A Tool for Understanding Object-Oriented Program Dependencies

Panagiotis K. Linos
Tennessee Technological University
Computer Science Department
Cookeville, TN 38505, USA
email : PKL3678@tntech.edu

Vincent Courtois
Hautes Etudes Industrielles
13 Rue de Toul
59046, Lille
France

Abstract

In this paper, we present a tool for understanding and re-engineering C++ programs called OO!CARE (Object-Oriented Computer-Aided Re-Engineering). OO!CARE demonstrates some practical solutions to the problem of extracting and visualizing object-oriented program dependencies (i.e. data-objects and their relationships). It is an extension of an earlier tool for maintaining C programs called CARE (Computer-Aided Re-engineering). In this paper, we also discuss some early experiences acquired from using the tool. For instance, an important observation made during a re-engineering exercise is that some characteristics of the object-oriented programming paradigm such as *inheritance* and *polymorphism* contribute significantly to the complexity of understanding program dependencies. Moreover, in this paper, we discuss how object-oriented program dependencies differ from the procedural ones and explain how they can be visualized within the same environment.

1. Introduction

Program dependencies include information regarding the relationships between various components in computer programs such as the interaction between modules (i.e. files), the use of variables and their types, as well as the calls among functions. Today, there are many software environments which facilitate the comprehension of programs written in procedural programming languages by systematically extracting a large number of program dependencies contained in the source code. These dependencies are stored in a database and then visualized in various graphical representations. Taxonomies of software visualization environments and their characteristics can be found in Price [8] and Stasko [12]. These environments have been reasonably successful in helping to reduce the time and effort spent to understand existing programs written in procedural programming languages. However, a new problem arises from the fact that object-oriented languages are quickly emerging from the procedural ones. Although, the new features introduced by object-oriented languages offer flexibility some complications during program understanding can arise [16]. In particular, the use of *classes* and *inheritance* often leads to a plethora of small program components (e.g. *objects*) with many relationships (e.g. *message passing*) [14]. Consequently, locating and understanding object-oriented program dependencies becomes a difficult problem. Today, there are several commercial and academic tools available for understanding and re-engineering object-oriented programs. Examples of commercial tools include the *ObjectCenter* by Centerline Software Inc. [1], the *ObjectWork* by Parcplace Systems and the *Objective C* from the Stepstone Corporation. In addition, several research prototype tools for maintaining object-oriented programs are available today. A software environment described in Lejter [3] entails a relational database with an interactive interface which supports queries about programs written in object-oriented programming languages. Another software tool provides browsing features through the source code of object-oriented programs using hypertext techniques [10]. Finally, some visualization mechanisms for maintaining object-oriented software are described in DePauw [15] which are based on a language-independent approach.

Many of the above tools utilize graphical representations which become difficult to understand when medium-to-large programs are displayed. In addition, they support limited abstraction mechanisms and transformation tools for facilitating the re-engineering of object-oriented programs. In this paper, we address the above issues by developing a software environment for understanding and re-engineering C++ programs called OO!CARE (Object-Oriented Computer-Aided Re-Engineering). This effort focuses on the dynamic behavior of object-oriented programs and it supports the general hypothesis that visualization of program dependencies is most effective for program comprehension [6]. The OO!CARE environment evolved from an earlier tool for maintaining C programs called CARE (Computer-Aided Re-engineering) [5]. The new environment facilitates the understanding of existing C programs as well as the difficulties introduced by the C++ language. The rest of this paper is organized as follows : the second section explains how object-oriented program dependencies differ from the procedural ones. Then, a description of the OO!CARE environment is given in the third section. The fourth section presents a brief history of the OO!CARE project and finally in the fifth section we present our conclusions.

2. Object-Oriented Program Dependencies

Programs written using a procedural programming language (e.g. Pascal, C) consist of various components which embody *data-elements*, *data-types* and *sub-programs*. Examples of *data-elements* in procedural languages include variables,

0-8186-5647-6/94 $04.00 © 1994 IEEE

constants and parameters of the program. *Data-types* can be standard or user-defined and sub-programs represent functions, procedures or modules (i.e. a group of functions or procedures). These components are linked via various relationships such as *calls* between functions, the *use* of parameters by a procedure or the *definition* of a variable as of a data-type. We call these components and their relationships *Procedural Program Dependencies* (PPDs). More formally, a PPD can be represented as a triplet PPD=<X, Y, R> where X and Y can be *data-elements, data-types* or *sub-programs* and R depicts a relationship between X and Y. For example, the triplet <Variable, Type, *is-defined-as*> presents the *is-defined-as* relationship between variables and data-*types*. An instance of this program dependency is <counter, integer, is-defined-as> meaning that a variable called *counter* is defined as an *integer* data-type in the program.

