtHandler, and Object. In this case, too, no extra documentation has been written for the class DProject under consideration.

Design Aspects

The most obvious way to realize the programming environment presented in the previous chapter is to scan a whole project each time the tool starts up in order to get information about the logical structure of the software system and to load all the related pieces of text into main memory. This straightforward approach has some drawbacks, however.

First of all, large software systems consist of hundreds of thousands of lines of source code and documentation which comprise dozens of megabytes of data. Keeping masses of textual information in memory is well supported by standard text classes of most class libraries and therefore simple to achieve, but inadequate for our purposes. Related tools like file browsers implemented with such memory-consuming text classes are able to display files only if they entirely fit into memory. This restriction prevents users from browsing really large amounts of data on systems with moderate-sized main memory shared by many processes.

Another issue is performance. Quick response to selected actions highly influences the acceptance of any software tool by users. Early prototypes of DOgMA were substantially too slow. It took a few minutes to start up the tool and to present the overall structure of a project. Selecting an item in the selection lists did not immediately cause the appropriate text to be displayed.

In order to avoid these disadvantages, we tried another approach. First, the project is scanned and the information about its structure is stored on disk. When the programming environment is started, it suffices to read in the structure information. This can be done quickly even with very large projects. The source itself is not kept in main memory. The parts the user wants to see or work on are read in whenever needed.

This results in two different tools, depicted in Fig. 6. The tool dogmamake analyzes the source code and stores the structure information on separate files. DOgMA, the actual programming environment, reads the structure information of an entire project, presents it to the user, and reads source code whenever necessary.

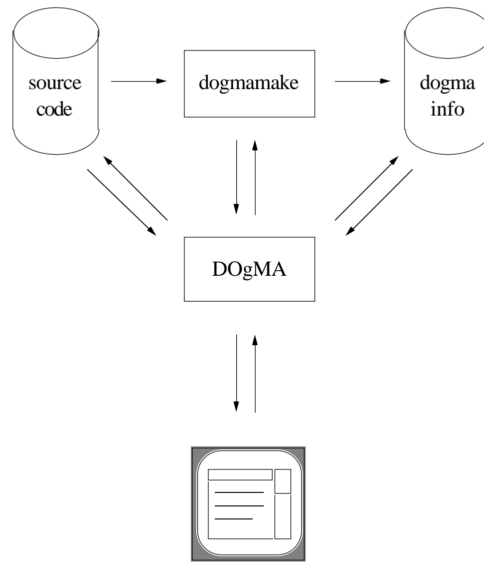

Fig. 6: General Design

We use the structure information for the documentation also. Chapters are stored in files and are treated like classes and methods. While the user is editing with DOgMA, the structure information is updated. However, it is not stored on disk because the stored structure information is updated anyway when DOgMA is restarted. This guarantees consistent structure information with little effort. We must mention that the structure information can get inconsistent when files are edited with other tools simultaneously. When this happens, the displayed text does not correspond with the class to be shown. In this case the user has to quit DOgMA, run dogmamake, and restart DOgMA in order to obtain consistent information.

Structure Information

Of the information about the logical structure of a software system in C++, certainly the inheritance hierarchy is the most important. Additionally, file names, line numbers, and scope information are used. The following list shows the information we use:

- *file*: name of file, classes and methods contained in the file, number of lines
- *class definition*: name of the class, names of the superclasses, line number where class definition starts in its containing file, names of methods defined in class definition, line number where class definition ends
- *method definition*: name of method, line number of definition, scope (private, protected, public), in-line or separate implementation
- *method implementation*: name of method, name of corresponding class, line numbers where method starts/ends in its containing file
- *documentation*: name of the section, name of the class or method the section belongs to, name of the file containing the section, line numbers where the section starts/ends in this file.

```
file: StyleDialog.h {
  class: StyleDialog:Dialog 16 {
     Control 29 private def
     DispatchEvents 30 private def
     StyleDialog 34 public def
     ~StyleDialog 35 public def
     GetStyle 37 public inline
     Show 40 public def
     DoCreateDialog 41 public def
  } 42
} 45

file: StyleDialog.c {
  method: StyleDialog::StyleDialog 35 160
  method: StyleDialog::~StyleDialog 162 164
  method: StyleDialog::DoCreateDialog 166 368
  method: StyleDialog::Show 370 373
  method: StyleDialog::Control 375 456
  method: StyleDialog::DispatchEvents 458 461
} 462

file: StyleDialog.d {
  docu: StyleDialog 1 {
     section: Description 87
     section: Setup 143
     section: Validation 189
     section: Cancellation 243
} 244
```

