(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

USING AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM TO ENHANCE THE REUSABILITY OF CLASS DEFINITIONS

by

Tilemahos Poulis

September, 1992

Thesis Advisor: Michael L. Nelson

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
USING AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM TO ENHANCE THE REUSABILITY OF CLASS DEFINITIONS

12. PERSONAL AUTHOR(S)
Tilemahos Poulis

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 01/92 TO 09/92	1992, September, 24	119

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Object-Oriented, browser, reusability, class definition

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

One of the main benefits of object-orientation is reusability which allows modules (classes/objects) developed for a previous application to be used again for a new application. A Browser in object-oriented systems is a highly specialized system used in class libraries for viewing, writing, changing, and saving code. However, as the number of classes grows from the hundreds to the thousands to the tens of thousands, the Browser approach is not sufficient. The Class Storage and Retrieval System (CSRS) is a proposed system based on the Computer Aided Prototyping Sytem that aims to overcome the insufficiencies of the Browser approach and therefore to enhance the reusability of classes in object-oriented systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Michael L. Nelson	22b. TELEPHONE (Include Area Code): (408) 646-2026; 22c. OFFICE SYMBOL: CS/Ne

Approved for public release; distribution is unlimited.

Using an Object-Oriented Database Management System
to Enhance the Reusability
of Class Definitions

by

Tilemahos Poulis
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1983

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Author: _____
Tilemahos Poulis

Approved by: _____
Michael L. Nelson, Thesis Advisor

_____
Luqi, Second Reader

_____
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

One of the main benefits of object-orientation is reusability which allows modules (classes/objects) developed for a previous appication to be used again for a new application.

A Browser in object-oriented systems is a highly specialized system used in class libraries for viewing, writing, changing, and saving code. However, as the number of classes grows from the hundreds to the thousands to the tens of thousands, the Browser approach is not sufficient.

The Class Storage and Retrieval System (CSRS) is a proposed system based on the Computer Aided Prototyping System that aims to overcome the insufficiencies of the Browser approach and therefore to enhance the reusability of classes in object-oriented systems.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to take this opportunity to sincerely thank the Greek Navy; my thesis advisor Michael Nelson, for his guidance and assistance, to accomplish my goal.

To my wife Myriam and my newborn son Dimitrios I dedicate this thesis for the infinitely invaluable encouragement during the last nine months.

I. INTRODUCTION

A. GENERAL

The 1990's will most likely be characterized by an increasing complexity and diversity of software applications. The need for computerized solutions in almost every human activity is increasing daily. In order for software production mechanisms to meet these requirements it is necessary to achieve faster development, a quality finished product, and easier maintenance and extensibility.

Object-oriented programming (OOP) is a relatively new approach that promises to serve software development needs better than more traditional approaches. These needs can be briefly described as creating reliable, sharable, easily reusable, extensible, and maintainable modules of code. These modules of code are organized into classes which form a hierarchy.

In order to maximize reusability of classes, a storage and retrieval mechanism for their definitions is needed. With this mechanism (typically a browser), software developers are able to search previously defined classes for those that are appropriate to their needs. This mechanism needs to be as efficient and as automated as possible in order to minimize software development costs and therefore fulfill the promise of reusability in object-oriented (OO) environments.

In most object-oriented environments the browser is a tool useful in viewing, writing, and saving code, but only if the number of classes is relatively small. This is

because the developer has to manually search the class library to determine if a specific class already exists. But as the number of classes grows from the hundreds to the thousands to the tens of thousands, the browser approach is not sufficient [NB92].

The Computer Aided Prototyping System (CAPS) is a rapid prototyping environment for hard real-time systems [LK88, Luqi91]. CAPS uses an object-oriented database management system (OODBMS) for the storage and retrieval of ADA components, achieving a high degree of reusability. A system such as CAPS which could be used in the storage and retrieval of class definitions would be a great improvement over browsers.

B. OBJECTIVES

The primary objective of this thesis is to determine the suitability of a CAPS-like approach to object-oriented class storage and retrieval. This thesis will also determine if CAPS' storage and retrieval system can be used "as is", or if modifications will need to be designed and implemented.

C. ORGANIZATION

Chapter II is a survey of the literature of basic object-oriented concepts, reusability in both conventional and object-oriented environments, and CAPS. Chapter III gives a detailed description of the problem statement. Chapter IV presents the Class Storage and Retrieval System, our proposed solution. Chapter V includes conclusions and suggestions for future research.

II. SURVEY OF LITERATURE

This chapter describes some useful background concerning various concepts that will be used throughout this Thesis. The first section introduces the basics of object-oriented programming. The next section examines the concept of reusability in both conventional and object-oriented environments. Finally, the third section gives a general description of the Computer Aided Prototyping System.

A. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is a relatively new approach that promises to serve software development needs better than more traditional approaches have; that is, faster development, quality finished product, and easier maintenance and extensibility. To achieve this, object-oriented systems introduce some fundamental ideas which distinguish it from more conventional programming languages. We now give a brief description of these basic concepts independent of any specific implementation language.

1. Basic Concepts

a. Objects

"An object has state behavior and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable." ([Booc91]: pp.77)

In every day life we use the term object to describe a tangible real-world entity that has a specific structure and behavior. For example, **my_bicycle** is an object. It has a specific structure that differentiates it from other objects, such as **my_dog**. It also has a specific behavior: with the help of a driver it can be moved within a predetermined range of speed. OO systems implement this idea of an object, including intangible real-world entities[1].

Associated with an object is a set of *instance variables* which have specific values for that object and defines its structure. In the previous bicycle example, candidate instance variables would include the serial number, model, production date, color, etc. As we shall see later, the only way to manipulate this set of instance variables is with a set of methods (procedures) defined exclusively for that purpose[2] [Nels90a].

The notion of object is fundamental in object-oriented programming because it is one of the primary building blocks. Even the term object-oriented implies that we need to view the world in terms of objects or collections of objects.

b. Classes

"A class is a set of objects that share a common structure and a common behavior." ([Booc91]: pp.93)

The analytic description of an object in the previous section leads us to categorize them into different classes according to the information that they maintain

---

[1]For example, an Array A (8x1) of integers is an object because it has a specific structure and behavior; it has eight rows and one column and it can store only integers.

[2]This assumes an encapsulated approach - see Subsection c for more information on this.

(structure) and their abilities (behavior) . Thus, we find that "an object is an instance of a class" ([WEK90]: pp.31).

In the previous example, **my_bicycle** is an instance of a class **Bicycle** that includes all bicycles; **maria's_bicycle** is another instance of the same class. A class can have an infinite number of instances, and can be viewed as a factory that produces objects [Cox86].

All instances of a class have the same set of instance variables, but their values are specific for each instance. For example **my_bicycle** has serial_number, model, production_ date, etc. while **maria's_bicycle** has its own serial_number, model, production_date, etc.

Also associated with a class is a set of *class variables.* "A class variable is shared in both name and value by all instances of a class" ([Nels90a]: pp.2). For example we could define a class variable number_of_wheels for the **Bicycle** class to hold the value 2 as the number of wheels for each bicycle. The name and the unique value of this class variable is shared by all instances of the class.

c. Methods

"The procedures or operations that are defined for the object are called methods." ([Nels90a]: pp.2)

Methods are associated with either the instances of a class or the class itself and define their behaviors. The former are called *instance methods* and the latter *class methods.* However, both are defined as part of the class definition.

Ideally the state of an object or a class (i.e., the values of its instance or class variables respectively) can be retrieved and updated only through its methods [KA90]. The only way to communicate with an object is via instance methods defined for the class that it belongs to. The only way to create new instances of a class is via a class method. This property enforces the principle of information hiding and is called *encapsulation.* A class/object encapsulates its structure from the environment.

In order for a method to be invoked, a *message* with the same name as the method is sent to a specific object (receiver).

Continuing with our bicycle example, there may be a method **get_serial_number** defined for the **Bicycle** class. Sending the message **get_serial_number** to **my_bicycle** would cause it to respond with its serial number.

d. Inheritance and Hierarchies

(1) Inheritance. "The principle that knowledge of a more general category is applicable also to the more specific category is called inheritance." ([Bud91]: pp.5)

Inheritance is a powerful mechanism that distinguishes object-oriented programming from traditional programming. It is the result of the creation of a kind-of hierarchy. Within this class hierarchy a *subclass* is allowed to use all or part of the code (methods and variables) defined in its *superclass(es),* as though it were defined within the subclass itself.

A subclass is not restricted to using inherited methods and variables as is. In most object-oriented languages they can be redefined, and possibly even excluded from the definition of the new (sub)class [Nels90a].

They are two kinds of inheritance: *single inheritance* (referred to simply as inheritance) in which a subclass may inherit from a single superclass, and *multiple inheritance* (MI) that allows inheritance from several superclasses [KL89]. Some OOP languages provide both single and multiple inheritance, others provide only single inheritance.

(2) Hierarchies. "Hierarchy is a ranking or ordering of abstractions." ([Booc91]: pp.54)

Trying to categorize objects in general, it is very useful to find relationships among them. The most critical relationships are the *kind-of* and the *part-of*. A bicycle is a kind-of a vehicle for example, but it is not a part-of a vehicle. The kind-of relationship is generally implemented by inheritance, and the part-of relationship is generally implemented through composition (i.e., the variables making up an object are in turn other objects).

The result of the application of a kind-of or a part-of relationship is a kind-of hierarchy or a part-of hierarchy respectively. In most object-oriented programming languages the kind-of hierarchy is the class hierarchy and the part-of hierarchy indicates the object structure, that is the type of the instance variables [Booc91, LP91].

e. Polymorphism

"Polymorphism means the ability to take several forms. In object-oriented programming, this refers to the ability of an entity to refer at run time to instances of various classes." ([Meye88]: pp.224)

Polymorphism, sometimes called operator overloading, is the property that allows the same message name to be used for different objects of different classes, and for each object to react accordingly. Thus, the control of the reaction of a method invocation is transferred to the object itself. The user can send a message and leave the implementation details entirely up to the receiving object [WEK90]. Modifying the code of one object does not affect the behavior of another object. For this reason polymorphism together with inheritance enhances the concept of extensibility in object-oriented programming.

They are two kind of polymorphism: *simple polymorphism*, which allows several classes to each have their own implementation of an operation; and *multiple polymorphism*, which allows each class to have several operations with the same name. [Mica88]

2. Object-Oriented Database Management Systems

The development of applications that require more complex data, such as computer-aided design (CAD), computer-aided software engineering (CASE), and computer-aided manufacturing (CAM), along with the need to maintain that data for use by several people and/or programs, has led to the need for new database systems. This

is because conventional database systems typically offer only a fixed set of data types (such as integers and character strings) and lack the capability to define even simple new types and operations, much less complex types and an inheritance scheme [NMO90].

An object-oriented database management system (OODBMS) generally implements collections of objects, where each object represents a physical entity, an idea, an event, or some aspect of interest to the database application [EN89].

A data model is a set of concepts that can be used to describe the structure of a database [EN89]. Although there is no standard object-oriented data model, an object-oriented database management system can be defined as a database system which directly supports an object-oriented data model [Kim90].

B. REUSABILITY

"Reuse is the use of previously acquired concepts or objects in a new situation. Reusability is a measure of the ease with which one can use those previous concepts or objects in the new situation." ([PF87]: pp.7)

In software development process, reusability plays a very important role in order to reduce the effort of development and maintenance, and therefore to increase software productivity. A way to achieve this is through source code reuse; that is, the reapplication of code modules. We now describe the current state of the art in source code reuse in both conventional and OO systems, giving more emphasis to OO systems.

1. Conventional Systems

Two factors have stimulated a lot of research efforts towards the direction of code reuse: (1) dramatic increase in the cost of software in recent years; and (2) several studies that indicate that much of the code of one system is virtually identical to previously written code for another system [FG89].

Most of the systems developed with reusability in mind are composition-based systems. That is, they are based on the reuse of atomic and unchanged components of previously written code (building blocks) that are stored in large software libraries. [BR87]

In order to be able to use a software library a retrieval mechanism must exist. That is, a representation of and a search method for the software components must to exist. Several representation and search methods for software components have been proposed, including traditional library science methods, knowledge based methods, and hypertext [FG89].

There are three main approaches for retrieval of reusable components: browsers, informal specifications, and formal specifications [Ste91]. Browsers are of particular interest to this thesis, and are described extensively in Section II.B.2.a.

Informal specifications are based on the idea of looking (searching) for specified attributes of the desired component. There are several such search methods, including keyword search, multi-attribute search, and natural language interfaces. Keyword search is a search based upon a specified list of words relevant to the desired

component. Multi-attribute search is a search based upon an extended list of component's attributes that could include the class of the component (procedure, function, etc.), the number and type of parameters used, etc. Natural language interfaces in which natural language queries are used to search for the appropriate component. Advantages of informal specifications are simplicity and easy manipulation. Their most serious disadvantage is that is difficult to create systems that achieve a high degree of precision.

Formal specifications require that the software components and also the queries upon them be written in a formal specification language, consistent with the underlying implementation language, promising that automated and precise syntactic and semantic matching can be achieved. Disadvantages of this approach are that automated matching can be time consuming, and also the difficulty of writing formal specifications for the components. [McDo91]

Many of the recently developed reusable component systems, including Draco [Neig84], Reusable Ada Packages for Information Systems Software (RAPID) [Vog89], Reusable Software Library (RSL) [BW87], and Common Ada Missile Packages (CAMP) [CAMP89, Ande88] use one or more of the above described retrieval methods.

2. Object-Oriented Environments

"Using object-oriented techniques, we can construct large reusable software components." ([Bud91]: pp.14)

One of the primary purposes of object-orientation is to maximize the reusability of the constructed software modules. This is accomplished through the concepts that were introduced in the previous section.

A software module in an OO environment is simply a class. A class, as previously described, encapsulates its behavior from the environment by allowing manipulation only through its methods. "Classes provide not only modularity and information hiding but also reusability enhanced by inheritance and polymorphism" ([WEK90]: pp.47).

Classes are related to one an other via inheritance (single or multiple), thus creating a class hierarchy. This class hierarchy constitutes the base upon which applications can be built. "Most OO systems provide a set of predefined classes (libraries of classes are also available from various sources) which can be used either as is or for inheritance in designing new classes" ([NB92]: pp.562).

In order for an application to be built, users have to create their own classes as subclasses of the already existing classes (if any). Because a subclass inherits all or part of the behavior (code) defined for its superclass(es), the only thing that the user has to do is to add variables/methods or modify some inherited methods to make the new class(es) appropriate for the new application. It has been said that the entire generated class hierarchy is nothing but reusable code [Mul90]. Adding more classes to the class library increases code reusability, making programming more a technique of composition.

The construction of the class hierarchy is very critical for the kind of applications that can be directly supported, because it determines the actual inheritance paths that can be followed (i.e., the type and quantity of code that can be reused).

In order for a programmer to build an application, the appropriate superclass(es) among all of those predefined in the class library must first be detected. This means that in an OO environment a search and retrieval mechanism for class definitions is fundamentally necessity.

a. Browsers

A browser is a window-based software tool which allows potential users of existing class libraries to retrieve and view classes and their code at various levels of abstraction [Meye88]. It is an interactive tool, with characteristics and properties dependent on the implemented OO system, that applies the idea of programming by looking around, permiting some form of navigation through the class library [WEK90]. A browser helps the programmer to understand and become familiar with the class hierarchy, allowing efficient reuse of the predefined code.

b. Example Browsers

Different OO systems implement provided browsing facilities differently. We will now examine the browsers of some of the more common OOP languages in more detail, considering their advantages and disadvantages.

(1) Smalltalk-80. Smalltalk-80 [Gold84] includes a System Browser and a number of Class Browsers (see Figure 1), distinguished by the particular subset of classes that can be accessed.

