ABSTRACT

COMPUTER AND INFORMATION SCIENCE

ENGLISH, JANTH B. **B.S. TENNESSEE STATE UNIVERSITY, 1972**

DEVELOPMENT OF AN OBJECT ORIENTED PROGRAM INFORMATION DATABASE WITH SUPPORT FOR SOFTWARE REUSE

Advisor: Dr. Roy George

Thesis dated May, 1996

This research addresses the need for organizations to have a project support environment that can produce large, complex, quality systems at a reasonable cost. Many researchers recommend a program database as the basis for this project support environment. The development of an object-oriented database to store and retrieve program information is described along with the design for a software reuse library. The program objects are based on the common taxonomy of all software, e.g. composed of modules, programs, functions, and procedures, rather than the domain knowledge that software represents. This design permits maximum flexibility in accommodating most applications. The reuse library is composed of reusable components in the program database. The multi-attributes of keywords and component signatures are used to classify and retrieve reusable software objects.

This research demonstrates a means for an organization to provide immediate improvement to its project support environment and to implement a software reuse program. These benefits translate to improved software quality and productivity which are needed to remedy the current software crisis.

DEVELOPMENT OF AN OBJECT ORIENTED PROGRAM INFORMATION DATABASE WITH SUPPORT FOR SOFTWARE REUSE

A THESIS

SUBMITTED TO THE FACULTY OF CLARK ATLANTA UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

BY

JANTH B. ENGLISH

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

ATLANTA, GEORGIA
MAY 1996

R = V   T = 57

ACKNOWLEDGMENTS

I would like to thank my husband, James, for his kind and patient endurance as I pursued a lifelong dream over the past two years. He graciously gave me the space and support that I needed to accomplish this goal. He always believed that I could do it. I would also like to thank my advisor, Dr. George, for his gentle corrections, strong persuasions, and constant nurturing as I matured in my understanding and this research effort evolved.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. Introduction

This section describes the current software crisis and how Computer Aided Software Engineering tools, which were designed to alleviate the software crisis, have not lived up to the expectations of many organizations. Next, a discussion is presented on the advantages of a program database and how its implementation can help organizations that cannot procure CASE tools at this time. Finally, the contribution of this research effort is discussed.

B. The Software Crisis

It is widely accepted that current software development techniques are unable to produce high quality software at the rate required to keep pace with demand [2,11]. The trend continues toward larger more complex systems with millions of lines of code created by multiple software development groups. This necessarily generates volumes of information about the software product including multiple revisions of requirements documents, specifications, code, documentation, and test cases. It is, therefore, an important task to provide an environment that manages and coordinates access to this information. In fact, the software development environment should give the programmer the facilities to create, view, modify, check, translate, and execute portions of this information [19].

Brown [2] has called the software development environment an Integrated Project Support Environment, IPSE, to point out the need to provide consistent, coordinated support throughout the software development lifecycle. He states that all development groups can be classified as having either a first, second, or third generation IPSE. A first generation IPSE has a set of existing tools whose only link is a file system, e.g. development under the UNIX operating system. A second generation IPSE consists of a set of tools built on top of a database, and a third generation IPSE is built on knowledge based techniques. Brown [2] further states that most organizations can be classified as having first generation IPSE. The file system environment is incapable of providing the data consistency and security required by the software projects of today. Furthermore, there is a direct correlation between the project support environment and the price and quality of the product delivered [2]. The support environment has directly contributed to cost overruns, low productivity, low product quality, and high maintenance costs. Therefore, an improvement in the project support environment would bring about a corresponding improvement in software products.

Some years ago, it was recognized that due to its complex nature, software development could no longer be considered an art and a consistent methodology needed to be applied. The software engineering discipline brought with it rigorously defined software development methods based on mathematical and engineering principles. When these methods are supported by automated tools improved productivity and efficiency are the expected results. Thus, Computer Aided Software Engineering, CASE, was introduced as the answer to the software problem. The promise of CASE is that automated support for some aspects of software development and maintenance will increase productivity, reduce the cost of software development, and improve the quality of software products [3].

CASE tools have had many successes, but just as important to note is the reports of partial usage or abandoning CASE tools in practice [5,12,13]. CASE tools often fail to live up to the expectations of its purchasers and users [3]. The complaints against existing

CASE tools include issues of usability, learnability, flexibility, and effectiveness. Perhaps the most consistent of these complaints is the issue of flexibility. According to Henninger [13], a fundamental flaw in CASE tools is that their methods focus exclusively on the project life cycle with only incidental references to previous development efforts, development infrastructure, and other organizational factors affecting the development process. In short, the current generation of CASE tools force an organization to adapt their styles to that of the tools and lack the flexibility necessary to meet the needs of many organizations.

When an organization adopts a CASE tool, it is most often adopting a software development methodology as well. Many CASE tools strictly enforce their methodology with no deviations permitted. Moving from a first generation environment to a CASE development methodology may be too big a step for many organizations. An organization's corporate culture must be considered. Judging from some survey results [5, 10 12], the corporate culture of many organizations will not allow them to make the commitment to "marry" a particular software development methodology. What is needed by these organizations is an intermediate step that allows them to reap some benefits a CASE tool provides without full integration of the tool or the methodology.

C. Program Databases

Organizations need an environment that can produce large complex software systems to a strict time scale, and produce a finished product of high quality. To do this, they must be able to provide a means to share data while exercising control over that data to ensure consistency and integrity. Further, they need tools in place that can provide that functionality while being effective and flexible. Many in software engineering advocate a database as the basis of a project support environment [2,10,19,21]. A database for the project support environment is called a program database. It consists of requirements documents, specifications, data definitions, software, test cases and test results, and records data items and their relationships in a structured and accessible form, allowing controlled access to shared data.

Database systems were first introduced in the commercial environment to correct the same problems as those being experienced in the project support environment. However, the requirements of the typical commercial database differ substantially from that of a program database. Commercial data are usually simple and fixed in length. Program data are complex data types with variable lengths. Commercial databases usually have few types with a large number of instances for each type. Program data usually has many types with few instances of each type. A commercial database typically has short atomic transactions with single valued data items updated in place. Program data has multiple versions of the data with dependencies on versions; the transactions are long lived and can leave the database inconsistent for long periods of time. It can be seen from these differences that program data is more complex that typical commercial data. Researchers have found that an object-oriented, OO, database models this complexity more readily than other database models [19, 22]. In fact, using an OO model allows these complexities to be hidden from the user.

Besides the traditional advantages afforded by a database -- reduced data redundancy, data consistency, integrity, and security -- a program database can provide other benefits such as enforcing standards. Since all software is centrally located, rules can be invoked to ensure that programming standards are maintained. Consistent programming standards have been shown to lower program maintenance costs, the main source of expenditures in the software development lifecycle. Another benefit of a program database is that software reuse can be promoted. Reuse of existing software is a means to improve productivity and reduce costs. Reuse also improves software reliability and offers a means of improving quality by using components that have proven their integrity and effectiveness over time. Of course, reuse does not occur just because the software is centrally located; there must be an effective means to classify and retrieve reusable components.

D. Contribution

This research describes the design and implementation of an OO database to store and retrieve program information. The program objects are based on the common taxonomy of all software, e.g. composed of modules, programs, functions, and procedures, rather than the domain knowledge that software represents. This permits maximum flexibility in accommodating a variety of applications from many organizations. A methodology to classify and retrieve reusable software components is also presented.

Many organizations desire the improvements that CASE promises, but find the tools too restrictive and inflexible for their environments. Researchers have basically ignored this sizeable section of the information systems community. The focus of this research is to provide a practical means for organizations to make meaningful, effective improvements in their environment. This research proposes an OO program database to provide the immediate benefits of reduced data redundancy, data consistency, integrity, and security to improve the project support environment. This is accomplished without supporting any particular development methodology.

It is widely accepted that software reuse improves software productivity and reliability [9, 14, 17, 25]. Many CASE tools have a central repository but offer little support for structured reuse. This research project includes a software reuse library that is composed of software components identified as candidates for reuse. A multi-facet classification scheme is used to describe both the semantic and syntactic meaning of each Reusable Software Component, RSC. The user can retrieve RSCs by querying the database based on facet values; a browser is also supplied.

This research shows that the introduction of a program database to the project support environment is a feasible and desirable undertaking. The approach of basing the design on the taxonomy of applications makes a Commercial Off The Shelf, COTS, database application for program data both technically and economically feasible. Organizations whose corporate culture prevents them from adapting a specific CASE methodology would not have the same problem with a program database application since database products are familiar. The fact that no particular methodology is enforced addresses the problem of flexibility. The addition of a program database is desirable because of the many benefits it offers, especially the benefit of an improved project support environment. Improving the project support environment translates directly to improved software productivity and quality. For these reasons, a program database is an acceptable step toward a completely integrated project support environment. This research presents a means for an organization to take that step.

CHAPTER II

TECHNICAL BACKGROUND AND PREVIOUS WORK

A. Introduction

This chapter introduces the reader to some of the terms encountered when discussing object-oriented concepts and gives a brief explanation of object-oriented methodology. The advantages of using an object-oriented model for a program database is presented. Software reuse in a program database environment is discussed along with an introduction to the problem of information classification and retrieval. This is followed by a synopsis of previous work in this area including TULUM, a CASE environment for software documents and GRAS, a graph-oriented software engineering database.

B. Object-oriented Terminology

Object orientation is the process of organizing software as a collection of discrete objects that incorporate both data structure and behavior. The OO approach includes the concepts of identity, classification, polymorphism, and inheritance. Identity implies that data is represented as discrete distinguishable entities called objects. Two objects are distinct even if all their attribute values are identical. Classification is the process of grouping objects with the same data structure, attributes, behavior, and operations into a class. A class is an implementation of an object type. It describes both the data structure and permissible operations for each object in its class. Each class describes a possibly infinite set of individual objects, and each object is an instance of its class. An object contains an implicit reference to its own class, e.g., "it knows what kind of object it is" [21]. Polymorphism describes the concept that the same operation may behave differently on different classes [26]. An operation is the specification of an action or transformation that an object is subject to or performs. The implementation of an operation by a certain class is called a method. Each object knows how to perform its own methods. Inheritance is a means of sharing the properties and methods among classes based on a hierarchical relationship. A class can be broadly defined and then refined into successively finer subclasses. Each subclass inherits all the properties of its superclass and adds its own unique properties and operations.