On the other hand, programs written in an object-oriented language entail different kinds of components and relationships due to the different programming paradigm supported. Today, there are two families of object-oriented languages; the *pure* object-oriented ones where all computation is based on *message passing* (e.g. Smalltalk, Eiffel) and the *hybrid* object-oriented languages which have evolved from the procedural ones (e.g. C++) [11]. They usually include a mixture of features from both families of languages. Programs written in a pure object-oriented language consist of *data-objects, class-types* and *methods*. We call these entities and their relationships *Object-Oriented Program Dependencies* (OOPDs). Similarly, an OOPD can be represented by a triplet OOPD=<X, Y, R> where the entities X and Y can be *data-objects, class-types* or *methods* and R represents a relationship between X and Y. For example, the triplet <Class, Method, *implements* > defines the *implements* relationship between *Classes* and *Methods* . An instance of this relationship is <Shape, draw, *implements*> meaning that the class *Shape* implements (i.e. defines within its body) a method for this class called *draw*. Finally, programs written using a *hybrid* object-oriented programming language entail a combination of both procedural and object-oriented program dependencies. An example of such dependency is represented by the triplet <Class::Method, Function, *calls* > meaning that class methods *call* user-defined functions. An instance of this dependency is <Shape::draw, PrintLabel, *calls*> indicating that the *draw* method of the class *Shape* calls a regular function called *PrintLabel*.

In this work, we have selected C++ as a hybrid object-oriented programming language in order to study how object-oriented program dependencies can be visualized. C++ supports the object-oriented programming paradigm while maintaining the procedural features of the C language. In the object-oriented programming paradigm data and behavior of a program are strongly connected. C++ implements this concept by the use of *classes* whose instances are *objects* [2]. A *class* consists of a set of values (*data members*) and a collection of operations (*methods* or *member functions*) that can act on those values. For example, we can create a class called *GeometricFigure* where the data members are the *center* and *perimeter* and where methods are operations using those members such as *draw-figure*. *Inheritance* is a way of deriving a new class from existing classes called *base classes*. The derived class is developed from its parent by adding or altering code. Inheritance is said to be *single* if a class is derived from only one parent, or *multiple* if it is developed from several base classes. Moreover, access privileges to classes and their members can be managed and limited to whatever group of functions needs to access their implementation (this is accomplished in C++ by the use of the keywords *public, protected* and *private*). Also, functions and operators can be overloaded in C++. Overloaded functions or operators are also known as *polymorphic* entities because they can take many different forms (i.e. can have several distinct implementations). In particular, polymorphic functions are implemented by *virtual* member functions in C++.

A. Polymorphic Program Dependencies

The above mentioned features of object-oriented programming languages introduce some additional complexity towards extracting program dependencies from source code. In particular, the use of polymorphic entities create *dynamic* program dependencies in object-oriented programs. As we know, *static* program dependencies are extracted directly by analyzing (i.e. scanning) the source code and execution of the program is not necessary. However, *dynamic* program dependencies are only determined at run-time and they require program execution. *Message passing* to a *virtual* member function in a C++ program is an example of a *dynamic* program dependency. In order to demonstrate this dynamic behavior we present a C++ example shown in Figure 2.1. As we can see in this figure, the classes named *Box* and *Circle* are derived from the class *Shape*. Also, the class *Square* is a subclass of *Box*. Moreover, a variable P is declared as a pointer to a *Shape* or to any of its subclasses (i.e. *Box*, *Circle* and *Square*). When the program statement *P->draw()* is executed *a message* is sent to the virtual member function *draw()*. However, all four classes in the program of Figure 2.1 implement their own version of this method. Therefore, the version being called can only be determined at run-time, depending on what kind of shape P points to at the particular time of execution. This dynamic behavior is also demonstrated graphically in the same figure. When the while loop of the program in Figure 2.1 is executed for the first time the message is sent to the method *Shape::draw* because P points to a *Shape* object but on the next iteration P points to a *Box* so the message is sent to *Box::draw*. During the following loop iteration, the same message is sent to the *Circle::draw* method and then to *Square::draw* . This sequence continues until execution of the program stops. With this example, we demonstrate the fact that a statement such as *P->draw()* can have several different meanings (i.e. *polymorphic* entity) and can be

determined only at run-time. We call such dependencies *Polymorphic Program Dependencies*.