The System Browser is made up of five panes, each one having pop-up submenu choices that provide several editing and source code compiling facilities. In the Class Categories Menu (first pane), categories of classes are displayed. Choosing one category causes the Class Names Menu (second pane) to display the classes that belong to that category. Selecting a class from the Class Names Menu causes the definition of the class to appear on the Text area (bottom pane), and its list

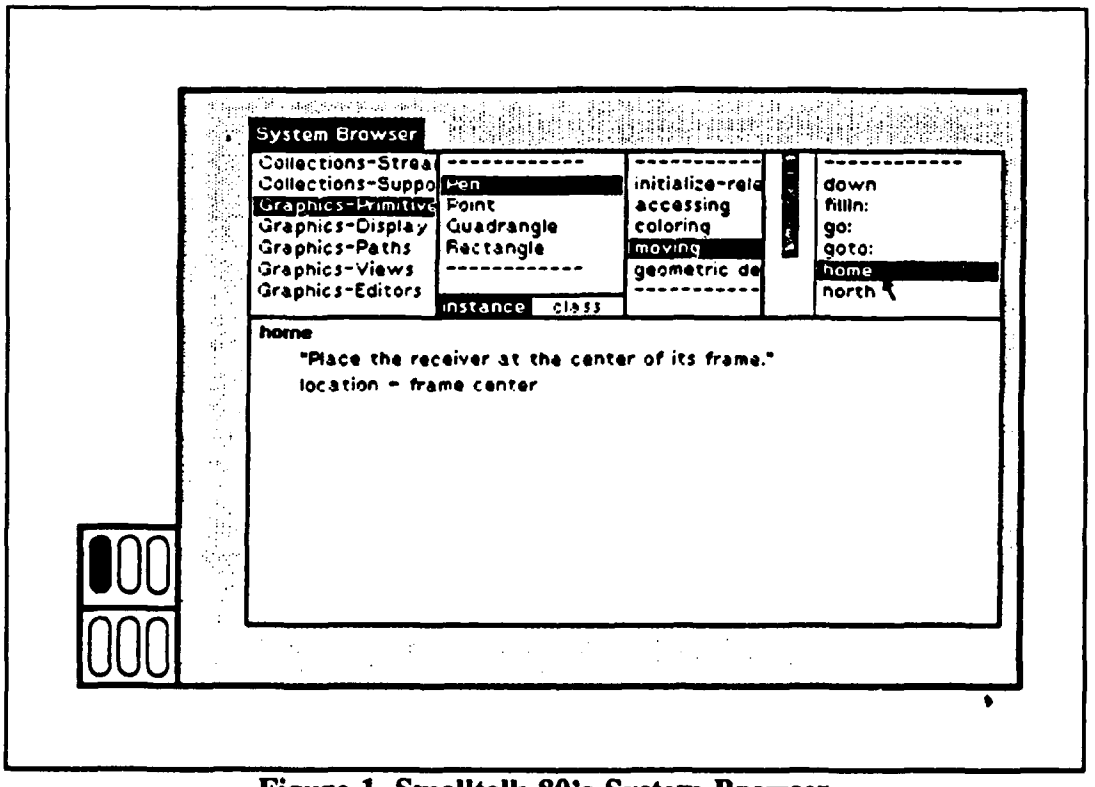


Figure 1. Smalltalk-80's System Browser

of categories of messages appears on the Message Categories Menu (third pane). Finally, selecting a Message Category causes the methods that belong to this category to appear on the Message Selectors Menu (fourth pane), and the source code for that method appears on the Text (bottom pane).

The functionality of the System Browser requires that the user know in advance the category that a class belongs to in order to find it. If the user does not know the construction of the system's library well, it becomes more and more difficult to find a class. This results in a gradual degradation of the entire performance of the System Browser. It should also be obvious that as the number of classes grows, it will become more difficult for the user to know which category to search.

An alternative way to create a browser (for only one class) is by creating an instance of the class Browser for a specific class. The result is a browser for only that specific class.

(2) Smalltalk/V 286. Smalltalk/V 286 [Dig88] introduces the notions of Class Hierarchy Browser (see Figure 2) and Class Browser. They are used as follows to facilitate program development.

The Class Hierarchy Browser is a window consisting of five panes. The class hierarchy pane displays all of the classes in the system in a hierarchical order using indentation. Selecting a class from the library causes the class definition, including its superclass, its instance and class variables, and its pool dictionaries, to be displayed on the contents pane. In addition, the class or the instance methods are displayed on the

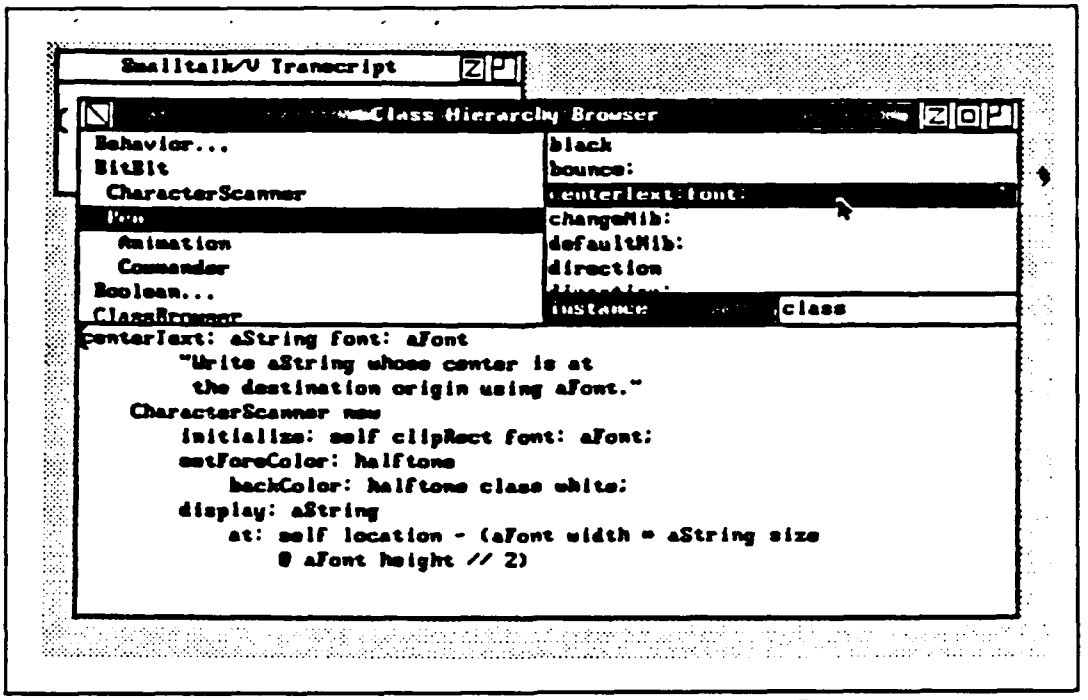


Figure 2. Smalltalk/V 286's Class Hierarchy Browser

method list pane, depending on the choice of the user on the corresponding class or instance pane. Choosing a particular method from the list pane causes its source code to be displayed on the contents pane. The contents pane is not only a displaying pane, it has also editing capabilities.

All the information concerning a class is stored in a separate file for each class (with the extension .cls). This file is recompiled each time anything in the class definition is changed.

Several other facilities are also offered in the Class Hierarchy Browser. The most important of these are the Senders and Implementors choice. With Senders, Smalltalk searches all of the methods in the environment for senders of a

selected message. With Implementors, Smalltalk searches all of the classes for implementors of the selected message.

The Class Browser is a window that is opened for a specific class and allows the manipulation of the source code of that class.

The Class Hierarchy Browser allows navigation through the class hierarchy that is much easier if the hierarchy is organized as a tree. From the moment that the programmer knows which is the desired class, its definition and its source code are easily manipulated with the provided menu choices. But when the programmer does not know where to begin, all class definitions must be searched to find the appropriate one. This problem becomes worse as the number of classes in the library is grown.

(3) Actor. Actor [WG90] (version 3.0) is a pure OO language like Smalltalk. A Browser in Actor (see Figure 3) is a highly specialized editor for viewing, writing, changing, and saving code. It is a window with several menu choices that allows the user to create new classes, create/modify methods/variables, and study the classes and methods in the development environment. [WG90]

A Browser is created by clicking the mouse on the Browse menu of the Actor Workspace (the main working window of Actor's environment), or by sending a browse message to a class, or by clicking on the browse menu of an already active Browser. With this way it is possible to have multiple browsers open.

A Browser has four areas. The Class Box area displays all of the classes in the library and their objects (instances), indented according to the class

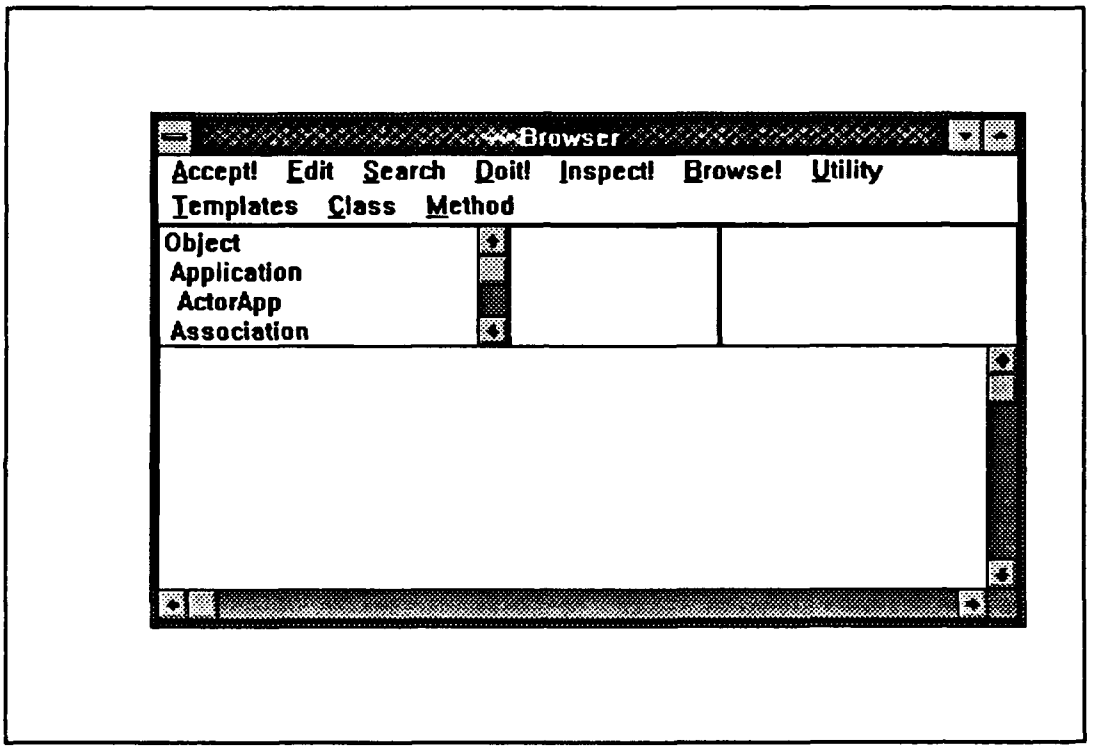


Figure 3. Actor's Browser

hierarchy, or alphabetically. The Variables Box displays class and instance variables and also inherited variables from ancestor classes for a selected class of the Class Box. The Methods Box displays a list of the class or object (instance) methods, defined for the particular selected class (or object) in the Class Box. By clicking a method, its source code appears in the Editing area of the Browser. There is no way to display inherited methods from ancestor classes. The user has to manually search the ancestor classes and their methods to discover the code that is inherited.

Several useful editing, compiling, debugging, and other facilities exist that allow the user to easily manipulate the source code and navigate through the system supplied class library. Actor comes with a library of about one hundred

predefined classes forming a standard class hierarchy. Every newly defined class must be a descendant of an existing class. Although the system supplied class library is not large, a lot of time is spent in manual search of the library every time a new class is defined because:

1. The appropriate superclass has to be found.

2. All inherited methods of the ancestor classes have to be examined separately to discover the useful ones for the new user's application.

3. All inherited class variables have to be examined to discover what is inherited and from where.

The time to do this increases considerably as the class library grows.

(4) Sun C++. The Sun C++ version 2.1 [Sun91] Browser is called the Sourcebrowser. The Sourcebrowser (see Figure 4) was developed to help new programmers in joining a programming team, especially in the development of large programs. Using a "what you see is what you browse" paradigm, it is a tool to help understand how applications work by locating all occurrences of desired symbols and strings. Sourcebrowser can be used with Sun C, ANSI C, FORTRAN, Pascal, and Modula-2, in addition to Sun C++ [Sun91].

When issuing a query, Sourcebrowser searches in a specialized database (that contains pertinent information about the files that are browsed) to find matches of the symbol or the string constant that has been specified. This database is built using the -sb option during the compilation of source files. More specifically, the

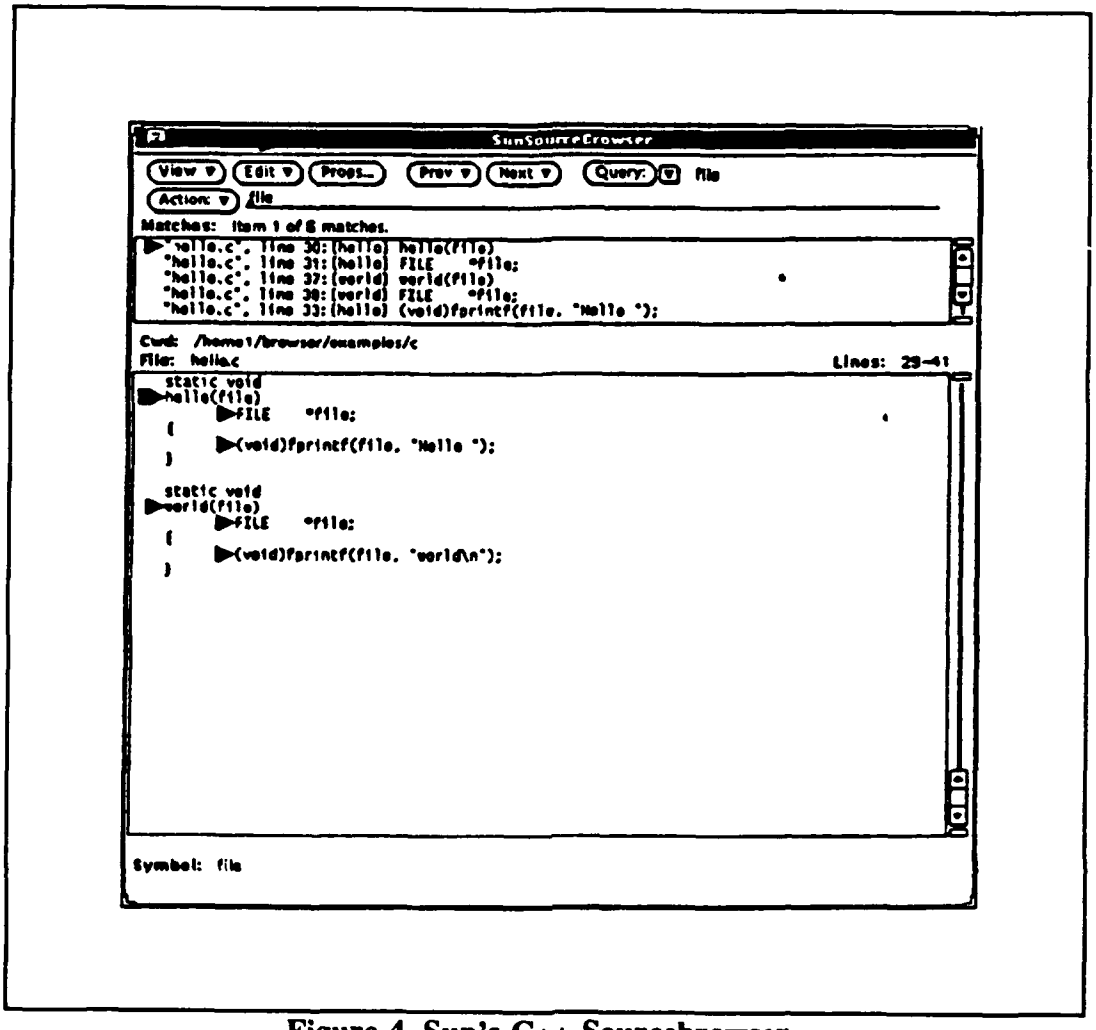


Figure 4. Sun's C++ Sourcebrowser

-sb option causes the compiler to create a .bd (browser data) file for each source file. Then, prior to responding to the initial query following a compilation, the Sourcebrowser creates an index file which it uses to locate information in the .bd files. The .bd files together with the index file, are stored in a separate subdirectory .sb (SourceBrowser). Each time a source file is recompiled with the -sb option a new .bd file is created. The index file is updated when issuing the first query following the compilation.

Sourcebrowser's user interface consists of a main window with several menu/submenu choices and some pop-up windows necessary for controling its functionality. Matches are displayed on the Match pane section of the main window, and the respective source code on the Source pane section.

Due to Sourcebrowser being developed in this way, it offers a more useful editing facility than might be expected in a search tool for a library in an object-oriented programming language. For this reason it can also be used in languages that are not object-oriented as previously mentioned.

The latest version, SPARCworks Professional C++ 3.0, includes a set of programming tools including a SourceBrowser and a ClassBrowser. The functionality of the SourceBrowser is the same as in Version 2.1. ClassBrowser offers the ability to display a class hierarchy, navigate through it, and display source code, class data, or member functions for a particular class.

(5) Borland C++. The latest version of Borland Turbo C++ (3.0) [Bor91] includes an ObjectBrowser that graphically shows relationships between objects and allows the programmer to navigate through the source code.

The ObjectBrowser is a window that can be accessed from the Browse menu of Turbo C++, or from the source code of a program by clicking the right mouse button on a specific class, function, or variable. The source code has to be compiled before the ObjectBrowser is called.

