# A Semantic Browser for Object Oriented Program Development

Peter Grogono and Benjamin Cheung
Department of Computer Science, Concordia University
1455 deMaisonneuve Blvd., Montréal, Québec
Canada H3G 1M8

In *Proc. 25th Hawaii International Conference on System Sciences, Volume II*, January 1992, pages 38–45.

## Abstract

*Object oriented methods allow programmers to construct software with a simple and uniform structure. Object oriented programs should be simple to maintain and extend. Source code browsers are not sufficient for understanding object oriented programs.*

*We have combined a strongly-typed object oriented language with an integrated, interactive development environment. For several reasons, we designed the compiler as an integral component of the environment. Coupling the compiler and the browser simplifies symbol table management in the compiler. Conversely, the same coupling ensures that information is semantically checked before the browser displays it. Also, programmers do not have to understand the class hierarchy because the compiler creates class views.*

## 1 Introduction

The lack of reusability of software is a serious problem in the software industry. Object oriented programming has been proposed as a solution. By itself, however, the object paradigm is not sufficient. The paradigm requires support from both a language and a development environment.

Biggerstaff [1, 2] identifies four elements that would support software reuse: finding components, understanding components, modifying components, and creating components. A *software component* may be a library function, a module, or a class.

We describe a development environment which provides semantic browsing capabilities for an object oriented programming language. Both the language and the environment are called "Dee" [5]. The Dee browser satisfies two of the criteria identified by Biggerstaff as fundamental to software reuse: finding suitable components and understanding them.

A significant aspect of the Dee environment is that we designed the language and the compiler in parallel. The compiler is tightly integrated into the environment. Our approach contrasts with the conventional scenario in which a team first designs a language, then writes a compiler for it, and finally develops a programming environment.

Program development in object oriented languages typically requires more interaction with existing code and its documentation than does development based on other paradigms. The object oriented paradigm encourages programmers to extend existing classes by inheritance rather than to write completely new modules. Since the "inheritance" relation is more intimate than the "client" relation, programmers must acquire familiarity with the details of many classes before they can program productively. While programming, they will frequently need to confirm their recollections of the services provided by the classes they are using.

For these reasons, programmers who use object oriented languages need rapid access to accurate documentation about the software components they intend to use. Typical object oriented environments support a *browser* which provides views of the class hierarchy and source code. The browser is usually syntactic, deriving information from the source text only. The use of inheritance makes adequate browsing capabilities important to the programmer. At the same time, the presence of inheritance complicates the design of environments which support object oriented languages.

Object oriented languages have supported browsing since their beginning. For example, the Smalltalk–80 programming environment allows programmers not only to inspect the class hierarchy, the classes themselves, and individual methods, but also to ask questions such as "What messages does this method send?" [4]. Since Smalltalk is interpreted, however, browsing reveals only what the programmer wrote: the browser does not provide any semantic information or any indication of validity.

Interactive browsing is more commonly associated with dynamic systems, which interpret or compile incrementally and do not perform static type checking, than it is with systems which perform semantic analysis during compilation. Development environments for typed object oriented languages, such as C++ [3] and Eiffel [10], usually appear after the language itself has been implemented rather than being developed as an integral part of the language design. In fact, many programmers are required to develop object oriented programs with no tools other than an editor, compiler, and debugger. Meyers reports that debugging an object oriented program under these circumstances can be difficult and frustrating because the information provided by the debugger is often not relevant to the problem [11].

In comparison, our approach has three major advantages. First, the information that the browser displays is correct and up-to-date. Moreover, the browser has access to information obtained during semantic analysis. Second, programmers do not usually need to be aware of the inheritance hierarchy because the compiler processes inheritance information. Third, the origin of a query can control the level of detail provided in the response. Section 4 describes these advantages in detail.

## 2 The Language

Dee is a strongly-typed, class-based, object oriented language which provides multiple inheritance for both protocol and implementation.

The source text of a Dee program consists of a number of class definitions. Each class is defined by a single document called the *canonical document* of the class. The canonical document is read and modified only by the *owner* of the class, who may be a programmer or a team of programmers.

All the information about a class is contained in the *canonical document* of the class. A class is not visible to anyone other than its owner until it has been compiled. The compiler, in addition to semantic checking and code generation, constructs an interface for the class and writes it to a database. Subsequent access to the class, by both programmers and the compiler, is made through this interface. During semantic analysis, the compiler reads the interfaces of other classes from this database.