Fig. 7: Structure Information

The structure information shown in Fig. 7 describes the file StyleDialog.h (45 lines) containing the definition of the class StyleDialog, the file StyleDialog.c (462 lines) containing several method implementations of the class StyleDialog, as well as the file StyleDialog.d (244 lines) containing documentation describing the class StyleDialog. The class definition starts at line 16 and ends at line 42. StyleDialog is a subclass of Dialog and has two private and six public methods, one of which is written in-line. For example, the public method DoCreateDialog is defined at line 41 in the file StyleDialog.h; its implementation starts at line 166 and ends at line 368 in the file StyleDialog.c. The documentation in file StyleDialog.d contains 244 lines and consists of four sections entitled Description, Setup, Validation, and Cancellation.

The structure information can be stored in a single file for an entire software system, or an extra structure file can be created for each source-code file. A single structure file might necessitate rescanning of the entire software system every time a change has been made in the system. Multiple files require an extra structure file for each source-code file. We decided to use both single and multiple structure files. We pack the structure information of parts of the software system that are not likely to be changed into a single file, e.g., reused class libraries or application frameworks. All other parts of a project, i.e., files that are currently being worked on, have an extra structure file. This strategy minimizes both the number of structure files and the number of source-code files having to be scanned after any changes. The user can choose between these alternatives by specifying an option for the dogmamake command.

Generating structure information does not require syntactical correctness of a software system. It must be guaranteed, however, that braces and parentheses match.

Internal Administration of the Structure Information

In order to provide users with comfortable browsing capabilities, the structure information is read in at startup time and represented internally by a set of interrelated objects. Various objects exist, describing files, classes, methods and documentation sections (see Fig. 8). These objects are entirely held in memory and are necessary for providing users with any needed information about a software system. The following interrelations are held in memory:

- *class*: list of methods, list of superclasses, list of subclasses, corresponding file, list of documentation sections, corresponding source-code text
- *method*: corresponding class, corresponding file, list of documentation sections, corresponding source-code text
- *file*: list of classes and methods contained in this file, corresponding source-code text
- *documentation sections*: corresponding class or method, corresponding file (containing the section), corresponding documentation text

This information web provides users with a lot of useful information. Besides simple capabilities like listing the methods of a class or the documentation sections of a method, the tool can easily determine inherited methods, methods that are intended for clients, heirs or friends, overridden methods, overridden documentation sections, the inheritance path, etc.

As mentioned above, another big benefit is the fact that the corresponding source-code (or documentation) text does not have to be kept in main memory. It is read in from the files only when the user chooses to display a certain text by selecting, for example, a class or method. This text can be thrown away as soon as the user inspects another text. Multiple text pieces have to be kept in main memory when several browsing windows of the tool are active. When any changes are made in a text, the changed text is also kept in main memory as long as the changes are not saved to disk or canceled by the user.

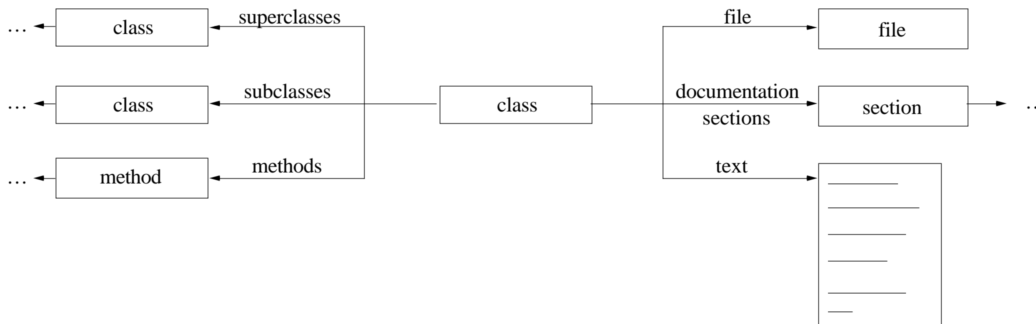

Fig. 8: Internal Information Web

Updating the Structure Information

The programming environment manages the structure information, presents it to the user, and reads in parts of the source code (e.g., classes, methods) whenever needed. In order to keep the structure information correct, it has to be updated whenever any source-code file has been modified. Updating the structure information is also required whenever a directly or indirectly included file has been modified because macros might have been changed that are used in the source file.