Other OO concepts include abstraction and encapsulation. Abstraction is the selective examination of aspects of a problem; it focuses on what an object is and does before deciding how it should be implemented. All abstractions are incomplete, but they serve the purpose of limiting the universe so that things can be accomplished [26]. Encapsulation is separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object. It combines data structure and behavior in a single entity. This protects the data by allowing access to the data only through predefined methods. Encapsulation allows the implementation of an object to be changed without affecting the application that uses it.

C. Advantages of an OO Program Database

Since a database is recommended as the basis of a good project support environment, and program data requires a different approach than typical commercial data, the question arises as to what database model should be chosen. The options in selecting a database are to choose a commercially available one or to build a customized database to suit the individual needs of the organization. Developing a customized database is complicated and may take many man-years to achieve. Most software development shops would not attempt such a task when there are commercially available databases on the market. Besides this, the time frame would be such that no benefit could be realized immediately from the expenditures incurred.

There are different models of commercially available databases, but the most widely used are the relational and object-oriented, OO, models. Relational databases have the advantage that they are widely used, and, thus have immediate acceptance in the work place. However, because of the complex nature of the program data, it becomes difficult to model some of the associations using relational technology. In an object-oriented database, modeling these complex associations is made easier. An object-oriented database approach allows the complexity of the data types to be hidden from the users. The OO concept of encapsulation makes the access methods and implementation of these complex types become a part of the object. Defining these abstract data types facilitates the modularizing of large software systems which also reduces complexity. These complex types and relationships cannot be implemented using a file system such as UNIX, and are difficult to express and impossible to hide in a relational model.

Multiple views of the same data (i.e. programs) may exist in a program database. In an OO database, it is possible to implement access methods for the different views. This is achieved through the two important properties of polymorphism and inheritance. By defining one view as a subclass of the other and defining different methods in each view for the same operation, operations can be tuned independently for different views. Two important aspects of project support, the automation of version control and an incremental compilation system [19], can be easily implemented.

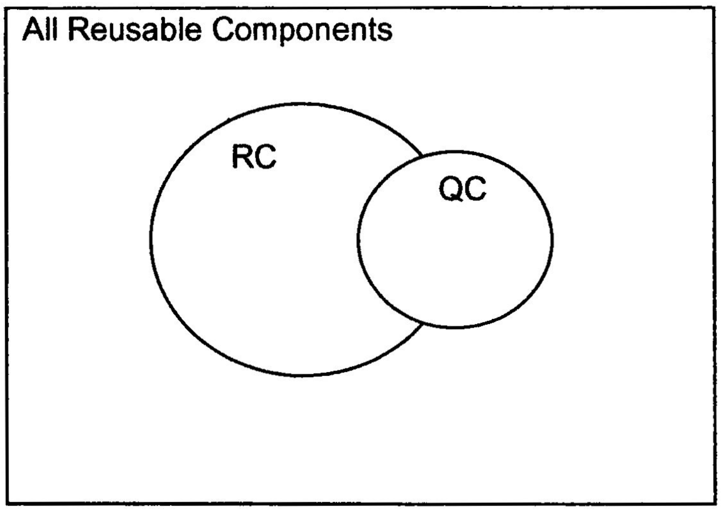
D. Reuse and Program Databases

Reuse is not an unfamiliar concept to experienced programmers. Analysts recycle their own previously developed code based on their understanding of the new code to be generated. This improves productivity and perpetuates the same level of quality as the recycled code. A program database containing all software components of an organization presents the opportunity to formalize software reuse and apply this principle on a larger scale. This will improve productivity and, because methods will be in place to ensure that only qualified components are reused, improve quality as well.

A project support environment that supports software development with reusable software components needs a library of Reusable Software Components, RSCs, that can be easily accessed and understood [28]. Accessing RSCs is a type of information retrieval problem. Information retrieval tools can be measured in terms of precision and recall [30]. Precision is the ratio of the number of components retrieved that satisfy the request to the total number of components retrieved, and recall is the number of qualified components retrieved relative to the total number of qualified components in the software database. The Figure 1 explains this relationship.

Information retrieval tools have both a method of representation for retrieval and a search methodology. The method of representation is important because the object should be structured to facilitate retrieval. This will be discussed in Section III, Program Database Design.

The method of search can generally be categorized as browsing, formal specifications, and informal specifications. Browsers depend upon the user to make a selection by physically viewing the available components. This methodology is effective only with very small repositories, and then may have low recall, e.g. there may be available components not retrieved.



RC = Retrieved Components, QC = Qualified Components

Figure 1. Relationship of Precision and Recall to Total Components

Formal specifications can be very accurate in describing the semantics of a component as most are based on predicate calculus. Theoretically, this means that any two specifications can be normalized to the same formal specification regardless of the manner in which the formal specifications were written. This methodology would have total recall, however, it would require that software engineers learn a specification language such as OBJ3, and that a set of rewrite rules be mechanized to normalize any specification written [20]. Mechanization of rules to normalize specifications is a non-trivial matter that has not yet been resolved by researchers. Utilizing formal specifications as a basis for software retrieval is, therefore, not a viable alternative at this time for an organization.

Informal specifications allow the user to describe some attributes of the component they are looking for. This search method includes such things as natural language search, keyword search, and a multi-attribute search [30]. A natural language search offers many potential advantages such as allowing different styles and sentence structures to be normalized to the same representation. The domain vocabulary would have to be very restricted to apply this approach. Besides being an expensive approach, researchers have not mastered the inherent ambiguities in natural language processing. Here too, the promise is far off and is not a viable alternative for implementation at this time.

Keywords offer a more practical approach to software retrieval. The main difficulty lies in assigning appropriate keywords to components. An uncontrolled vocabulary for keywords can lead to low precision and low recall. A controlled vocabulary may be too restrictive to describe some components. Because keywords offer the most hope for immediate implementation, researchers have applied differing modifications to this approach including the multi-attribute search and facet classification systems.

A multi-attribute search uses keywords that describe the semantics of the component as well as other attributes such as the type of component and its signature [22]. The facet classification system is a type of multi-attribute search. The facet classification system proposed by Pietro-Diaz [24] assigned a facet descriptor composed of values taken from each of the three classifications Function, Object Type, and System Type. Other types of facet schemes have been proposed such as the classifications of Abstraction, Operations, Operates On, and Dependencies proposed by Sorumgard [29].

In keeping with the goal of providing a practical solution, a combination of the multi-attribute search and a facet classification scheme similar to Sorumgard [29] is used. The classification scheme differs from Sorumgard's in that the facet Dependencies is not used. This facet was thought to unnecessarily restrict retrieved components that could be modified to suit the environment, or, at least, understood for reuse of the component design. The attributes used include the component signature and keywords that include the facets of Abstraction, Operations, and Operates On. The component signature is used to describe the syntax of the component and keywords are used to describe the semantics of the program. Abstraction refers to the abstract concept being implemented by the components. Operations is usually a verb that refers to the function being performed, e.g. update, print. Operates On is the class or classes of objects on which the component operates, e.g. integer, record. Each reusable component is described by its signature and a non-empty set of keywords as a facet descriptor based on these facets.

Another problem to address in software reuse is the granularity of the software component to reuse, e.g. at what point in the Systems Development Life Cycle, SDLC, should reuse occur. Different design methodologies vary in the type of support provided for reuse. For example, OO methods support reuse throughout the lifecycle [28]. Structured design methods support reuse at a functional level, and typically benefit from reuse mostly in detail design, coding, and testing [28]. It is generally accepted that reuse is desirable at all phases of the SDLC with the construction of an entire application from reusable components being the ultimate success. However, large scale software reuse has not been realized [14,15]. In fact, it is unrealistic to expect to take large pieces of software and connect them to each other without a firm scientific foundation at the most basic level [14], and there is no such foundation established for component based software engineering [14]. To date, the most successful model for software reuse has been components that implement a single function [14, 15, 28].

Another fundamental question is at what cost should reuse be attempted. It has been argued that the costs of building the "glue" to compose these functions into a complex application is too much work [15]. However, with the lack of a formal methodology to guide in the construction of applications from large software objects, this may be all that can be realistically expected. To suggest that reuse cannot begin until everything is known about it is impractical, especially in view of the fact that reuse is practiced and is especially successful with small granularity software components. Ramamoorthy, et. al. [25], states experience on the Genesis project shows that generally it is a good idea to reuse code even if time has to be spent in understanding it and some changes have to be made to reuse it. He further states that only after an initial effort finds the changes required to be too extensive is it more pragmatic to abandon reusing the component.

Based on the discussions in this section, an underlying software reuse philosophy was developed. That philosophy is that the most successful model for an RSC is that of a single function, and that reuse should be attempted even if modifications must occur.

E. Previous Research

TULUM was developed as a Computer Aided Software Engineering environment for software documents by Luis Miguel [22]. Miguel [22] proposed CASE database requirements and examined the relevancy of these requirements for data manager classes. The data manager classes examined include custom DBMS, custom DBMS from a generator, persistent object store, relational DBMS, relational object shell, OO DBMS, and extended relational DBMS. TULUM is the proposed CASE environment architecture that addresses physical issues, logical issues, and computational paradigms. These proposals are based on experiments that were conducted to measure the performance and space utilization characteristics of the different database designs. The effects of data representation, the number of relations, and the granularity of the data are isolated to obtain meaningful results.

After analyzing the needs of a CASE data manager, each data manager class was examined to determine if they satisfy those requirements. Conclusions were reached based on how each class satisfied the requirements. A custom DBMS, whether designed from scratch or a generator, only satisfies a subset of the required features. The proprietary environment prohibits sharing data with other environments. Also, applications built on a custom DBMS tend to be non-portable. Persistent object stores lack sophisticated features such as rules, procedures, query language, abstract data types, and inheritance. Relational data managers have no object management features such as object encapsulation, inheritance, etc. Both relational data managers and relational object shells --systems that provide object management on top of a relational

DBMS-- have no rules or procedures for knowledge and process management. The extended relational manager --a relational model that provides some object management and knowledge management features such as data encapsulation and rules system-- and OO DBMS both satisfy most of the requirements for a CASE data manager. TULUM uses an extended relational model as a data manager.

It was concluded from the experiments that data representation had minimal effect on space usage but could have a profound impact on operation time. The closer the internal representation to the actual object the less time spent in expensive conversions. It was also found that small granularity design is very expensive in terms of time and space. Larger granularity design pays off in terms of time and space but provides a more limited functionality than a small granularity. However, for most applications, the larger granularities may be sufficient.