The canonical document of a class defines each attribute of the class. An *attribute* is either an *instance variable* or a *method*. Each attribute is either *public* or *private*. Clients of a class may use its public attributes but not its private attributes. Heirs of a class may use both its public and its private attributes. For example, a client of class *Part*, shown in Figure 2, could use *desc*, *satisfies*, and *show*, but not *cost*. The value of a public variable, such as *desc*, can be accessed, but not altered, by a client. The canonical document also specifies the classes which the defined class needs. The classes which the defined class inherits are listed explicitly. Other needs are indicated by declaration: for example, the declaration *desc*:*String* indicates that the defined class is a client class of *String*, the class of strings.

The canonical document may contain comments. Programmers cannot write comments in arbitrary places because a comment is a terminal symbol in the grammar of Dee. For example, there may be a comment between two statements but not within a simple statement. In practice, the restrictions on the placement of comments are a minor inconvenience for programmers. The compiler writes selected comments to the class interface. Specification comments, which answer the question "What does it do?" are distinguished syntactically from implementation comments, which answer the question "How does it work?" A programmer who enquires about a method will see only its signature and specification documentation.

## 3 The Environment

The development environment maintains a *class interface database*. A class interface contains, in encoded form, some of the text of the canonical document and some information added by the compiler. It does not contain the code for methods or the implementation comments. The compiler writes the interface of a class to the database after successfully completing the semantic analysis of that class. This is the only way in which the database is ever updated.

Class interfaces are managed by the *class interface manager* (CIM). The CIM provides information in response to queries. The compiler issues queries to obtain information about classes other than the class it is compiling. Programmers issue queries, *via* the browser, to learn about classes for which they are writing clients or descendants. The information provided by the CIM is determined by class relationships: a descendant class has access to more information than a client class.

Figure 1 shows the relationships between the major components of the environment. The compiler,

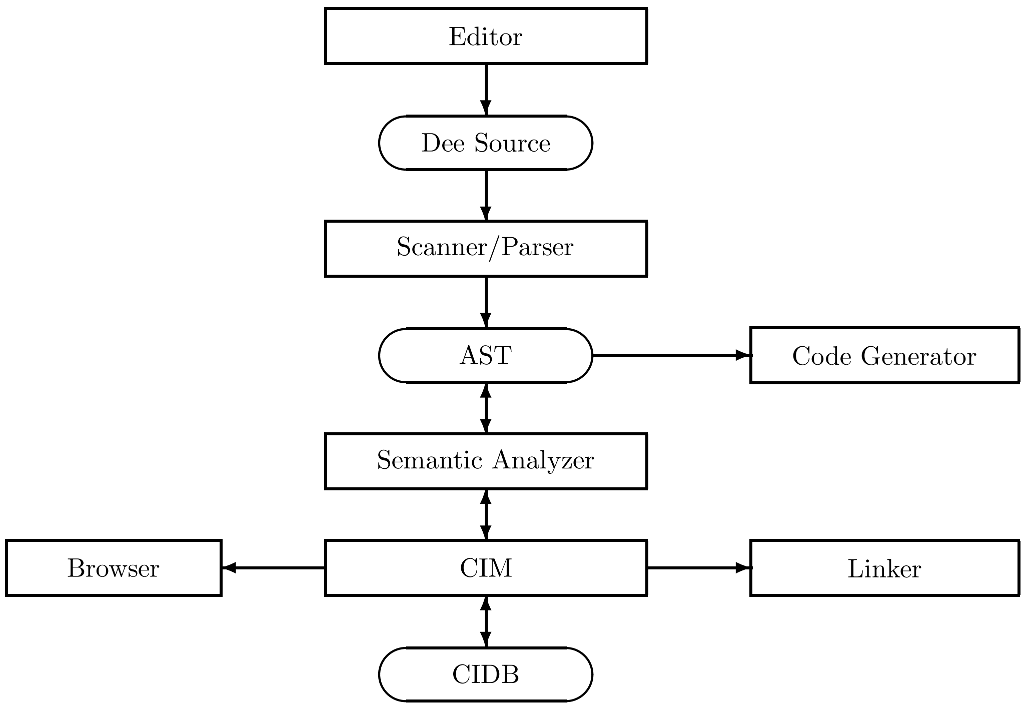

Figure 1: Dee System Organization

consisting of a scanner/parser, a semantic analyzer, and a code generator, is an integral part of the system, not an isolated component. This is the key to our approach, since it allows all components of the environment to benefit from semantic information derived during compilation. The abstract syntax tree, denoted by AST in Figure 1, is an intermediate data structure created by the scanner/parser and used by the semantic analyser and the code generator. Class interface information is stored in CIDB, the class interface database. The diagram shows that all access to class interfaces is mediated by the CIM and that the class interfaces contain only information that has been semantically checked.