The Classes choice from the Browse menu of the ObjectBrowser offers a horizontal tree display of all the classes of an application. This allows the programmer to see the class hierarchy of the particular application. By choosing a specific class from the class tree it is possible to display the source code that defines this class and inspect its functions and data elements.

With the Functions or Variables choice from the Browse menu a window with all the functions or all the global variables, respectively, of the program is opened. Class member functions are listed together by class. It is possible for the programmer to choose a function or variable and go to the source code that defines it to inspect its declaration.

The way that predefined source code is treated by the ObjectBrowser is very useful. The programmer can search in the existing class tree hierarchy to discover the class(es) that could become superclass(es) in their application, examining the previously defined functions and variables. However, as the number of classes in the class library grows, this manual search becomes more and more time consuming.

(6) Prograph. Prograph [Gun90a,90b,91] is a language that combines object-oriented, pictorial, and dataflow features. At this time it is designed to run only on Macintosh machines.

Even the latest version of Prograph (2.5) [Gun91] does not directly support a browser. The term browser does not even exist in the Prograph language.

Nevertheless, most of the functionalities of a typical browser that the previously described languages usually support are included in Prograph through different menu selections.

The Info choice of the Info menu displays the Info window on the screen which offers information about different categories of elements. Among these categories the most important for us are the Primitives, Classes, Attributes, Methods, and Universal Methods that list the functions, classes, defined and inherited variables, methods for each class, and global methods of the system, respectively. In order to find a particular primitive, class, variable, or method, the user has to know in advance its existence in the system's library, and scroll through the respective list of the Info window. Clicking the mouse on a specific primitive, class, variable, or method causes its comments to appear on the right pane of the Info window.

The submenus of Windows menu open respective windows with useful information about the class structure and behavior. With the submenu Class choice, a pictorial tree representation of the class hierarchy is displayed on the screen. Clicking the mouse on the icon that represents a specific class gives the user the ability to display the variables or the methods of the class. Again, every search is manual, requiring users to have knowledge about the structure of the class library in order to facilitate the building of their application.

C. THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS)

1. General Overview

Building large real-time systems and systems which have hard real-time constraints using the traditional classical project life cycle approach does not allow the designer to know if the system can be built with the necessary timing and control constraints until much time and effort have been spent on the implementation.

In contrast to the classical project life cycle, the prototyping method extracts, presents, and refines a user's needs by building a working model of the ultimate system quickly and in context [Boa84]. This model is called a prototype, and is only used to model the system's requirements.

The Computer Aided Prototyping System CAPS [LK88, Luqi91], an ongoing project in the Naval Postgraduate School Computer Science Department, is a software development environment that provides a means to rapidly construct an executable prototype representing a large real-time software system with hard real-time constraints [Cum90].

The major subsystems of CAPS (see Figure 5) are as follows:

1. The User Interface, that has graphics capabilities and allows a graphical representation of the prototype(s).

2. The Execution Support System, that gives the designer the ability to execute the constructed prototype(s).

3. The Software database, that provides a repository of reusable Ada components (Software base) and an engineering database management system with an embedded design management system.

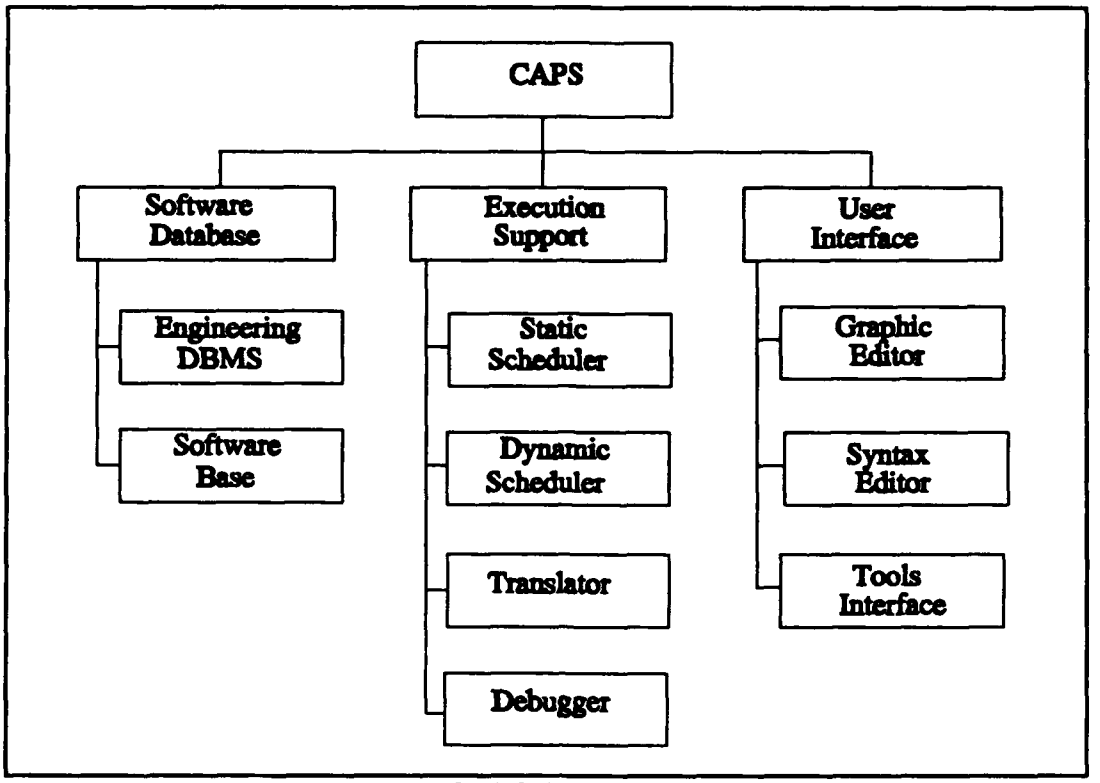


Figure 5. CAPS Structure [Ste91]

CAPS prototypes a system through translation of the high level specification language, Prototyping System Description Language (PSDL) [LBY88], into Ada code along with the incorporation of atomic Ada reusable components [LBY88]. PSDL is used to specify the interface and functionality of the atomic components in order to make automated searches of the reusable component library feasible.

In order to generate a prototype the designer uses the graphic editor to create a graphic representation of the proposed system. From the graphic representation a part of an executable description of the proposed system is generated in PSDL as shown in Figure 6, using a rewrite subsystem. PSDL descriptions are used to search the software

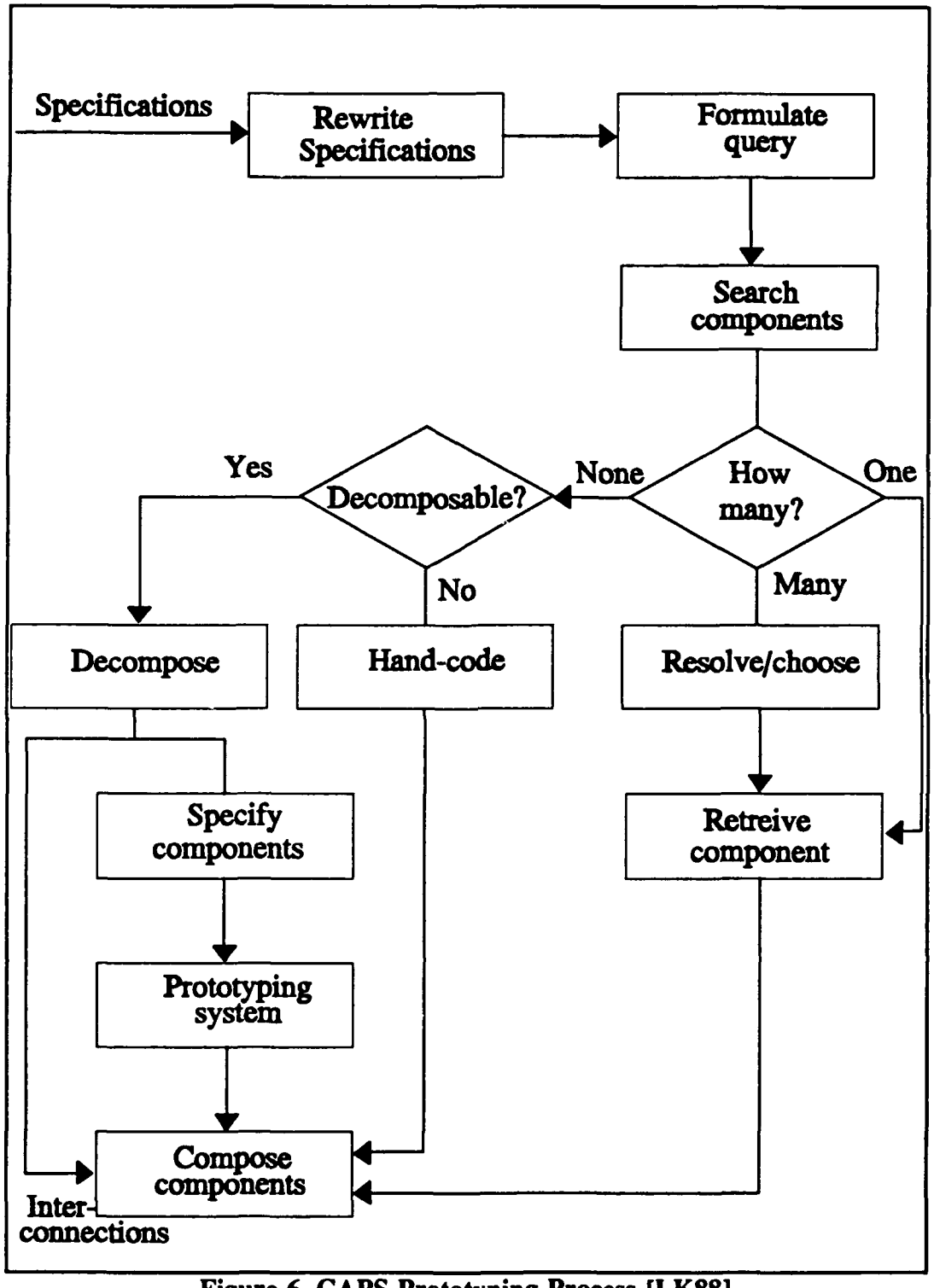


Figure 6. CAPS Prototyping Process [LK88]

base to find reusable components that match the specifications. If only one component meeting the specifications is found, it is retrieved; if more than one are found, the designer has to choose one; otherwise the specification has to be decomposed into simpler specifications applying the entire process to those specifications, or the user has to implement the component manually. A transformation schema is then used to transform the PSDL specifications into Ada code that controls and connects the retrieved reusable components. The prototype is then compiled and executed and the designer evaluates its behavior to see if the requirements are met or if the entire process needs to be repeated. In this way a system that will finally meet the users requirements will be produced. [Ste91]

2. The Prototype System Description Language

PSDL forms the basis of CAPS, allowing the system designer to create executable prototypes of the prospective system. It is built using the concepts of data abstraction, function abstraction, and control abstraction, and is capable of supporting hierarchically structured prototypes that preserve their modularity. [LBY88]

The grammar of PSDL (included in Appendix A) allows a prototype to be either an abstract data type (ADT) or an operator. A software module is modeled as a network of operators that communicate via data streams [Ste91]. Each data stream carries values of a fixed abstract data type. Formally, the computational model on which PSDL is based on is an augmented graph:

$$G=(V,E,T(v),C(v))$$

where G is the graph that represents the prototype, V is the set of vertices, E is the set of edges, $T(v)$ is the maximum execution time for each vertex v, and $C(v)$ is the set of control constraints for each vertex v. [LBY88]

The prototypes constructed in PSDL can be executed in CAPS if they are supported by software components in an underlying programming language. The software components are stored in the CAPS Software base. The current version of CAPS uses Ada as the underlying programming language, but plans are for future versions to support more languages. [LBY88]

3. Storage/Retrieval of Reusable Components In/From CAPS

As previously discussed, one of CAPS tasks is to provide the designer a tool for rapid prototyping. In order to achieve this, CAPS is designed to take full advantage of reusable components stored in a Software base, minimizing the search time of the library by automating the search.

The Software base uses the ONTOS Object-Oriented Database Management System (OODBMS) [Onto90]. ONTOS supports a multi-user networked environment and is not constrained by a particular data model such as relational or hierarchical systems. It was developed to be able to store components, to browse them, search them by query, and integrate them after locating so that the execution support system can produce an executable prototype. [McDo91]

In order to be able to automate the search mechanism of the software base, the PSDL specification of every software component to be stored in the Software base passes through a *syntactic* and then a *semantic normalization*. Syntactic normalization involves format changes and statistical calculations that standardize the component's interface characteristics [MacDo91]; semantic normalization requires specification expansion and transformations and standardizes the component's behavior. The produced normalized specification is stored with each component in the software base [Ste91].

a. Syntactic Matching

The purpose of syntactic matching is to find those software components stored in the Software base that have PSDL specifications that syntactically match the PSDL specification of the posed query. For this reason the interface characteristics of each component are exploited and components that do not meet the specification of the query are eliminated from further consideration.

The syntactic matching rules (described in [MacDo91]) are first used to derive a set of module attributes that will be used to eliminate components with unsuitable interfaces. Examples of these module attributes include:

1. If the number of input parameters in S(q) is not equal to the number of input parameters in S(m), then S(m) can be eliminated from the search.[3]

2. If the number of output parameters in S(q) is greater than the number of output parameters in S(m), then S(m) can be eliminated from the search.

[3]S(q) is the PSDL interface specification for a query module q, and S(m) is the PSDL interface specification for a Software base module m.

The rest of the components are checked one by one against the syntactic rules until the final candidates will be selected. [MacDo91]

It is very important to realize that this process involves only the PSDL specification; neither the implementation specification nor the implementation code of the component is used.

b. Semantic Matching - OBJ3

An optional item for each PSDL component is its formal description, called the axioms. An algebraic specification language known as OBJ3 [GW88] is used for the axioms of each component that express the semantics of the PSDL specification of the component. Before a component is stored in the Software base its OBJ3 specification is normalized. The specification of each query is also normalized and a technique called *query by consistency* is used to exploit the OBJ3 formal semantics in order to achieve semantic matches. [Ste91]

Semantic matching is not yet integrated into CAPS Software base, but is proposed to be used on those components that have first achieved a syntactic match in order to produce a final list of candidate components for evaluation by the designer. Writing formal specifications in OBJ3 may be a difficult and time consuming procedure, but it ensures high degree of precision [Ste91].

c. Alternative Methods

This automated storage and retrieval process may slow down the execution of the system, but it ensures precision, maximum degree of reusability, and also avoids

the time consuming manual search of the library, especially as the number of the stored components grows.

Alternative ways to search the software base for candidate components are with a *keyword query* that lists all the components that possess one or more of the query keywords, and *named look up* that browses all components of a particular library alphabetically [MacDo91].

All of these functionalities, combined with addition, deletion, and update operations, are provided by the CAPS Software Base graphical user interface, a windowing system that integrates the communication of the Software base with the designer.

III. THE PROBLEM STATEMENT

A. CLASS DEFINITIONS IN AN OO ENVIRONMENT

Associated with each class in every OO environment is its definition. By *class definition*, we mean all of the necessary information about a class that the potential user searching the class library needs to be able to understand what this class includes, and how instances of the class are manipulated.

The information provided for each class definition should be minimized so that the performance of the system is not degraded as the number of classes grows. At the same time, however, it must be detailed enough to ensure that the user knows all that he needs to know about the class.

As discussed in Chapter II, there is no universally accepted definition of OOP. However, the language independent class definition, as shown in Figure 7, does encompass the class definition of every OOPL that we have surveyed. That is, actual

```
Class <class_name>
  Superclasses          : <superclass_1>, <superclass_2>, ...
  Class Variables       : <class_var_1>, <class_var_2>, ...
  Instance Variables    : <inst_var_1>, <inst_var_2>, ...
  Methods               : <method_name_1>, <method_name_2>, ...
```

Figure 7. Language Independent Class Definition [Nels90a]

class definitions from any OOPL are easily converted to and from this language-independent form. Therefore, this is the form of class definition that we will work from in this thesis.

B. THE BROWSER APPROACH

The promise of object-orientation is to create reliable, sharable, easily reusable, extensible, and maintainable modules of code (i.e., classes). Existing OO systems support these fundamental principles by implementing some form of browser. A browser is usually a window-based software tool that allows the user to manipulate a predefined class library facilitating class definition and code search and retrieval.

We now summarize the advantages and disadvantages of the browser approach as a result of the examination of browsers as contained in Chapter II.

1. Advantages

The browser approach is based on the idea of programming by looking around [WEK90]. For that reason browsers are developed in order to facilitate a manual search, allowing programmers to search previously defined classes for those that are appropriate to their needs.