TULUM is presented here because of the work done in the area of data representation and because of the comparisons that were made between different database models. Many design and implementation decisions for this research follow the findings of Miguel on TULUM and others, e.g. [2, 19]. Some of those decisions for this research project include the use of a non-proprietary OO database, internal representation of objects that are close to that of the original object, and use of large granularity for storage.

A different approach to data representation for program database objects is that of an attributed graph. This model was investigated because (1) graph theory is the underlying mathematical model for some computer sciences formalisms, (2) a comparison between this model and other database models was warranted, and (3) this approach was different and invoked curiosity. The database system selected for study was GRAS, GRAph Storage.

GRAS is a software engineering database system that utilizes the attributed graph as the underlying model for complex software objects. Objects are modeled as nodes with attributes, and relations between objects are modeled as edges. Attributes of nodes may be either intrinsic, assigned explicitly, or derived. Edges are binary, bidirectional, directed relations with two distinct end nodes, sink and source. Edges do not carry attributes. Composite nodes act as source, and Component nodes act as sink. Edges represent relations such as "Contains," "Precedes," and "RefersTo." Paths are derived relations that are calculated from edge and node properties. Paths provide a way to define abstract views on graph structures.

Graph schemes, which describe the components of attributed graphs, are defined using a formal specifications language called PROGRES, PROgrammed Graph REwriting System. PROGRES is both a data definition language, defining graph schemes, and a data manipulation language, performing complex graph transformations. Graph transformations specified in PROGRES are mapped onto basic operations provided by GRAS. GRAS serves as the kernel of a database development environment for PROGRES.

Internally each graph is stored in a data structure called a graph base which consists of separate storage areas, including the Node, Attribute, Index, and Edge storage areas. All these storage areas are collectively known as the GraphStorage which is the kernel of the system architecture. Storing different types of data in different stores has the advantage of more efficient navigational queries, e.g., traversing the graph. However, operations which affect a node including all its attributes and edges such as creation or deletion, are adversely affected [16]. The GraphStorage has an underlying storage layer called the VirtualRecord storage layer which is a record-oriented interface. It is designed for efficient access to medium-sized graph bases -- a graph base having the size of a typical document such as a program module.

GRAS is an active database system. Action routines are evoked when specific event patterns are triggered. An event refers to a database transition which is an atomic change to a graph object. These changes may involve multiple database transitions, for example, deleting a node causes all associated edges to also be deleted. Any changes to the graph database requires GRAS to preserve consistency of the graph schema and recompute derived attribute values and derived relations or paths.

GRAS has been used in a variety of software engineering projects. In the IPSEN project [16], a program database was constructed consisting of related documents such as requirements, specifications, software architectures, modules, test plans, etc. Documents were viewed as having a fine-grained internal structure which were constructed with a corresponding database scheme. The documents were viewed as the "natural" objects for distribution, concurrency control, version management, etc., and not the internal representation thereof. From a user perspective, a document is a hierarchically structured piece of text or diagram and modifications occur to this structure using a text editor. The developer views the document as a complex graph structure and tools are used to manipulate this structure. The internal representation of the document is an abstract syntax tree augmented by context-sensitive edges which is maintained by GRAS. As can be seen, the internal representation of the document object is very different from that of the natural object. The overhead incurred for transformations from the natural object to its internal representation and back can be expensive in terms of efficiency. This problem is avoided in the current research effort by using course granularity and an internal representation close to that of the natural object, e.g. the entire document is stored as one object whose data type is Text.

A program or software engineering database can be thought of as a set of interrelated documents, and those documents are represented as graphs with each graph corresponding to a certain document [16]. However, one drawback is that

GRAS does not provide built in support for representing inter-document relations. The OO model, however, facilitates representing inter-document relations. Associations such as "has" and "consists of" between objects are maintained by the program database application.

Other software databases were studied during this research effort. However, none contributed significantly in the design or implementation of this program database.

CHAPTER III
PROGRAM DATABASE DESIGN

A. Introduction

This chapter describes the design of the Program Information Database. The scope of the project's initial phase is stated including an explanation of which program database components will be implemented. Design issues are discussed including the selection of a DBMS and the concept of an application as an abstract data type, ADT. The analysis model for this program database is presented as well as the Reuse Library design.

B. Project Scope

Since a program database is a large undertaking consisting of requirements documents, specifications, data definitions, software, test cases and test results, the first issue to decide was the scope of the initial phase of this research project. Since producing correct software in a timely manner is at the core of the software crises, the initial phase of our implementation includes software objects. It was also decided to include data definitions in the initial phase because incorrect and inconsistent usage of data types are a typical source of errors in developing an application. Due to space considerations, no object code or executables are stored in the database. These objects are easily derived through methods used to compile the source code. Requirements documents and specifications were not exploited fully in this version; however, due to their importance in verifying correctness, documentation objects were created as an attribute of the software objects.

It is generally accepted that software reuse will improve software productivity, and, thus, will help to alleviate the software crises [2,9,20,24,27]. With such overwhelming conviction of its effectiveness, it was decided to include mechanisms for software reuse within the initial implementation of the Program Information Database. This has been accomplished by storing Meta-Data information describing the reusable software components in the Program Information Database. A browser and query mechanism are provided as part of the application. In addition, a separate application called REuse QUEry Subsystem, REQUES, was developed to access and share reusable components.

C. Database Design Issues

The first design issue to address was that of selecting a database manager. This is arguably an implementation issue, however, it is contended that the choice of a DBMS influences other design decisions, and can effect the scope of the project, e.g. make some desirable features either feasible or not feasible. Thus the choice of a DBMS should be made during the design phase. It is generally accepted that an object-oriented database is more suited for a program information database [19, 22]. An OO DBMS was, therefore, sought for implementation. O2 was examined and found to support all functionality for OO applications including data encapsulation, inheritance, and persistence. O2 has a data definition language and data manipulation language, o2c, which is a superset of the C language. In addition to o2c, C and C^{++} is supported. O2 has the ability to handle complex objects such as text and graphics; it also has an SQL like query facility. O2 was selected because it is a complete environment that is commercially available with all the features required to implement our application.

To have a program database flexible enough to accommodate most applications, it was decided to base the design on the nature and structure of software, e.g. its taxonomy, rather than tailor the design to a specific application domain. To decide what objects best represent this domain, it is necessary to study a software application as an abstract data type. The size and complexity of today's software systems makes it necessary to structure them as modules, a software component which constitutes a coherent unit that provides a certain functionality [8]. The modules that represent the basic functionality of the software system are called system modules. An application can then be defined as a non-empty set of system modules. This definition of a module can define any software component that performs a function, so, for clarity, modules, in implementing this system, refer only to system modules.

A module may require several software components -- programs, functions, and procedures -- to implement the desired functionality. A program is defined as a collection of statements from a programming language that implement a certain functionality. In the Program Information Database, a program is distinguished from a module in that a module is the functionality or concept to be implemented and a program is the implementation of the concept. Therefore, in this system, a module contains a non-empty set of programs. Programs may contain other programs and subprograms, a process abstraction that allows details of program implementation to be hidden [27]. Functions and procedures are classified as subprogram types. By definition, procedures are software objects that are allowed to produce results in the calling program unit [27]. Functions, however, are not allowed to modify variables outside its environment and may return a result to the calling program. Programs then are composed of a set of programs, functions and procedures.

Programs, functions, and procedures have an "is-a" relationship with the Super class Source_Code. They are defined within the database by using the property of inheritance from the Source_Code class and adding those attributes that make each specialized. The Source_Code class definition is follows.

```
class Source_Code public type tuple
        (       public name : string,
                public short_description : string,
                public doc : set (Documentation),
                public language : string,
                public compiler_directive : string,
                public code : Text,
                public variables : unique set (Variable),
                public called_functions : unique set (Function),
                public called_procedures : unique set (Procedure),
                private last_modified : Date,
                private date_created : Date,
                private modified_by : string,
                private checked_out : boolean,
                private application : string,
                private where_used : set (Source_Code)   )
        end;
```

The Variable class contains instances of data definitions for an application. Unique names and consistent data types are maintained by Module within an Application. Only those variables selected by the user are stored and maintained as objects. The Variable class definition follows.

```
        class Variable public type tuple
        (       public var_name : string,
                public var_type : string,
                public description : Text,
                private first_declared : string,
                private where_used : list (Source_Code)   )
        end;
```

The Documentation class, as stated previously, was created to contain instances of all types of documentation including requirements documents and specifications. Programmers can also create documentation and save it with the object it represents. The Documentation class as well as all other class definitions can be found in Appendix A.

The Program Information Database can store multiple applications within the database. This allows different development groups to share this resource and creates a larger pool of reusable software components to share. Figure 2 is an Object Diagram of the Analysis Model depicting these relationships.

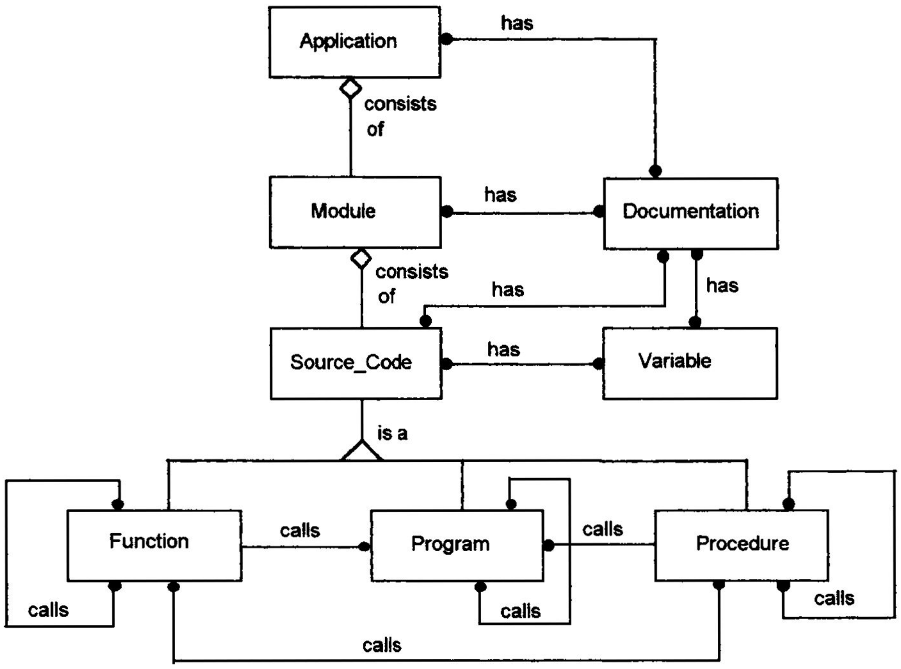


Figure 2. Object Diagram -- Analysis Model

The next major design issue to resolve is that of data representation. One of the issues involved in data representation is that of granularity. Questions like "should each statement in the code of a Source_Code object be an identifiable object or should it be the entire text file" must be answered. Research done on TULUM and Allegro suggest that using coarse granularity of the data improves efficiency. It can be seen that the smaller the granularity the more complex the conversions required to present the object in the format required by the application user. Course granularity is used in this design with the smallest identifiable unit of the code being the entire text file. In other words, each Source_Code object has an attribute "code" which is a Text object. This internal representation is sufficient for the functionality implemented as no operations are required for any "sub-code" level, e.g., words, phrases, lines. This design is also efficient in that O2 has methods that handle variable length text files so that no efficiency is lost in converting data. Miguel [22] found that data representation has minimal effect on space usage, therefore, our representation does not adversely affect disk space.