Figure 2 shows the canonical document for an abstract class called *Part*. The document has been simplified for the purpose of illustration; typical documents would be much longer than Figure 2. A part has a description, a cost, and two methods. The method *satisfies* is abstract because it has no implementation in this class. The method *show* has an implementation and is therefore concrete. Since there is an abstract method, the class as a whole is abstract and can have no instances.

The class *Hook*, shown in Figure 3, inherits from *Part* and provides an implementation for *satisfies*. A programmer who wanted to inherit from *Hook* would see the view shown in Figure 5. The view contains the attributes inherited from *Part* as well as the attributes declared in *Hook* itself. The programmer would see the specification comment "Return true if the hook size satisfies the constraint" for the method *satisfies* but not its implementation comment "Use integer comparison".

The compiler, compiling a client of *Hook*, would see an encoded form of the information shown in Figure 4. The attribute *cost*, declared `private` in *Part*, is not visible to a client of *Hook* and is therefore not included in the interface.

Programmers can ask for the client view or the descendant view of a class. They see the signatures of public variables and public methods together with selected comments. A view contains information that does not explicitly appear in the canonical document, such as a list of the suppliers of the class.

```
class Part
    -- An abstract class describing an inventory part
    -- which satisfies a constraint.
inherits Any
public var desc:String
    -- Description of the part.
private var cost:Float
    -- Cost of the part.
public method satisfies (c:Int):Bool
    -- Return true if this part satisfies the constraint.
public method show:String
    -- Return a string corresponding to the part.
    begin
        result := desc + " " + cost.show
    end
```

Figure 2: Canonical Document for Class *Part*

```
class Hook
inherits Part
var size:Int
public cons make_hook (hook_cost:Float; hook_size:Int)
    -- Construct a hook with given cost and size.
    begin
        cost := hook_cost
        size := hook_size
        desc := "hook"
    end
public method satisfies (hook_min_size:Int):Bool
    -- Return true if the hook size satisfies the constraint.
    begin
        -- Use integer comparison.
        result := size ≥ hook_min_size
    end
```

Figure 3: Canonical Document for Class *Hook*

```
class Hook
ancestors Any Part
var desc:String
var size:Int
cons make_hook (hook_cost:Float; hook_size:Int)
method show:String
method satisfies (hook_min_size:Int):Bool
```

Figure 4: Client Interface of Class *Hook*

```
class Hook
inherits Part
uses Int String Float Bool
ancestors Any Part
desc:String
    -- Part descriptor
cost:Float
    -- Cost of a part
size:Int
show:String
cons make_hook (hook_cost:Float; hook_size:Int)
    -- Construct a hook with given cost and size.
satisfies (hook_min_size:Int):Bool
    -- Return true if the hook size satisfies the constraint.
```

Figure 5: Inherited View of Class *Hook*

## 4 Advantages of the Design

The design has advantages for both the implementors of the development environment and the programmers who use it. For the implementors, symbol table management within the compiler is simplified. When the compiler requires information for type checking, it issues a query to the CIM.

One of the design principles of Dee requires that programmers should not have to provide the same information more than once. Another important principle is that all of the information about a particular entity should be in one place [5]. The canonical document and views support these principles: programmers are not required to write separate interface and implementation modules. In the canonical document, the definition of an instance variable or method consists of a single block of text; there are no export lists.

The environment can determine whether a change to a canonical document changes the interface of the corresponding class. It can therefore decide how much recompilation is necessary and may even make fine distinctions, such as to recompile descendants but not clients.

Programmers do not need to know the ancestry of a class except when they are actually involved in designing or modifying a class hierarchy. As Figure 5 shows, a class interface is presented with all of its accessible attributes, and programmers do not usually need to know the origins of these attributes. For some applications, it is useful to know the inheritance hierarchy, and for these purposes the browser can answer queries such as "which classes implement the method *satisfies*?"

The CIM maintains precise control over the content of a view. In particular, it can decide whether the information is needed for a client or for a descendant, and it reveals attributes accordingly. It may also reveal information to the compiler but not to programmers, for example the fact that a method is to be compiled in-line.

