# Sniff—A Pragmatic Approach to a C++ Programming Environment[1]

Walter R. Bischofberger
*UBILAB (UBS Information Technology Laboratory)*
*Bahnhofstrasse 45*
*CH-8021 Zurich/Switzerland*
*Phone: (0041) 01 236 31 83 (direct)*
*Fax: (0041) 01 236 46 71 (direct)*
*Email: Walter.Bischofberger@ubilab.ubs.ch*

## Abstract

Sniff is a C++ programming environment which runs on different UNIX workstations under OSF Motif, OpenWindows, and SunView. Sniff is an open environment providing browsing, cross-referencing, design visualization, documentation, and editing support. It delegates compilation and debugging to any C++ compiler and debugger of choice. Sniff has been in internal use at UBS (Union Bank of Switzerland) since August 1991. Since then several developers are applying Sniff in writing serious software systems as well as in evolving Sniff.

In developing Sniff we took a pragmatic approach. We chose simple and efficient techniques in implementing Sniff's components and concentrated on combining these components into a seamless application. The aim of this paper is to describe Sniff's components and how they cooperate, to show the decisions which had to be taken, the trade-offs which have resulted, and to discuss our experience in applying Sniff.

## 1. Introduction

The main goal in developing Sniff was to create an efficient and portable C++ programming environment which makes it possible to edit and browse large software systems textually and graphically with a high degree of comfort, without wasting huge amounts of RAM or slowing down in an annoying way.

To achieve these goals in a reasonable time we decided to take a pragmatic approach. We chose simple and efficient techniques in implementing Sniff's components and concentrated on combining these components into a seamless application.

The aim of this paper is to describe Sniff's components and how they cooperate, to show the decisions which had to be taken, the trade-offs which have resulted, and to discuss our experience in applying Sniff.

Section 2 gives a short overview of Sniff's basic building blocks. Section 3 consists of a discussion of Sniff's overall architecture with emphasis laid on the topic of openness. The

---

[1] This paper appeared in the proceedings of the USENIX C++ conference, Portland, Oregon, August 1992.

following three sections give in-depth information about Sniff's basic building blocks. Section 7 discusses the portability related aspects of Sniff. Section 8 compares certain aspects of Sniff with the solutions chosen in the development of other C++ development environments. The paper ends with a discussion of the experience gained in developing and applying Sniff.

We do not discuss the look and feel of Sniff's user interface. For this reason we included Figure 1, a screen dump which gives an impression of a subset of the tools Sniff provides. It was optimized for showing as many tools as possible and shows no typical working configuration

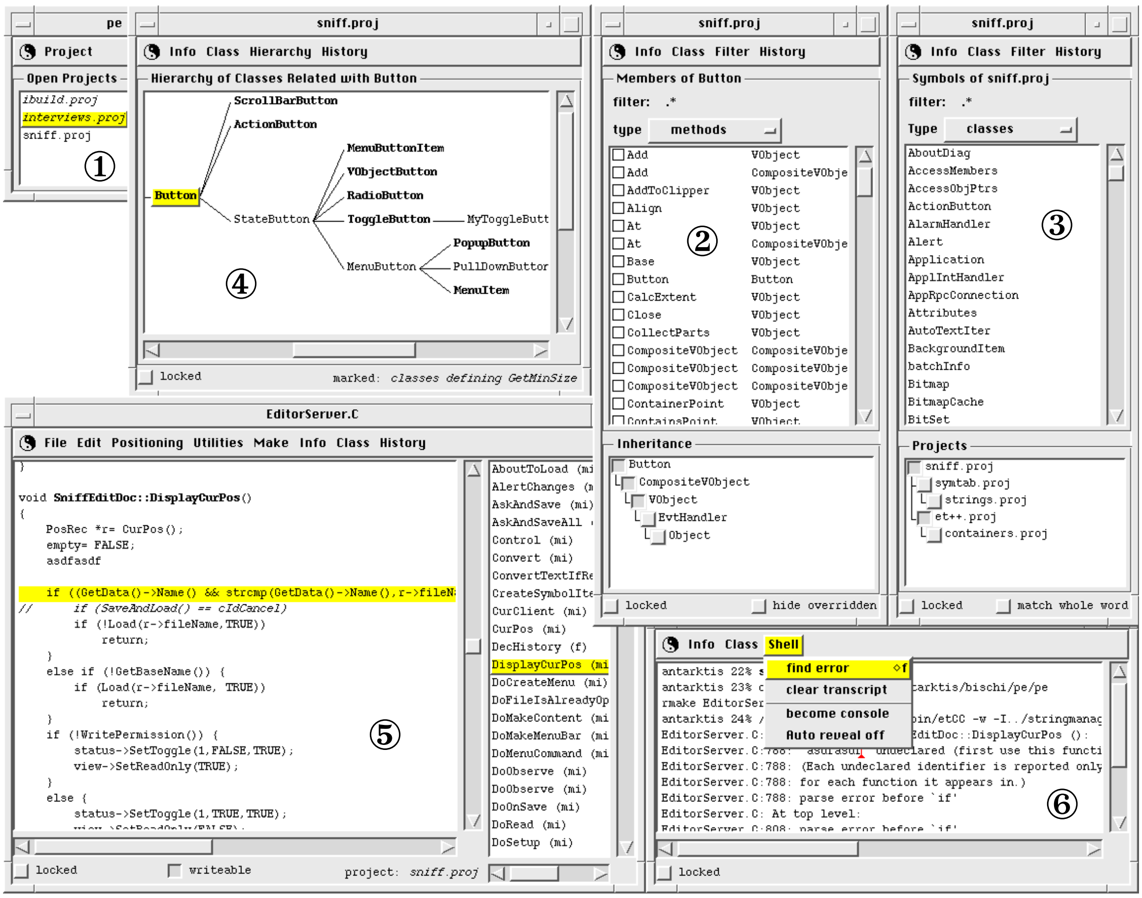


Figure 1. Overview of Sniff's user interface

## 2. The Building Blocks of Sniff

From a general point of view the tasks of a programming environment are to

- gather information about the software system under development
- update this information after the source code was modified
- give support for browsing and editing this information
- execute the software system
- provide means for inspecting the execution process (debugging)