D. The Reuse Library Design

The objective in the design of the Reuse Library is to take full advantage of the software components stored in the program database. Our research is focused on software development and maintenance groups not presently utilizing CASE tools. According to Fiejs [8], application domains are in one of the following phases.

1. No reuse
2. Ad hoc reuse
3. Structured reuse
4. Automation of the domain

This application is geared toward taking an organization from step 1 or 2 to step 3, structured reuse. Research indicates that reuse has been most successful with small atomic functions such as I/O, mathematical software, and string manipulations [15, 28]. While it is recognized that reuse is desirable at all phases and levels of software production, it was decided best to start with the successful model of reusing small atomic operations. Design and implementation is based on the philosophy that programs can be constructed by putting together components that perform atomic operations. Therefore, the Reuse Library is composed of a set of Library_Function which inherits from the Function class and adds signature information. (For a formal class definition, see Appendix A.) An Object Diagram of the Analysis Model including the Reuse Library is shown in Figure 3.

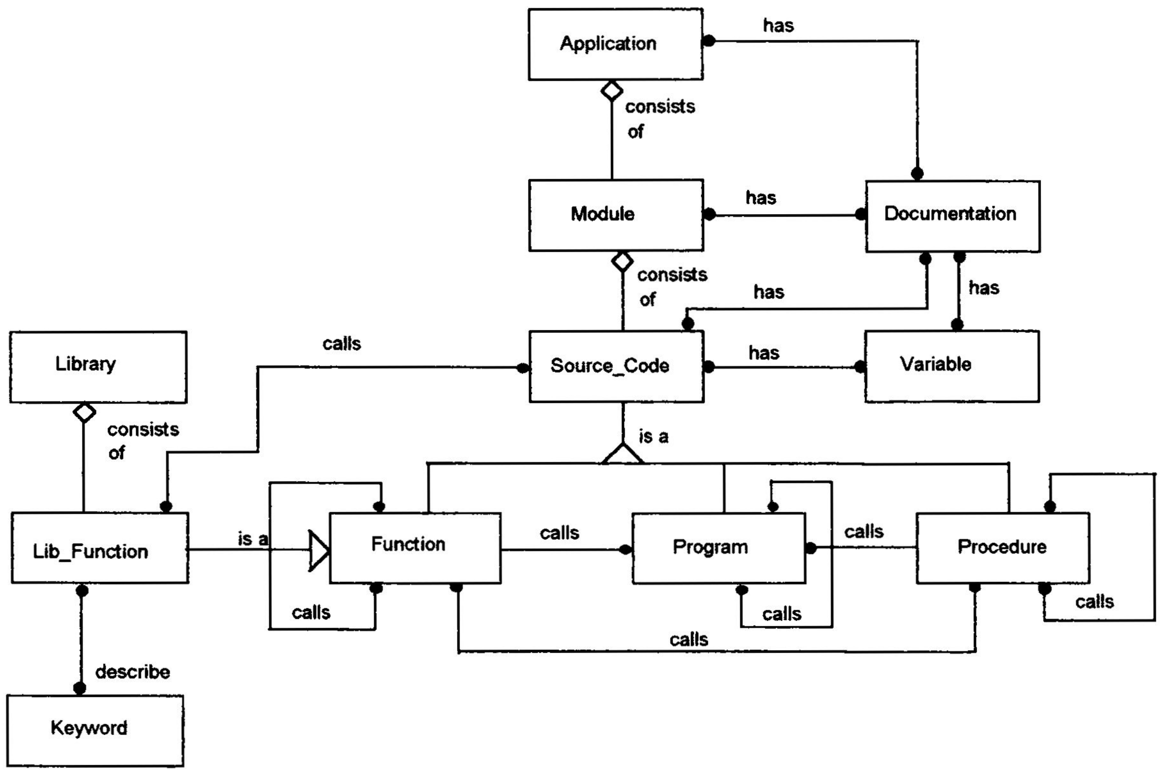


Figure 3. Object Diagram -- Analysis Model With Reuse Library

The first design issue for the Reuse Library is to represent the reusable components so as to aid efficient retrieval. The design employs a multi-attribute search and a facet classification scheme. The multi-attributes include keyword descriptions and the component signature. The keywords used to describe a component are actually the facet Abstraction and one or more of the facets Operations, and Operates On. This can be expressed as follows:

$$\text{Keyword} = \{<\text{Abstraction}>^{+}(<\text{Operations}>)^{*}(<\text{Operates On}>)^{*}\}$$

The Abstraction facet is usually a noun describing the type of component. The Operations facet describes the actions that the component performs, and the Operates On facet states the type of object that the component acts on. The Abstraction keyword is assigned when the component is placed in a particular library. The Library Name is the first level of abstraction indicating the concept being implemented by its components. The Library Name then is equivalent to the facet Abstraction. For example a Library Name of Matrix would contain all components performing matrix operations such as multiply, add, subtract, etc. One or more keywords are selected to indicate what functionality or operations the component performs, e.g. multiply. These keywords would then satisfy the abstraction Operations. Keywords are also selected to satisfy the facet of Operates On, e.g. real. A component, C1, that multiplies a matrix of real numbers will have a set of Keywords as follows:

$$K_{C1} = \{\langle\text{Matrix}\rangle, \langle\text{Multiply}\rangle, \langle\text{Real}\rangle\}$$

All components are thus classified by the Librarian who also maintains the Keyword class. Each component can be an instance in only one Library and has a non-empty set of Keywords that describe it.

The signature of the component is used to describe the syntactic representation of the component. A syntactic query match would indicate little or no revisions would be required for reuse when a match has occurred. Each component's signature is described in terms of it's input and output data types. All data types are accepted. The program performs a "normalization" procedure on the signature data types and stores both the original and normalized data types with the object. This normalization procedure attempts to match data type queries on the component signature with those in the database. For example, if a query is received with keywords and signature of

$$K_C = \{\langle\text{Matrix}\rangle, \langle\text{Multiply}\rangle\} \text{ and } S_C = \{\text{Inputs: } \langle\text{Real}\rangle; \text{ Outputs: } \langle\text{Real}\rangle\}$$

respectively, and, if the system can not find a component that has input data type of Real and output data type of Real, it will normalize Real to Integer. The search will then be conducted with the normalized input. This process is repeated until a match is found or the data types can not be normalized further. All data types except "Private" eventually normalize to "Integer". If the original input type is not recognized, it is mapped to the

"Private" data type. The private type includes all user defined data types. The Type class contains predefined mappings for most data types which are maintained by the Librarian.

This normalization process is done in keeping with our philosophy that reuse should be attempted even if revisions are required. Also, in this spirit, queries where no syntactic match is found on the signatures will receive a response based on keyword match.

E. REQUES Application Design

REQUES is a client/server application that interfaces with the O2 database. It is written primarily in C with some of the server functions written in o2c. The server program, o2_server, is the client's interface to the O2 database. The server performs the following functions.

1. Opens a socket to communicate with clients.
2. Opens the O2 database.
3. Accepts and interprets requests from a client.
4. Calls O2 functions to satisfy client requests.
5. Formats O2 data for transmission to clients.
6. Transmits results to clients.
7. Closes O2 database.

Valid transaction requests are OPEN, open the database, KEYWORD, get a list of the valid keywords, QUERY, find components based on the semantic and syntactic information sent, and CLOSE, close the O2 database. The OPEN transaction opens the O2 database if it has not already been opened by a previous client request. When the server receives a KEYWORD transaction request, the O2 database is queried for all existing keywords which are then sent to the client. A QUERY transaction request causes the server to query the database for all components in the library that match the accompanying component attributes. These attributes include the keywords used to describe the semantic meaning of the component along with the data types of the inputs and outputs. If matching components are found, they are transmitted to the client requesting the data. If no matching components are found, a message is transmitted indicating zero as the number of components found. Upon receipt of a CLOSE transaction request, the server determines if there are any active clients before closing.

The client application consists of a user interface and four executables that communicate with the server program. The user interface performs the following functions.

1. Displays and accepts user request options.
2. Formats user request for use by executables.
3. Calls executables to implement user requests.
4. Presents data received from the server.

Each executable performs one of the following functions -- open o2 database, get valid keywords, query o2 database, or close o2 database functions. The REQUES Application configuration is shown in Figure 4.

