Object-Oriented Programming Environments: Requirements and Approaches

Walter R. Bischofberger, Thomas Kofler, and Bruno Schäffer

UBILAB (UBS Information Technology Laboratory), Union Bank of Switzerland, Bahnhofstr. 45, CH-8021 Zürich
e-mail: {bischofberger, kofler, schaeffer}@ubilab.ubs.ch

Abstract. The wide spread of object technology has strongly increased the requirements for quality and functionality of programming environments. In this paper we discuss the reasons for this and work out the concrete demands for browsing, editing and executing large object-oriented software systems. We consider how to meet these requirements with available technology, and what we can expect in the future.
Much of our experience is based on the development and use of the C++ programming environment Sniff. Therefore, our discussion is focused on C++. Nevertheless, most of our conclusions are valid for other programming languages, too.

Keywords: object-oriented programming, programming environments, C++

1 . Introduction[1]

For several years we have been engaged in the development of large object-oriented software systems based on the ET++ application framework [Wei88], [Wei89]. Working with an editor, a compiler, and a debugger, this was difficult and often tedious. Hence, we developed a class browser, hoping it would help us to get a better overview of our software systems. The first prototype was internally used and so successful that we added development tools until we had a full-fledged C++ programming environment called Sniff. At the end of 1992, Sniff became a commercial product which is now available for most UNIX platforms[2]. Since then we have been working on the Beyond-Sniff project, a platform for cooperative software development (CSCSE).

After the completion of Sniff we began to reason on our experience with programming environments for object-oriented languages and what is of general interest in this context. We present the results of this process here, viz. discussions of:

- the impact of object-oriented programming on the requirements for a programming environment (section 2),
- what browsing really is and which aspects of a software system are amenable to browsing (section 3-7),
- what support is necessary for the modification of a software system (section 8),
- the requirements for executing and debugging software systems (section 9),
- approaches for the management of large software systems (section 10),
- implementation aspects which significantly influence the quality of a programming environment (section 11).

We illustrate our theoretical considerations with examples taken from Sniff.

Our concrete experience is derived from C++ programming environments. The statements in this paper primarily relate to these environments. Most of our conclusions are also valid for programming environments for other languages. For the sake of brevity, we restrict ourselves to the functionality that can be provided based on source code. We also refrain from specifying general quality criteria, unless it is of particular interest.

1.1 Definition

We consider a tool or a tool set to be a programming environment if the tool:

- manages the components of a software system, ie, pieces of source code,
- extracts information contained in the source code and presents it appropriately,

[1] Published in Software–Concepts and Tools, Vol. 15, No. 2, Springer Verlag 1994.

[2] The product version of Sniff (SNiFF+) is free for universities and can be downloaded by ftp from eunet.co.at (/pub/vendor/takefive) or from self.stanford.edu (/pub/sniff).

- supports modification of the source code,
- constructs and executes the software system and presents information related to the execution.

As a further important requirement, a programming environment should support team-work. Since team-work is not yet well understood, we have not included this requirement in the definition, and we will only touch team-work in the sequel.

2. General Requirements due to Object-Oriented Programming

The concepts of object-oriented programming have significant influence on the requirements for a programming environment. Inheritance and polymorphism make an object-oriented software system less readable in a linear way than a system written in a procedural style. This is aggravated by the fact that objects respectively classes are relatively small units. Code reading therefore requires frequent context switches, which is a challenge to the navigational capabilities of the programming environment.

Many aspects of an object-oriented system emerge only at run-time. Therefore, a programming environment has to support the inspection of objects and object graphs at run-time, in cooperation with a debugger.

Conventional software engineering is based on the assumption of a more or less sequential development process. This approach is generally known as the "waterfall model" [Boe76]. Experience has shown that this development process is not adequate for object-oriented systems. To illustrate this, we consider two essential tasks in object-oriented software development, namely using and developing reusable components.

The key advantage of object-oriented software development is that it does not start from scratch but builds on reusable building blocks and frameworks. A framework not only contains reusable components but also embodies an abstract architecture. The concrete domain-specific classes have to be embedded in this architecture. This requires a thorough knowledge of the architecture and how the framework classes interact with the domain specific classes, which necessitates intensive inspection of the corresponding implementation. As a consequence, the amount of source code a developer has to browse increases significantly which asks for an efficient programming environment. In addition, the use of frameworks belittles the importance of the design phase in the classic sense, because a framework already represents an abstract design. The primary task is to become familiar with a framework in order to derive and embed domain specific classes. For this reason, the browsing capabilities of programming environments get increasingly important. More information on frameworks can be found, eg, in [Joh88] and [Sch86].

Experience has shown that reusable building blocks and frameworks cannot be developed in a sequential top-down fashion. Johnson writes in [Joh88]: "…useful abstractions are usually designed from the bottom up, ie, they are discovered not invented". Useful abstractions are likely to be discovered only after a set of implementations of several similar classes. This leads to the creation of an abstract class, and subsequent modification of its derived classes. Thus, design and programming are closely coupled and carried out iteratively. According to our experience, this activity is mostly carried out in the programming environment.

We conclude that the requirements for object-oriented programming environments regarding navigation and visualization of information, as well as the requirements for time and memory efficiency, are significantly higher than for environments for conventional procedural languages. This conclusion is illustrated by the fact that developers of object-oriented programs typically spend much more time studying and restructuring than writing code.

3. Browsing Software Systems

Reading and understanding code is one of the main activities of a software developer. Goldberg writes in [Gol87]: "We read programs in order to learn to write, we read to find information, and we read in order to rewrite." The main goal of a programming environment is to optimally support the developer with this information retrieval, in the following called browsing.

There are two distinct ways to browse source code, namely top-down browsing and bottom-up browsing.