To provide these services a complete programming environment has to consist of at least the following parts:

- an information extractor, which extracts information from the source code
- an information repository, which manages the extracted information, as well as certain project specific information
- an execution component, which compiles and executes, or interprets the software system and provides debugging support
- a user interface and several tools which make it possible to browse and manipulate the information in the data repository as well as to carry out other programming environment specific tasks (e.g. generation of makefiles and software metrics)

Sniff's information extractor is a fast fuzzy C++ parser which runs as a server process, parses entire source files on request, and sends a tokenized stream of extracted information to the client. It can be accessed by any program which needs information about the internals of a C++ source file.

Sniff's information repository is a symbol table which is kept in main memory and which is rebuilt every time a project is loaded. After a source file was changed all information about this file is discarded and rebuilt. The symbol table stores information about where and how the symbols (e.g., classes, members, variables) of a software system are declared and defined. Information about where the symbols are used is extracted on request.

Sniff does not provide an execution and debugging component on its own for reasons of portability, efficiency, and manpower. It delegates compilation to the compiler of choice and debugging is carried out with the corresponding debugger. We use Sniff, for example, on our SUN workstations together with AT&T, SUN, and GNU C++ compilers and the corresponding debuggers.

Sniff provides several tools for browsing and manipulating a software system based on the information stored in the symbol table and on information which is collected on the fly. These tools provide support for

- project management (1[2])
- browsing the symbols declared in a software system (3)
- getting information about usage of symbols (Figure 4)
- accessing their source code (5)
- for browsing the interface of a class (e.g., which members are declared in the class itself and which members are inherited from which base class) (2)
- visualizing the inheritance graph and information about protocols (4)
- compilation management (6)
- displaying documentation about symbols
- editing and browsing source code (5)

---

[2] The numbers behind the points reference the markings in Figure 1. They link the described functionality with the corresponding user interface elements on the screen dump (only a subset of the user interface element is displayed) .

## 3. Openness and Tool Cooperation

One of the main topics in actual discussions about programming environments is openness and the closely related topic of how the tools of a programming environment communicate and cooperate.

Openness means in this context that the tools of a programming environment are decoupled in a way which makes it possible to replace existing tools or insert new tools without having access to the source code of the entire system and without having to mess around in a large monolithic source code.

Many authors such as Gabriel et al. [Gab90] and Reiss [Rei90a], [Rei90b] are arguing for absolute openness. This absolute openness is achieved by running all tools in separate operating system processes and by making the tools communicate via well defined protocols.

In the field of open programming environments the coordination of the cooperation of various tools (control integration) is not the only challenge. The other challenge is the sharing of the large amount of information which has to be handled by a programming environment (data integration). This is especially true for modern programming environments because they provide sets of tools which visualize and manipulate the same information in different ways.

In developing a completely open programming environment we have to build tools which manage control and data integration. To support control integration we need a tool that manages cooperation and provides services such as message passing, starting of tools if they are not already running, and sensible handling of exceptions such as the breakdown of a certain tool. To achieve data integration we need a central data management unit which administers shared information and which makes this information available to arbitrary tools via abstract protocols. This central data management unit has to coordinate the updating of the information and to prevent consistency violating simultaneous updates. An example for such an open environment is Energize (formerly called Cadillac [Gab90]).

Openness and the resulting decentralization, achieved by communication over general common protocols, are general desirable goals in developing large software systems. Nonetheless, there are two reasons why their importance has to be relativated in developing the kernel of a programming environment.

First, the kernel of a programming environment consist of a set of closely cooperating tools sharing large amounts of information, and providing a homogeneous user interface. The advantage of a kernel with a completely open architecture is the possibility to replace entire tools if the protocols are known. This kind of replacement does not make much sense because there are no general purpose tools which could replace kernel tools. The text editor of a modern programming environment, for example, is usually very much customized providing not only editing but also browsing capabilities (examples are the editors of Energize [Gab90] and Sniff). Therefore, it would not make sense for a developer to integrate his preferred editor into such a programming environment.

Second, complete openness is expensive both in terms of development effort and runtime overhead. The runtime overhead is neglectable for many software systems because they do not need all of the available processing power. This is not the case for available C++ programming environments, as many C++ programmers painfully learned.

While we do not consider openness to be important in integrating the kernel of a programming environment an open architecture makes sense for flexibly integrating and replacing new and peripheral tools which do not need the same degree of data, control, and user interface integration as the kernel tools.

In designing Sniff we decided, therefore, for a medium degree of openness. We built a centralized core which contains the symbol table and all tools which make intensive use of the information stored therein. The class browser and the symbol browser are typical examples for kernel tools. We decentralized services which do not require the same degree of integration as kernel tools and which can be accessed via simple protocols (e.g., the information extractor and the compiler). Furthermore, we provided the external access manager which permits external tools to access the centrally managed information, as well as to send requests and queries to the kernel tools. The overall architecture of Sniff is depicted in Figure 2.