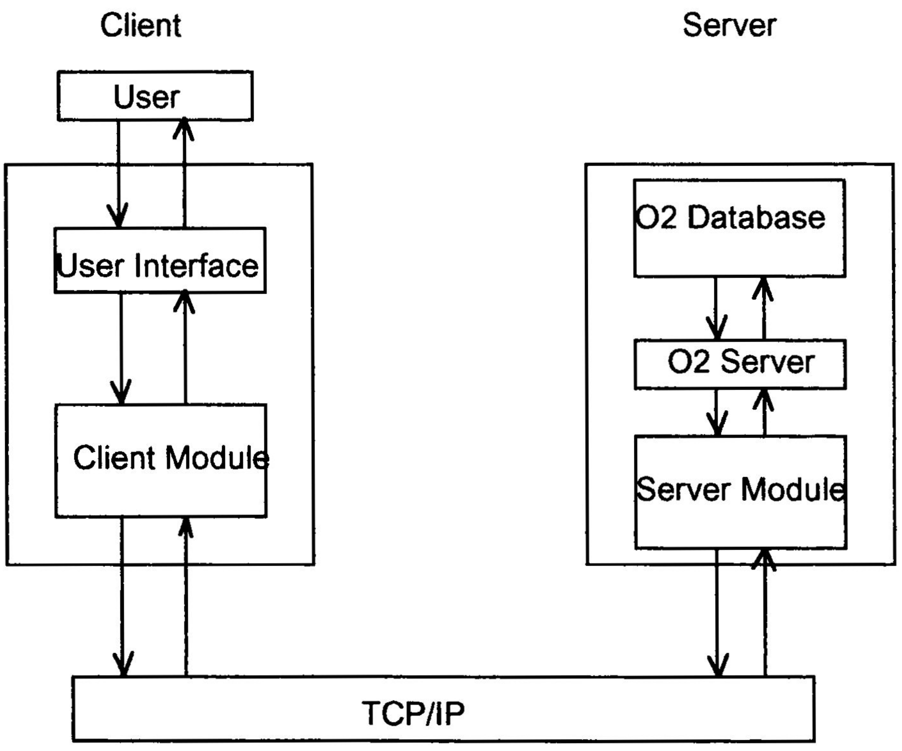


Figure 4. REQUES Application Architecture

CHAPTER IV

PROGRAM DATABASE IMPLEMENTATION

A. Introduction

This chapter describes issues addressed while implementing the Program Information Database. The implementation environment is defined, and implementation issues such as improving efficiency by adding redundant associations and locking are addressed. The function of the Librarian, the person(s) who maintains the Reuse Library, is explained. Finally, implementation details of the REuse QUEry Subsystem, including transaction management and component retrieval is discussed.

B. Implementation Environment

The Program Information Database runs on a Sun workstation under the UNIX operating system. It was written primarily in o2c, since, o2c is a very expressive programming language. Due to O2 design, those programs accessing the database from outside the O2 environment must be written in C or C^{++}. REuse QUEry Subsystem, REQUES, was written in C and calls O2 functions written in C or o2c. The user interface to REQUES is written in TCL which uses Motif. This gives the REuse QUEry Subsystem the same look and feel as the Program Information Database since the O2 interface is Motif based as well.

The REuse QUEry Subsystem is a client server application using TCP/IP network protocol. The server application runs on a Sun workstation and client applications may run on any hardware having a UNIX operating system and supports TCP/IP. TCL is public domain software.

C. Implementation Issues

Since the focus of this research is on providing immediate, practical assistance to software development groups, it was a safe assumption that existing applications would be the primary source of data for the database. An application that exemplifies managing shared resources, Dining Philosophers, was selected as the target application. This application was selected because it is large enough to exercise all parts of the system, yet small enough to have a controlled experiment. The Dining Philosophers application was also selected because it was not constructed using a CASE tool. Using an existing application constructed without CASE demonstrates that the program database can accommodate applications typical of the target group identified.

It is recognized that during design optimization, the designer must add redundant associations to minimize access cost and maximize convenience [26]. In theory, redundancy is undesirable as it adds no additional information. However, the associations of the analysis model may not provide the most efficient access patterns for implementation. This represents the situation for the Program Information Database. For convenience, the Source_Code class definition is restated below.

```
class Source_Code public type tuple
(     public name : string,
      public short_description : string,
      public doc : set (Documentation),
      public language : string,
      public compiler_directive : string,
      public code : Text,
      public variables : unique set (Variable),
      public called_functions : unique set (Function),
      public called_procedures : unique set (Procedure),
      private last_modified : Date,
      private date_created : Date,
      private modified_by : string,
      private checked_out : boolean,
      private application : string,
        private where_used : set (Source_Code)    )
    end;
```

To demonstrate this problem, the relationship between a Program and a Library Function it calls is examined. This relationship is expressed in the association "Source_Code-calls-Library_Function". An analyst desiring to modify a function in the Library needs to know what objects might be affected. If the "Source_Code-calls-Library_Function" is used all Source_Code objects would have to be searched to determine which objects are affected. However, if a redundant association "Library_Function-called_by-Source_Code" is added, the Source_Code objects that call the function are pointed to by the function called. In the Dining Philosophers application, both the server.c and client.c programs call the Library Function sema_signal.c which sends a signal operation to a semaphore. This information is available by selecting the method Display_Usage from the sema_signal.c Library_Function object. Several redundant associations were added to the analysis model to improve efficiency of the test-to-hit ratio and, thereby, improve performance. An analysis of these situations are explained in detail.

A unique property of an OO database is that no two objects are the same even if all of their attributes are the same. This presents a challenge to the Program Information Database when trying to ensure a correct application structure. This challenge is compounded when the existence of multiple applications in the database is considered. The question is whether to enforce unique identities on Source_Code objects and if so, at what level of implementation -- database, application, or module -- should this enforcement take place.

It was decided that each Source_Code object be identified by a unique name, however, enforcing unique names from one application to another was deemed impractical. Therefore, enforcement at the database level was ruled out with the exception that all Library_Function objects must be uniquely named because two functions having the same name implies they provide the same functionality. This is undesirable in a Reuse Library. The next question is the practicality of enforcing unique names at the application level. It was decided that most applications would not have two different objects with the same name, and, if this situation did exist, it would be cost efficient to correct. Therefore, it was decided to enforce name uniqueness at the application level for all Program, Function, and Procedure objects.

To accomplish this, a private attribute identifying the application was added to the Source_Code object. An index composed of the application name and the object name ensures a test-to-hit ratio of 1:1. This is important in a potentially very large database. A global variable "Current_Application" is used to determine the application in which the user is currently working. This allows the program to find and modify the correct object and prohibits adding a new object with the same name as an existing object in that application. An example of this would be adding a Program object named "server.c" to the Dining Philosophers application. The system would check the persistent data stores to verify if any Source_Code object existed with name = "server.c" and application = "Dining Philosophers". If such an object existed, an error message would be displayed and the object would not be added to the database. However, if the same Program object were added to another application where no Source_Code object had the name "server.c", it would be added to the database.

Similar questions concerning Variable objects are present. It was assumed that variables could possibly have the same name across applications and modules, yet have different meanings and uses. Therefore, unique variable names are enforced at the module level. A redundant association between the Module class and Variable class was created to implement this functionality. A Module "has" Variable objects and all Variable objects associated with Source_Code objects are a subset of those associated with a Module. Users may select a variable to store at any Program, Function, or Procedure object. A reverse engineering technique is used to maintain this relationship of Source_Code Variable objects and Module Variable objects. If the variable is added at the Source_Code level, the program will update the object's list of variables used as well as add the variable to the Module's selected variables if it has not been added previously. A check for data type consistency is made against the Module's variable information if the name of the variable is found to already exist. In the Dining Philosophers application, there was no existing data dictionary type documentation. A Variable object, "operation", which indicates the activity a client is requesting was added to the Program object "server.c". The system automatically updated the server Module object to reflect "operation" as one of the Variable objects that it "has". It should be noted that only those variables selected by the user are added to the system. This avoids using system resources to monitor insignificant variables such as temporary storage variables and counters.

Another redundant association, "where used", is given between the Variable class and the Source_Code class. It is recognized that one access pattern for variables in the Program Information Database will be the operation to find all objects where a Variable is used. This association is maintained automatically by the system. When a user selects the Add_Variable method, a new Variable object is created and the "where_used" attribute is modified to include the Source_Code object adding the variable. If the variable is not new to the module, it is retrieved and the "where_used" attribute is updated to reflect the addition of another Source_Code object, the object adding the variable. To find all objects where a Variable object is used, the only step is to find the variable and select the appropriate method. Without this association, it would be necessary to check all Variable objects for each Source_Code object within an application in order to find where a variable is used. This redundancy is well worth the improved performance.

Finally, a redundant association, "called by", was added between the Library_Function and Source_Code classes. This association allows efficient retrieval of all Source_Code objects that call any Library Function. This is necessary in the event a Library_Function object needs to be modified. It is also useful for determining how much reuse is taking place and to what extent. This association is updated whenever a Source_Code object executes the method "Add_Library_Function" or "Delete_Library_Function". These methods add a Source_Code object to and deletes a Source_Code object form the "called_by" attribute. Figure 5 shows an Object Diagram of the Implementation Model with all associations.

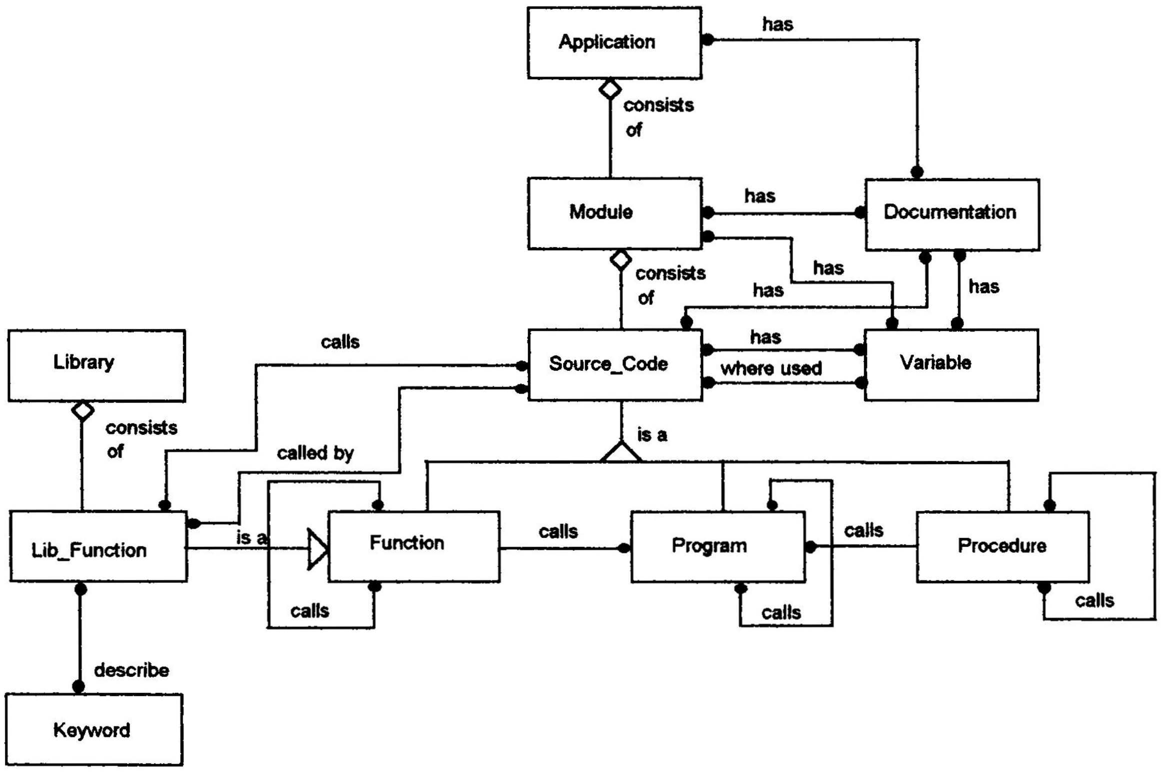


Figure 5. Object Diagram -- Implementation Model

Locking is a significant issue in a program database. Program data normally have long lived transactions that can leave the database inconsistent for long periods of time. Our approach to this problem is a to implement updates using a two phase transaction methodology. When the method to update a Source_Code object is invoked, the object locks itself, makes a copy of itself, and marks itself as "checked out". The lock is then released thus completing the first phase. Only read type transactions are permitted on objects in this state. When the user selects the option to save the update, a second transaction is started in which the object again locks itself then overwrites the old version with the new version and marks itself "in". The lock is then released which completes the second transaction phase.