Programmers can trust information they obtain from the browser because the original source of the information is the compiler. Information provided by the CIM is correct, consistent, and up to date. The CIM provides access to a rich repository of useful information about existing classes. A browser need do no more than access this information in response to appropriate queries.

As class libraries evolve, they tend to deepen: designers notice new abstractions and introduce new levels into the inheritance hierarchy. The resulting classification of capabilities is useful to programmers because they can select classes which are closely suited to their requirements. Once they have chosen appropriate classes, however, the hierarchy is a distraction. Programmers need to know the complete specification of a class, not its pedigree. Class views contain inherited information; it is only rarely necessary for a programmer to study or understand the inheritance hierarchy.

The Dee environment can support distributed development by maintaining the source code as private files and the database as a shared resource. In a natural way, programmers retain full control over their own classes while benefitting from classes developed by others.

Multiple versions of both individual classes and complete programs must be supported. With our approach, version control requires that the CIM maintain multiple versions and provide access to them as required by the compiler and by programmers.

## 5 Implementation Issues

There are currently three implementations of the Dee compiler and environment. $\text{Dee}_1$ was written in Turbo-Pascal for the IBM PC and compatibles by Grogono. $\text{Dee}_2$ is a revised version of $\text{Dee}_1$ designed and implemented by Cheung. With other graduate students, we are currently completing $\text{Dee}_3$, a version which runs on Unix workstations. In all three versions, the abstract syntax tree of an entire class is stored in memory during compilation. Individual classes are typically small, consisting of no more than a few hundred lines of code, and the AST is usually less than 200K bytes. Even the PC version, with 640K bytes, can easily store the abstract syntax tree of the largest class we have written to date.

### 5.1 Efficiency

The $\text{Dee}_1$ compiler writes class interfaces as small, separate files. Although the compiler must open and close many files while compiling a class, the speed of compilation is acceptable. $\text{Dee}_2$ stores class interfaces in a B-tree indexed by class name and attribute name. Access to class interfaces is considerably faster with this organization, but the overall performance of the compiler is little better than with $\text{Dee}_1$. There are two reasons for this. First, $\text{Dee}_2$ was obtained by modifying $\text{Dee}_1$ rather than by starting afresh: efficiency is lost where new code interacts with old code. Second and more significantly, after compiling a class, the compiler rebuilds the B-tree index, which takes longer than we anticipated.

$\text{Dee}_3$ represents an attempt to combine the best features of $\text{Dee}_1$ and $\text{Dee}_2$. The database is again an indexed B-tree but the CIM is tightly coupled to the needs of the compiler so that performance is not sacrificed.

### 5.2 Circular Dependencies

Although the Dee compiler does not allow cycles in the inheritance graph, there may be cyclic dependencies in the client-supplier relation. For example, the class *Bool* has a method *show*, which returns a string representation of the receiver (`"true"` or `"false"`). The class *String* has a method "=" which returns an instance of *Bool*. According to the account given above, neither *Bool* nor *Int* can be compiled first, because each requires the interface of the other.

Circular dependencies are not a problem for languages which require programmers to write separate definition modules ( "header files") and implementation modules ("code files"). In these languages, the compiler can compile either implementation module under the assumption that the declarations in the definition module for the other class are correct. Our requirement that all information about a class be in a single document prevents us from using this method.

In this situation, the programmer must assist the compiler by using an option called *Force Interface.* To compile *Bool* and *String*, the following steps are performed.

1. Set *Force Interface.*
2. Compile *Bool*. The compiler will report errors because `Bool` refers to the unknown class *String*, but it will write an interface for *Bool* to the CIM anyway. The interface will be flagged "insecure".
3. Clear *Force Interface.*
4. Compile *String*. The compilation will succeed because the interface for *Bool* is present in the CIM.
5. Compile *Bool* again. The compilation will succeed because the interface for *String* is present in the CIM. The interface for *Bool* will be rewritten without the "insecure" flag.

We could have adopted an approach in which the compiler performed these steps automatically. We decided against automatic compilation because the problem of circular dependencies does not arise frequently and we did not want the compiler to surprise programmers by performing unexpected compilations.

## 6 Related Work

Database management systems have long been associated with software engineering [9, 13]. The databases described by these authors, however, are primarily concerned with project management. The CIM in Dee contributes to project management, but its primary purpose is to store specific information needed by the compiler and by programmers.