A developer browses a software system top-down to learn more about its structure. A developer of an object-oriented software system can learn about classes, their interfaces, their relationships among each other, and their implementations. To accomplish this task, a programming environment has to collect the necessary information from the source code, condense it, and present it appropriately.

A developer browses a system bottom up to learn more about a certain entity in the program, its purpose, and its embedding in the software system. Understanding a certain piece of code requires, for example, knowing what variables are of which type, where they are declared, where they are referenced, which functions are called, and what the definitions of these functions are. During bottom-up browsing, a programming environment has to visualize this implicit hypertext structure and support its traversal.

Browsing a software system not only means for a programming environment to answer the corresponding questions but also to provide appropriate navigation aids. In this context, it is of interest what queries are feasible and how the information required to provide efficient browsing options can be acquired and managed.

The following three sections discuss the requirements and solutions for various browsing tasks: browsing definition and usage of symbols (Section 4), browsing classes (Section 5), and browsing object-oriented applications at run time (Section 6). Section 7 discusses aspects that are relevant for the browsing tasks presented in sections 4 through 6.

4. Browsing Definition and Usage of Symbols

During bottom-up browsing, a developer needs to know where a symbol (ie, symbol table entry such as a class or variable) is defined and where it is used. Providing this information makes the implicit hypertext structure of a software system explicit. The C++ browser Dogma [Sam90] provides a user interface that is a good example for this particular notion of a hypertext.

Conceptually, it is straight forward to retrieve and present the information about the locations where symbols are defined and used. The information can be retrieved from the symbol table or extracted from the source code on demand.

Programming environments can be distinguished on how they support a developer who does not know exactly what he or she is looking for, on how they support a developer in filtering large amounts of information, and on how they present the information.

Most programming environments neglect the fact that at the beginning of a query, the developer often does not know exactly what he or she is looking for (especially during top-down browsing). As a consequence, it is important that the environment can handle general queries which can be refined incrementally.

A programming environment has to provide appropriate filter mechanisms to help a developer to find his way through large amounts of information. Filters can be based on regular expressions, information about the context of a location in the source code, information about the project structure (see 10.1), etc.

The presentation of information is a matter of the user interface and an important issue. For the sake of brevity, however, we will not discuss it in further detail.

Solution Implemented in Sniff

Every Sniff browser presents the result of a query. Queries about the definition of a symbol can be based either on exact matches or on regular expressions. The result of regular expression based query are all symbols of a certain type, the name of which matches the regular expression. This allows a developer to ask questions such as: show me all classes the name of which contains the string “menu”.

The starting-point for Sniff's cross-referencing is that conventional cross-referencing is too specific. In conventional systems, a developer can ask, for example, where a certain variable is referenced. But he or she will not find out whether there are similar variables or where a variable is mentioned in a comment. Therefore, we developed fuzzy cross-referencing for Sniff. The idea is to retrieve, in a first step, all locations that match a regular expression. If the result is too extensive, the developer may apply semantic filters in a second step. Semantic filters are also based on regular expressions. A typical example for a semantic filter is: show only those positions where something occurs in the context of an assignment or a comparison.

With this two-step approach we can achieve almost the same results as with conventional cross-referencing, but we can do more. An example is searching for the string “menu” in an application framework. As a result we get hundreds of positions in the source code. If in the second step the assignment filter is applied we get all positions where something is assigned to a variable whose name contains “menu”. As long as variables referencing objects have reasonable names, this query returns (among others) all locations where a reference to a menu object is assigned to a variable. Experience with this type of query has shown that we can achieve amazing results in various object-oriented libraries.

Static Call Graphs

Static call graph browsers visualize usage relationships between functions. In a procedural software system the static call graph is the only meaningful information about the cooperation of components. This graph can be reasonably visualized in two dimensions.

In object-oriented software systems, member functions (methods in Smalltalk terminology) usually are bound dynamically. When calling an operation on an object, the selection of the member function to be executed depends on the type of the object. As a consequence, the call graph has to show all possible variants. The graphical presentation of the call graph, therefore, gets usually so messy that it is worthless for a developer. Nevertheless, the programming environment should be able to show which functions could be executed on the invocation of a dynamically bound member function.

5. Browsing Classes

Studying the source code of a class is not sufficient to comprehend a class. We need to know its base classes and all classes that cooperate with it. This information may be distributed across any number of files but logically belongs together. The programming environment should be able to present the information as one coherent unit.

5.1 Browsing the Inheritance Graph

In order to gain an overview of a class library it is useful to graphically visualize the inheritance graph. Unfortunately, a simple drawing of the graph becomes unwieldy if a class library comprises hundreds of classes.

A common solution to this problem is that the user specifies a particular class and then a specific subgraph he or she wants to see. In our opinion, this solution is not satisfactory because it requires too much user interaction.

There must be a better way to restrict the number of classes visualized. We propose to provide filters to pick out the classes of interest.

Possible filters are:

- Use the information contained in the inheritance graph itself (eg, focusing on a certain class family, ie, an abstract class and all derived classes).
- Use regular expression matching on class names.
- Use library information (eg, focusing on the classes of a certain library).
- Use design information, such as design patterns [Gam92], to restrict the inheritance graph.

Additional information can be visualized in an inheritance graph by blending it with function and data members (in Smalltalk terminology instance/class method and instance/class variable) of particular classes. We think that this is a good idea, but it leads to useful visualizations only if the set of shown members can be restricted in a simple and straightforward way. If this is not possible, the overview is quickly lost.

Solution Implemented in Sniff

Sniff's hierarchy browser visualizes a class hierarchy as a graph. The user can restrict the classes in a graph as follows:

- to a class family, ie, a class together with its derived classes,
- to the classes of a certain project[1],
- or to all classes whose name matches a regular expression.