This source file dependency is very similar to the dependencies among source files and object files. A source file that is newer than the corresponding object file requires recompilation to create an up-to-date object file. A source file that is newer than the corresponding structure-information file requires reparsing to ensure valid links from structure information to source code.

Due to the similarity of these dependencies, we choose the *make* mechanism [Fel79] to keep structure information up to date. DOgMA is wrapped by a shell script *startDogma* that calls the UNIX command *make* to check for source-code changes which in turn invokes the parser whenever necessary. After this is done, *startDogma* starts DOgMA. However, when the user alters the source code using the editing facilities of DOgMA, then changes to the structure information are made directly and saved to disk as soon as the user confirms the modifications. (Further explanations of the *make* mechanism are given below).

Implementation Aspects

DOgMA was implemented with C++ [Str91] under UNIX on a SUN workstation using the application framework ET++ [Wei88, Wei89]. The implementation is separated into a language-independent browser, a language-dependent parser for C++ (used to get needed information about the inspected program text), and some shell scripts that guarantee that the structure information is kept up to date.

Language-Independent Browser

The logical structure of object-oriented software systems is very similar regardless of the programming language used. Only the notation is different in Eiffel, Smalltalk, and C++, for example. However, programs written in these languages consist of classes and methods and use the inheritance mechanism. Therefore the concepts presented so far are applicable and advantageous to all of these languages.

This means that the presented programming environment should be usable with most other object-oriented programming languages. In fact, DOgMA was designed with these considerations in mind to support various languages with minor changes.

The browser component controls the user interface, reads the structure information of a software system (see Fig. 7), and manages the following information (see Fig. 8):

- class definitions, method implementations, documentation sections, files
- any relations among these pieces for browsing (e.g., inheritance)
- additional information (e.g., file location, corresponding class)

Language-Dependent Parser

Obviously each programming environment has to have a part specialized for the supported language. We tried to bundle all language-specific code into a separate component, - the language-dependent parser, which may be replaced completely and easily by another language-dependent version.

The parser scans the source code, recognizes classes and methods, and stores relevant information in a file. Syntactic correctness of the source code is not necessary; however, it is assumed that class and method declarations are correct and that braces match. This simple and light-weight parsing is also referred to as *fuzzy parsing* (see [Bis92]). An example parser output can be seen in Fig. 7.

A well-known problem for developers of programming environments for C or C++ is the use of macros, which make it impossible for simple parsers to easily extract structure information from source-code files. This is the case when macros are used instead of syntactic constructs of the language, e.g.:

```
#define begin {
#define end }
…
if (…) begin
    …
end
```

Macros can also be used to combine variants in a single source-code file, e.g.:

```
#ifdef sun
…
#else
…
#end
```

Depending on whether `sun` is defined, either the then or the else part has to be considered by the parser. This simple construction can lead to serious parsing problems when it is combined with parts of a syntactic construct, e.g.:

```
#ifdef sun
void xyz::abc(int a) {
#else
void xyz::abc(int a; int b) {
#end
```

Without considering the preprocessor statements, any parser would recognize a syntax error because of nested method definitions (as would Sniff, see [Bis92]).

For the above reasons, we used the C preprocessor output rather than the original source-code file as input for the parser. That allows a maximum of flexibility and makes

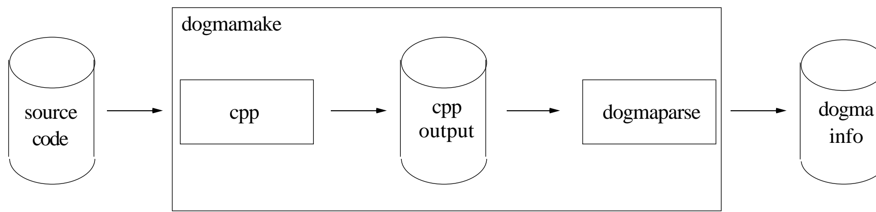

Fig. 9: Preparsing with integrated C preprocessing

us independent of future expansions of the macro language. The C preprocessor *cpp* and the parser *dogmaparse* are connected together within the shell script *dogmamake* (see Fig. 9).