B. Implicit Program Dependencies

In addition, program dependencies can be *implicit* or *explicit*. Explicit program dependencies appear in the program and can be directly extracted from the source code. However, implicit program dependencies do not arise explicitly in the source code and thus some additional complexity is introduced for extracting them.

```
#include <iostream>
class Shape : { virtual void draw(void)
    { cout << "drawing a Shape"; } };
class Box : public Shape { virtual void draw(void)
    { cout << "drawing a Box"; } };
class Circle : public Shape { virtual void draw(void)
    { cout << "drawing a Circle"; } };
class Square : public Box { virtual void draw(void)
    { cout << "drawing a Square"; } };
main () {
Shape * P;  int i=0; int limit=10;
Shape Sh; Box Bo; Circle Ci; Square Sq;
while (i<limit)
{
   switch (i%4) {
   case 0: P=Sh;
      break;
   case 1: P=Bo;
      break;
   case 2: P=Ci;
      break;
   case 3: P=Sq;
      break;
   }
   P->draw();
   i++;
}
}
```

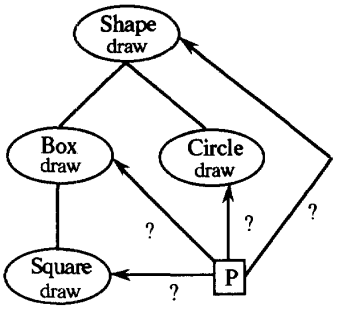

Figure 2.1 : A C++ program demonstrating dynamic program dependencies

In order to demonstrate some caveats regarding implicit program dependencies we consider a small C++ example shown in Figure 2.2. The output (with some explanations) generated by this program is given in Figure 2.3. Although only four objects (i.e. *ashape, abox, somesquare* and *othersquare*) are created in the *main()* function of the program as shown in Figure 2.2, nine *constructors* (i.e. member functions used to create objects) are executed (see their output in Figure 2.3). This happens because in the program, a *square* is defined to be a special case of *box* and a *box* a special case of *shape*. Thus, in order for the program to create a *square* object, it first builds a *shape* object, then constructs a *box* object of this shape, and finally turns this *box* into a *square*. This results in the execution of additional statements which do not appear explicitly in the source code. Moreover, in this example, there is no explicit call to *destructors* (i.e. member functions which deallocate space for an object). However, several destructors are invoked implicitly in the program for the four objects created (see their output in Figure 2.3). Notice that the program sends a message to three different destructors in order to destroy a *square*.

```
// Example of implicit operations performed by C+
#include <iostream.h>
class shape {
  private: double area;
  public :  shape(void)
  { cout << "I am a shape " << endl; }
  ~shape(void)
  { cout << "I am the shape destructor" << endl;}
};
class box : public shape {
  private: double l,L;
  public : box(void)
   { cout << ", i am a box " << endl; }
   ~box(void)
   { cout << "I am the box destructor "<< endl;}
};
class square : public box {
  private: double side;
  public : square(void)
   { cout << ", i am a square " << endl; }
   ~square(void)
   { cout << "I am the square destructor" << endl;}
};
main() {
shape ashape;
box abox;
square somesquare, othersquare;
somesquare = othersquare;
}
```

Figure 2.2 : C++ source code demonstrating implicit program dependencies

In addition, C++ provides default constructors and destructors (i.e. none defined by the programmer). Consequently, another version of the program in Figure 2.2 is shown in Figure 2.4. The two programs are functionally equivalent however, the program of Figure 2.4 depicts a higher abstraction of implicit program dependencies (i.e. constructors and destructors are not defined explicitly in the source code) than the program of Figure 2.2. Evidently, the above mentioned features of object-oriented programs can complicate the task of understanding program dependencies. In this paper, we focus on the above issues with respect to extracting object-oriented program dependencies from C++ programs. Also, we explore ways to efficiently visualize such dependencies in order to facilitate the comprehension of object-oriented programs. To this end, we have designed

and implemented a software tool for understanding and re-engineering C++ programs. An overview of its main features is given in the next section.