D. The Role of the Librarian

The quality of the components in the Reuse Library is very important. Errors in a library component can propagate not only within an application, but, from one application to another. For this reason, there must be controls on all methods implemented by a Library_Function object. The Program Information Database maintains this control by allowing only those logins defined as "Librarian" to perform update type transactions on Library_Function objects. In addition, separate menus are presented to other users. There is also a separate Librarian application that only the Librarian can run.

The Librarian has the job of maintaining the Reuse Library. This includes creating libraries, adding qualified components to the library, and modifying a component in the library if necessary. The Librarian also maintains the Keyword and Type classes. The Librarian application must be used to create a Library or add, modify, and delete instances of the Keyword and Type classes. Other functionality such as adding or modifying a Library_Function can be performed either in the Librarian or the ProgramDB application. All functions are performed through menu selections from the applications.

The Librarian must have a thorough understanding of the facet classification system used in our Program Information Database as she is responsible for classifying or reclassifying every component. The Librarian decides the names of each Library which is also the first classification facet, Abstraction. The Librarian decides what Keyword objects are contained in the database and maintains the Keyword class. The Type class is already predefined but may be modified by the Librarian. Therefore, a thorough knowledge of how the Library_Function signatures and keywords are used to retrieve reusable components is also necessary. In short, the success of the Reuse Library can depend on how well the Librarian knows and performs her job.

E. Library Implementation Issues

Selecting matching components for a query was the biggest challenge for the Reuse Library. The signature and a non-empty set of keywords are a part of each reusable component. The keywords describe what exactly the component does. Therefore, a match on keywords is the most important aspect of finding a reusable component. A match of signatures is an indication that little or no modifications will be needed to reuse the component. Our philosophy includes the idea that reuse should be attempted even if modifications must occur. In implementing this philosophy, it becomes necessary to not only look for exact matches of the signature, but to match on equivalent data types, or just keywords if no equivalent signature matches can be found.

Sorumgard [29] states that reuse progresses from the idea phase to a description phase, and after formalizing the description, to a requirements phase. The requirements are then matched against the classifications in the Library. (See Figure 6 below.) The requirements phase identifies more specifically how the component performs by, among other things, defining the inputs and outputs needed, or, signature information. Another school of thought is that over specification of components should be avoided because it can preclude reuse of available RSCs [23]. This methodology promotes the concept of software development "for" reuse with reuse driving the design process. This is comparable to other engineering disciplines where new products incorporate existing components. Our query methodology accommodates both views. The user may be as specific as desired in identifying a component. The design also allows the user to go from the description stage directly to matching component classifications by not requiring signature information as part of the query. This allows the user to skip formalizing the design at this point and allow the syntactic structure of retrieved components to influence the formal design.

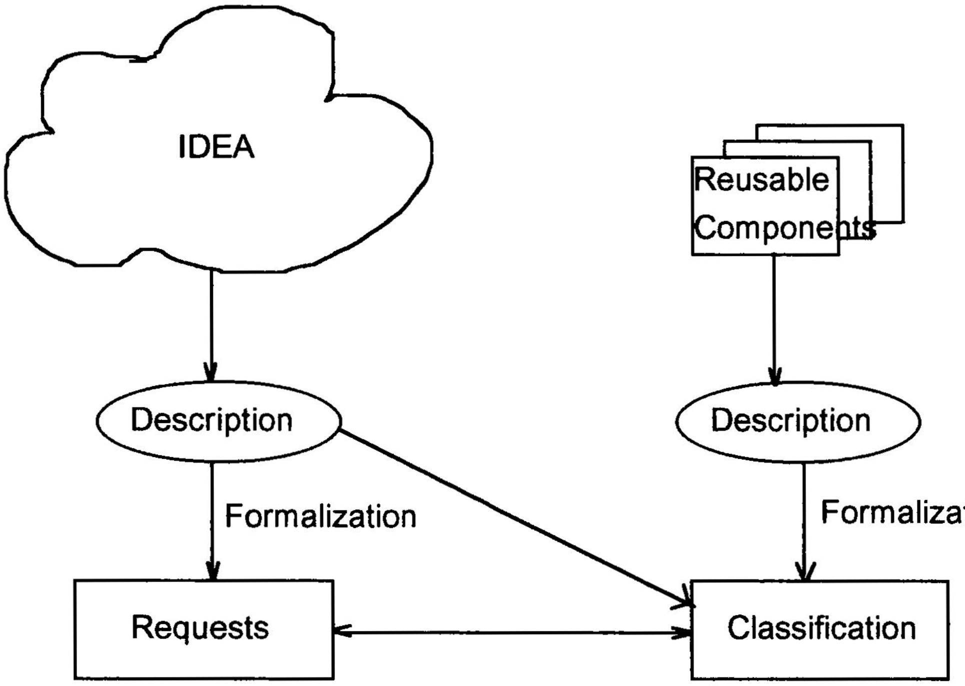


Figure 6. The Retrieval and Classification Process

Our retrieval scheme first finds all components having a set of keywords that match exactly the set of query keywords. If no components were found having an exact match, then components having the largest subset of keywords in the set of query keywords are selected. There will never be an empty set of components retrieved because

(i) each component has a non-empty set of keywords that describe it,

(ii) only keywords describing existing components can be used in the query, and

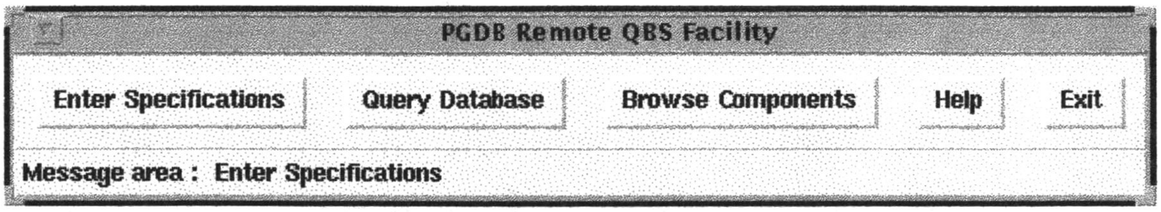
(iii) no query is processed with an empty set of keywords.

The query signature is converted to an equivalency data type based on a Type class equivalencies. The signature of the retrieved components are then compared to the normalized query signature. If no match is found, the query signature equivalency conversion is performed until no additional equivalencies exist. If a signature match is found, the subset of components found matching the query signature are retrieved. If no signature match is found, the set of components matching the keyword search is retrieved.

F. REQUES Implementation Issues

REQUES is a client/server application, as such the server must manage transaction requests among the clients. Valid transaction requests are OPEN, open the database, KEYWORD, get a list of valid keywords, QUERY, find components based on the semantic and syntactic information sent, and CLOSE, close the database. It was decided that an entire client request would be implemented as one transaction in a critical section. It can take more than one transmission from the server to fulfill a client request to send valid keywords or a query result. During these times all new requests from other clients are placed on a wait queue. When all data for the current request is sent, the server then handles the next request in its entirety.

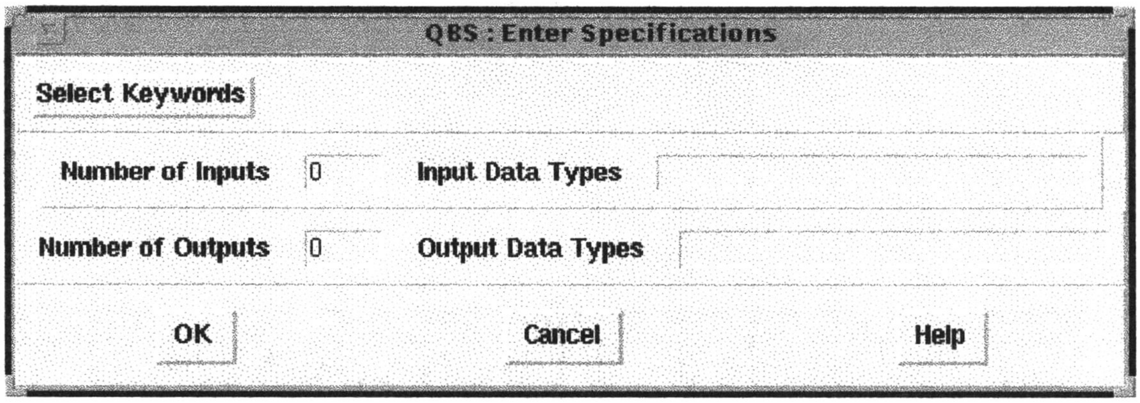
The OPEN transaction opens the O2 database if it has not already been opened by a previous request. If the transaction completes satisfactorily, a satisfactory message is sent to the client, else an error message is sent. An OPEN request is sent by the client to the server when the REQUES application is started. Figure 7 shows the REQUES menu selections.



Message area : Enter Specifications

Figure 7. REQUES Menu

The user must first select "Enter Specifications" from the menu. A new window is displayed by the interface program that permits the user to describe the semantic and syntactic descriptions of the desired component. Figure 8 below depicts the window presented to the user to enter component specifications.



Number of Inputs	0	Input Data Types	
Number of Outputs	0	Output Data Types	

Figure 8. REQUES Enter Specifications Window

When the "Select Keywords" button is selected from the "Enter Specifications" window, a KEYWORD request is sent to the server. When the server receives a KEYWORD transaction request, the O2 database is queried for all existing keywords which are then sent to the client.