In addition it is possible to explicitly hide subgraphs.

The visualization of a class family is no problem since it is just a subgraph. Difficulties may arise, however, if the classes of certain projects are hidden: this may lead to disruptions in the graph, because classes which are hidden may be parents of classes which are still visible. The empirically found solution for this problem is to avoid hiding abstract classes, whereby the skeleton of the graph is preserved. Hidden concrete classes that have derived visible classes are contracted to a dot. With this solution, it is possible to grasp the meaning of a large class graph or parts thereof quite efficiently. However, a prerequisite is that abstract classes are marked (ie, in C++ by defining at least one pure virtual function.)

Another important aspect that can be visualized in a class graph is the overriding of member functions. In the hierarchy browser, all classes of a class family or all classes that define a certain member function can be marked.

Possible Enhancements

Reasonable further solutions require the availability of design knowledge that cannot be extracted from the source code. For example, it is desirable that the developer can explicitly define protocols (in Smalltalk terminology method categories). A protocol is an aggregation of methods that implements a certain functionality for an object (eg, layout, activation/passivation etc.). On the basis of protocols all classes that override member functions of a protocol could be marked. These protocols could also be used for the selection of member functions to be shown in the class graph.

5.2 Browsing a Class with its Base Classes

The study of a class is centered around its defined and inherited members (member functions and data members). In order to gain overview of all members of a class, its members have to be merged with the set of all inherited members by flattening the inheritance hierarchy. It must also be possible to apply filters.

If a language has a notion of access rights for members then this has to be taken into account as well.

Solution Implemented in Beyond-Sniff

Beyond-Sniff's[2] class browser supports the flattening of a class hierarchy. Figure 1 shows a class browser, focused on the class Collection. The list in the middle contains all members and the structure view below display the inheritance hierarchy.

The two icons in the list of member functions visualize the following information:

- The first icon indicates whether the member function overrides a member function of a base class (triangle pointing left) and whether the member function is overridden in a derived class (triangle pointing right).
- The second icon visualizes further attributes of a member function. A circle indicates a “regular” member function, a triangle indicates a virtual one, and a square indicates a static one.
- The shading of the geometric objects visualizes the access rights of the member function. White means

[1] Sniff allows to aggregate files into projects and to build up a software system as a tree of projects (see also section 10)

[2] Beyond-Sniff is a platform for the development of cooperative software engineering tools. Based on this platform we developed Beyond-ClassBrowser, which is similar to Sniff's class browser but provides substantially refined filtering mechanisms.

"public", gray means "protected", and black means "private".

Basically, the following filters can be applied to the list of members:

- Restriction to a subset of members (method category) defined in the class graph.
- Restriction to all members whose name matches a certain regular expression.
- Hiding of overridden methods.

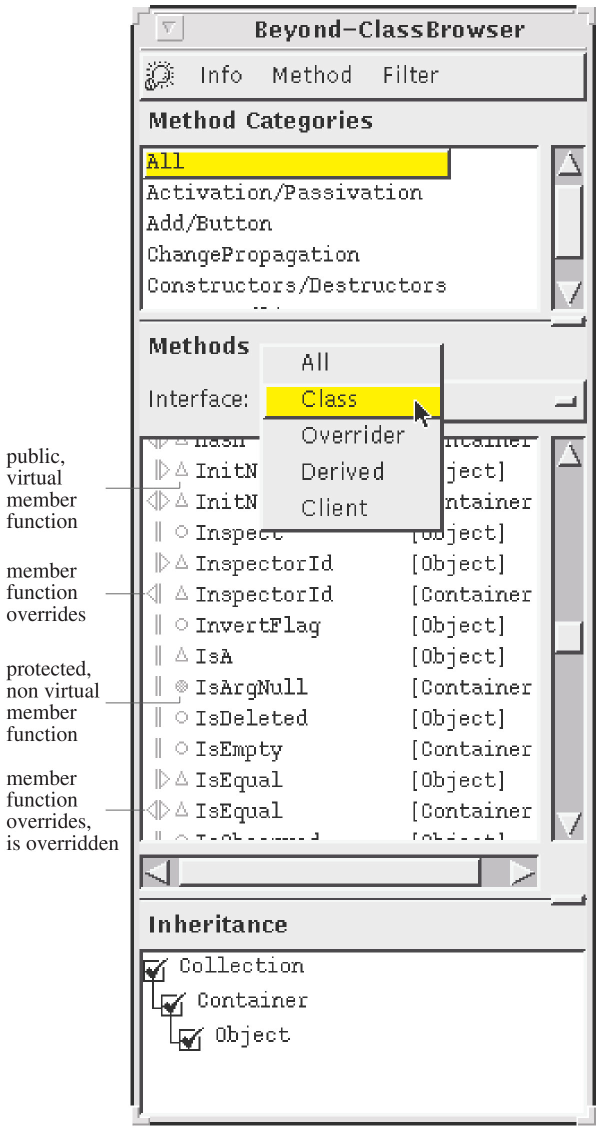

Figure 1. Beyond-Class Browser

Furthermore, five different views can be displayed by selection from the pop-up menu above the member function list:

- all: all members defined in the class and inherited from its base classes,
- class interface: all members that can be referenced in methods of this class (ie, all members of this class and all public and protected members of its base classes),
- derived interface: all members that can be referenced in a derived class (ie, all protected and public members of this class and its base classes),
- overrider interface: all overridable member functions (ie, all protected and public virtual members of this class and its base classes),
- client interface: all members that can be referenced anywhere (ie, all public members of this class and its base classes).