OUTPUT	PRODUCED BY
I am a shape	constructor of ashape
I am a shape	constructor of abox
, i am a box	constructor of abox
I am a shape	constructor of somesquare
, i am a box	constructor of somesquare
, i am a square	constructor of somesquare
I am a shape	constructor of othersquare
, i am a box	constructor of othersquare
, i am a square	constructor of othersquare
I am the square destructor	destructor of othersquare
I am the box destructor	destructor of othersquare
I am the shape destructor	destructor of othersquare
I am the square destructor	destructor of somesquare
I am the box destructor	destructor of somesquare
I am the shape destructor	destructor of somesquare
I am the box destructor	destructor of abox
I am the shape destructor	destructor of abox
I am the shape destructor	destructor of ashape

Figure 2.3: Output produced by the C++ program of Figure 2.2

in Figure 3.2. The *type* node in the figure represents C++ standard or user defined data types. The functions of the program are depicted by the *function* node. The *constant* and *variable* nodes in the graph represent constants and variables respectively. Moreover, the *object* node represents data-objects (i.e. instances of a class).

```
// A modified version of the C++ program
// shown in Figure 2.2

#include <iostream.h>
// constructors and destructors are not defined explicitly
// in the source code but provided by C++
class shape : { double area; };
class box : public shape { double l,L; };
class square : public box { double side; };
main() {
shape ashape;
box abox;
square somesquare, othersquare;
somesquare = othersquare;
}
```

Figure 2.4 : A C++ program with highly abstracted implicit program dependencies

3. The OO!CARE environment

This section explains how OO!CARE (Object-Oriented Computer-Aided Re-Engineering) evolved from the CARE (Computer-Aided Re-Engineering) tool. Then, it presents the architecture of OO!CARE and the data model used to construct its database. Finally, it describes the presentation model used to display data-flow and control-flow information graphically.

The OO!CARE environment evolved from an existing software tool called CARE which facilitates the comprehension and re-engineering of existing C programs [5]. The OO!CARE tool extends the existing features of CARE in order to facilitate the comprehension of C++ programs. Specifically, the original functionality and user-interface of CARE are extended in order to extract and visualize object-oriented program dependencies. The user-interface in OO!CARE is done through specially designed windows including the main panel as well as graphical and textual windows. Each window in OO!CARE is equipped with a group of typical operations for manipulating graphs or text (e.g. hide, highlight, zoom, code etc.). The architecture of OO!CARE is comprised of the *code analyzer*, the *dependencies database* and the *display manager*. The code analyzer extracts program dependencies from C++ source code and populates them into the database. In OO!CARE, eight kinds of program components are extracted from C++ code namely *data-types*, *user-defined functions*, *constants*, *variables*, *parameters*, *objects*, *classes* and *member functions* (*methods*). These components and their relations are populated in the database based on the data model shown

Information passed to a function or a method is depicted by the *parameter* node which can be defined as a class or as a type. A *class* encapsulates data with member functions. A *member function* is a function declared within a class and it can only be invoked by instances of this class. The relationships between these components are also shown in Figure 3.2 by connecting arrows. For example, the connecting arrow between *class* and *member function* means that a *class* implements (defines) a *member function* within its body. The program dependencies included in the database can be displayed using a presentation model designed for the OO!CARE environment. This model includes hierarchical displays for presenting class inheritance, control-flow program dependencies and file dependencies. In addition, an original display called *colonnade* (i.e. a sequence of columns displayed at regular intervals) is utilized in order to present data-flow program dependencies graphically. Such graphical displays are created and manipulated by the use of several graphical editors in OO!CARE. These include the *data-flow*, *control-flow*, *inheritance-hierarchy* and *file-dependencies* editors. Textual information (i.e. code) can also be manipulated in the OO!CARE environment using a traditional text editor (e.g. emacs, vi). The file dependencies editor presents the *include* relationships between files using a hierarchical display. Each file is represented by a node and included files are linked via connecting lines. The control-flow graph editor displays user-defined and member functions using different graphical notations. Table I includes a list of C++ program function types supported by the editor and their graphical notation. Function calls or message passing is denoted by connecting lines between

functions and methods (i.e. member functions). Specifically, a connecting line between two functions (or between a method and a function) depicts a call relationship. A connecting line between two methods (or a function and a method) represents a message being passed. Moreover, information regarding polymorphic program dependencies is also displayed in the control-flow graph.