As window-based tools they offer the capability to display a variety of information regarding a class library such as the class hierarchy, the class definitions, the superclasses or subclasses of a specific class, locally defined variables and methods, inherited variables and methods, and underlying code. How all this information is

organized and provided to the potential user is a matter of user interface and is the only real difference among the browsers we have surveyed.

For a small number of classes the browser approach is a very powerful tool that helps programmers to develop their own applications by building upon existing code, one of the primary benefits of the OO approach.

2. Disadvantages

The main common characteristic and also the largest disadvantage of browsers is that they obligate potential users of the class library to manually examine the class definition of each class in the library in order to find candidate class(es) for their application. Even if the class library is not so large (some hundreds of classes) and even if the system comes with manuals for the predefined classes, a large amount of time is spent until the user becomes familiar with this set of classes. Increasing the classes of the library (say to the thousands) considerably increases the search time, gradually eliminating the reusability benefits of object-orientation.

3. Capitalizing on the Advantages

In order for the browser approach in a large class library to be useful it should be integrated with an automated class definition storage and retrieval mechanism. This mechanism should standardize class definitions, store them in a class definition database, and each time a class is needed this mechanism should perform an efficient automated search of the class database to detect the appropriate class(es). That is, an automated system should search the available set of classes, providing a relatively small

number of classes that can then be searched manually using a browser. We believe that software development time under this approach will be considerably reduced and that the reusability benefit of object-orientation will then be fully realized.

C. USING CAPS TO STORE AND RETRIEVE CLASS DEFINITIONS

CAPS, as described in Chapter II, uses an efficient automated storage and retrieval mechanism (which is basically based on syntactic and semantic matching of PSDL specifications) to enhance the reusability of predeveloped software components. This mechanism ensures precision, maximum degree of reusability and avoids the time consuming manual search of the software library.

Using the CAPS Software base as a class definition database would benefit the user with its automated storage and retrieval mechanisms, and therefore would be a great improvement over browsers.

1. PSDL Specifications and Class Definitions

The idea to use CAPS's storage and retrieval mechanism to store and retrieve class definitions arises from the realization that a class definition is simply a software component. Further consideration of this observation leads to the functional commonality of a software component's PSDL specification and a class definition. A software component's PSDL specification consists of the specification of the component's underlying code and is also the basis upon which storage and automated retrieval is achieved. A class definition consists of the specification of the class' underlying code

and is also the basis upon which manual storage and retrieval is achieved in most OO systems (via browsers).

In order to be able to store a class definition in the CAPS Software base, it must first be transformed into a PSDL specification. Therefore transformation rules need to be developed that will transform a class definition into a PSDL specification. These rules must create proper PSDL grammar and also preserve the fundamental concepts of the class definition.

Each PSDL specification in the CAPS Software base is accompanied by an implementation specification and an implementation body file that include the Ada specification and body of the component, respectively. Since we will not be working with Ada, the use of these files will need to be considered in our solution.

2. Syntactic Normalization/Matching

A PSDL class definition would be treated by the CAPS Software base automated storage and retrieval mechanisms as though it were a regular component's PSDL specification (i.e., Ada specification and body files available). A normalized version of the PSDL specification is automatically created and stored with each class definition. Each time the user needs to know if a class with a particular structure and behavior exists in the Software base, the prospective class definition must be transformed into a PSDL specification in order to use the automated retrieval mechanism for potential syntactic matches.

3. Semantic Normalization/Matching

Semantic normalization and matching, as described in Chapter II, requires the semantics of each component to be written in OBJ3. This is a difficult and time consuming process that is not yet integrated in the Software base. Therefore, it will not be considerd in this thesis.

IV. THE CLASS STORAGE AND RETRIEVAL SYSTEM (CSRS)

PSDL, as previously described, was originally designed for describing prototypes of hard real time software systems. It forms the basis of CAPS, and in particular of the storage and retrieval mechanisms of the Software base. Therefore, whatever extensions to the usefulness of the language in order for OO concepts to be supported should not modify its grammar constructs as this would detract from the current use of the system.

Following are the necessary assumptions in order for the basic OO concepts to be represented in PSDL. These assumptions are derived from a subset of the PSDL grammar rules that is included in Appendix B.

A. CLASS DEFINITION TRANSFORMATION INTO PSDL COMPONENT

There are two basic bulding blocks in the PSDL grammar: abstract data types (ADT) and operators (see Figure 8).

An ADT is both a data type and a set of operators valid for that type. Because an ADT encapsulates both its structure and operators from the environment, it is a close relation to a class definition. The only thing missing from an ADT is the concept of inheritance. Therefore during the transformation of a class definition into a PSDL component we will represent a class as an ADT, addressing the problem of inheritance later.

A data type (see Figure 9) is described by the reserved word "type" and the identification (id) of the type (in our case this corresponds to the class name), followed

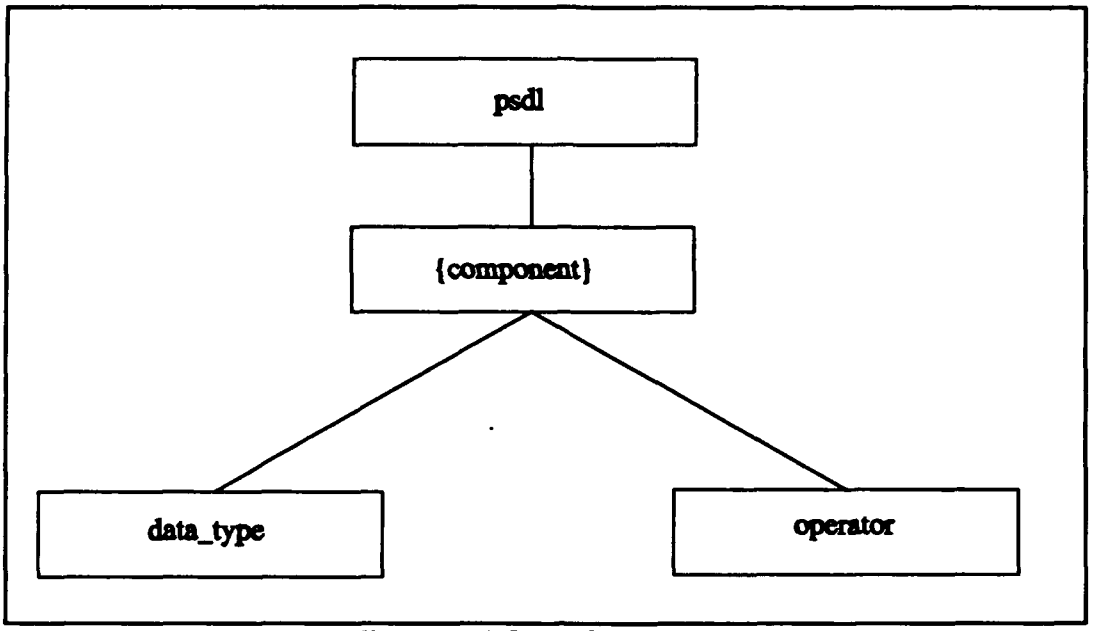


Figure 8. PSDL Component

by the type specification (type_spec). The type specification requires the reserved word "specification" to be written after the identification of the type (class name) followed by one or more from a set of optional items which will be discussed later.

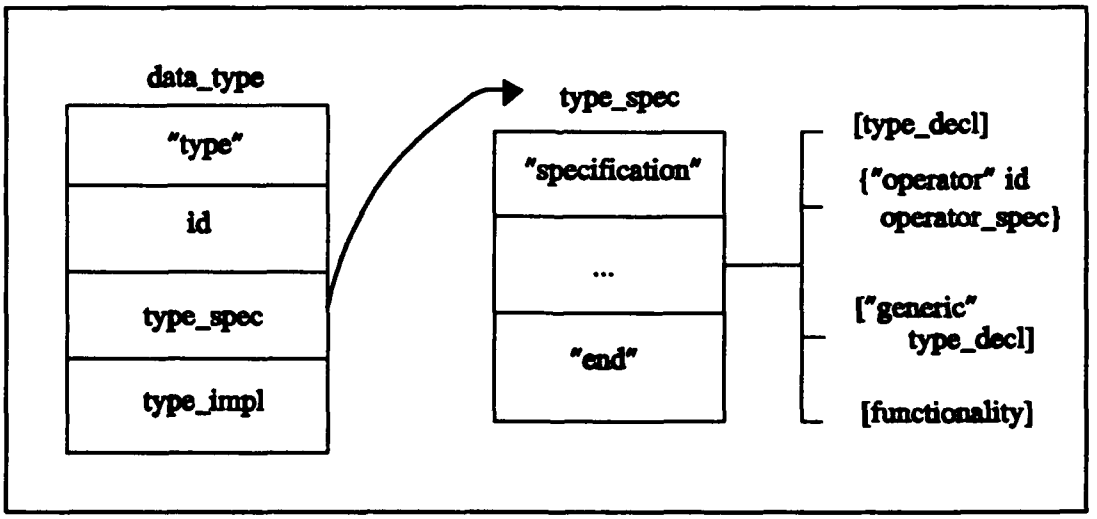


Figure 9. PSDL Data Type

Returning to the bicycle example of Chapter II, the class definition of the class **Bicycle** written in PSDL would begin as follows:

TYPE **Bicycle** SPECIFICATION

The class definition of the class **Vehicle** that includes all kind of existing vehicles (and could be a superclass of the class **Bicycle**) would be:

TYPE **Vehicle** SPECIFICATION

1. Variables

Among the provided set of additional optional items for the type specification is the declaration of the variables making up that type, the type_decl. The grammar rules for this declaration (see Figure 10) require each variable's name (type_name) to be specified, and also provide for more than one type_name to be declared.

a. Class Variables

PSDL does not include grammar rules to support class variables. Class variables could be included in the same way that instance variables are, but the CSRS would not differentiate between them. Inclusion of class variables is left as a suggestion for future research (see Chapter V).

b. Instance Variables

In our transformation a type declaration corresponds to an instance variable declaration. Each time an instance variable is declared it has to be followed by its type, called the type_name. One or more instance variables can be included in the

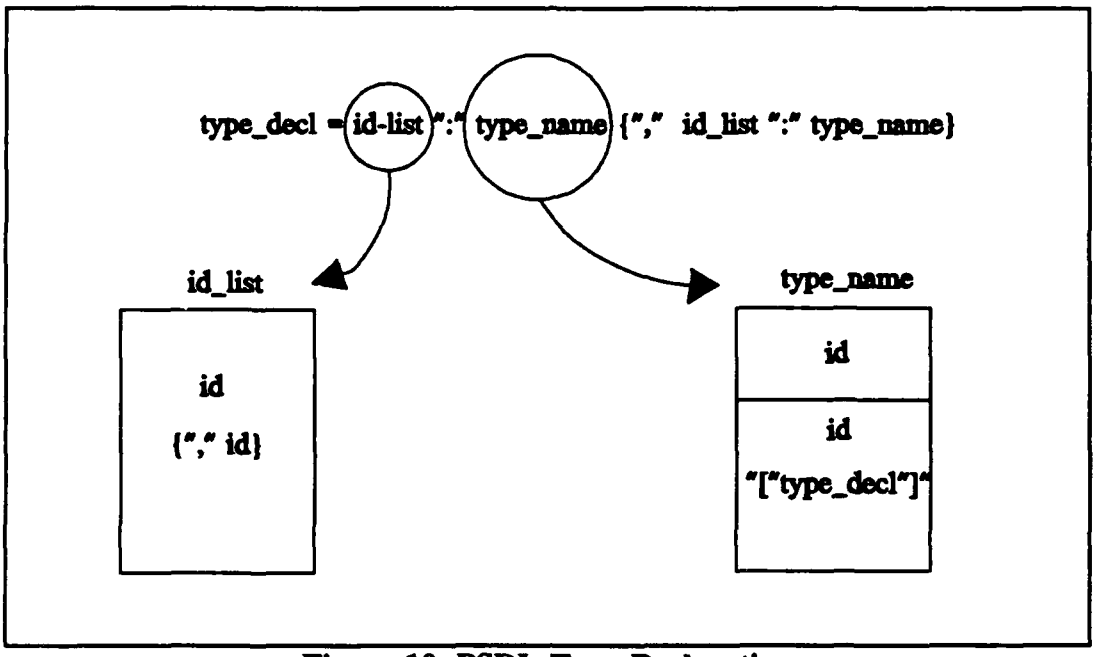


Figure 10. PSDL Type Declaration

specification of each class definition.

With the addition of instance variables the PSDL class definitions of our previous example would now be:

TYPE **Vehicle** SPECIFICATION

serial_number : integer

TYPE **Bicycle** SPECIFICATION

serial_number : integer

2. Methods

Another optional item for the type specification that may also appear zero or more times is the declaration of an operator. An operator is declared by the reserved word "operator" followed by its identification and then by the reserved word

"specification". The specification of an operator (see Figure 11) consists of its interface and its functionality, describing it through a set of optional attributes by various reserved words ("input", "output", "states", etc.).

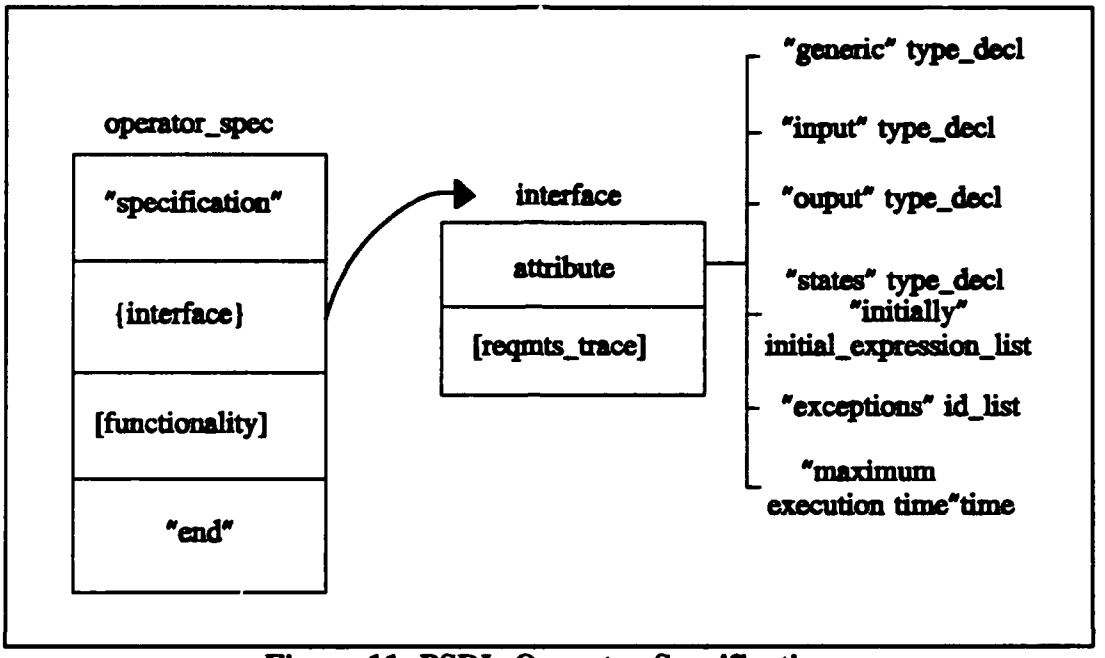


Figure 11. PSDL Operator Specification

In our transformation the declaration of an operator will correspond to the declaration of a method. The input and output parameters (if any) of each method will be described by the type declaration following the reserved words "input" and "output" of the attribute optional items. Each input or output parameter consists of its name and its type. The optional items [functionality] of operator_spec, [reqmts_trace] of interface, and "generic", "states", "exceptions", and "maximum execution time" of attribute are not necessary for our transformation, and are therefore not included in the grammar of Appendix B.

Continuing with our bicycle example the PSDL class definition of the **Vehicle** class, including its methods, would now be:

```
TYPE Vehicle SPECIFICATION

  serial_number : integer

  OPERATOR get_serial_number SPECIFICATION
    OUTPUT
      number : integer
  END
```

The PSDL class definition of the **Bicycle** class would be:

```
TYPE Bicycle SPECIFICATION

  serial_number : integer

  OPERATOR get_serial_number SPECIFICATION
    OUTPUT
      number : integer
  END
```

3. Inheritance

According to its specification, PSDL supports hierarchically structured prototypes [LBY88]. For this reason the PSDL grammar provides constructs for a *part-of* hierarchy. It is not possible with the current provided facilities of the language to have a *kind-of* hierarchy that would allow inheritance among our class definitions. We have identified two solutions to this problem. The first, called 'automatic' inheritance, assumes that the CSRS will be able to construct a class definition based upon its ancestor classes. The second, called 'manual' inheritance requires the designer to manually enter all of the inherited variables and methods.

a. 'Automatic' Inheritance

This approach should completely eliminate the problem of inheritance if CSRS were to be further integrated into a more independent system. The actual implementation, however, is left as a suggestion for future research, as discussed in Chapter V.