Possible Enhancements

When a developer studies a class and its base classes he or she is often confronted with a large number of methods. An aggregation of member functions into protocols as mentioned in section 4.1 would be helpful. For example, a browser can then display only the member functions that belong to a certain protocol. This functionality is already implemented in the Beyond-Sniff browser as shown in figure 1.

5.3 Browsing other Relationships between Classes

Besides inheritance information there is a number of other interesting relations between classes. On a lower abstraction level these are "use" and "contains" relationships, as described in [Boo94]. Relations like these can be extracted from the source code and graphically visualized, eg, by means of Booch diagrams. The difficulty is how to cope with the large amount of information. Although most programming environments extract the necessary information, we do not know of any that also visualize it.

On a higher abstraction level there are descriptions and visualizations of the cooperation between classes in the form of contracts [Hel90] and design patterns [Gam92]. Such tools are currently under development, but we do not know of any results that can be used in practice.

6. Browsing the Application at Run-Time

An object-oriented software system comprises a multitude of cooperating objects that often form a complex object graph. A good example is the implementation of graphical user interfaces with the ET++ application framework, where the user interface is realized by a hierarchy of visual objects. This hierarchy is defined by a "contains" relationship that is established dynamically and determines the forwarding of events, which can only be understood at run-time. The visualization of such structures is an indispensable aid for a developer.

Intuitively, one tends to assume that object graphs are preferably visualized graphically. However, experience has shown that this is reasonable only in rare cases and only if additional knowledge is available to filter the wealth of information.

The support provided in existing programming environments for filtering run-time information is insufficient. Usually, they merely allow navigation through dynamic data structures one step at a time, following pointer references.

Solution Implemented in Sniff

Sniff does not provide tools of its own to visualize objects at run-time. However, Sniff is tightly integrated with the ET++ run-time browsers. Therefore, all necessary information for ET++ applications can be visualized exemplary.

The ET++ inspector (figure 2) lets a developer quickly gain an overview of all objects of a certain class. The top left view lists all known classes. The number in brackets following the class name indicates the number of currently existing instances. If a class is selected, all objects of this class are displayed in the middle list. If an object is selected, it is displayed in the large subview together with its instance variables. The right list serves to display all objects that reference the currently selected object. Object graphs can be inspected by selecting an instance variable, whereupon its referenced object is loaded into the large subview.

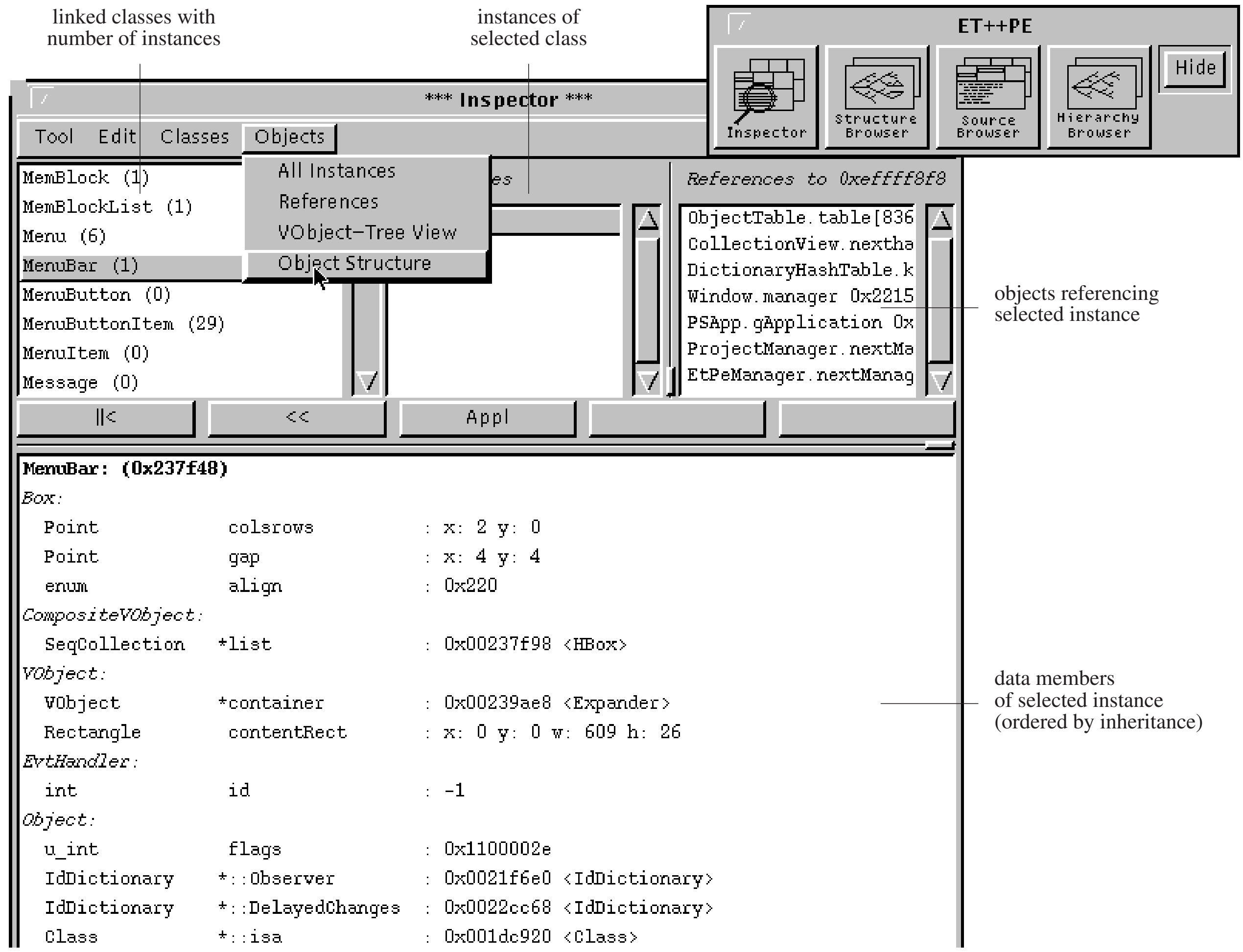

Figure 2. ET++ inspector

For certain objects there is additional information available, by means of which particular object graphs can also be visualized graphically. This holds, for example, for ET++ container objects and objects used for the creation of graphical user interfaces. Figure 3 shows the visualization of the object graph representing the ET++ inspector shown in figure 2. This graph marks explicitly how events from the TextItem object are forwarded and which objects can handle these events.