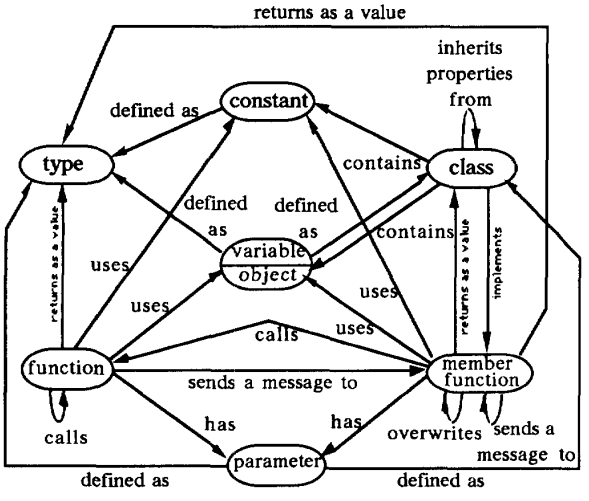

Figure 3.2 : The data model used in OO!CARE

For example, in the case of a message passed to a virtual member function (i.e. known at run time) the parser determines all possible entities where a message could be sent. In this case, OO!CARE creates a dummy member function of an unknown class (displayed as ?::member_function) which is connected to all possible member functions with the same name. This approach is illustrated by an example in Figure 3.5. In this example, a message to a polymorphic member function called *draw* is sent. The function *draw* is implemented differently within the classes *shape*, *box*, *circle* and *square*. OO!CARE displays a dummy node called *?::draw* which is linked to all the member functions named *draw* via connecting lines. An example of the control-flow graph of a C++ program is displayed by OO!CARE as shown in Figure 3.6. This example contains four user-defined functions namely *main*, *getname*, *getid* and *getsal* (all displayed by an oval shape according to Table I) and three virtual member functions called *student::print_virtual*, *teacher::print_virtual* and *person::print_virtual*. Moreover, we can see in Figure 3.6 that the *main* function of this program sends a message to a polymorphic function called *print_virtual* (all possible paths of this message are shown in Figure 3.6). In addition, the *main* driver function sends a message to three different types of constructors namely *student::student*, *person::person* and *teacher::teacher* and to their corresponding destructors namely *student::~student*, *person::~person* and *teacher::~teacher*. The above constructors call the user-defined functions *getid*, *getname* and *getsal* respectively. Finally, each of the three virtual member functions *student::print_virtual*, *person::print_virtual* and *teacher::print_virtual* sends a message to a *print* member function to their respective class.

NOTATION	MEANING
	User-defined function
	Public member function
	Protected member function
	Private member function
	Virtual member function

Table I : Shapes used in the control-flow graph and their meaning

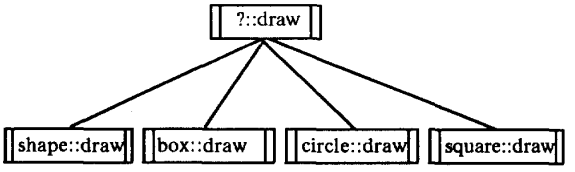

Figure 3.5 : Visualization of a message sent to a polymorphic function called draw

The data-flow program dependencies are presented separately in OO!CARE using a graphical display called *colonnade* (i.e. a sequence of columns drawn at regular intervals). Experimental data have shown that the colonnade display appears to be a promising graphical model for visualizing data-flow program dependencies [7]. It produces crossing-free, easy-to-draw and aesthetically pleasing layouts. The colonnade includes information about *Variables* or *Objects*, *Types*, *Parameters*, *Functions*, *Methods*, *Classes* and *Constants*. The relations between entities are represented by connecting lines between columns. Figure 3.7 gives an example of a colonnade display of a C++ program produced by OO!CARE. From this figure we can see that the first column displays the variables (or objects) of the program and the second column depicts the data-types. The next column contains the parameters and the following one embodies the functions (i.e. user-defined or member functions).

Finally, the fifth column displays the classes and the last one entails the constants of the program. Among others, Figure 3.7 depicts two constructor member functions for the classes *student* and *teacher* respectively. The *main* function is also displayed in the colonnade which uses an integer variable called *i* and six objects named *aperson*, *myperson*, *ateacher*, *myteacher*, *astudent* and *mystudent*. These objects are instances of their corresponding classes.