The optional item [functionality] of the type_spec allows for a component to be described by a set of optional keywords (see Figure 12) that are specified after the reserved word "keywords". In our proposed solution, the id_list that follows the reserved word "keywords" will be used to specify the superclass(es) (if any) of the class. An automated system could then be developed that would scan the class

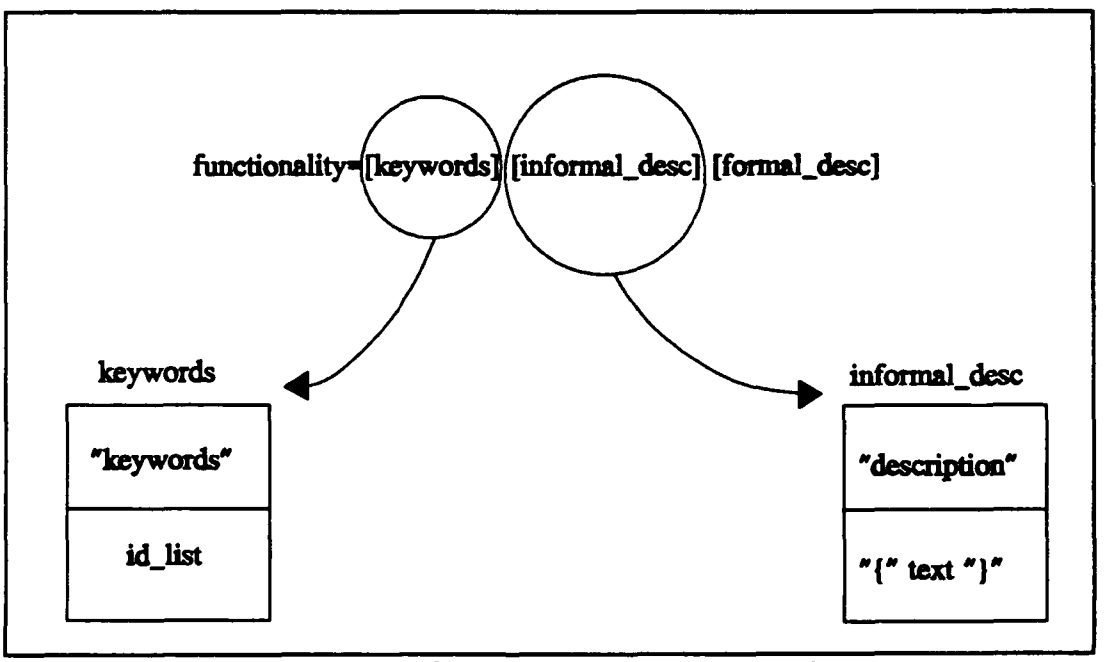


Figure 12. PSDL Component Fuctionality

definitions of the specified superclass(es) in this id_list to see if more superclasses are specified in their "keywords" id_list, etc. In this way the system would "walk up" the class hierarchy, and it would therefore be possible to automatically inherit variables and methods from a class' ancestors.[1]

The final form of the PSDL class definitions of the bicycle example using 'automatic' inheritance would be:

```
TYPE Vehicle SPECIFICATION

  serial_number : integer

  OPERATOR get_serial_number SPECIFICATION
    OUTPUT
      number : integer
  END

  KEYWORDS
    vehicle

  DESCRIPTION
    { A class that includes all vehicles }
END

TYPE Bicycle SPECIFICATION

  KEYWORDS
    vehicle, bicycle

  DESCRIPTION
    { Subclass of vehicle }
END
```

[1] The item [formal_description] consists of the "axioms" of the the component that, as described in Chapter II, would have to be be written in OBJ3. Therefore, they are not included in this discussion or in Appendix B.

b. 'Manual' Inheritance

In this approach it is the programmer that has to do the work of the automated system. That is,the superclasses specified in the [keywords] of a specific class must be manually searched for inherited variables and methods which are then included in the class specification. Continuing with our example, the PSDL class definitions using 'manual' inheritance would be:

```
TYPE Vehicle SPECIFICATION

  serial_number : integer

  OPERATOR get_serial_number SPECIFICATION
    OUTPUT
      number : integer
  END

  KEYWORDS
    vehicle

  DESCRIPTION
    { A class that includes all vehicles }
END

TYPE Bicycle SPECIFICATION

  serial_number : integer

  OPERATOR get_serial_number SPECIFICATION
    OUTPUT
      number : integer
  END

  KEYWORDS
    vehicle, bicycle
```

```
    DESCRIPTION
      { Subclass of vehicle }
END
```

B. IMPLEMENTATION SPECIFICATION AND BODY

In order for the designer to be able to store and execute the developed prototypes in CAPS the PSDL specification files must be accompanied by actual software components. That is, an implementation specification file and an implementation body file. The current version of CAPS is restricted to the use of Ada for these components, but plans are for future versions to provide the ability to choose from other programming languages.

At this time, however, we only want to use CAPS to store and retrieve class definitions. We are not interested in a system that will also provide mechanisms to execute these classes. Therefore, we will not be concerned with supporting CSRS's classes with actual implementation specification and body files (especially as Ada does not include inheritance).

For these reasons the item type_impl of the data_type (see Figure 9) is not included in the grammar of Appendix B. The implementation specification and implementation body files will be created (as they are required by the current system), but they will be empty. Therefore continuing with our bicycle example the following empty files would be created:

- vehicle.imp.spec
- vehicle.body.a

- bicycle.imp.spec
- bicycle.body.a

If OO languages (such as C++, Smalltalk, etc.) are supported by CAPS in the future, each PSDL class definition could be accompanied by its actual code, thereby achieving a more integrated system.

C. STORAGE/RETRIEVAL OF CLASS DEFINITIONS IN/FROM CSRS

All of the storage and retrieval facilities provided by the CAPS Software base can be accessed by the Software base graphical user interface or by the command line interface [MacDo91]. The command line interface requires the designer to manually enter several commands in order to communicate with the system. We believe, however, that this approach would not be efficient for CSRS in terms of overall performance and simplicity; therefore we will explain the storage and retrieval of PSDL class definitions only through the graphical user interface which is a window based environment consisted of a main window with the File, Browse, and Query menu choices.

1. Storage

With the Add Component submenu choice of the File menu the Input File Selection pop-up window appears on the screen (see Figure 13) prompting for the PSDL specification, implementation specification, and implementation body files to be specified.[2]

---

[2] The Software base does not provide an editing facility; therefore any available editor could be used to create these three files.

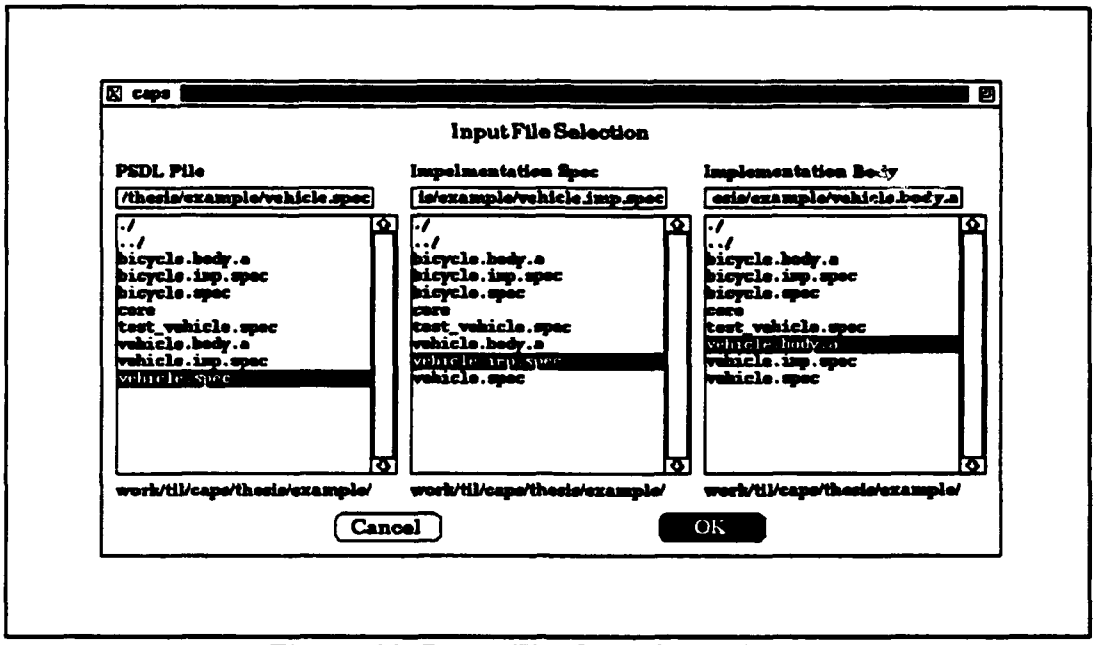


Figure 13. Input File Selection Window

After selecting the appropriate files followed by the OK button, storage of the PSDL class definition is achieved according to the automated mechanisms described in Chapter III. Continuing with our bicycle example the vehicle.spec, vehicle.imp.spec, and vehicle.body.a files correspond to the vehicle class definition. Similarly, the bicycle.spec, bicycle.imp.spec, and bicycle.body.a files correspond to the bicycle class definiton. Therefore they would be selected as triplets of files into the Input File Selector of the Software base.

2. Retrieval

a. By Query

Retrieval of stored classes by Query is the primary retrieval method of the CAPS Software base because it benefits the designer with an automated search

method of the stored classes. Selecting the Query menu choice of the Software base main window, the Query File Selection window (see Figure 14) appears on the screen prompting the user for a PSDL specification file based upon which the Software base is searched for syntactic matches. The matches (if any) appear on the Component Selection window (see Figure 15). Selecting one of the classes that matches causes its PSDL specification to appear on a View (see Figure 16), allowing the designer to examine the PSDL class definition and manipulate it in several usefull ways (such as print, save, and delete).

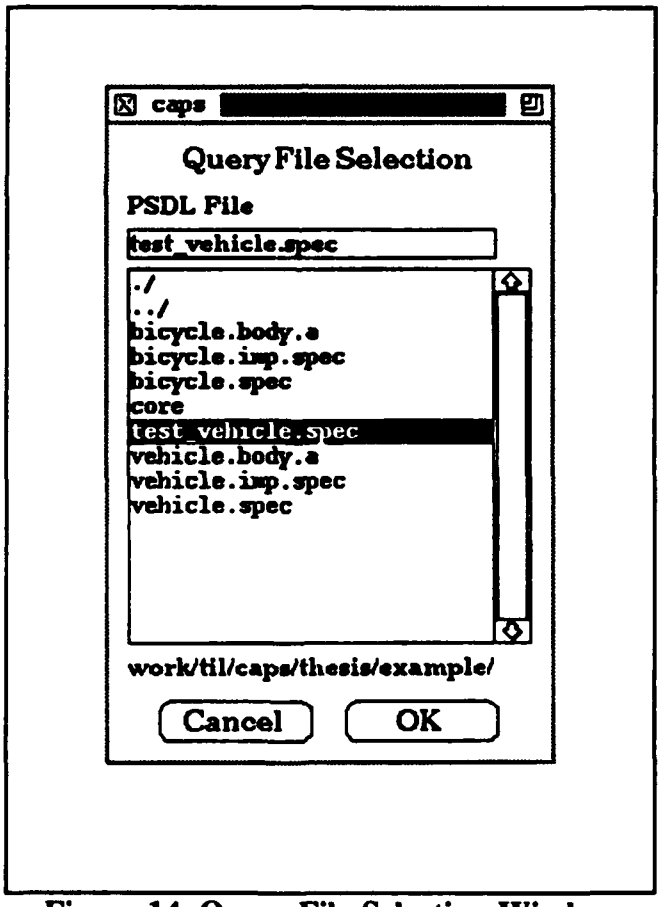


Figure 14. Query File Selection Window

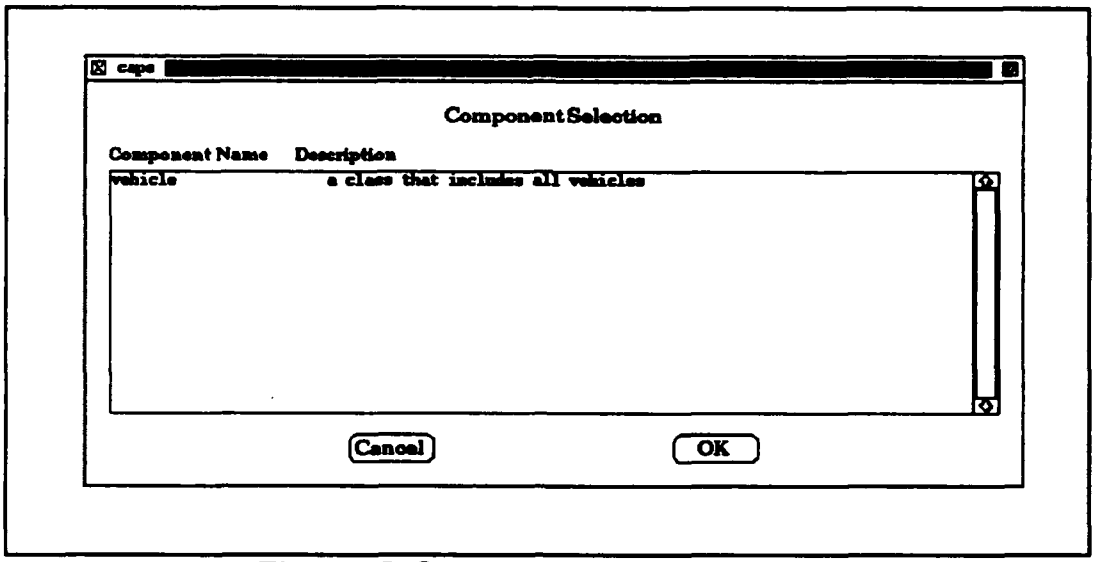


Figure 15. Component Selection Window

b. Browse

The Browse option is provided as an alternative method to search the Software base. It does not include a browsing capability like those of the examined browsers in Chapter II, but it offers adequate help to the designer that desires to manually search the Sotware base. It provides three submenu choices: search By Type, By Keyword, or By Operator.

(1) By Type. This selection causes the Component Selection window (see Figure 17) to appear on the screen again, but now it contains all of the stored Software base types (classes) in alphabetical order. Selecting a class causes its PSDL specification to appear on a view as previously described.

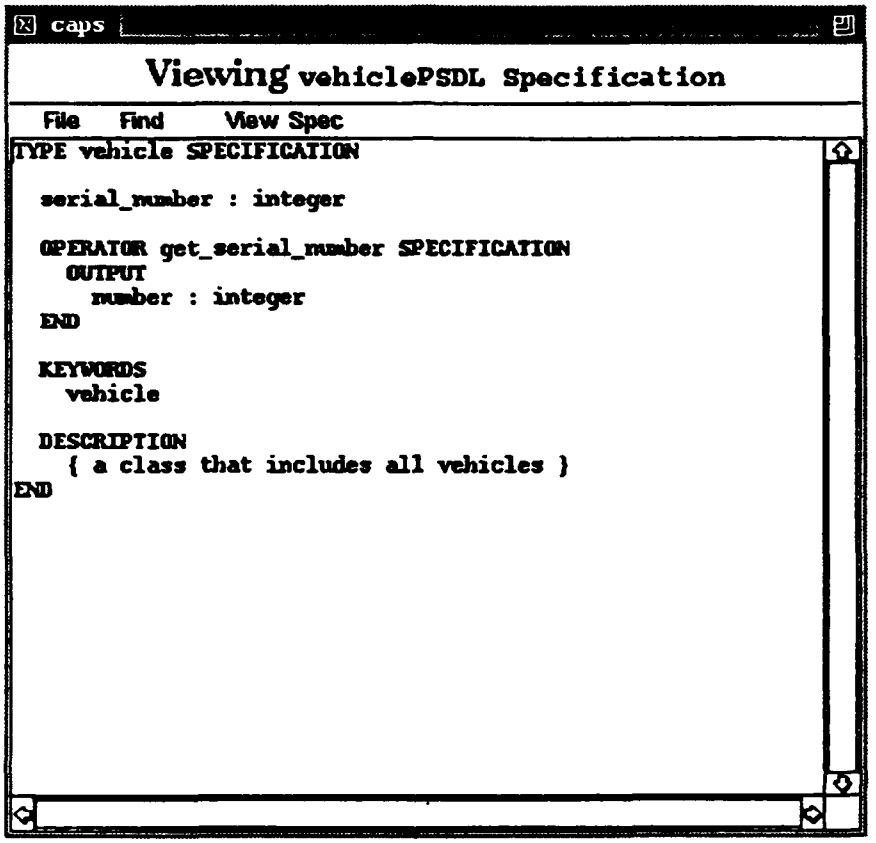


Figure 16. A Component's Specification View

(2) By Keyword. This selection causes the Keyword Selection Menu window that contains all keywords of all the stored Software base classes to appear on the screen (see Figure 18). Selecting a keyword causes the PSDL specification of the class(es) that include that keyword to appear in a View. In CSRS we include each class's superclasses in its keyword list; therefore, selecting a keyword that corresponds

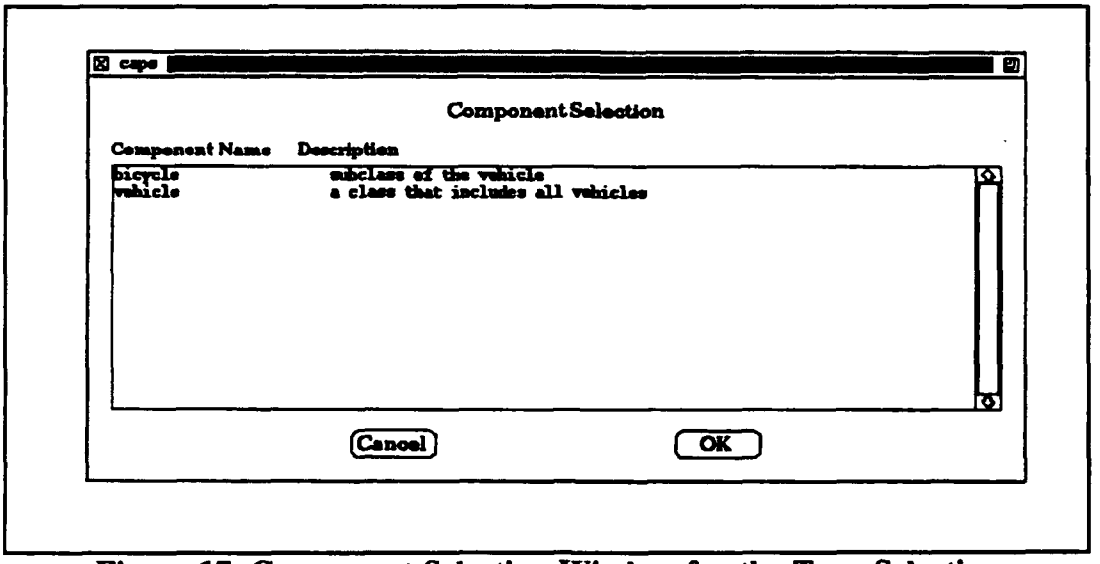


Figure 17. Component Selection Window for the Type Selection

to a class with a subclass will cause the PSDL specification of both classes to appear on the screen.