Additional information about this run-time browser developed by Erich Gamma can be found in [Gam89] and [Bis92/2].

Possible Enhancements

The currently available functionality of Sniff in cooperation with the ET++ inspectors for inspection and visualization of object graphs proved very valuable. Nevertheless, this solution has two basic drawbacks:

- A developer cannot specify interesting relationships dynamically and have them visualized at run-time.
- The ET++ inspector works only for ET++ applications. Moreover, the necessary meta-information has to be manually supplied by the developer, who has to write two macro calls for each class.

Currently, we are working to overcome the first disadvantage. The basic idea is that by means of a dedicated tool a developer describes relationships of interest between objects. Using this description, a visualization tool generates the desired graphical representation at run-time.

The second disadvantage is inherent to C++. The language definition does not provide any meta-information at run-time and therefore each programming environment has to implement its own solution (eg [Gam89]). It would be indisputably attractive if the compiler provided meta-information at request. But as far as we know, there is no compiler available that provides sufficient information to implement tools comparable with the ET++ inspectors. However, the optimal solution would be, if the C++ standard would define sufficient meta-information available at run-time. The C++ standardization committee is currently discussing run-time type information, but the proposed support is not sufficient.

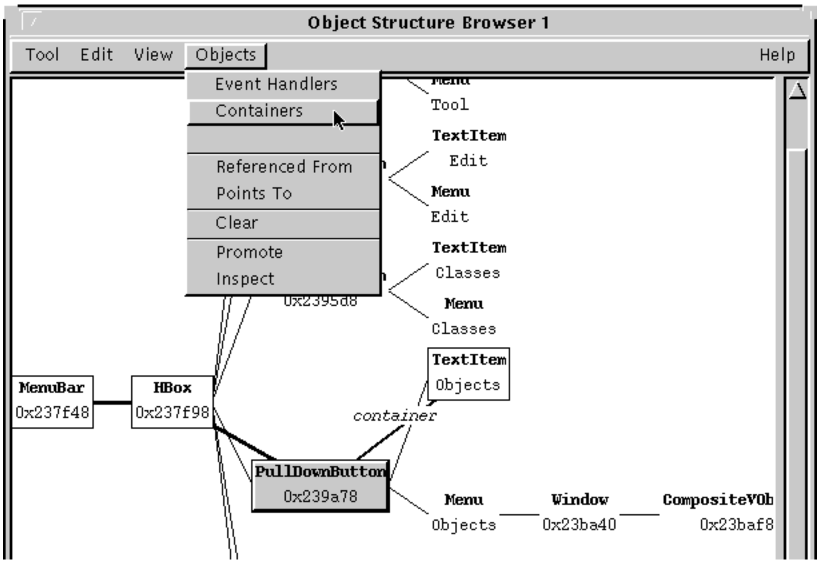

Figure 3. ET++ object structure browser

7. General Aspects of Browsing Support

The homogeneity of the user interface and short response times are major factors that contribute to the overall quality of a programming environment. For the sake of brevity, however, we will only address issues related to browsing in this section. The first is how to prevent situations where a developer loses overview. The second issue concerns reading source code itself.

7.1 Keeping Track of Browsing States

During browsing a developer walks through a software system in an arbitrary and seemingly chaotic way, because answers often lead to new questions. Attempting to answer an initial question can result in a lot of new information. Since it is almost impossible to remember all the details, it is crucial to be able to go back to a browsing state to access the information collected there. At the end of a browsing process, the developer usually would like to return to the starting point. Therefore, it is important that a programming environment provides navigational support.

In many programming environments, like for example Objectworks\Smalltalk [PAR90], a new window is opened for every query result. From a user's point of view, this is very unsatisfactory because at the end of a browsing process there are innumerable windows. To proceed with work, the user will have to close these windows one by one, at the same time losing direct access to the information found.

Solution Implemented in Sniff

We were aware of this problem when designing Sniff and tried to avoid the opening of too many windows. Therefore, Sniff tools use a single window, and the developer usually creates only one instance of each tool. The results of all queries are displayed in the tool's window. A new instance of a tool is created only if the developer wants to keep a certain state. Each tool provides a history menu, which allows the user to directly go back to earlier browsing stages.

Possible Enhancements

In order to improve navigation it is important that the developer can save any number of states which involve more than one tool. This way the context for understanding a certain aspect can be restored at any time.

7.2 Reading Source Code

When browsing a developer will focus a major part of his or her attention on reading source code. In the Sniff project we made an important experience: at the beginning we assumed that emphasizing specific syntactic constructs would be futile. Nevertheless, we implemented it, primarily because ET++ already provided a simple code formatter. We were subsequently rather surprised that users liked it, that they even felt that they lost focus when switching to a normal editor. The biggest effect was achieved by the introduction of colored comments: these proved to be substantially better noticed, and, to our surprise, encouraged developers to insert more comments in the source code. Comments are now also used for structuring purposes (eg, the methods of a class are aggregated into protocols and the comment names the protocol).