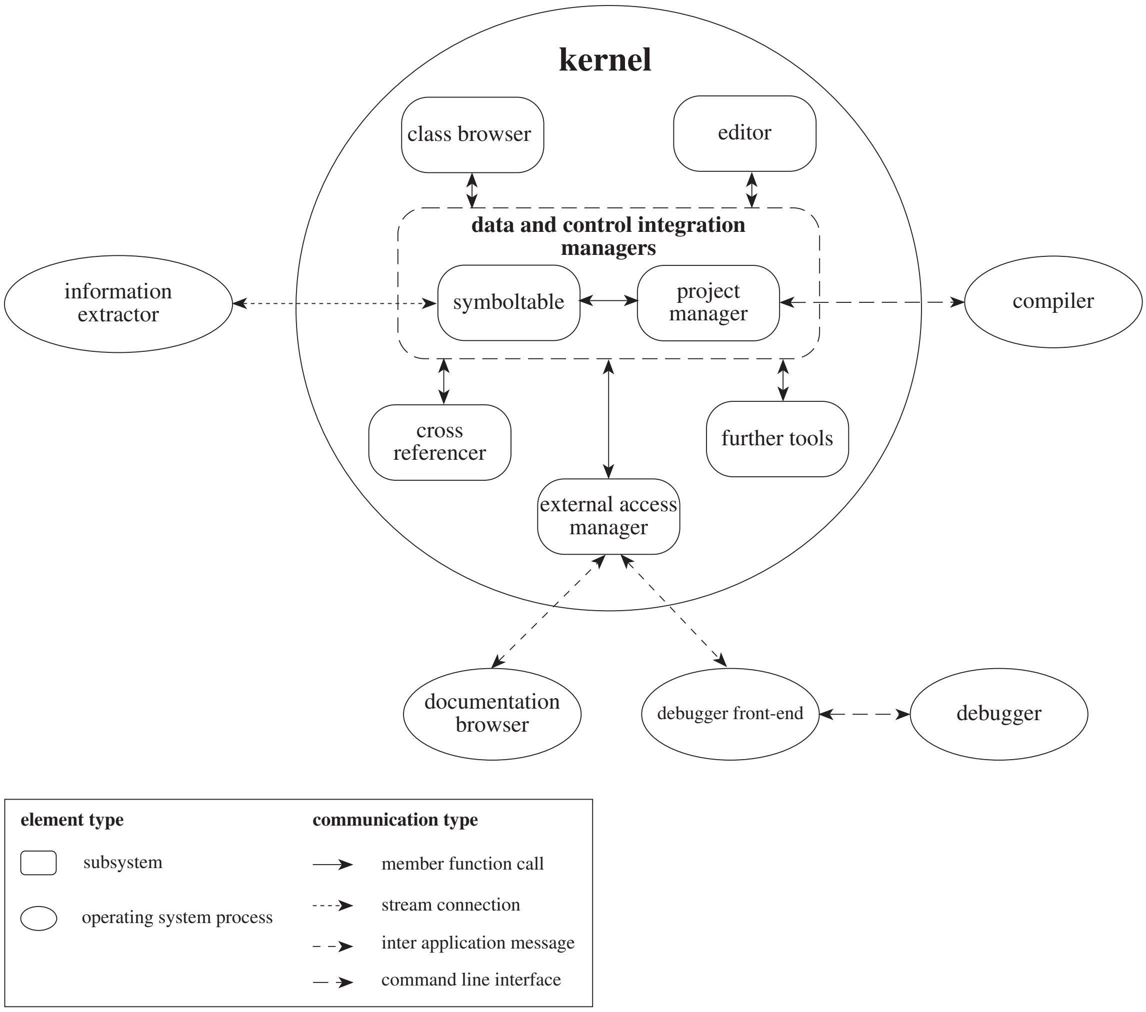


Figure 2: Overview of Sniff's architecture.

Figure 2 visualizes the operating system processes of which a running version of Sniff consists as well as the communication between them. The kernel process is depicted in detail showing its logical subsystems. It consists of the project manager and the symbol table which are accessed by all subsystems and which manage control and data integration.

The subsystems of the kernel communicate with member functions calls. The peripheral tools are integrated with the kernel in the following ways:

- The information extractor communicates with the symbol table over a stream connection.
- The compiler is invoked via a command line interface tool.
- All other tools communicate with the kernel's control and data integration managers via the external access manager. The external access manager provides an open interface based on ET++'s general inter application communication mechanism and translates external requests into internal member function calls.
- The debugger is not connected directly. It is wrapped with ET++'s generic debugger front-end which provides a comfortable graphical user interface and handles communication with the kernel via the external access manager. The debugger front-end uses the debugger's command line interface for communication.

This design was chosen pragmatically to obtain a solution making use of the increased flexibility and openness provided by decentralization without having to pay the penalty in efficiency and in implementation costs resulting from complete openness. The centralization of Sniff makes it possible to extract the entire information about declaration and definition of symbols from the source code when a project is opened. Furthermore, it allows us to store the extracted information in a simple symbol table. Our approach eliminates possible performance and portability problems resulting from the use of a database shared by several processes and it eliminates all consistency problems rising from possible changes to source files between Sniff sessions. Furthermore we have no coordination and locking problems because the core tools run in one process and can therefore not attempt to change shared information in parallel.

A further advantage of our solution is that it becomes possible to update the information displayed in the open kernel tools without having to rely on complex incremental parsing and updating algorithms (which is a basic requirement in a completely open environment). After changes were applied to a source file Sniff reparses the entire file, updates the symbol table, and notifies all tools via change propagation. This easily implementable and maintainable scheme results in a completely satisfying runtime efficiency. Externally connected tools are notified about changes with a broadcast message.

The decentralized information extractor uses a simple stream based protocol to communicate with the kernel. It obtains requests for the parsing of a source file and it returns a tokenized stream of symbol related information which is analyzed by the symbol table. The decentralization of the information extractor has several positive effects. It makes it possible to share an information extractor between different instances of Sniff running on various workstations. It can positively affect response time if it runs on a fast server machine. It makes it easy to replace the information extractor which could, for example, make sense to give a developer the possibility to choose between a fast version without macro expansion and a slower version expanding macros (see also Section 4). A further interesting effect of the decentralization of the information extractor is that it makes it easy to support other object-oriented programming languages than C++, by simply writing an information extractor supporting the same protocol as Sniff's C++ parser.