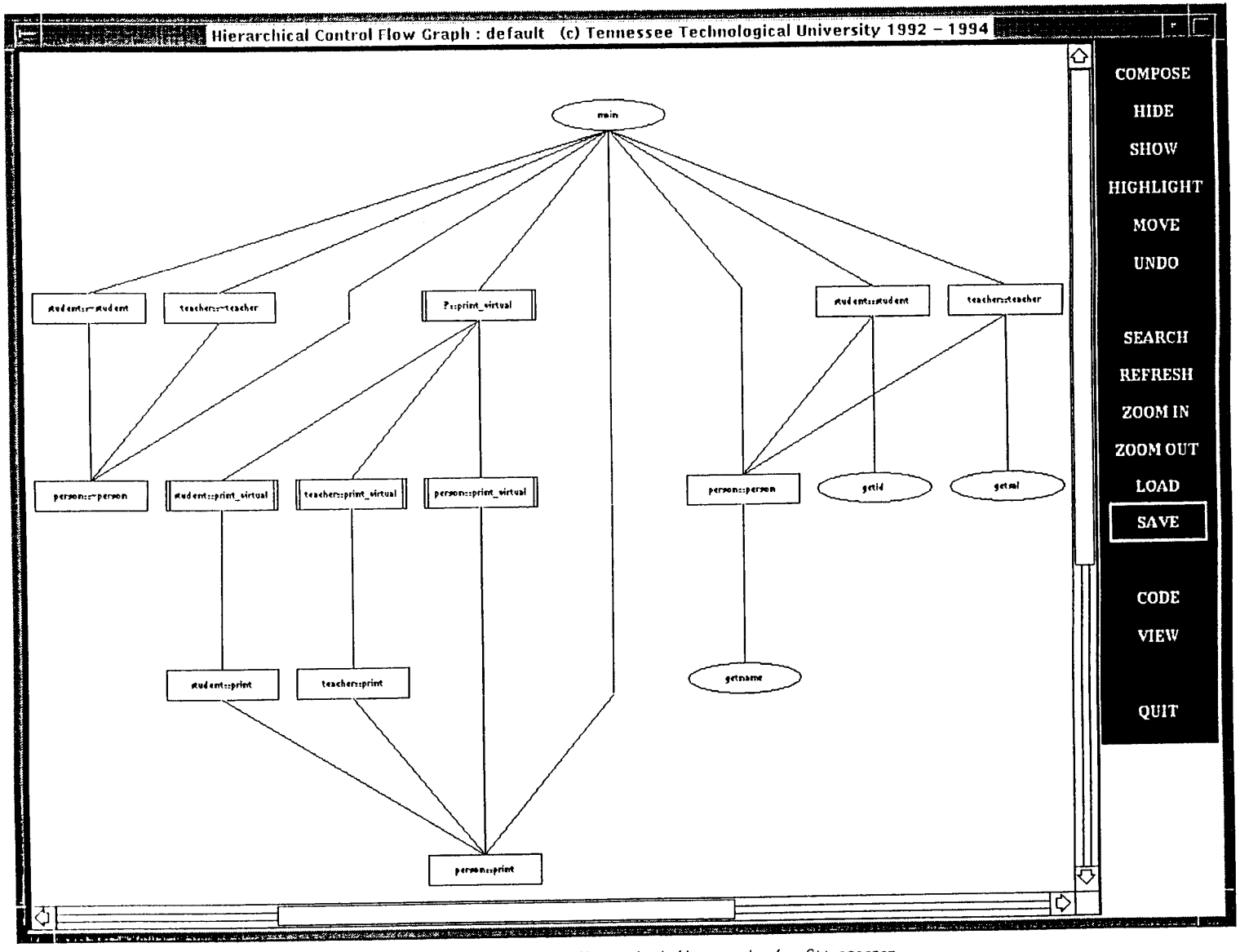

Figure 3.6: OOICARE displays the control-flow graph of a C++ program

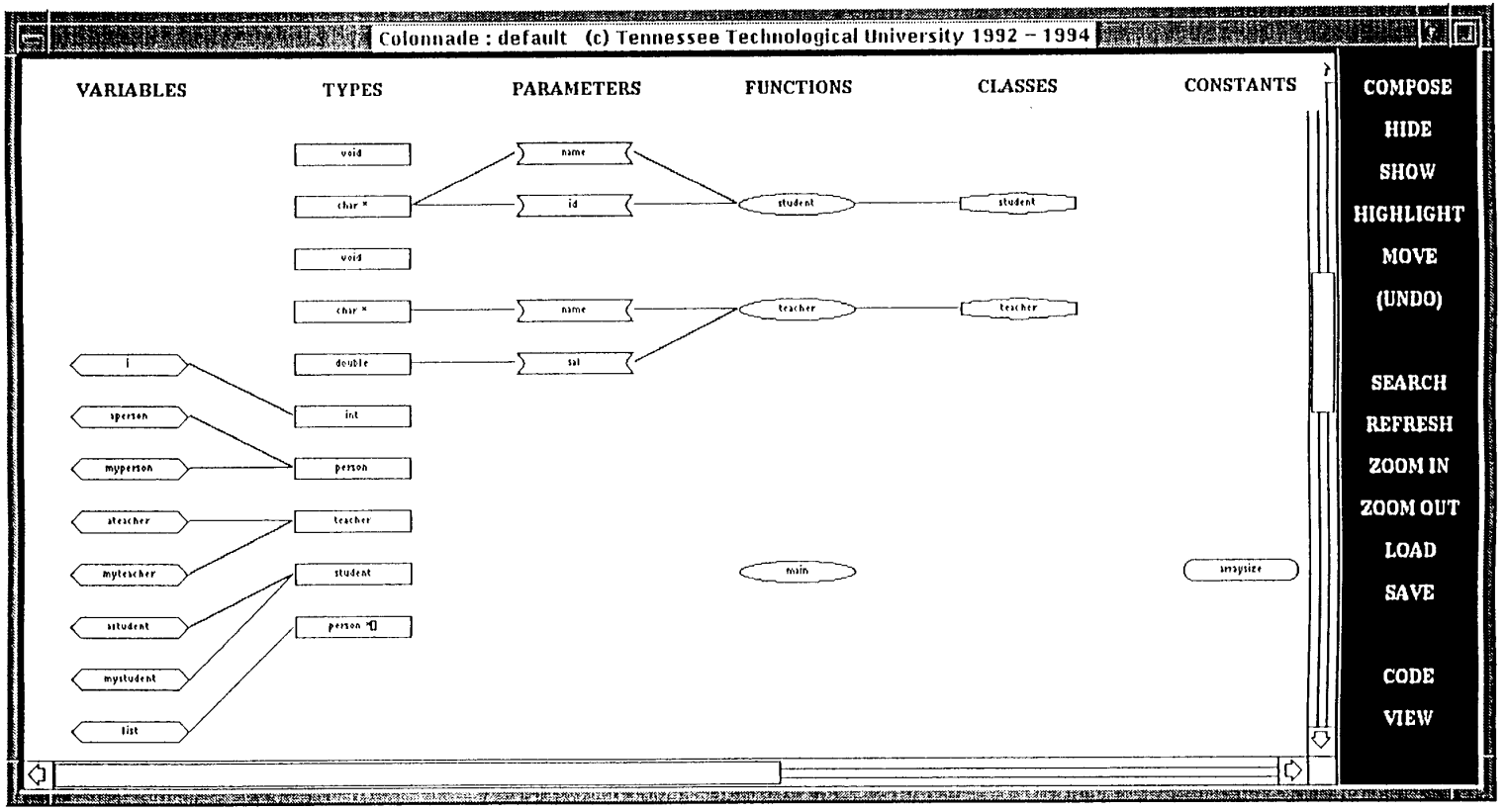

Figure 3.7: A colonnade display by OOtCARE

Also, the *main* function references a variable called *list* which is an array of pointers to a *person* object. Finally, *main* utilizes a constant called *arraysize*. If two columns are not adjacent, the relations between their entities are hidden. In order to display such relations, the colonnade allows the user to change the placement of the columns using a *move-column* operation. For instance, if the relationships between functions and types need to be visualized, the move-column option can be used to create a different layout. Finally, the inheritance graph includes all the classes defined in a C++ program and their inheritance relationships (i.e. both *single* and *multiple* inheritance).