This concept lets the user specify any flags as well as control the output of the preprocessor, thus obtaining the special view to the software system wanted.

Dependency Mechanism

Each time a source file is modified, it is necessary to update the corresponding structure-information file. In this section we describe in more detail how this dependency is managed using shell scripts and the UNIX *make* facility (see [Fel79]).

Let us assume that an application myTool consists of three files, namely main.C, myClass.C and myClass.h. The main program additionally uses a library whose interface is declared in the directory `/local/include/C++/common.h`.

Using *make* to automatically perform recompilation requires the definition of the dependencies among all files involved. This is done by writing a make-file as shown in Fig. 10.

Lines 1 and 2 define two macros. INCDIR specifies a nonstandard include directory and CCFLAGS instructs the C++ compiler (strictly speaking, the C preprocessor) to look into the directory specified in INCDIR whenever encountering the include statements. Line 3 contains the dependency for the entire application, which states that myTool has to be rebuilt if main.o and/or myClass.o do not exist or are newer than myTool. The command to be used for making myTool is given in line 4. It is the invocation of the C compiler (actually the C linker), which links the object files into an executable application. Lines 5 and 6 define the dependencies of the object files; e.g., main.o has to be rebuilt if it is older than main.C or $(INCDIR)/common.h ($(INCDIR) just expands to the string defined in line 1). For the benefit of short rules, it is not

```
1    INCDIR=/local/include/C++
2    CCFLAGS=-I$(INCDIR)

3    myTool: main.o myClass.o
4      cc -o myTool main.o myClass.o

5    main.o: main.C $(INCDIR)/common.h
6    myClass.o: myClass.h
```

Fig. 10: Simple make-file

necessary to explicitly specify the dependency between a source file and its corresponding object file (as in line 5). Line 6 contains such a hidden dependency which omits the .C file from the right side; *make* knows that a .o file always depends on a .C file with the same base name and automatically establishes the proper relationship.

Information for the Preprocessor

DOgMA uses the C preprocessor *cpp* to correctly expand sources and header files. As with "ordinary" compiling, cpp needs to know something about the environment, e.g., where to find header files or what flags to use for conditional compilation. This information is commonly collected in the CCFLAGS macro of the make-file. Using that macro, a simple UNIX pipeline like

```
cpp $CCFLAGS $sourceOrHeader | dogmaparse >$structInfo
```

is sufficient to precompile a source or a header file by cpp and convert the resulting output into structure information by dogmaparse.

However, the question still is where to place a command containing such a pipeline in the make-file and how to call that command whenever something has changed.

Dependencies between Source Files and Structure-Information Files

A good way to get along with that is to copy the original make-file to a new one, leaving all contents untouched, and to append a few lines for our purposes. First, we install a directory .dogma, which is used to keep all the structure-information files. Then the existing make-file is copied into this directory and automatically adapted for preparsing purposes; i.e., the same dependencies are used, but structure information is generated into the .dogma directory instead of compiling object code. The generated make-file for the above example is shown in Fig. 11.

All the appended lines are created by the shell script *dogmamake.make*, which reads the original make-file on its standard input channel and writes the converted file to standard output.