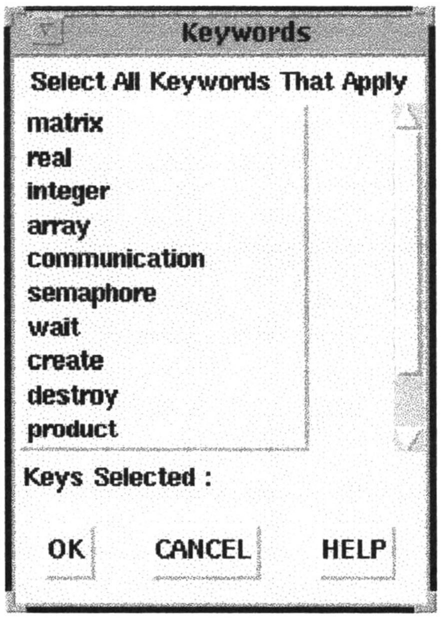


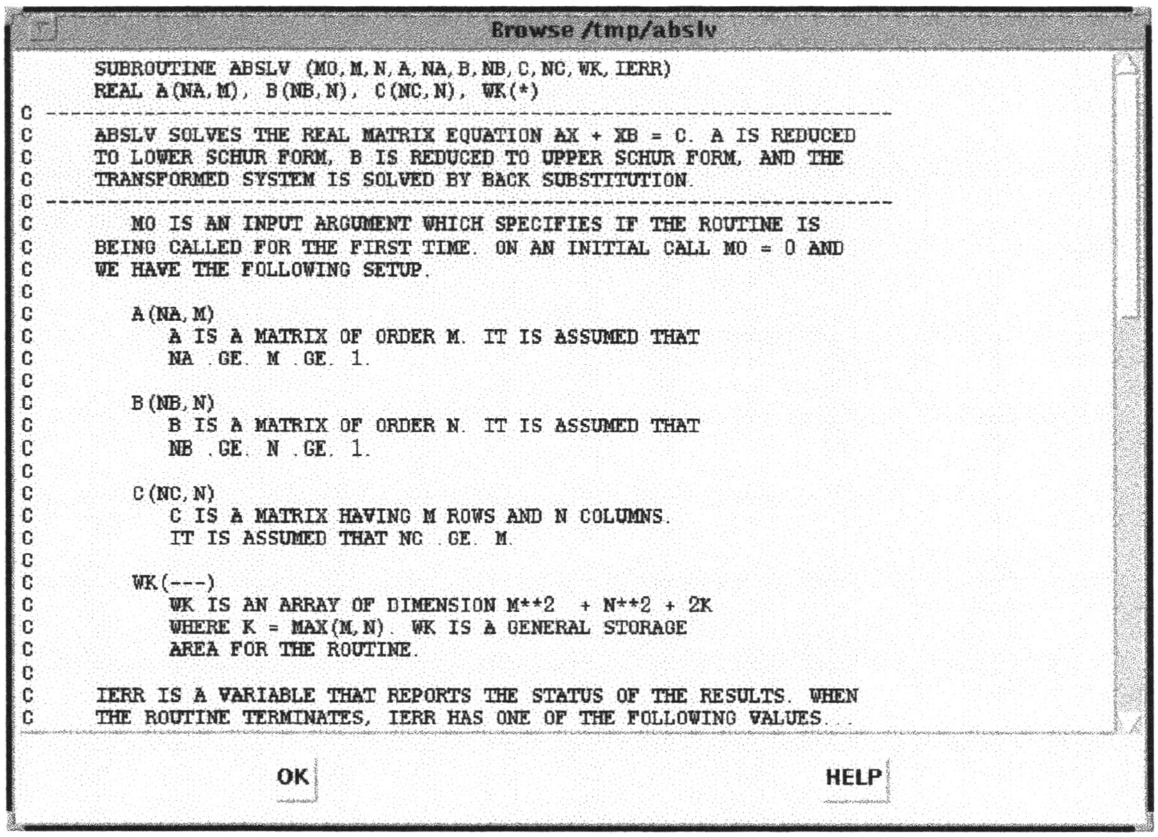
Figure 9. REQUES Select Keywords Window

When the client application requests and receives valid keywords, the executable, o2_get_keys, writes the keywords received to the file /tmp/keyword. This file is used by the interface program to allow the user to select keywords that describe the desired component.

The interface program includes functionality that permits the user to describe the signature of the function. This helps the system to retrieve components that have similar syntactic matches. The format of the input and output data types should be specified using C-like syntax. An array is specified by the data type followed by an open and close bracket, e.g., integer[] describes an integer array. A pointer data type is expressed by the data type followed by an asterisk, e.g., integer* indicates an integer pointer. All data types are accepted by the user interface program and passed as parameters to the server program. The interface program permits an empty set for the signature information, but the set of keywords must not be the empty set.

Once the keyword and signature data is collected, the interface program allows the user to select the query database option. The selected keywords and signature information are formatted and passed as parameters to the query executable, o2_query. A QUERY transaction request causes the server to query the database for all components in the library that match the accompanying component attributes. Those attributes include the keywords used to describe the semantic meaning of the component along with the data types of the inputs and outputs. The server first calls a function to convert the input and output data types to a normalized version of those types. Next, the server calls a function that uses SQL-like syntax to find components matching the input description. If matching components are found, they are transmitted to the client requesting the data. If no matching components are found, a message is transmitted indicating zero as the number of components found.

The executable writes the query result to /tmp/<name> where <name> is the program name of a retrieved component. Each retrieved component is written to a separate file. In addition, a file, /tmp/reuse_list is created. The /tmp/reuse_list file contains one line per retrieved component containing the name of the component and a brief description. The interface program allows the user to browse the components in /tmp/reuse_list and select components from the list to preview. An example of a retrieved component selected to be previewed is shown in Figure 10.



```
      SUBROUTINE ABSLV (MO,M,N,A,NA,B,NB,C,NC,WK,IERR)
      REAL A(NA,M), B(NB,N), C(NC,N), WK(*)
C ------------------------------------------------------------------
C     ABSLV SOLVES THE REAL MATRIX EQUATION AX + XB = C. A IS REDUCED
C     TO LOWER SCHUR FORM, B IS REDUCED TO UPPER SCHUR FORM, AND THE
C     TRANSFORMED SYSTEM IS SOLVED BY BACK SUBSTITUTION.
C ------------------------------------------------------------------
C        MO IS AN INPUT ARGUMENT WHICH SPECIFIES IF THE ROUTINE IS
C     BEING CALLED FOR THE FIRST TIME. ON AN INITIAL CALL MO = 0 AND
C     WE HAVE THE FOLLOWING SETUP.
C
C        A(NA,M)
C           A IS A MATRIX OF ORDER M. IT IS ASSUMED THAT
C           NA .GE. M .GE. 1.
C
C        B(NB,N)
C           B IS A MATRIX OF ORDER N. IT IS ASSUMED THAT
C           NB .GE. N .GE. 1.
C
C        C(NC,N)
C           C IS A MATRIX HAVING M ROWS AND N COLUMNS.
C           IT IS ASSUMED THAT NC .GE. M.
C
C        WK(---)
C           WK IS AN ARRAY OF DIMENSION M**2  + N**2 + 2K
C           WHERE K = MAX(M,N). WK IS A GENERAL STORAGE
C           AREA FOR THE ROUTINE.
C
C     IERR IS A VARIABLE THAT REPORTS THE STATUS OF THE RESULTS. WHEN
C     THE ROUTINE TERMINATES, IERR HAS ONE OF THE FOLLOWING VALUES...
```

Figure 10. REQUES Browse Component Window

Upon receipt of a CLOSE transaction request, the server decrements the number of clients accessing the database. If no clients are currently active, the database is closed. No return message is sent to the client when a CLOSE message is received.

If an error occurs at the host site, an error message is transmitted to the client application. The server decrements the number of active clients, and if no clients are communicating, the O2 database is closed. If the client receives an error message, it displays the text of the error and closes the socket to the server. The user cannot send any requests to the server until the server application at the host is corrected and restarted. Once an error is received, the user may only exit from the interface program.

It should be noted that all of the executables o2_open_db, o2_get_keys, o2_query, and o2_close_db are stand alone programs. It is an easy task to develop user interfaces that call these programs for environments where TCL cannot operate.

CHAPTER V
CONCLUSIONS/FUTURE WORK

A. Introduction

This chapter summarizes the work done in this research project. It explains the benefits an organization can derive from implementing a program database and compares the results of this research with an Information Repository. Finally, future research goals are stated.

B. Conclusions

This research describes the design and implementation of an object-oriented database to store and retrieve program information. It has focused on providing a practical solution for those organizations looking for a way to control aspects of the project support environment and to improve productivity without making a commitment to a specific CASE architecture. It is generally agreed that a centralized repository should be the basis of a project support environment. We have shown that it is practical to develop an off-the-shelf database application that is flexible enough to accommodate most business applications. This flexibility is achieved by basing the design on the taxonomy of software rather than targeting a specific application domain.

A query method to automatically retrieve components from the database was also defined. The method used emphasizes both the semantic description through the use of keywords to retrieve software components, and the syntactic description through the component signature. This approach allows an organization to tailor their design to reuse the available components by querying the Library based on a description of the functionality desired, minimizing modifications needed to reuse components while maximizing the reuse effort.

Many organizations have a 1st generation Integrated Project Support Environment with no reuse or ad hoc reuse capabilities. Their corporate cultures can not take the leap from this loosely coupled project support environment to the structured, unyielding methodologies imposed by many CASE tools. The Program Information Database offers a bridge between the two philosophies. Since most software development groups are familiar with database applications and no particular software engineering methodology is imposed, this application should be easily accepted and unobtrusive. The benefits to be gained by an organization utilizing this application are many including

- Reduced data redundancy,
- Data consistency,
- Data integrity,
- Data security, and
- Structured software reuse.

These benefits can translate into improved productivity and reduced software costs.

A general purpose DBMS was used in the implementation. This has an additional advantage of sharing the repository with other company data thus allowing an exchange of information. The unrestricted querying capabilities allows an organization to use the data in unanticipated ways.

A closely related concept is that of an Information Repository whose components include enterprise information, corporate business models, the corporate data architecture, and application descriptions and components [1]. The long term benefits of a repository are simplification of application maintenance because system components and information are managed by the repository and software reuse because the repository provides an inventory of reusable code and a means of easily depicting the code and components [1].

The initial implementation of the Program Information Database contains application descriptions and most related components. Because this initial implementation manages the components it contains and supports software reuse, it can be considered a subset of and is a significant step towards an Information Repository.