Compilation is triggered from a command line interface tool by invoking the make facility or by calling any shell script provided by the user. This decentralization strongly increases portability and flexibility. Environments providing their own compilers do not only have to adapt them to changes in C++ (which are not scarce) but they also have to adapt code generation when the tool is ported to a new hardware architecture. A further advantage is that a developer can always work with the “best” available C++ compiler.

The connection of different debuggers is no trivial enterprise. Our approach is to provide a generic debugger front-end [Gam92] which can be instantiated for different kinds of debuggers. We currently work with a stand alone instantiations for the gdb and dbx debuggers. The next step in evolving Sniff is to integrate this generic front-end with Sniff as depicted in Figure 1.

In designing Sniff we have tried to pragmatically balance the advantages of openness with the development and runtime costs it causes. The result is a fast, user friendly, partially open programming environment. The degree of openness is appropriate for the current configuration of tools which were added with custom designed mechanisms. If our general access mechanism is appropriate will be evaluated when we integrate further tools such as the generic debugger front-end and design tools.

One architectural improvement we foresee is the unification of the external and internal control integration mechanisms. If the external interface for sending queries and requests proves suitable we will route the internal requests and queries over the external access manager.

## 4. The Fuzzy C++ Parser

The C++ parser is the foundation of Sniff because it provides most of the information which is managed by Sniff. The requirements for the parser were speed, adaptability and portability.

To satisfy these requirements we have developed a fuzzy recursive descent parser which has only a partial understanding of C++, which can deal with incomplete software systems containing errors, and extracts information about where and how the symbols (e.g., classes, members, variables) of a software system are declared and where they are defined. This way only declarations and the headers of (member) functions (i.e., return code, name and argument list) have to be parsed while the executable code in the bodies of (member) functions can be ignored. The extracted information is transmitted as a tokenized stream to the client.

The parser does not expand macros but parses the macro definitions. This makes it possible to drastically reduce the amount of code which has to be parsed (because included header files are only parsed once) and to provide information about macros. The disadvantage of this approach is that macros can confuse the parser. (Non syntactical macros result in an error recovery process. Information about the next declaration may be lost. Macros used in software systems to be developed with Sniff should therefore be syntactical.)

Our approach makes only sense if the information loss and the extraction of erroneous information are negligible. For this reason we discuss some important applications of macros in C and C++ and the effect of not expanding them.

1 Macros are used as constants. These constants are usually used in the executable code and represent therefore no problem for the parser.

2 Macros are used to facilitate library integration by putting a unique, distinguishing string before all class names (e.g., "iv_" in InterViews and "ET_" in ET++). For a developer it is an advantage if the effect of these macros is not visible in the programming environments.

3 Macros are used to generate supporting code fragments from a few parameters such as class descriptors (e.g., [Gam90]). The generated code is usually of no interest to the user and should be ignored in browsing. If these macros resemble declarations they can cause extraction of erroneous information.

4 Macros are used to hide differences between C++ versions. These macros are not syntactic if they are used to delete or replace key words. A typical example is the removing of the "= 0" for C++ versions which do not support pure virtual functions.

5 Macros are used to localize multiply used strings for improving changeability. E.g., the replacement of the name of the base class with a "SUPER" macro in the class declaration and in all places where an overridden member function is called. This application of macros leads to a heavy loss of information. Software systems containing this kind of macros should therefore not be developed with Sniff.

To prevent the problems caused by macros described in points 3 and 4 a developer has the possibility to define a list of strings to be ignored and a list of strings to be replaced with other strings. This simple mechanism makes it possible to ignore macros looking like declarations and to insert the correct key words where a macro was used to hide differences between different C++ versions.

During the last months Sniff was used to browse a lot of software systems and most of them did not use macros in a problematic way. Examples for such software systems are InterViews [Lin87] and ET++ [Wei89]. The only unsuited software system we tested is the NIH library [Gor87], where base class names are inserted with the macro "SUPER". This makes Sniff believe that the NIH library has a pretty shallow inheritance hierarchy. The problem of localizing the name of the base class could be solved elegantly by inserting a typedef for "SUPER" into each class declaration and by referencing the type "SUPER" to invoke overridden member functions [Str92].

In designing the C++ parser we had to balance parsing speed, adaptability, development effort, and the possibility to work with incomplete software systems containing errors against absolute correctness. We decided for a pragmatic solution. From our point of view it is more important to have a simple fast parser which works good for most software systems than to have a considerably slower parser which always extracts the correct information.

Most measurements we present were obtained based on the source code of the main ET++ directory (release 2.2). For this reason we list the most important information about the size and content of our test data in Table 1. If it is not mentioned explicitly the measurements were taken on a SUN SPARCstation1.

| | |
|---|---|
| number of files | 243 |
| total size of files | 862 KB |
| lines of code | 40,300 |
| number of classes | 235 |
| number of member functions | 2893 |

Table 1. Properties of the source code used for measurements.

The actual source code size of the parser is 30 KB on 1500 lines of code. It was implemented and tested together with the symbol table in about three months. It parses the source files in the main ET++ directory (see Table 1) without constructing the symbol table in about 10 seconds. Together with the symbol table construction it takes about 20 seconds.

The adaptability of Sniff's parser is best exemplified by two extensions we applied. Once we recognized the importance of supporting old stile C as defined in [Ker88] the parser was extended in one hour to correctly read the old parameter lists. Support for the template construct was implemented in two days (including the adaptation of the symbol table and the user interface).

With the growing speed of processors it could be possible that our pragmatic decision has to be revised. In this case we will be able to profit from Sniff's openness by writing a new parser which expands macros, has a better understanding of C++, and supports the same stream protocol as the old version.

## 5. Symbol Table and Cross Reference Information

A C++ symbol table is a complex data structure which has to be able to cope with large amounts of data. Sniff's symbol table is a nested structure of container objects provided by the ET++ foundation classes and basic information elements. The symbol table makes extensive use of hashed data structures. The class hierarchy of objects managed by Sniff's symbol table is visualized in Figure 3 which shows Sniff's hierarchy browser. The browser is displaying Sniff's class hierarchy restricted to all classes which are base or derived classes of class SymtabObj.