8. Modifying Source Code

Source code can be modified on two levels. The lower level comprises the usual text editing operations, eg, inserting and deleting characters. The higher level takes the semantics of the text into account.

8.1 Editing

The discussion about programming environments tends to focus on the efficiency of editing. There is no doubt that the quality of the editor is an essential part of the overall quality. However, a developer spends only relatively little time with the actual editing and therefore the efficiency of the editor contributes only little to the overall efficiency. Nevertheless, it is important that the editor is seamlessly integrated with the rest of the programming environment. In particular, it is essential that the developer is always supplied with sufficient information, even during major restructuring.

A common problem with editing is to globally change names and structures. Manually renaming a symbol, as for example a class or a member, can be very time-consuming. Every location that references this symbol has to be found and updated. This operation is error-prone and its automation would be a big help for the developer.

8.2 Refactoring

Object-oriented programming promises that reusable software components can substantially increase the productivity of software development and improve the quality of the software. Even with object-oriented programming, reusable components do not come for free. Their production is a challenging task and requires several redesigns of a class library, of a framework or parts thereof. Experience shows that a developer spends more time on redesign or restructuring than on the first implementation. The most important tasks during restructuring are:

- Introduction of abstract classes, where these serve as new base classes for already existing classes. As a consequence, instance variables and methods are migrated to the abstract class and the derived classes have to be adapted. Class families are built this way.
- Classes which have too many responsibilities have to be split up into several smaller classes. This is the way class teams are built.
- Responsibilities (eg, of a class team) are newly distributed.

These tasks are usually called refactoring. In doing so, the source code is manipulated on a semantic level. As an example, methods have to be moved between classes, instance variables renamed, etc. With a simple editor, this is very error prone. Tools which provide the operations mentioned above should substantially accelerate the development of reusable components. Research has merely begun in this area (eg, [Joh93], [Opd93]). Early results, however, give hope for a significant increase in productivity in the development of reusable software.

Due to the complexity of C++, we do not expect applicable results for this language in the near future. Hence, refactoring has to be done manually. During refactoring, the software system often is in an incomplete or inconsistent state. It is essential for refactoring that the programming environment supports browsing even then (see 11.1).

9. Executing and Debugging the Software System

As long as the typical programming environments consisted only of an editor, compiler, and debugger, its most important task was to execute and debug software systems. In the meantime, weights have significantly shifted, as was shown above.

Nevertheless, the quality of a programming environment depends on how well testing and debugging are supported. Important quality criteria are:

- How long does it take to execute a modified software system again (edit-compile-go cycle)?
- What information is provided by the debugger?
- How well is the debugger integrated into the programming environment? Ideally, the debugger uses the programming environment's editor to display the execution state and browsing features should be available during debugging as well.

Solution Implemented in Sniff

Sniff does not provide execution tools of its own. It rather integrates the best suited compilers and debuggers for a concrete project [Bis92]. Debuggers are integrated using a generic user interface and an adapter architecture that mimics the interaction between user and debugger.

Possible Enhancements

The functionality as described in section 6 could be largely realized in the debugger, where it also belongs conceptually. However, this requires intervention in basic mechanisms of the execution, such as allocation and deallocation of objects, which in turn demands a common evolution of compiler and debugger. For C++, we do not expect much progress in the near future.

10. Management of Large Software Systems

An important task of a programming environment is to help a developer in the definition and control of all artefacts belonging to a software system.

10.1 Definition of the Project Structure

Extensive software projects can comprise several hundred thousands lines of code which usually are spread over hundreds of files. The programming environment is supposed to support the developer at defining and managing the project structure. Essential information is, what files belong to a project and what subprojects a project consists of. There are two approaches defining a project structure:

- Constructive, ie, a description contains all the necessary information, how a target system is built from its various parts. The most commonly used constructive description is the UNIX makefile [Fel79]. Programming environments can determine all parts that belong to a software system by examining such a description.
- Descriptive, ie, all parts belonging to a project are explicitly defined by means of a tool.

The advantage of the constructive approach is that already existing projects can easily be loaded into the programming environment. The two most essential disadvantages are that the makefile approach typically manages only the information necessary for compiling a software system and that one is tied to a certain formalism that constrains flexibility and portability.

The advantage of the descriptive approach is that it defines a logical view on a project, which can be used for source code management as well as various other things (eg, filtering of symbolic information). Furthermore, it is easier to manage a hierarchical project structure since the descriptive approach has more degrees of freedom than the constructive approach.

Solution Implemented in Sniff

Sniff has an explicit definition of projects: a project consists of a number of C/C++ source files and a number of attributes (eg, writable, prelinked etc.). A project can contain several subprojects, which themselves are complete projects again. Figure 4 shows the project editor of Sniff with the loaded project “sniff.proj”.

A project needs only be defined once but it can be reused as a subproject in any project where required. Figure 4 shows two examples for reused subprojects, namely the projects “symtab.proj” (symbol table) and “et3.proj” (ET++ application framework)