(3) By Operator. This selection causes a Component Selection Window that contains all of the stored Software base operators to appear on the screen. In CSRS a component (class) is implemented only as a data_type and never as an operator, so we do not use this option.

D. AN EXAMPLE APPLICATION: STORAGE/RETRIEVAL OF COMPUTER ARCHITECTURE SIMULATION CLASSES IN/FROM CSRS

In order to examine the feasibilty of CSRS we will now apply our solution to a previously developed set of OO classes. The Tanenbaum [Tan90] microarchitecture simulation classes [NFZ92, Font91] (see Appendices C,D) have been chosen for this purpose.

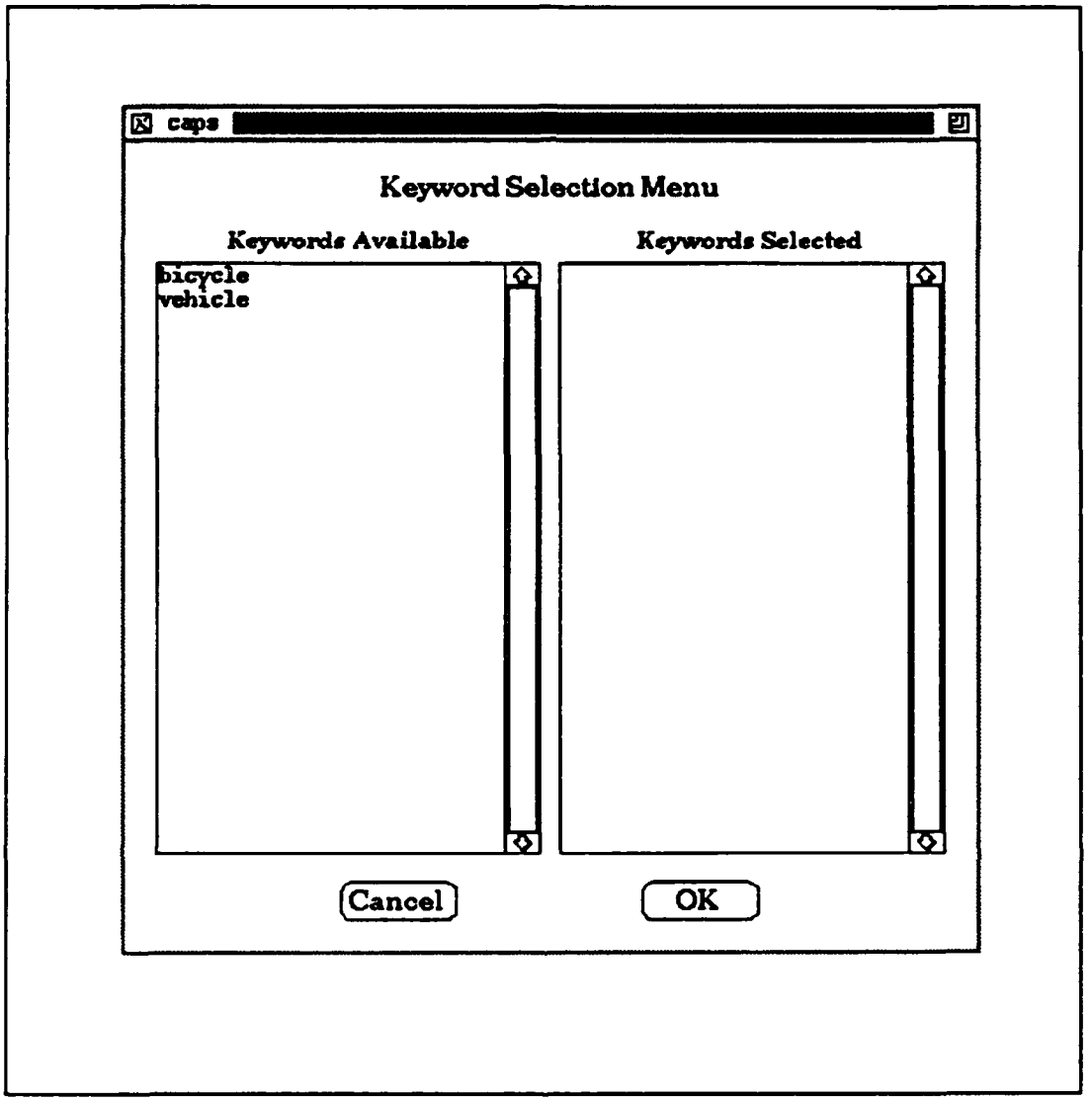


Figure 18. Keyword Selection Menu Window

1. Transforming Classes into PSDL Specifications

From the Tanenbaum microarchitecture simulation classes, two sets of PSDL specifications have been developed. The first (see Appendix E) is derived by applying the above described method of 'automatic' inheritance, and the second (see Appendix F) is derived applying the 'manual' inheritance method.

The Tanenbaum classes were developed using Prograph as the implementation language. We have attempted to transform them into PSDL specifications in such a way that Prograph's language dependent features are not lost. We now discuss this transformation.

All types of instance variables and input/output parameters of the methods that correspond to input/output data to the various components are of the general type string_of_bits, which is our adaptation of the actual Prograph code.

All input/output parameters of the developed methods that correspond to control signals are of type boolean if they are binary operations, or of type bit if they correspond to a one bit signal.

The methods **logical_and**, **logical_or**, **logical_not**, and **math** of the class **Alu** produce as outputs the n and z signals. However the n and z signals are provided by the **positive?**, and **zero?** methods of this class. For this reason it is assumed that whenever one of the three above methods is called, the underlying code of the method would call the methods **positive?** and **zero?** to provide the necessary outputs. This approach is similar to the concept of public and private methods that some OOP languages provide. The methods **logical_and**, **logical_or**, **logical_not**, and **math** are public methods and the methods **positive?** and **zero?** are the private methods of the class **alu**.

The **Alu_shifter** class inherits all methods of the **Alu** class without any modifications. It then specifies three more methods for the necessary functionality of a shifter.

The **Control_store**, **Memory_bank**, and **Register_bank** classes inherit the methods of **Storage_bank** class and modify the type of the inherited instance variable **contents** from array_of_storage_locations to array_of_microinstructions, and from array_of_memory_locations to array_of_registers, respectivelly.

The **Memory_location** and **Register** classes inherit all code from the **Storage_location** class without any additions or modifications.

The **Mir** class modifies the type of the inherited instance variable **contents** from string_of_bits to microinstruction in order to be able to hold an entire microinstruction as it has been defined by the superclass.

The **Mar**, **Mbr**, and **Mpc** classes inherit all the code of **Register** class without any modifications. However, additional methods are specified for the **Mpc** class.

2. Storage/Retrieval of Classes in/from CSRS

The storage and retrieval of computer architecture simulation class definitions in/from the Software base is achieved according to the mechanisms described in the previous section.

Every PSDL specification file is accompagnied by an empty implememtation specification file and an empty implementation body file (see Figure 19) before being stored in the Software base.

When the user needs to automatically search the Software base for a particular class, a PSDL specification of that class is written for use with the previously described Query mechanism for possible matches. As an example, Figure 20 shows the

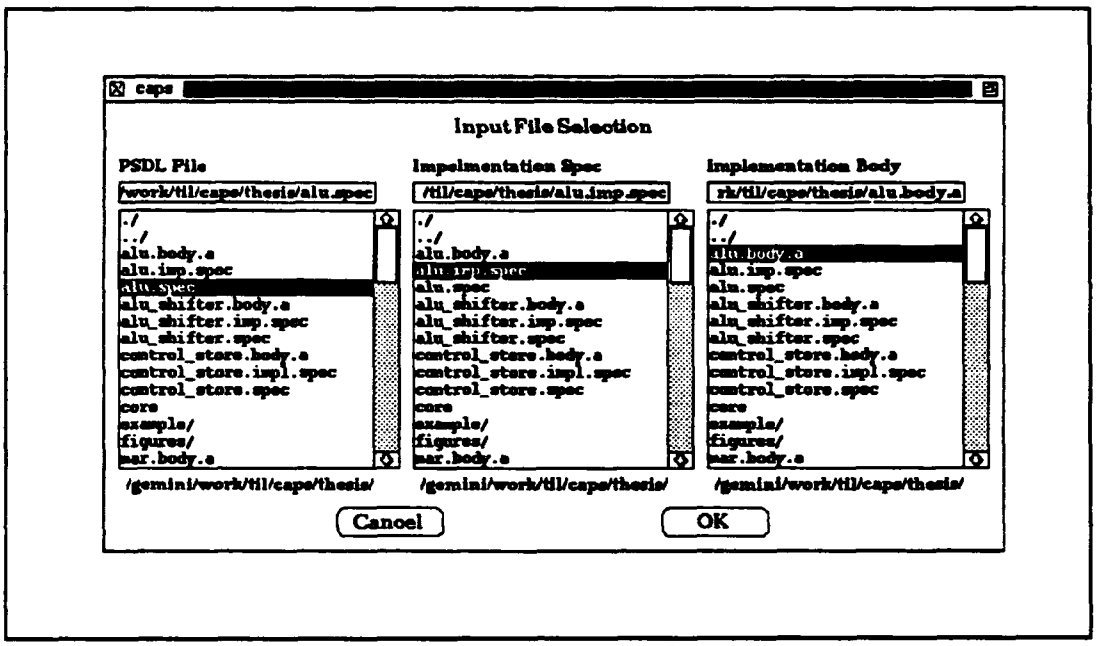


Figure 19. Storing a Class

classes that match the following PSDL specification:

```
TYPE Test_mux SPECIFICATION

  OPERATOR test_multiplexer SPECIFICATION
    INPUT
      inc : string
      mir : string
      sign : signal
    OUTPUT
      out : string
    END

  KEYWORDS
    test_mux
  DESCRIPTION
    { test class for the mux class }
END
```

When the user needs to manually search the Software base, either the Keyword Selection Menu window or the Component Selection window may be used to

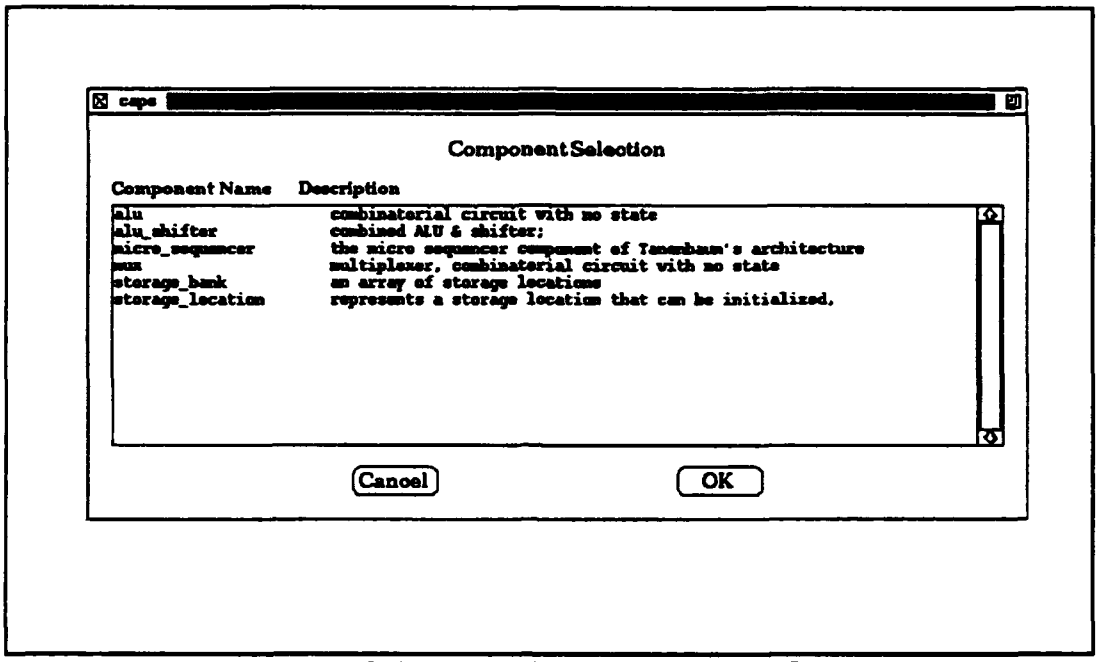


Figure 20. Matches for the Test_mux Class

provide information about the keywords of each class or offer all of the stored classes, respectivelly.

E. ANOMALIES

After the development of the PSDL class definitions for the Tanenbaum microarchitecture simulation classes and their storage in CSRS, the following anomalies have been encountered:

1. The Update Component submenu of the File menu choice of the Software base main window does not work. If the designer needs to modify a PSDL class definition it must be deleted from the Software base and then the modified definition is added using the Add Component submenu.

2. Selecting a keyword (class) in the Keyword Selection Menu that belongs to more than one class causes the system to respond with an error message. The same error occurs when selecting two different keywords that belong to two different classes.

Because CAPS is an on going research project these anomalies are not considered to be serious at this time as they should be eliminated in some future version of the system.

V. SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE RESEARCH

A. SUMMARY

Chapter I of this thesis introduced the increasing need for reusable software, object-orientation as a new approach to serve this need, and the Computer Aided Prototyping System as a system that could be used in the storage and retrieval of class definitions in order to enhance their reusability.

In Chapter II the fundamental principles of object-oriented programming were described independently of any specific implementation language, and then reusability in both conventional software systems and object-oriented environments was examined. Because a browser is the only actual tool that most OO environments provide to enhance the reusability of class definitions, we also examined the browsers of several of the more common OOP systems. Finally an extensive description of the Computer Aided Prototyping System, and especially of its automated storage and retrieval mechanism for the software components, was given.

In Chapter III we concluded that the main disadvantage of the browser approach is that the time required to manually search the class library increases considerably as the number of classes in the library increases. This disadvantage has stimulated our research effort towards the direction of CAPS because it provides an automated search

mechanism for the library of the stored components that in our case could be used to automatically search previously stored class definitions.

Because CAPS requires the specification of the components to be written in PSDL before stored in the Software base, transformation rules have been developed as explained in Chapter IV, creating the Class Storage and Retrieval System. These rules transform a class definition into a PSDL specification creating proper PSDL grammar while preserving the fundamental concepts of the class definition. The transformed class definition can now be stored in a standardized form in the Software base, benefiting from its syntactic matching rules whenever a query for a particular class is posed to it. Finally, an example application demonstrating the entire process of transforming a set of classes into PSDL specifications in order to store them in the Software base and then using the automated search mechanism whenever the designer needs to know if a particular class exists in the class library.