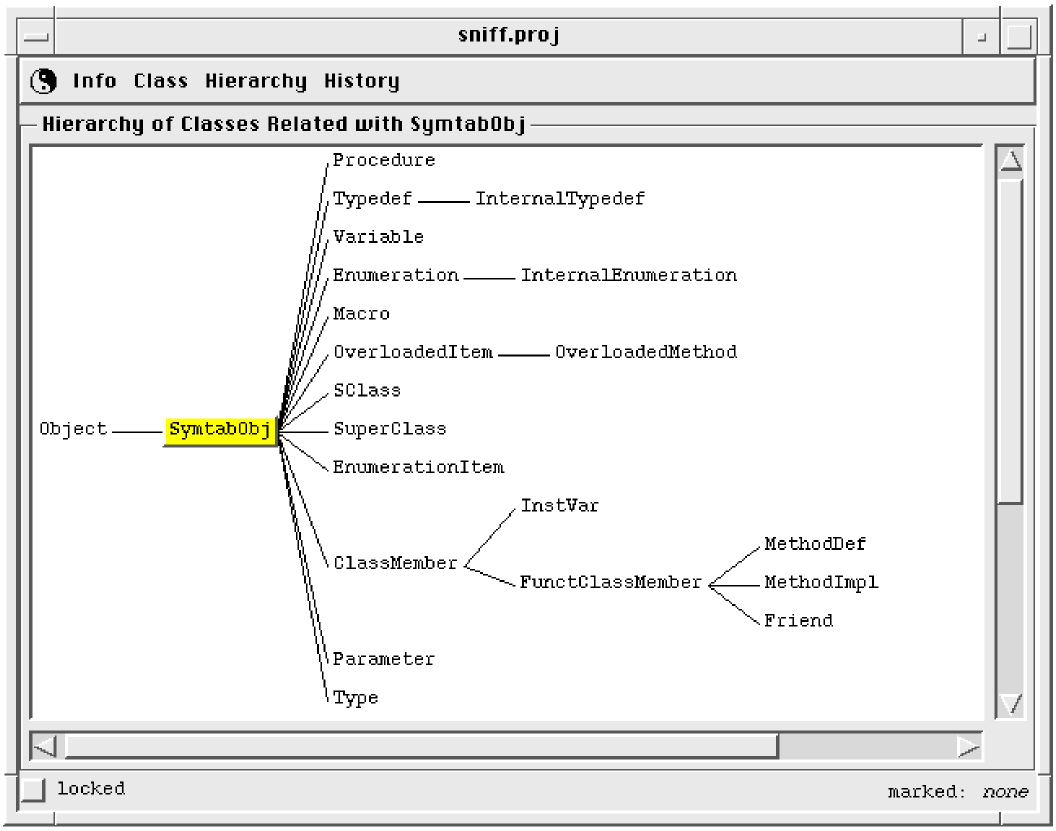


Figure 3: Class hierarchy of the objects stored in a symbol table.

The quality of a symbol table depends on what kind of information it stores, on the amount of storage it needs to represent a software system, and on the time required to retrieve an information subset. A symbol table storing the information extracted from the main ET++ source directory (see Table 1) needs about 3 MB of RAM. The time required to search the symbol table for all symbols matching a regular expression is less than a second.

Sniff's symbol table stores no cross reference information because this would increase the required amount of storage by an order of magnitude and because the extraction, updating, and storing of cross reference information after every change would consume too many resources. For this reason we tried to extract the cross reference information on the fly. The first prototype of the retriever tool shown in Figure 4 searched all source files sequentially with a regular expression matcher. The results were so satisfying that we did not consider any more sophisticated approaches for implementation.