The OMEGA programming system uses a relational database to provide various views of a program, including configurations, versions, call graphs, and slices [8]. The approach is similar to ours but, as in Smalltalk [4] and Eiffel [10], the stored information is obtained from program text rather than from the compiler.

Trellis is a programming environment which supports programming in Trellis/Owl [12, 6]. The Trellis browser provides access to source code. Realizing that source code alone is inadequate for object oriented development, the designers of Trellis provided additional information to programmers, including information generated by the compiler. In this respect, Trellis is quite similar to Dee.

Shilling and Sweeney propose that a class should be able to expose different interfaces to different clients [14]. Dee effectively provides four different interfaces. Which interface is used depends on the viewer, who may be the compiler or a programmer, and on the class relation, which may be descendant or client.

Programmers need ways of discovering classes which might be useful to them. Class names and method names are inadequate for this purpose. Comments are the most useful source of information. Programmers need formal specifications to make a final choice, but helpful comments get them onto the right track. A tool which searches for comments containing given patterns would assist in discovery. Li and O'Shea, for example, report that navigation in Smalltalk can be difficult, and they have implemented a tool called BRRR (Browser for Retrieval by Reformulation) which helps programmers to find their way around the Smalltalk–80 class library [7]. Although we have not yet implemented anything along these lines, we feel that such a tool would fit very naturally into the environment.

## 7 Conclusion

The benefits of concurrently designing a language and its development environment have exceeded our expectations. The unified design has yielded simplifications in the compiler, in the environment, and in the user interface.

The most important single feature in the design of the Dee environment is the class interface manager. The central role of the CIM reflects the importance of class interfaces in software design and implementation. The CIM provides efficient access to the information needed by the compiler and the programmer. The integrity of the information is ensured by the compiler.

Our experience with Dee, based on a number of application programs and a library of about 100 classes, has been positive. The browser has proved to be a useful tool for programmers, providing all of the information that they really need to use a class without revealing details of its implementation.

The CIM in the Dee environment provides solutions to the first two of the problems identified in Section 1: it helps programmers to find the code they need and to understand it. In future work, we will address the other two problems, modifying and creating components.

## Acknowledgements

The Dee language and development environment has benefitted from our discussions with many people, including Patrice Chalin, Mitch Cherniack, Lawrence Hegarty, and Wai Ming Wong. Trellis is a registered trade mark of Digital Equipment Corporation. Unix is a registered trade mark of AT&T Bell Laboratories. The research was funded in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

## References

[1] T. Biggerstaff and A. Perlis, editors. *Software Reusability. Volume II: Applications and Experience.* ACM Press (Addison Wesley), 1989.

[2] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. In T. Biggerstaff and A. Perlis, editors, *Software Reusability. Volume I: Concepts and Models*, pages 1–17. ACM Press (Addison Wesley), 1989.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[4] A. Goldberg. The influence of an object-oriented language on the programming environment. In D. Barstow, H. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, chapter 8, pages 141–174. McGraw-Hill, 1984.

[5] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.

[6] M. Kilian. Trellis: Turning designs into programs. *Comm. ACM*, 33(9):65–67, September 1990.

[7] Y. Li and T. O'Shea. BRRR: a tool for facilitating user's navigation in Smalltalk–80. In *Proc.*

*Symp. on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 175–189, September 1990.

[8] M. Linton. Implementing relational views of programs. In P. Henderson, editor, *Proc. ACM Software Engineering Symp. on Practical Software Development Environments*, pages 132–140. ACM, April 1984. Published as *SIGPLAN Notices*, 19(5), May 1984.

[9] L.-C. Liu and E. Horowitz. Object database support for a software project management environment. In P. Henderson, editor, *Proc. ACM Software Engineering Symp. on Practical Software Development Environments*, pages 85–96. ACM, December 1988. Also published in *SIGPLAN Notices*, 24(2), February 1989.

[10] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988. Second Edition, 1997.

[11] S. Meyers. Working with object-oriented programs: the view from the trenches is not always pretty. In *Proc. Symp. on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 51–65, September 1990.

[12] P. O'Brien, D. Halbert, and M. Kilian. The Trellis programming environment. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 91–102, 1987.

[13] M. Penedo. Prototyping a project master data base for software engineering environments. In P. Henderson, editor, *Proc. ACM Software Engineering Symp. on Practical Software Development Environments*, pages 1–11. ACM, December 1986. Also published in *SIGPLAN Notices*, 22(1), January 1987.

[14] J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 353–361, October 1989.