B. CONCLUSIONS

The browser approach as implemented in most OO systems today fails to meet software development needs because it obligates the user to manually search a class library for candidate classes. This increases software development time, especially when the number of classes in the library increases beyond a relatively small number.

The Class Storage and Retrieval System enhances the reusability of class definitions and reduces software development time giving the designer a powerful tool that automatically selects the candidate classes from a class library (i.e., those classes

that have a functional description similar to the one that is needed). The benefit of this approach becomes even more apparent as the number of classes in the class library increases because more classes are eliminated from the manual search, thus reducing the search component of the overall software development cost. Therefore, we conclude that the use of CSRS greatly enhances the promise of reusability in OO environments.

Transforming a class definition into a PSDL specification using the developed transformation rules is a straight forward process that adds very little overhead and does not pose any difficulties even though PSDL was developed for completely different applications.

C. RECOMMENDATIONS FOR FUTURE RESEARCH

The Class Storage and Retrieval System serves to store and automatically retrieve class definitions. Suggested improvements to the system include the following:

Make the system capable of including class variables' representation in the PSDL class definition. This could be achieved by modifying the PSDL grammar rules.

The development of a system that would automatically scan the class definitions of the specified superclass(es) in the id_list of the keywords of a PSDL class definition to see if more superclasses are specified in their "keywords" id_list (i.e., implement the 'automatic' inheritance method introduced in Chapter IV). In this way the system would "walk up" the class hierarchy, and it would therefore be possible to automatically inherit variables and methods from a class' ancestors.

The development of a browser with the standard manual browsing capabilities described in Chapter II that would browse the set of classes that consist a match for a specific Query of the Software base. This would allow the CSRS to provide a set of candidate classes which could then be manually searched using a conventional browser approach.

Finally, if OO languages (such as C++, Smalltalk, etc.) are supported by CAPS in the future, then actual code could be included in the implementation specification and body files. This would extend the usefulness of CSRS to support execution of the designed classes, thus achieving a complete system.

APPENDIX A

This Appendix contains the PSDL grammar [LBY88].

****

Optional items are enclosed in [square brackets]. Items which may appear zero or more times appear in { braces }. Terminal symbols appear in " double quotes ". Groupings appear in (parentheses).

****

```
psdl
      = {component}

component
      = data_type
      | operator

data_type
      = "type" id type_spec type_impl

type_spec
      = "specification" ["generic" type_decl] [type_decl]
        {"operator" id operator_spec}
        [functionality] "end"

operator
      = "operator" id operator_spec operator_impl

operator_spec
      = "specification" {interface} [functionality] "end"

interface
      = attribute [reqmts_trace]

attribute
      = "generic" type_decl
```

```
        | "input" type_decl
        | "output" type_decl
        | "states" type_decl "initially" initial_expression_list
        | "exceptions" id_list
        | "maximum execution time" time

type_decl
        = id_list ":" type_name {"," id_list ":" type_name}

type_name
        = id
        | id "[" type_decl "]"

id_list
        = id {"," id}

reqmts_trace
        = "required by" id_list

functionality
        = [keywords] [informal_desc] [formal_desc]

keywords
        = "keywords" id_list

informal_desc
        = "description" "{" text "}"

formal_desc
        = "axioms" "{" text "}"

type_impl
        = "implementation ada" id "end"
        | "implementation" type_name {"operator" id operator_impl} "end"

operator_impl
        = "implementation ada" id "end"
        | "implementation" psdl_impl "end"

psdl_impl
        = data_flow_diagram [streams] [timers] [control_constraints]
          [informal_desc]
```

```
data_flow_diagram
      = "graph" {vertex} {edge}

vertex
      = "vertex" op_id [":" time]
        -- time is the maximum execution time

edge
      = "edge" id [":" time] op_id "->" op_id
        -- time is the latency

op_id
      = id ["(" [id_list] "|" [id_list] ")"]

streams
      = "data stream" type_decl

timers
      = "timer" id_list

control_constraints
      = "control constraints" constraint {constraint}

constraint
      = "operator" op_id
        ["triggered" [trigger] ["if" expression] [reqmts_trace]]
        ["period" time [reqmts_trace]]
        ["finish within" time [reqmts_trace]]
        ["minimum calling period" time [reqmts_trace]]
        ["maximum response time" time [reqmts_trace]]
        {constraint_options}

constraint_options
      = "output" id_list "if" expression [reqmts_trace]
      | "exception" id ["if" expression] [reqmts_trace]
      | timer_op id ["if" expression] [reqmts_trace]

trigger
      = "by all" id_list
      | "by some" id_list

timer_op
```

```
        = "reset timer"
        | "start timer"
        | "stop timer"

initial_expression_list
        = initial_expression {"," initial_expression}

initial_expression
        = "true"
        | "false"
        | integer_literal
        | real_literal
        | string_literal
        | id
        | type_name "." id ["(" initial_expression_list ")"]
        | "(" initial_expression ")"
        | initial_expression binary_op initial_expression
        | unary_op initial_expression

binary_op
        = "and" | "or" | "xor"
        | "<" | ">" | "=" | ">=" | "<=" | "/="
        | "+" | "-" | "&" | "*" | "/" | "mod" | "rem" | "**"

unary_op
        = "not" | "abs" | "-" | "+"

time
        = integer_literal unit

unit
        = "microsec"
        | "ms"
        | "sec"
        | "min"
        | "hours"

expression_list
        = expression {"," expression}

expression
        = "true"
```

```
        | "false"
        | integer_literal
        | time
        | real_literal
        | string_literal
        | id
        | type_name "." id ["(" expression_list ")"]
        | "(" expression ")"
        | initial_expression binary_op initial_expression
        | unary_op initial_expression

id
        = letter {alpha_numeric}

real_literal
        = integer_literal "." integer_literal

integer_literal
        = digit {digit}

string_literal
        = """ {char} """

char
        = any printable character except "}"

digit
        = "0 .. 9"

letter
        = "a .. z"
        | "A .. Z"
        | "_"

alpha_numberic
        = letter
        | digit

text
        = {char}
```

APPENDIX B

This Appendix contains the subset of the PSDL grammar that is used by our system.

****

Optional items are enclosed in [square brackets]. Items which may appear zero or more times appear in { braces }. Terminal symbols appear in " double quotes ". Groupings appear in (parentheses).

****

```
psdl
      = {component}

component
      = data_type
      | operator

data_type
      = "type" id type_spec type_impl

type_spec
      = "specification" ["generic" type_decl] [type_decl]
        {"operator" id operator_spec}
        [functionality] "end"

operator_spec
      = "specification" {interface} [functionality] "end"

interface
      = attribute [reqmts_trace]

attribute
      = "generic" type_decl
      | "input" type_decl
```

```
      | "output" type_decl
      | "states" type_decl "initially" initial_expression_list
      | "exceptions" id_list
      | "maximum execution time" time

type_decl
      = id_list ":" type_name {"," id_list ":" type_name}

type_name
      = id
      | id "[" type_decl "]"

id_list
      = id {"," id}

functionality
      = [keywords] [informal_desc] [formal_desc]

keywords
      = "keywords" id_list

informal_desc
      = "description" "{" text "}"
```

APPENDIX C

This Appendix contains the Tanenbaum microarchitecture simulation classes [NFZ92, Font91].

Class : Alu
Superclasses : none
Variables : none
Methods : and, or, math, zero?, positive?

Class : Alu_shifter
Superclasses : Alu
Variables : none
Methods : shift_left, shift_right, no_shift

Class : Control_store
Superclasses : Storage_bank
Variables : contents : array_of_microinstructions
Methods : none

Class : Mar
Superclasses : Register
Variables : none
Methods : mar

Class : Mbr
Superclasses : Register
Variables : none
Methods : mbr

Class : Memory_bank
Superclasses : Storage_bank
Variables : contents : array_of_memory_locations
Methods : none

Class : Memory_location
Superclasses : storage_location
Variables : none
Methods : none

Class : Micro_sequencer
Superclasses : none
Variables : none
Methods : generate_signal

Class : Microinstruction
Superclasses : none
Variables : instruction : string_of_bits
Methods : none

Class : Mir
Superclasses : Register
Variables : contents : microinstruction
Methods : decode

Class : Mpc
Superclasses : Register
Variables : none
Methods : set, increment, jump

Class : Mux
Superclasses : none
Variables : none
Methods : mux

Class : Register
Superclasses : Storage_location
Variables : none
Methods : none

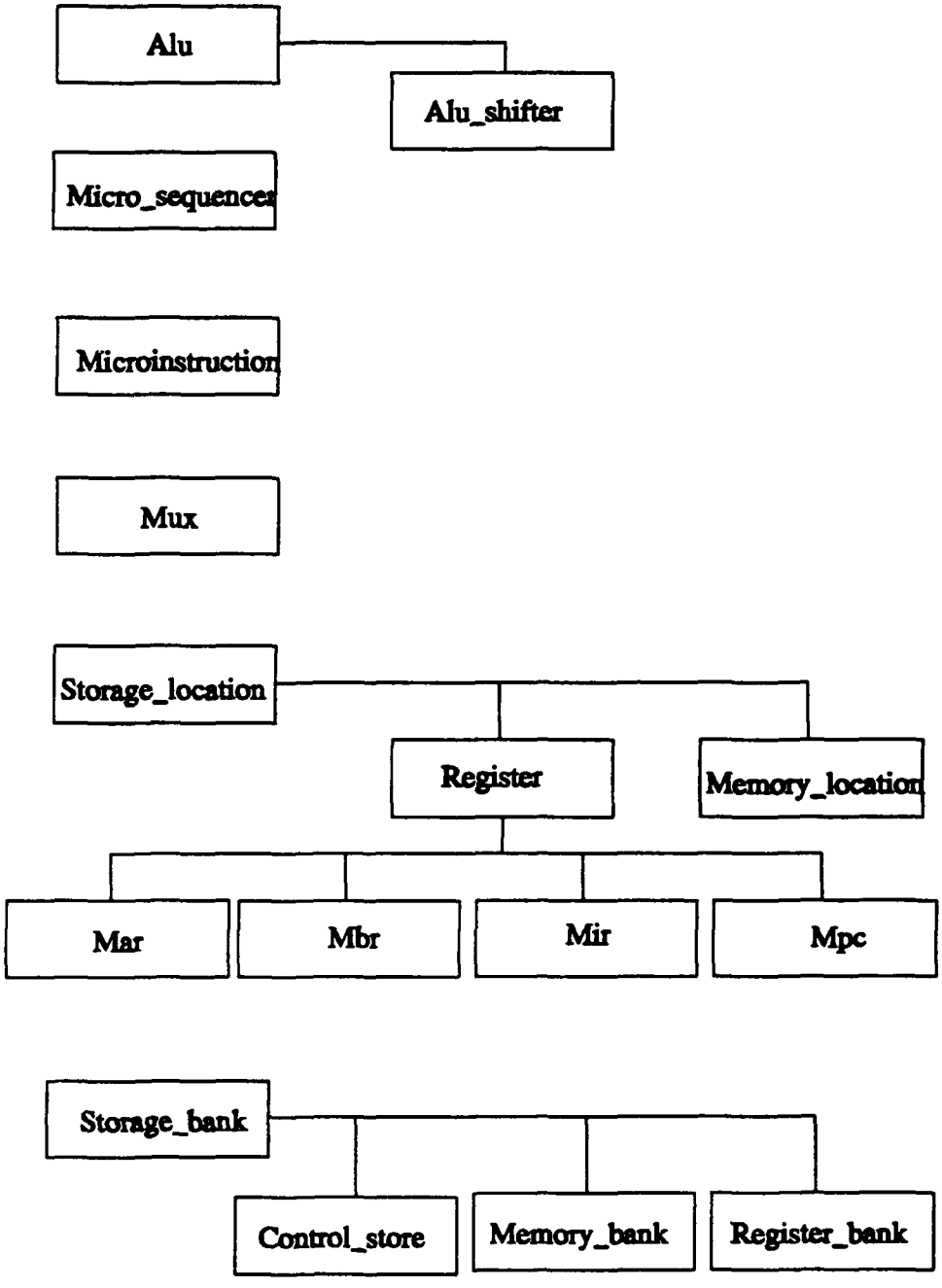
Class : Register_bank
Superclasses : Storage_bank
Variables : contents : array_of_registers
Methods : none

Class : Storage_bank
Superclasses : none
Variables : array_of_storage_locations
Methods : read, write, initialize, load

Class : Storage_location
Superclasses : none
Variables : contents : string_of_bits
Methods : initialize, read, write, binary_read, binary_write

APPENDIX D

This Appendix presents the class hierarchy of the Tanenbaum microarchitecture simulation (adapted from [NFZ92, Font91]).



APPENDIX E

This Appendix contains the PSDL specifications for the Tanenbaum microarchitecture simulation classes that have derived applying the 'automatic' inheritance.

```
TYPE Alu SPECIFICATION

 OPERATOR logical_and SPECIFICATION
  INPUT
   input_one : string_of_bits,
   input_two : string_of_bits
  OUTPUT
   result : string_of_bits,
 END

 OPERATOR logical_or specification
  INPUT
   input_one :string_of_bits,
   input_two :string_of_bits
  OUTPUT
   result : string_of_bits,
  END

 OPERATOR logical_not SPECIFICATION
  INPUT
   input_one : string_of_bits
  OUTPUT
   out_inverted : string_of_bits,
 END

 OPERATOR math SPECIFICATION
  INPUT
   input_one : string_of_bits,
   input_two : string_of_bits
  OUTPUT
```

```
    result : string_of_bits,
  END

  OPERATOR zero? SPECIFICATION
   OUTPUT
    z : boolean
  END

  OPERATOR positive? SPECIFICATION
   OUTPUT
    n : boolean
  END

  KEYWORDS
   alu

  DESCRIPTION
   { combinatorial circuit with no state }
END
```

```
TYPE Alu_shifter SPECIFICATION

  OPERATOR shift_left SPECIFICATION
   INPUT
    data : string_of_bits
   OUTPUT
    left_shifted_data :string_of_bits
  END

  OPERATOR shift_right SPECIFICATION
   INPUT
    data : string_of_ ts
   OUTPUT
    right_shifted_data : string_of_bits
  END
```

```
  OPERATOR no_shift SPECIFICATION
    INPUT
      data : string_of_bits
    OUTPUT
      data : string_of_bits
  END

  KEYWORDS
    alu, alu_shifter

  DESCRIPTION
  { combined ALU & shifter;
    shifts functions added to inherited ALU functions }
END



TYPE Control_store SPECIFICATION

  contents : array_of_microinstructions

  KEYWORDS
    storage_bank, control_store

  DESCRIPTION
    { contains the microprogram which must be loaded
      before the simulation begins }
END



TYPE Mar SPECIFICATION

  OPERATOR memory_address_register SPECIFICATION
    INPUT
      signal : control_signal
  END

  KEYWORDS
```

```
    register, mar

  DESCRIPTION
    { a memory_location within a memory_bank }
END



TYPE Mbr SPECIFICATION

  OPERATOR memory_buffer_register SPECIFICATION
    INPUT
      signal : control_signal
  END

  KEYWORDS
    register, mbr

  DESCRIPTION
    { interface to the memory_bank }
END



TYPE Memory_bank SPECIFICATION

  contents : array_of_memory_locations

  KEYWORDS
    storage_bank, memory_bank

  DESCRIPTION
    { an array of memory locations }
END
```

```
TYPE Memory_location SPECIFICATION

  KEYWORDS
    storage_location,memory_location

  DESCRIPTION
    { the smallest individually addressable memory unit }
END
```

```
TYPE Micro_sequencer SPECIFICATION

  OPERATOR generate_signal SPECIFICATION
    INPUT
      n : bit,
      z : bit,
      mir_cond : string_of_bits
    OUTPUT
      out_mux : control_signal
  END

  KEYWORDS
    microsequencer

  DESCRIPTION
    { the micro sequencer component of Tanenbaum's architecture }
END
```

TYPE **Microinstruction** SPECIFICATION

instruction : string_of_bits

KEYWORDS
microinstuction

DESCRIPTION
{ defines the various fields necessary to make up a microprogram instruction }
END