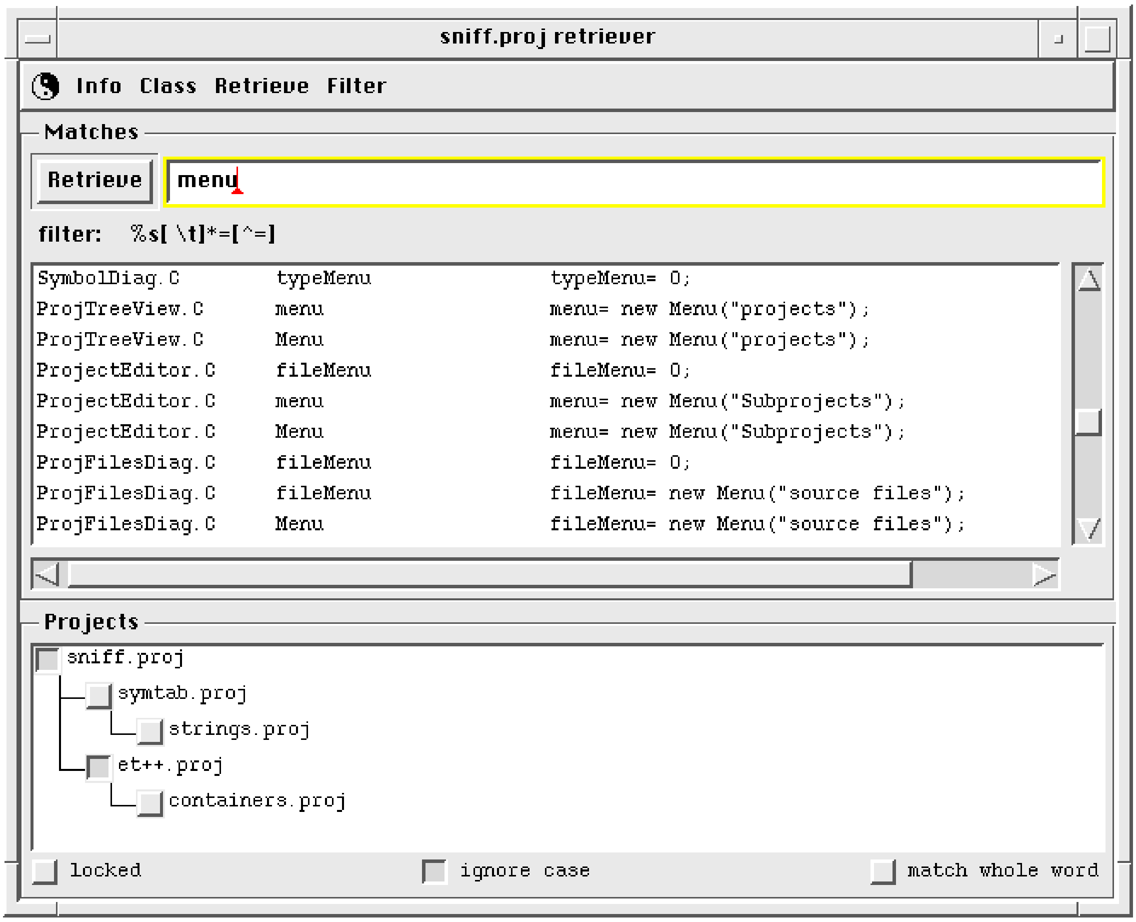


Figure 4: Sniff's retriever tool.

Sniff supports a similar approach as taken by developers searching their source files with the grep utility for occurrences of a regular expression. The difference is that Sniff provides considerable support for filtering and browsing the results of a query. The filter mechanism we use most frequently allows to restrict the results of a query to matches in the source files of a subset of the nested projects (see "Projects" subview in Figure 4).

The following list describes a subset of further filtering criteria.

- Does the match consist of a whole word or substrings?
- Is the matching carried out case sensitive or not?
- Is the match part of an assignment or comparison operation?
- Is the match a parameter for an invocation of "new"?
- Is the match followed by a "(" and therefore probably a (member) function invocation?
- Does the line containing the match match a further regular expression?

The disadvantage of our approach is that a subset of the reported matches may be irrelevant. Fortunately, our experience in browsing large software systems has shown that this is almost never a problem because the relevant matches are easily recognized by a developer.

Experience has even shown that our approach is frequently more convenient than a conventional cross referencing mechanism which exactly reports all uses of a certain symbol. In using Sniff's retriever tool it is possible to issue fuzzy queries. These fuzzy queries make it possible to get information which is not easily detectable with classic cross referencing mechanisms.

Derived classes are, for example, frequently named by appending something to the name of the base class (e.g., Button, ActionButton, ToggleButton). By searching all occurrences of "Button" and applying the "new" filter all places can be obtained, where any kind of button is created on the heap. Another strategy we frequently apply bases upon the fact that in most software systems variables storing objects have sensible names containing parts of the names of the class of the objects they store. By retrieving all occurrences of the string "menu" and applying the assignment filter it is, for example, possible to obtain all places in a software system where a menu is assigned to an (instance) variable (see also Figure 4).

Fuzzy queries are very helpful in browsing large software systems because they make it possible to extract and browse related information in one step. Their usefulness depends only to a small degree on the search process itself but rather on the expressiveness of identifier names and on the cleverness of the employed filtering techniques.

Searching all strings matching a regular expression in 850 KB of ET++'s source code takes about 6 seconds on a SPARCstation1. In working with Sniff the range of projects which has to be searched for a query can easily be restricted so that according to our experience the average response time of cross reference queries is about two seconds. The symbol table lookup of all declarations matching the same regular expression takes less than one second as discussed in the previous section.

The implementation of Sniff's cross referencing mechanism is another example for our pragmatic approach. We had to balance the advantage of a slow space consuming solution providing *exact* results against the fast solution which can result in too many matches. We first decided for the fast solution because we go with Andrew Koenig in believing that "Interestingly, sometimes a fast answer that's slightly wrong, but almost right, is preferable to a slow one that's closer to being right." [Koe92]. In retrospect we believe that our solution is not only faster but also better because of the power of fuzzy queries.

## 6. Realization of the User Interface

Programming environments are highly interactive tools which visualize information about a software system and give support to manipulate and execute it. This is why their user interfaces consume a large percentage of the overall development effort and why they are an important factor affecting user friendliness.

While a discussion of the look and feel of Sniff's user interface goes beyond the scope of this paper we feel that it is important to discuss some technical aspects such as its overall architecture, the tools we used for its implementation, and the experience we made with both of them.

Sniff's architecture is characterized by the clear division between data management and user interface. The data management consists of the project manager which manages the project related information and the symbol table which manages information that is extracted from the source code (i.e., the model). The user interface consists of several tools which serve as views on project and symbol data. This model/view view of Sniff' s architecture is depicted in Figure 5.

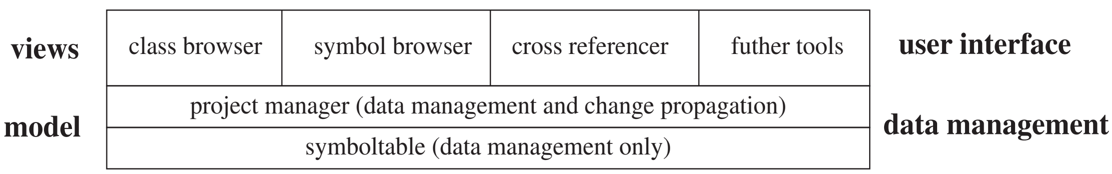


Figure 5. Sniff's architecture (model/view view).