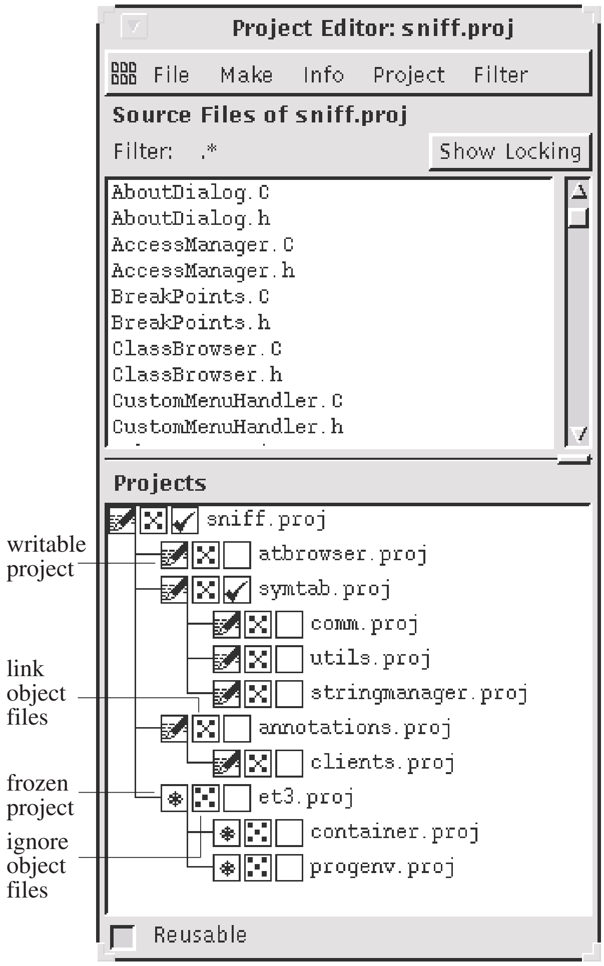

Figure 4. Sniff project editor

Possible Enhancements

The information provided by an explicit project model makes it easy to add additional functionality. Examples are configuration management and version control systems and the coordination of development teams. In both cases the approaches available today are not satisfying, what can be, among others, attributed to the lack of a project concept in widely used programming environments.

Source code files are not the sole parts of a project. An advanced project concept should also allow the integration of documentation and other project relevant artefacts.

10.2 Configuration Management and Version Control

In addition to managing project structure, a programming environment has to support a developer in the control of source code. Most of today's tools provide an interface to version control systems. The functionality of this integration is usually restricted to issuing commands for checkin and checkout from the programming environment. We do not know, though, of any programming environment which supports versioning and configuration management sufficiently.

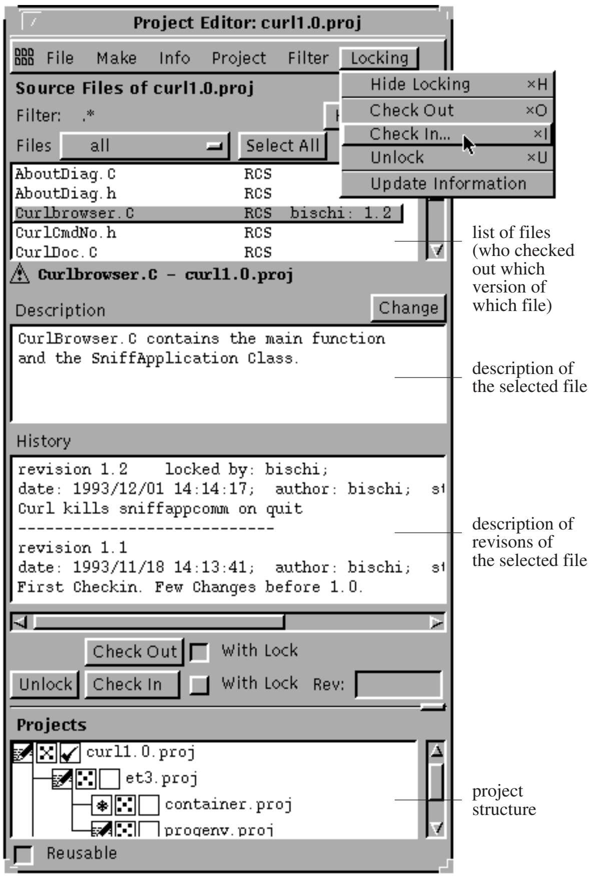

Figure 5. Extended Sniff project manager

Solution Implemented in Sniff

Sniff is equipped with an adapter architecture that enables a developer to integrate various version control systems, such as SCCS [Roc75] and RCS [Tic85]. A developer can trigger the checkin or checkout of a file in the editor. Figure 5 shows the extended user interface of the project manager. It allows to check in or out several files at once and to study the modification history of single files.

11. Implementation Aspects

As mentioned above, a programming environment has to extract information about definition and declaration of symbols and their usage from the source code of a project. It also has to keep that information ready for quick access.

The concrete solution is mostly transparent for the developer but the behavior of a programming environment depends on the solution chosen. Therefore, in the remainder of this section, we will briefly discuss different approaches for the extraction and management of information. For an in-depth discussion of these aspects we refer to [Bis92].

11.1 Extraction of Information from the Source Code

There are two important questions in this context:

- How is information extracted?
- What information is extracted?

For the extraction of information there are two approaches: either the programming environment provides a compiler of its own and uses the information gained during compilation (symbol table etc.), or the programming environment uses a dedicated parser that extracts the necessary information.

Due to weaker requirements, a dedicated parser can be implemented much easier. Moreover, such a parser is significantly faster than a compiler. Therefore, it is a lot easier to keep the information about a software system always up-to-date.

Since a dedicated parser does not need to generate object code, it can be much more error tolerant compared to a compiler. Hence it can extract information even from an incomplete or inconsistent software system, which is essential during large restructuring tasks.

Further advantages of a dedicated parser are high portability and the possibility to integrate different compilers in the programming environment.

A feasible solution to the efficiency problems of the compiler approach would be to develop an incremental compiler. Due to the complexity of C++ we do not expect any results in the near future.

The basic goal of a programming environment is to extract as much information as possible and to provide efficient access to it. However, this can lead to problems with large software systems (several hundred thousand lines of code), because a large amount of information has to be managed. For hundred thousand lines of code we expect about 20 Megabytes of data.

This size can be reduced if some information is extracted on demand. Using a compiler for this purpose cannot be considered due to lack of efficiency, however.