Line 12 defines the main target for preparsing. Lines 13-15 contain a list of all files that contain structure information. The files are written to the directory .domga and named according to the related sources, i.e., to the header files. Line 17 defines that every structure-information file depends on a file with the same name in the current directory. Line 19 calls the shell script dogmapase.sh (containing the pipeline mentioned above) for every dependent source file $<$ that is out of date with respect to the target structure-information file $@. Finally, from line 21 to the end of the make-file all dependencies are listed.

```
1  INCDIR=/local/C++/include
2  CCFLAGS=-I$(INCDIR)
3  myTool: main.o myClass.o
4    cc -o myTool main.o myClass.o
5  main.o: main.c $(INCDIR)/common.h
6  myClass.o: myClass.h
7
8  # -----------------------------------------
9  # Original makefile ends here.
10 # Next lines were appended by dogmamake.make.
11
12 dogma: \
13   .dogma/main.c \
14   .dogma/myClass.c \
15   .dogma/myClass.h
16
17 .dogma/%: %
18
19 @echo "parsing $<"; dogmaparse.sh $< $@
20
21 .dogma/main.c:/local/C++/include/common.h
22 .dogma/main.c:main.c
23 .dogma/main.h:/local/C++/include/common.h
24 .dogma/main.h:main.c
25 .dogma/myClass.c:myClass.c
26 .dogma/myClass.c:myClass.h
27 .dogma/myClass.h:myClass.c
28 .dogma/myClass.h:myClass.h
```

Fig. 11: Extended make-file

CONCLUSION AND PROSPECTS

Design and implementation aspects of the programming environment DOgMA have been presented. The use of external structure information served as a means of providing detailed information about a software system as well as high performance. Using the C preprocessor for obtaining structure information makes DOgMA applicable for any real-world C++ program (without imposing any restrictions on the use of macros). Furthermore, utilizing the *make* mechanism together with UNIX's powerful concepts of filters, pipelines, and shell scripts made it possible to reduce the capacity needed for keeping the structure information up to date.

DOgMA has been used in various projects at our institute and in industry. Its usefulness has been proven by a drastic reduction of necessary effort to develop and maintain software systems, and especially to get familiarized with large class libraries and application frameworks. DOgMA does not cover a wide range of supported activities like other tools do. But its representation of the logical model of software systems outperforms any other tools the authors know. Additionally, the concept of object-oriented documentation promises to bring the advantages of object-oriented programming to the documentation as well.

For the development of DOgMA we have integrated UNIX tools like *make*. Tools like compilers and debuggers can be used together with DOgMA. However, their integration is not as fully developed as is, for example, in TakeFive's SNiFF+ or HP's Softbench.

DOgMA offers less support for the various software life-cycle activities than SNiFF or Softbench do. However, its advantages are better support for documentation and better presentation of the logical model of a software system.

Besides various enhancements, we are currently working on the integration of a deductive database in order to provide users with a simple, yet flexible query mechanism for obtaining metrics or checking constraints. Additionally, visualization of dynamic aspects, like animating objects and their interactions, is being investigated.

Because of its experimental nature, DOgMA is not publicly available. Please contact the authors, if you are interested in more details.

REFERENCES

[Bis92] Bischofberger W.: Sniff—A Pragmatic Approach to a C++ Programming Environment, Proceedings of the USENIX C++ Conference, Portland, Oregon, August 1992.

[Fel79] Feldman S.I.: Make—A Program for Maintaining Computer Programs, Software—Practice and Experience, Vol. 9, No.4, pp. 255-266, April 1979.

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

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

[Sam91] Sametinger J.: DOgMA: A Tool for the Documentation and Maintenance of Software Systems, Technical Report, University of Linz, Department of Software Engineering, June 1991, available via anonymous ftp from ftp.swe.uni-linz.ac.at.

[Sam92] Sametinger, J., Pomberger G.: A Hypertext System for Literate C++ Programming. Journal of Object-Oriented Programming, Vol. 4, No. 8, pp. 24-29, January 1992.

[Sam94] Sametinger J.: Object-oriented Documentation, ACM Journal of Computer Documentation, Vol. 18, No. 1, pp. 3-14, January 1994.

[Shn86] Shneiderman B., et al.: Display Strategies for Program Browsing: Concepts and Experiment, IEEE Software, Vol. 2, No. 5, pp. 7-15, May 1986.

[Str91] Stroustrup B.: The C++ Programming Language (Second Edition), Addison-Wesley, 1991.

[Wei88] Weinand A., Gamma E., Marty R.: ET++—An Object Oriented Application Framework in C++, OOPSLA ‘88, ACM 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, 1989.

[Wu90] Wu C.T.: A Better Browser for Object-Oriented Programming, Journal of Object-Oriented Programming, Vol. 3, No.9, November/December 1990.