The views extract and store data by using the access member functions of the project manager and the symbol table. From this point of view the project manager and symbol table would form one single data management layer. This is not the case because the project manager shields the symbol table in two ways. First, it is the only data managing part known by the views and it forwards symbol specific queries to the symbol table. Second, it is the manager which notifies the views about changes in the model via change propagation. The effect of shielding the symbol table is that the symbol table knows absolutely nothing about the views displaying its data and serves as mere data repository. The symbol table can therefore be reused in any other software system requiring information about C++ source code without having to be modified. The project manager is less decoupled from the user interface because it does not only provide information but has to use the same change propagation mechanism as the user interface tools.

Sniff's user interface was implemented with ET++ ([Wei88], [Wei89]), an object-oriented application framework. In rating the influence of ET++ on the development of Sniff we see two positive influences.

First, the reusable high-level parts of ET++ such as the graph browser, the text building block, the shell view, and the list manager strongly reduced the implementation effort of the user interface. It took us, for example, one day to implement the first version of the hierarchy browser, the tool for visualizing the inheritance graph and providing support to visualize queries such as "mark all classes related with class X and defining member function Y".

Second, the standardized, reusable architecture embodied in ET++ made it possible to apply a prototyping-oriented, exploratory development strategy. Because we had an architectural framework it was possible to evolve Sniff instead of having to rewrite major parts after major redesigns of the user interface. According to our experience this is impossible if a toolkit providing only basic building blocks is used because in this case basic architectural decisions taken by the developer are frequently revised during redesigns. The most impressive experience we made in evolving Sniff was that it was possible to painlessly evolve the symbol browser, the first tool we had implemented as a quick shot to test the contents of the symbol table.

We did not only select pragmatic technologies in realizing Sniff but we took also a pragmatic development approach in starting with a small prototype which was then evolved into the first distributed version of Sniff. Our prototyping-oriented evolutionary strategy made it possible to evaluate the quality of the selected technical solutions early in the development process. Furthermore, it lead to a constant evolution of the user interface because we had a lot of feedback from early adopters. While the evolution of the user interface took us a certain amount of time it was carried out without painful global redesigns because of the high degree of reusability of the ET++ application framework.

## 7. Portability

An important goal in designing Sniff was to ensure portability, i.e., to make it possible to port Sniff to a new platform by simply recompiling it. While we did not yet practically validate the portability we are optimistic because of the following three reasons.

- ET++, the application framework used to implement Sniff's user interface, is easily portable. It runs under OSF Motif, OpenWindows, and SunView on a number of different UNIX workstations.
- The problem of porting the execution mechanism, i.e., a compiler or an interpreter, does not occur in porting Sniff, because Sniff delegates compilation to any specified compiler running on the new platform.
- Sniff is implemented entirely in C++ and compiles on a wide range of C++ compilers such as AT&T cfront, SUN C++, and Gnu C++.
- The gdb instatiation of Etdbg, the generic debugger front-end, runs on many platforms.

## 8. Comparison with Similar Tools

In comparing programming environments it is possible to have a look at their external behavior (i.e., what kind of functionality do they provide with which response times) or it is possible to study their architecture and the approaches which were taken in implementing their components.

The external behavior can only be fairly judged if all tools to be compared were applied for a certain time in a real world development project. This is exactly what we plan to do once distribution of Sniff has started. In this Section we discuss selected technical aspects.

In this discussion we distinguish two kinds of tools. One kind are C++ browsers supporting the study of software systems which are not undergoing changes during browsing. The other kind are full featured programming environments which support browsing, editing, and execution of software systems.

Available or currently discussed C++ browsers are CIA++ [Gra90], the XREF tools [Lej90], and the SUN Source Code browser [SUN91]. The basic idea underlying these tools is to extract information with adapted compilers from C++ source code, to store this information in a data base and to provide tools to browse the data base. CIA++ and the XREF tools store their information in a relational data base which can also be directly accessed.

These tools differ from programming environments in that they do not have to ensure short update times. From our point of view such tools are interesting if they make it possible to directly query the data base. This is, because such a query facility makes it possible to obtain additional information which can not be obtained from the available browsing tools. For examples see [Gra92]. Direct data base access is not provided by currently available C++ programming environments.

In the rest of this section we compare some aspects of Energize from Lucid, ObjectCenter from CenterLine, and Objectworks\C++ from ParcPlace Systems—three commercially available C++ programming environments—with Sniff. We will set Sniff in perspective with Objectworks\C++ and ObjectCenter based on a bench mark study obtained from ParcPlace Systems [Par91]. All performance measurements stem from this study which was written by the developers of Objectworks. Its objectivity, therefore, has to be relativated. The time measurements were taken on a SPARCstation2.

We do not intend to make any statements about the quality of ObjectCenter or Objectworks\C++ because that would require extensive discussion about how the time measurements were obtained. We only want to show the order of magnitude of the differences with the measurements obtained with Sniff.

An important architectural difference between Energize, ObjectCenter, and Objectworks\C++ is their degree of openness. Energize has an open architecture with flexible mechanisms for control and data integration [Gab90]. ObjectCenter consists of two parts, a monolithic kernel and a decoupled user interface which makes it possible to integrate the kernel into other tools. ObjectCenter's kernel provides browsing, hybrid execution (interpretation and direct execution), and debugging. Objectworks\C++ has a monolithic architecture. Sniff's architecture lies in between by providing a centralized kernel with flexibly connected peripheral tools and by providing an interface for integrating other tools which do not need the same degree of control, data, and user interface integration as the kernel tools.

All three discussed tools extract their information about the source code either with an adapted compiler or with an interpreter or compiler in the case of ObjectCenter. The information is then stored in an object-oriented data base (Energize), in the object code (ObjectCenter compiler) or in files (Objectworks). The advantage of this approach is that the information extracted is certainly correct and that cross reference information can be extracted. The disadvantages of information extraction with compilers/interpreters is that it works only for correct C++ code (which may be problematic during extensive reorganizations). Moreover the information extraction is slower because of macro expansion and manifold parsing of header files.