TYPE **Mir** SPECIFICATION

contents : microinstuction

OPERATOR decode SPECIFICATION

END

KEYWORDS
register, mir

DESCRIPTION
{ contains control signals for routing to other components }

END

```
TYPE Mpc SPECIFICATION

  OPERATOR set SPECIFICATION
    INPUT
      location : microinstruction_address
  END

  OPERATOR increment SPECIFICATION

  END

  OPERATOR jump SPECIFICATION
    INPUT
      location : microinstruction_address
  END

  KEYWORDS
    register, mpc

  DESCRIPTION
    { microprogram counter }

END
```

```
TYPE Mux SPECIFICATION

  OPERATOR multiplexer SPECIFICATION
    INPUT
      inc_addr : string_of_bits,
      mir_addr : string_of_bits,
      signal : control_signal
    OUTPUT
      out : string_of_bits
  END
```

```
  KEYWORDS
    mux

  DESCRIPTION
    { multiplexer, combinatorial circuit with no state }
END
```

```
 TYPE Register SPECIFICATION

  KEYWORDS
    storage_location, register

  DESCRIPTION
    { essentially the same as a memory_location;
      use,speed, and possibly size are the major differences }

END
```

```
TYPE Register_bank SPECIFICATION

  contents : array_of_registers

  KEYWORDS
    storage_bank, register_bank

  DESCRIPTION
    { an array of registers }
END
```

```
TYPE Storage_bank SPECIFICATION

 contents : array_of_storage_locations

 OPERATOR read SPECIFICATION
   INPUT
     location : memory_location_address
   OUTPUT
    element : string_of_bits
 END

 OPERATOR write SPECIFICATION
  INPUT
    element : string_of_bits,
    location : memory_location_address
 END

 OPERATOR initialize SPECIFICATION
  INPUT
    init : array_of_string_of_bits
  END

 OPERATOR load SPECIFICATION
  INPUT
    element : string_of_bits,
    location : memory_location_address
 END

 KEYWORDS
  storage_bank

 DESCRIPTION
   { an array of storage locations }
END
```

```
TYPE Storage_location SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 KEYWORDS
  storage_location

 DESCRIPTION
  { represents a storage location that can be initialized,
   read, or written in either integer or binary form }
END
```

APPENDIX F

This Appendix contains the PSDL specifications for the Tanenbaum microarchitecture simulation classes that have derived from applying the 'manual' inheritance.

```
TYPE Alu SPECIFICATION

 OPERATOR logical_and SPECIFICATION
  INPUT
   input_one : string_of_bits,
   input_two : string_of_bits
  OUTPUT
   result : string_of_bits,
 END

 OPERATOR logical_or specification
  INPUT
   input_one :string_of_bits,
   input_two :string_of_bits
  OUTPUT
   result : string_of_bits,
  END

 OPERATOR logical_not SPECIFICATION
  INPUT
   input_one : string_of_bits
  OUTPUT
   out_inverted : string_of_bits,
 END
```

```
  OPERATOR math SPECIFICATION
    INPUT
      input_one : string_of_bits,
      input_two : string_of_bits
    OUTPUT
      result : string_of_bits,
  END

  OPERATOR zero? SPECIFICATION
    OUTPUT
      z : boolean
  END

  OPERATOR positive? SPECIFICATION
    OUTPUT
      n : boolean
  END

  KEYWORDS
    alu

  DESCRIPTION
    { combinatorial circuit with no state }
END
```

```
TYPE Alu_shifter SPECIFICATION

 OPERATOR logical_and SPECIFICATION
  INPUT
   input_one : string_of_bits,
   input_two : string_of_bits
  OUTPUT
   result : string_of_bits,
 END

 OPERATOR logical_or specification
  INPUT
   input_one :string_of_bits,
   input_two :string_of_bits
  OUTPUT
   result : string_of_bits,
  END

 OPERATOR logical_not SPECIFICATION
  INPUT
   input_one : string_of_bits
  OUTPUT
   out_inverted : string_of_bits,
 END

 OPERATOR math SPECIFICATION
  INPUT
   input_one : string_of_bits,
   input_two : string_of_bits
  OUTPUT
   result : string_of_bits,
 END

 OPERATOR zero? SPECIFICATION
  OUTPUT
   z : boolean
 END

 OPERATOR positive? SPECIFICATION
  OUTPUT
   n : boolean
 END
```

```
OPERATOR shift_left SPECIFICATION
  INPUT
    data : string_of_bits
  OUTPUT
    left_shifted_data :string_of_bits
END

OPERATOR shift_right SPECIFICATION
  INPUT
    data : string_of_bits
  OUTPUT
    right_shifted_data : string_of_bits
END

OPERATOR no_shift SPECIFICATION
  INPUT
    data : string_of_bits
  OUTPUT
    data : string_of_bits
END

KEYWORDS
  alu, alu_shifter

DESCRIPTION
{ combined ALU & shifter;
  shifts functions added to inherited ALU functions }
END
```

```
TYPE Control_store SPECIFICATION

 contents : array_of_microinstructions

 OPERATOR read SPECIFICATION
   INPUT
     location : memory_location_address
   OUTPUT
    element : string_of_bits
 END

 OPERATOR write SPECIFICATION
  INPUT
    element : string_of_bits,
    location : memory_location_address
 END

 OPERATOR initialize SPECIFICATION
  INPUT
    init : array_of_string_of_bits
  END

 OPERATOR load SPECIFICATION
  INPUT
    element : string_of_bits,
    location : memory_location_address
 END

 KEYWORDS
  storage_bank, control_store

 DESCRIPTION
  { contains the microprogram which must be loaded
    before the simulation begins }
END
```

```
TYPE Mar SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 OPERATOR memory_address_register SPECIFICATION
  INPUT
   signal : control_signal
 END

 KEYWORDS
  register, mar

 DESCRIPTION
  { a memory_location within a memory_bank }
END
```

```
TYPE Mbr SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 OPERATOR memory_buffer_register SPECIFICATION
  INPUT
   signal : control_signal
 END

 KEYWORDS
  register, mbr

 DESCRIPTION
   { interface to the memory_bank }
END
```

```
TYPE Memory_bank SPECIFICATION

 contents : array_of_memory_locations

 OPERATOR read SPECIFICATION
   INPUT
     location : memory_location_address
   OUTPUT
    element : string_of_bits
 END

 OPERATOR write SPECIFICATION
  INPUT
    element : string_of_bits,
    location : memory_location_address
 END

 OPERATOR initialize SPECIFICATION
  INPUT
    init : array_of_string_of_bits
  END

 OPERATOR load SPECIFICATION
  INPUT
    element : string_of_bits,
    location : memory_location_address
 END

 KEYWORDS
  storage_bank, memory_bank

 DESCRIPTION
   { an array of memory locations }
END
```

```
TYPE Memory_location SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 KEYWORDS
  storage_location,memory_location

 DESCRIPTION
  { the smallest individually addressable memory unit }
END
```

```
TYPE Micro_sequencer SPECIFICATION

  OPERATOR generate_signal SPECIFICATION
    INPUT
      n : bit,
      z : bit,
      mir_cond : string_of_bits
    OUTPUT
      out_mux : control_signal
  END

  KEYWORDS
    microsequencer

  DESCRIPTION
    { the micro sequencer component of Tanenbaum's architecture }
END
```

```
TYPE Microinstruction SPECIFICATION

  instruction : string_of_bits

  KEYWORDS
    microinstuction

  DESCRIPTION
    { defines the various fields necessary to make up a microprogram instruction }
END
```

```
TYPE Mir SPECIFICATION

 contents : microinstuction

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 OPERATOR decode SPECIFICATION

 END

 KEYWORDS
  register, mir

 DESCRIPTION
  { contains control signals for routing to other components }

END
```

```
TYPE Mpc SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 OPERATOR set SPECIFICATION
  INPUT
   location : microinstruction_address
 END

 OPERATOR increment SPECIFICATION

 END

 OPERATOR jump SPECIFICATION
  INPUT
   location : microinstruction_address
 END
```

```
  KEYWORDS
    register, mpc

  DESCRIPTION
    { microprogram counter }

END
```

```
TYPE Mux SPECIFICATION

  OPERATOR multiplexer SPECIFICATION
    INPUT
      inc_addr : string_of_bits,
      mir_addr : string_of_bits,
      signal : control_signal
    OUTPUT
      out : string_of_bits
  END

  KEYWORDS
    mux

  DESCRIPTION
    { multiplexer, combinatorial circuit with no state }
END
```

```
TYPE Register SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
    init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
    element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
    element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
    binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
    binary_element : string_of_bits
 END

 KEYWORDS
  storage_location, register

 DESCRIPTION
  { essentially the same as a memory_location;
    use,speed,and possibly size are the major differences }

END
```

```
TYPE Register_bank SPECIFICATION

  contents : array_of_registers

  OPERATOR read SPECIFICATION
    INPUT
      location : memory_location_address
    OUTPUT
     element : string_of_bits
  END

  OPERATOR write SPECIFICATION
   INPUT
     element : string_of_bits,
     location : memory_location_address
  END

  OPERATOR initialize SPECIFICATION
   INPUT
     init : array_of_string_of_bits
   END

  OPERATOR load SPECIFICATION
   INPUT
     element : string_of_bits,
     location : memory_location_address
  END

  KEYWORDS
   storage_bank, register_bank

  DESCRIPTION
    { an array of registers }
END
```

```
TYPE Storage_bank SPECIFICATION

  contents : array_of_storage_locations

  OPERATOR read SPECIFICATION
    INPUT
      location : memory_location_address
    OUTPUT
     element : string_of_bits
  END

  OPERATOR write SPECIFICATION
    INPUT
      element : string_of_bits,
      location : memory_location_address
  END

  OPERATOR initialize SPECIFICATION
    INPUT
      init : array_of_string_of_bits
    END

  OPERATOR load SPECIFICATION
    INPUT
      element : string_of_bits,
      location : memory_location_address
  END

  KEYWORDS
    storage_bank

  DESCRIPTION
    { an array of storage locations }
END
```

```
TYPE Storage_location SPECIFICATION

 contents : string_of_bits

 OPERATOR initialize SPECIFICATION
  INPUT
   init : string_of_bits
 END

 OPERATOR read SPECIFICATION
  OUTPUT
   element : integer
 END

 OPERATOR write SPECIFICATION
  INPUT
   element : integer
 END

 OPERATOR binary_read SPECIFICATION
  OUTPUT
   binary_element : string_of_bits
 END

 OPERATOR binary_write SPECIFICATION
  INPUT
   binary_element : string_of_bits
 END

 KEYWORDS
  storage_location

 DESCRIPTION
   { represents a storage location that can be initialized,
     read, or written in either integer or binary form }
END
```

LIST OF REFERENCES

[Ande88] Air Force Armament Lab Technical Report, *The CAMP Approach: A Pragmatic Approach to Software Reuse*, by C. Anderson, 1988.

[Boa84] Boar, B.H., *Application Prototyping: A Requirements Definition Strategy For the 80's*, John Wiley & Sons, Inc., 1984.

[Booc91] Booch, G., *Object-Oriented Design: with applications*, The Benjamin/Cummings Publishing Company, Inc., 1991.

[Bor91] *Borland C++ Version 3.0 User's Guide*, Borland International Inc., Scotts Valley, CA, 1991.

[BR87] Biggerstaff, T.J., and Richter C., "Reusability Framework, Assessment, and Directions", *IEEE Software*, Vol.4, No2, March 1987.

[Bud91] Budd, T., *Object-Oriented Programming*, Addison-Wesley, April 1991.

[BW87] Burton, B.A., Wienk, R., and others, "The Reusable Software Library", *IEEE Software*, Vol.4, pp.25-33, July 1987.

[CAMP89] Air Force Armanment Laboratory, Contract F08635-88-C-0002, CDRL No. A009, *CAMP Parts Engineering System Catalog User's Guide*, McDonnell Douglas Missile Systems Company, 30 November 1989.

[Dig88] *Smalltalk/V 286 Tutorial and Programming Handbook*, Digitalk Inc., 1988.

[Gold84] Goldberg, A., *Smalltalk-80 The Interactive Programming Environment*, Addison-Wesley, 1984.

[Gun90a] *Prograph Tutorial*, The Gunakara Sun Systems Ltd, Halifax, Nova Scotia, 1990.

[Gun90b] *Prograph Reference*, The Gunakara Sun Systems Ltd, Halifax, Nova Scotia, 1990.

[Gun91] *Prograph 2.5 Updates*, The Gunakara Sun Systems Ltd, Halifax, Nova Scotia, 1991.

[GW88] Goguen, J.A., and Winkler, T., "Introducing OBJ3", *SRI International Report SRI-CSL-88-9*, August 1988.

[EN89] Elmasri, R. and Navathe, S.B., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.

[FG89] Frakes, W.B., and Gandel, P.B., "Representation Methods for Software Reuse", *Proceedings of Tri-Ada '89*, Pittsburgh, PA., ACM, pp 302-314, 1989.

[Font91] Fontes, K.A., *Object-Oriented Approach to Computer Architecture Simulation*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

[KA90] Khoshafian, S., and Abnous, R., *Object-Orientation: Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, Inc., 1990.

[Kim90] Kim, W., "Object-Oriented Databases: Definition and Research Directions", *IEEE Transactions on Knowledge and Data Engineering*, Vol.2 No.3, pp 327-341, September 1990.

[KL89] Kim, W., and Lochovsky, F., *Object-Oriented Concepts, Databases and Applications*, ACM Press, New York, 1989.

[LBY88] Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering* Vol.14, No.10, pp 1409-1423, October 1988.

[LP91] LaLonde, W., and Pugh, J. "Smalltalk: Subclassing ≠ Subtyping ≠ Is-a", *JOOP*, Vol.3, No 5, pp 57-62, January 1991.

[Luqi91] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer*, Vol.24, pp 111-112, September 1991.

[LK88] Luqi, and Ketabchi, M.A., "A Computer Aided Prototyping System" *IEEE Software*, pp 66-72, March 1988.

[McDo91] McDowell, J.K., *A Reusable Component Retrieval System For Prototyping*, Master's Thesis, Naval Postgraduate School, September 1991.

[Meye88] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall,Inc, Englewood Cliffs, NJ, 1988.

[Mica88] Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *JOOP*, Vol.1, No 1, pp 12-34, April/May 1988.

[Mul90] Mullin, M., *Rapid Prototyping for Object-Oriented Systems*, Addison-Wesley Publishing Company, Inc., July 1990.

[NB92] Nelson, M.L., Byrnes, R.B., "Rapid Prototyping in an Object-Oriented Pictorial Dataflow Environment", *Proceedings of the 25th Annual Hawaii International Conference on System Sciences (HICSS-25)*, Vol.2: Software Technology, pp. 562-563, 1992.

[Neig84] Neighbors, J.M., "The Drago Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, Vol. 10, pp. 564-574, September 1984.

[Nels90a] Naval Postgraduate School Report 52-90-024, *An Intoduction To Object-Oriented Programming*, by Nelson, M.L., April 1990.

[Nels90b] Naval Postgraduate School Report 52-90-025, *Object-Oriented Database Management Systems*, by Nelson, M.L., May 1990.

[Nels91] Nelson, M.L., "An Object-Oriented Tower of Babel", *OOPS Messenger*, Vol.2, No 3, pp. 3-11, July 1991.

[NFZ92] Nelson, M.L., Fontes, K.A., and Zaky, A., "An Object-Oriented Approach to Computer Architecture Simulation", *Proceedings of the 25th Annual Hawai International Conference on System Sciences (HICSS-25)*, pp. 476-485, Kawai, HI, January 7-10, 1992.

[NMO90] Nelson, M.L., Moshell, M., and Orooji, A., "A Relational Object-Oriented Management System" *IPCCC*, pp.319-323, 1990 .

[Onto90] *ONTOS Object Database Documentation Set Release 2.0*, Ontologic Inc., Burlington, MA, 1990.

[PF87] Prieto-Diaz, R., and Freeman, P., "Classifying Software for Reusability", *IEEE Software*, Vol.4, pp.6-16, January 1987.

[Ste91] Steigerwald, R.A., *Reusable Software Component Retrieval Via Normalized Algebraic Specifications*, PhD Dissertation, Naval Postgradute School, Monterey, CA, December 1991.

[Sun91] *Sun SourceBrowser Reference (OpenWindows)*, Sun Microsystems Inc., Mountain View, CA, 1991.

[Tan90] Tanenbaum, A.S., *Structured Computer Organization*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Vog89] Vogelsong, T., "Reusable Ada Packages for Information Systems Development (RAPID): An Operational Center of Excellence for Software Reuse", *Proceedings of Tri-Ada '89*, Pittsburgh, PA., ACM, 1989.

[WEK90] Winblad, A.L., Edwards, S.D., King, D.R., *Object-Oriented Software*, Addison-Wesley, 1990.

[WG90] *Actor User's Manual Vol.1*, The Whitewater Group, Evanston, IL., 1990.

INITIAL DISTRIBUTION LIST

1. Defence Information Center — 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52 — 2
   Naval Postgraduate School
   Monterey, CA 93943

3. MAJ Michael L. Nelson, USAF — 2
   Computer Science Department, Code CSNe
   Naval Postgraduate School
   Monterey, CA 93943

4. Prof. Luqi — 2
   Computer Science Department, Code CSLq
   Naval Postgraduate School
   Monterey, CA 93943

5. Embassy of Greece — 4
   Naval Attache
   2228 Massachusettes Ave., N.W.
   Washington, D.C. 20008

6. LT. Tilemahos Poulis, Hellenic Navy — 3
   Narkisson 3
   Philothei 15237
   Athens, GREECE