4. History

OO!CARE (Object-Oriented Computer-Aided Re-Engineering) is a software environment for understanding and re-engineering C++ programs. It is an on-going research project since 1990 and it evolved from an earlier effort towards maintaining C programs. It is partially funded by Tennessee Technological University (under grants #9211, #9423 and #9512) and it involves a faculty member and three research students. A prototype version runs on DEC stations under the Ultrix operating system and it uses the X window manager [9]. Finally, the *lex* utility available in the Ultrix environment is also used to implement the code analyzer [4].

5. Conclusions

In this work, we have designed and implemented a software tool for understanding and re-engineering C++ programs called OO!CARE (Object-Oriented Computer-Aided Re-Engineering). During the implementation of a prototype version of OO!CARE, we made some observations with respect to understanding object-oriented program dependencies (i.e. data-objects and their relationships). First, we have observed that the unique characteristics of the object-oriented programming paradigm such as *inheritance* and *polymorphism* can increase the complexity of understanding object-oriented program dependencies. In particular, tracing the control-flow program dependencies (i.e. message passing between objects) of a medium-to-large size object-oriented program becomes a difficult task because the maintenance programmer is forced to travel through a usually large hierarchy of classes. Moreover, *polymorphic* program dependencies contribute significantly to the complexity of object-oriented programs because their values cannot be known before the execution of the program (i.e. they are known only at run-time). Finally, the use of *implicit* object-oriented program dependencies (i.e. the ones that do not appear explicitly in the source code) adds some additional complications to the task of program comprehension. Although, we do not have any experimental data to support these observations at this point, our experience during the development of OO!CARE indicates that object-oriented program dependencies appear to be complicated enough in order to make tools for program comprehension a compelling area of research and investigation. These environments need to focus on efficient ways of visualizing object-oriented program dependencies, simple abstraction mechanisms and effective transformation tools. Such features can be useful to the maintenance programmer for understanding the difficulties introduced by object-oriented programs.

References

[1] Centerline Software, *Object Center Reference*, Centerline Software Inc., Cambridge, Massachusetts, USA, 1991.

[2] Gorlen K., Orlow S., Plexico P., *Data Abstraction and Object-Oriented Programming in C++*, Wiley and Sons.

[3] Lejter M., Meyer S and Reiss S., *Support for Maintaining Object-Oriented Programs*, IEEE Transactions on Software Engineering, Vol. 18, No 12, pp. 1045-1052, December 1992.

[4] Levine J., Mason T., Brown D., *Lex & Yacc*, O'Reilly & Associates, Inc.

[5] Linos P., Aubet P., Dumas L., Helleboid Y., Lejeune P., Tulula P., *CARE : An Environment for Understanding and Re-engineering C programs*, IEEE Conference on Software Maintenance, Montreal, Canada, 1993, pp. 130-139.

[6] Linos P., Aubet P., Dumas L., Helleboid Y., Lejeune P., Tulula P., *Visualizing Program Dependencies*, Software-Practice and Experience, Vol. 24(4), April 1994, pp. 387-403 .

[7] Linos P., Aubet P., Dumas L., Helleboid Y., Lejeune P., Tulula P., *Facilitating the Comprehension of C Programs : An Experimental Study*, 2nd IEEE Workshop on Program Comprehension, Capri, Italy, 1993, pp. 55-63.

[8] Price B., Baecker R., Small I., *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing, Vol. 4, 1993, pp. 211-266.

[9] Reiss L., Rodin J., *X Window inside-out*, McGraw Hill.

[10] Sametinger J., *A Tool for Maintenance of C++ Programs*, IEEE Conference on Software Maintenance, San Diego, California, 1990, pp. 54-59.

[11] Sebesta R., *Concepts of Programming Languages*, The Benjamin/Cummings, 1993.

[12] Stasko J., Patterson C., *Understanding and Characterizing Software Visualization Systems*, IEEE Visual Languages Workshop, Seattle, Washington, September 1992, pp. 3-10.

[13] Stroustrup B., The *C++ Programming Language*, Addison-Wesley.

[14] Taenzer D., Ganti M., Podar S., *Object-Oriented Software Reuse : The Yoyo Problem*, Journal of Object-Oriented Programming 1989, pp. 30-35.

[15] Wim DePauw, Helm R., Kimelman D., Vlissides J., *Visualizing the Behavior of Object-Oriented Systems*, OOPSLA'93, pp. 326-337.

[16] Wilde N., Huitt R., *Maintenance Support for Object-Oriented Programs*, IEEE Transactions on Software Engineering, Vol. 18, No 12, December 92.