According to ParcPlace's bench mark study the initial information extraction is time consuming. For extracting and loading the browsing information of the InterViews 3.0 ibuild software system consisting of about 36.000 lines of code (1MB disk space) ObjectCenter 1.02 needed 1 hour 21 minutes and Objectworks\C++ 2.4 needed 1 hour and 35 minutes. With Sniff the same activity took about 30 seconds. Both ObjectCenter and

Objectworks\C++ can store the information between sessions so that it does not take that long to open a project the second time.

ObjectCenter and Objectworks both keep all information about a loaded project in main storage, as Sniff does. The difference is that they store also cross reference information (and the interpretable representation of the source code in the case of interpretable files in ObjectCenter). This means that they need much more main storage than Sniff does. Energize keeps its information in an object-oriented data base. While this reduces main storage requirements it slows down accesses.

According to ParcPlace's bench mark study ObjectCenter 1.02 needs about 87 MB of main storage if all source files are loaded for interpretation Objectworks\C++ needs 19 MB of main storage. Sniff in comparison needs 5 MB with ibuild loaded and 7 MB when the InterViews library is also loaded. ObjectCenter and Objectworks\C++ would probably have problems to load InterViews together with ibuild.

There are two facts relativating these measurements. First, ObjectCenter would not be used this way. A developer would make use of the hybrid execution facility to obtain a sensible tradeoff between available information for browsing and main storage requirements. Second, both ObjectCenter and Objectworks\C++ have the information available to provide debugging support. The main storage requirements for Sniff grows significantly during debugging.

Sniff's approach has two advantages in the area of main storage requirements. First, the information required for debugging purposes is only loaded during debugging. Second, gdb [Sta89] makes it possible to load symbolic information on demand which reduces the overall storage requirement.

The updating of information after a source file was changed requires the recompilation (or preparation for interpretation) of this file in the case of ObjectCenter and Objectworks\C++, and an incremental compilation in the case of Energize. This takes certainly more time than the time it takes Sniff to update its information about a source file and update all open tools (1-2 seconds). If the change in the source code is tested immediately the approach of the commercial environments is no disadvantage, because Sniff has to compile the file too. Sniff's approach is an advantage if a developer makes several changes in different source and header files but wants to continue browsing.

## 9. Experience with Sniff

Since we started using Sniff in August 1991 about ten persons have applied it for software development. During this time the C++ parser and the symbol table were almost unchanged while the user interface evolved considerably.

The main experience in developing Sniff is, from a technical point of view, that it is possible to write a user friendly, efficient, complete, portable C++ programming environment in one personyear by taking a pragmatic approach and by building on a powerful class library. The resulting software system consisting of 18,500 lines of C++ code is easily expandable and maintainable because of Sniff's partially open architecture and because our pragmatic approach resulted in the selection of simple and quickly implementable solutions.

Besides the technical point of view the users point of view has to be considered as well. While it is difficult to measure user friendliness everybody using Sniff feels that the way he browses and edits software systems has changed. This is especially obvious with novice

ET++ programmers. According to our previous experience a novice ET++ programmer needed about two months to learn enough to become productive. The last two novices which used Sniff from the beginning became productive in less than half this time because Sniff made it much easier for them to understand the comprehensive application framework as well as the sample applications.

## Availability

Sniff can be obtained via anonymous ftp from iamsun.unibe.ch (130.92.64.10).

## Acknowledgments

Thanks to the referees for their constructive criticism. Thanks to Erich Gamma and André Weinand for their technical support and for many discussions about this paper. Thanks also to Andreas Birrer and Bruno Schäffer for many diverting lunches and the discussions we had about Sniff and this paper.

## References

[Gab90] Gabriel RP et. al.: Foundation for a C++ Programming Environment. Proceedings of C++ at Work-90, Secaucus, New Jersey, September 90

[Gam90] Gamma E, Weinand A: ET++—A Portable C++ Class Library for a Unix Environment. OOPSLA Tutorial, Ottawa, Canada, October 1990

[Gam92] Gamma E: Etdbg—A Generic Debugger Front-End. UBILAB Technical Report, 1992

[Gor87] Gorlen KE: An Object-Oriented Class Library for C++ Programs. Software—Practice and Experience Vol. 17., No. 12, 1987

[Gra90] Grass JE, Chen YF: The C++ Information Abstractor. USENIX C++ Conference Proceedings, San Francisco, CA, April 1990

[Gra92] Grass JE: Object-Oriented Design Archaeology with CIA++. USENIX Computing Systems, Vol.5, No.1, 1992

[Ker88] Kernighan BW, Ritchie DM: The C Programming Language (Second Edition). Englewood Cliffs, New Jersey,1988

[Koe92] Koenig A: Posting to the comp.lang.c++ news group on the USENET, 1992

[Lin87] Linton MA, Calder PR: The Design and Implementation of InterViews. USENIX Proceedings and Additional Papers C++ Workshop, Santa Fe, NM, 1987

[Lej90] Lejter M, Meyers S, Reiss SP: Adding Semantic Information to C++ Development Environments. Proceedings of the C++ at Work Conference, Secaucus, NJ, September 1990

[PAR91] ParcPlace Systems: A Performance Comparison of Objectworks\C++ and Saber-C++ Development Environments. ParcPlace Systems, Sunnyvale, CA, 1991

[Rei90a] Reiss SP: Interacting with the FIELD Environment; Software—Practice and Experience. Vol. 20, No.1, 1990

[Rei90b] Reiss SP: Connection Tools Using Message Passing in the Field Environment. IEEE Software, July 1990

[Sta89] Stallman RM: GDB Manual—The GNU Source-Level Debugger. Free Software Foundation, Inc., Cambridge, MA, 1989

[Str92] Stroustrup B: How to write a C++ language extension proposal. C++ Report, Vol. 4, No. 4, May 1992

[SUN91] Sun Microsystems: Sun SourceBrowser Reference. Sun Microsystems, Mountain View, CA, 1991

[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, 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