Solution Implemented in Sniff

Because of the arguments stated above we decided to implement a small and efficient fuzzy parser in Sniff. This dedicated parser runs in a separate process and communicates with the remaining programming environment using a simple protocol. Thanks to this separation, the parser can easily be replaced. Thus, the programming environment can be adapted with little effort to new languages whose concepts are similar to those of C++.

11.2 Symbol Table Management

Basically, there are two approaches for the management of the extracted information: using a DBMS or a main memory based data structure.

The advantage of a DBMS is that different tools can flexibly access it, even if they are not running in the same process. Moreover, the amount of information is virtually unlimited and a DBMS offers a rich set of information management functionality. Finally, a DBMS provides a general purpose query processor and a wealth of mechanisms for optimizing access.

Storing information after its extraction is relatively time-consuming. Apart from that, the synchronization of the tools with the database by change propagation is comparatively difficult. After each information extraction, all tools have to find out which information in use has to be updated. Subsequently, all modified information must be loaded again from the database. This can be time-consuming.

Solution Implemented in Sniff

Sniff maintains the information extracted by the parser in a symbol table in main memory. This was possible only because we refrained from extracting all information and storing it in the symbol table. Sniff only extracts information about definition and declaration of symbols. Thus the symbol table of Sniff is by a factor of about 10 smaller than that of programming environments that store all information.

Possible Enhancements

In the future it will be of great importance that different tools can access the information base of a programming environment in an object-oriented way. The current state of the art leaves many problems unsolved, however. At the heart of these are two questions: what is a tool's view of the object graph representing the information base and how can consistency be maintained while objects are being modified.

12. Conclusions and Prospects

We have pointed out why object-oriented programming increases the requirements for high-quality programming environments. We have described the functionality required, presented how it can be realized with current technology (using Sniff as an example), and what further development we can expect in the future.

We feel that the requirements are well understood today. Nevertheless, it is challenging to implement a good user interface and to provide sufficient extendibility and efficiency.

The next step is to investigate how programming environments can support development in teams. In this context, the recent advances in global networking have to be taken into account. This might allow projects to be set up in which members of a team live and work in geographically separate and remote locations.

A programming environment which supports cooperative software development at remote sites has to know much more about projects, its relevant parts and their internal structure. Such a programming environment consists of a set of cooperating tools and therefore has to provide communication components which enable the tools to coordinate themselves, to keep their data consistent, and to distribute the relevant data to remote sites.

Therefore such a programming environment is the ideal integration component for extensive software development environments, since it provides the basic communication and coordination infrastructure. It is also able to manage the wealth of bits and pieces created during the development process. Given the advances in networking and distributed computing, we expect that highly performant systems will reach the market within the next few years.

Acknowledgements

We would like to thank Marc Domenig, Ralph Johnson, and André Weinand for their constructive comments on drafts of this paper and encouragement for redesigning it.

13. References

[Bis92] Bischofberger WR (1992): Sniff–A Pragmatic Approach to a C++ Programming Environment. in Proceedings of the USENIX C++ Conference, Portland, Oregon, Aug. 1992

[Bis92/2] Bischofberger WR, Pomberger G (1992): Prototyping-Oriented Software Development–Concepts and Tools. Springer Verlag, Berlin

[Boe76] Boehm BW (1976): Software Engineering. IEEE Transactions on Computers, Vol. 25, No. 12

[Boo94] Booch G (1994): Object-Oriented Analysis and Design. The Benjamin/Cummings Publishing Company, Inc.

[Fel79] Feldman SI (1979): Make–A Program for Maintaining Computer Programs. Software Practice & Experience, Vol. 9, No. 4

[Gam89] Gamma E, Marty R: Integration of a Programming Environment into ET++–A Case Study. In: Proceedings of the 3rd ECOOP. University Press, Cambridge

[Gam92] Gamma E: Objektorientierte Softwareentwicklung am Beispiel von ET++. Springer Verlag, 1992

[Gol87] Goldberg A (1987): Programmer as Reader. IEEE Software, Vol. 4, No. 5, September

[Hel90] Helm R, Holland IM, Gangopadhyay D. Contracts: Specifying Behavioral Composition in Object-Oriented Systems. OOPSLA'90 Conference Proceedings, SIGPLAN Notices, Vol. 25 No. 10, 1990

[Joh88] Johnson RE, Foot B (1988): Designing Reusable Classes. Journal of Object-Oriented Programming, Vol. 1. No. 22

[Joh91] Johnson RE, Russo VF (1991): Reusing Object-Oriented Designs. Department of Computer Science, University of Illinois Champaign

[Joh93] Johnson RE, Opdyke WF (1993): Refactoring and Aggregation. In Lecture Notes in Computer Science #742, Springer Verlag

[Opd93] Opdyke WF, Johnson RE (1993): Creating Abstract Superclasses by Refactoring. Proceedings of CSC '93: The ACM 1993 Computer Science Conference, February 1993

[PAR90] Parcplace Systems (1990): Objectworks\-Smalltalk User's Guide. ParcPlace Systems, Mountain View, CA

[Roc75] Rochkind M (1975): The Source Code Control System (SCCS). IEEE Transactions on Software Engineering, Vol. 1, No. 4

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

[Sch86] Schmucker KJ (1986): Object-Oriented Programming for the Macintosh. Hayden Book Company. Hasbrouck Heights, New Jersey

[Tic85] Tichy WF (1985): RCS–A System for Version Control. Software Practice & Experience, Vol. 15, July

[Wei88] Weinand A, Gamma E, Marty R (1988): ET++–An Object-Oriented Application Framework in C++. In: OOPSLA 88, Special Issue of SIGPLAN Notices, Vol. 23, No. 1

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