The specific contribution of this research is the design and implementation of a program database that can be easily used for any application which gives that application the immediate advantages of a central repository and the productivity gained from automated retrieval of reusable software components. It allows an organization to go immediately from having no or ad hoc software reuse to structured software reuse. It demonstrates that a Commercial Off The Shelf program database application is a feasible and worthwhile undertaking.

C. Future Work

The existing application does not address some specific issues of importance to a program database, configuration management and incremental compilation. Configuration management is one such issue; in theory, this should be made easier to implement with an OO DBMS. Due to space considerations, no object code is stored in our database at this time. As a result, we were unable to take advantage of an incremental compilation system. Both issues will be addressed in the near future.

A methodology to provide greater security measures in the matter of who has update capabilities and access permissions for all aspects of the system and not just for the Reuse Library should be implemented. A rudimentary capability has been implemented as part of the existing application. Each user must log into the application. Currently the login data is only compared to those users permitted to access the O2 database, therefore, any O2 user has all permissions for all components except the Reuse Library components. When this problem is corrected, the hard coded "Librarian" identification should be corrected as well.

Deciding what keywords should be a part of the standard vocabulary to classify and retrieve reusable software components is a critical task whose success determines the success of the Reuse Library. Any two individuals may select very different keywords to express the same functionality of a component. A case-based reasoning, CBR, approach as suggested by Chen, et. al. [4], is better suited for describing reusable components. The CBR approach is more robust and doesn't rely on knowledge abstracted from experience [4], e.g. an individual expert. It is a long term goal for this research project to incorporate case-based reasoning to classify and retrieve reusable software components.

APPENDIX A

CLASS DEFINITIONS

```
/*******************************************************************
Application Class
*******************************************************************/
class Application public type tuple
        (public appl_name : string,
         public appl_icon : Bitsmap,
         public doc : set(Documentation),
         public modules : set(Module),
         public run_commands : string,
         public compile_link : Text )

method title : string,
      init,
      menu : list(string),
      bitmap : tuple (width : integer, height : integer, bitsmap : bits),
      write_application,
      public Update,
      public Add_Documentation,
      public Add_Module,
      public Display_Modules,
      public Compile_Application,
      public Run_Application,
      public display_icon : Bitsmap
end;


/*******************************************************************
 Module Class
*******************************************************************/
class Module public type tuple
        (public module_name : string,
         public doc : set(Documentation),
         public pgms : set(Program),
         public description : Text,
         public variables : set(Variable))

method title : string,
      init,
      menu : list(string),
      public Browse,
      public Update,
      public Add_Documentation,
      public Add_Program,
      public Display_Programs,
      public Delete_Program,
      public List_Variables
end;

/*******************************************************************
 Source_Code Class
*******************************************************************/
class Source_Code public type tuple
         (public name : string,
          public short_description : string,
          public doc : set(Documentation),
          public language : string,
          public compiler_directive : string,
          public code : Text,
          public variables : unique set(Variable),
          public called_functions : unique set(Function),
          public called_procedures : unique set(Procedure),
          public library_functions : unique set(Library_Function),
          private last_modified : Date,
          private date_created : Date,
          private modified_by : string,
          private checked_out : boolean,
          private application : string,
          private where_used : set(Source_Code) )


method title : string,
      init,
      public menu : list(string),
      public Browse,
      public Update,
      public Add_Documentation,
      public Add_Variable,
      public List_Variables,
      public Delete_Variable,
      public Add_Function,
      public List_Functions,
      public Delete_Function,
      public Add_Procedure,
      public List_Procedure,
      public Delete_Procedure,
      private Where_Used : set (Source_Code)
end;
```

```
/*******************************************************************
 Program  Class
*******************************************************************/
class Program inherit Source_Code type tuple
     (public called_programs : set(Program))

method  init,
      title : string,
      menu : list(string),
      Browse,
      Update,
      public Add_Documentation,
      public Add_Variable,
      public List_Variables,
      public Delete_Variable,
      public Add_Function,
      public List_Functions,
      public Delete_Function,
      public Add_Procedure,
      public List_Procedure,
      public Delete_Procedure,
      public Add_Called_Program,
      public List_Called_Programs,
      public Delete_Called_Program,
      public write_program,
end;


/*******************************************************************
 Function Class
*******************************************************************/
class Function inherit Source_Code type tuple
        (public return_type : string)

method  menu : list(string),
      init,
      title : string,
      Browse,
      Update,
      public Add_Documentation,
      public Add_Variable,
      public List_Variables,
      public Delete_Variable,
      public Add_Function,
      public List_Functions,
      public Delete_Function,
      public Add_Procedure,
      public List_Procedure,
      public Delete_Procedure,
      public write_function
end;


/*******************************************************************
 Procedure Class
*******************************************************************/
class Procedure inherit Source_Code

method  init,
      title : string,
      menu : list(string),
      Browse,
      Update,
      public Add_Documentation,
      public Add_Variable,
      public List_Variables,
      public Delete_Variable,
      public Add_Function,
      public List_Functions,
      public Delete_Function,
      public Add_Procedure,
      public List_Procedure,
      public Delete_Procedure,
      public write_procedure
end;

/*******************************************************************
  Library_Function Class
*******************************************************************/
class Library_Function inherit Function public type
   tuple (input_types : set (string),
         normal_inputs : set (string),
         output_types : set (string),
         normal_outputs : set (string),
         keys : set (string),
         state_machine : boolean)

method public title : string,
          init,
          public menu : list(string),
        public update_keys,
        public Update
end;


/******************************************************************
 Library Class
******************************************************************/
class Library public type tuple
     (public name : string,
      public category : string,
      public doc : set(Documentation),
      public functions: set(Library_Function))

method  title : string,
      menu : list(string),
      init,
      public Add_Documentation,
      public List_Functions
end;


/********************************************************************
 Keyword Class
********************************************************************/
class Keyword public type tuple
      (public key : string,
       public facet : string,
       public functions : set(Library_Function))

    method private init (keyname : string, facet : string),
         public title : string,
         public menu : list(string),
         public Get_Facet_Type : string
end;


/********************************************************************
 Type Class
********************************************************************/
class Type public type
   tuple (user_type: string,
        base_type: string)
    method public title : string,
         public menu : list(string),
         public Get_Base_Type : string
end;


/******************************************************************
 Variable Class
*******************************************************************/
class Variable public type tuple
        (public var_name : string,
         public var_type : string,
         public description : Text,
         public first_declared : string,
         public where_used : set(Source_Code) )

method init,
      menu : list(string),
      public List_Usage,
      public Update,
      public del_usage (pgm : string)
end;


/******************************************************************
 Documentation Class
*******************************************************************/
class Documentation public type tuple
        (public type_documentation : string,
       public subject : string,
       public content : Text )

method title : string,
      init,
      menu : list (string),
      public Browse,
      public Update,
      public Write_Documentation,
      public Add_Documentation(doctype : Documentation) : integer
end;
```

BIBLIOGRAPHY

1. Advanced Testing Technologies, Inc. FAA Information Repository Report: Analysis and Recommendations. Technical Report. Feb., 1993.

2. Brown, A.W. Database Support for Software Engineering. John Wiley & Sons Inc. 1989.

3. Brown, Alan W., Carney, David J., Morris, Edwin J., Smith, Dennis B., and Zarella, Paul F. Principles of CASE Tool Integration. Carnegie Mellon University Software Engineering Institute. 1994.

4. Chen, Yufeng F., George, Roy, and Warsi, Nazir A. A Knowledge-Based Framework for Software Reuse Using Multiple-View Approach. Proceedings of the International Society for Computers and Their Applications (ISCA). International Conference. p. 66-70. San Francisco, CA. June, 1995.

5. Church, Terry and Matthews, Philip. An Evaluation of Object-Oriented CASE Tools: The Newbridge Experience. CASE '95 Proceedings Seventh International Workshop on Computer-Aided Software Engineering. p. 4-9. Toronto, Ontario, Canada. 1995.

6. Desai, Bipin C. An Introduction to Database Systems. West Publishing Co. 1990.

7. Deubler, H. H. and Koestler, M. Introducing Object Orientation into Large and Complex Systems. IEEE Transactions on Software Engineering. vol. 20. no. 11. p. 840-847. Nov., 1994.

8. Feijs, Loe. A Formalisation of Design Methods. Ellis Horwood Limited. 1993.

9. Frakes, William B. and Pole, Thomas P. An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering. vol. 20. no. 8. p 617- 629. Aug, 1994.

10. Godart, C. and Charoy, F. Databases for Software Engineering. Prentice Hall. 1993.

11. Hall, Patrick A. U. Proceedings of Software Engineering 90. Cambridge University Press, July, 1990.

12. Hardy, Colin, Thompson, Barrie, and Edwards, Helen. A Survey of SSADM Usage in the UK. School of Computing and Information Systems, Unversity of Sunderland, U.K. May, 1995.

13. Henninger, Scott. Supporting the Domain Lifecycle. CASE '95 Proceedings Seventh International Workshop on Computer-Aided Software Engineering. p. 10-19. Toronto, Ontario, Canada. 1995.

14. Jazayeri, Mehdi. Component Programming - a fresh look at software components. Technical Report. Technical University of Vienna. 1995.

15. Kaiser, Gail E. and Garnlan, David. Melding Software Systems From Reusable Building Blocks. IEEE Software. p. 17-24. July, 1987.

16. Kiesel, Norbert, Schuerr, Andy, and Westfechtel, Bernhard. GRAS, A Graph-Oriented (Software) Engineering Database System. Information Systems. vol. 20. no.1. p.21-51. 1995.

17. Liao, Hsian-Chou and Wang, Feng-Jian. Software Reuse Based on a Large Object-Oriented Library. ACM SIGSOFT Software Engineering Notes. vol. 18. no. 1. p. 74-80. Jan., 1993.

18. Lieberherr, K. J. and Xiao, C. Object Oriented Software Evolution. IEEE Transactions on Software Evolution. vol. 19. no. 4. p. 314-341. April, 1993.

19. Linton, Mark A. Distributed Management of a Software Database. IEEE Software. p. 70-76. Nov., 1987.

20. Luqi. Normalized Specifications for Identifying Reusable Software. Proceedings of Fall Joint Computer Conference (FJCC) '87. p. 46-49. Dallas, Tx. Oct., 1987.

21. Martin, James. Principles of Object Oriented Systems. James Martin Insight, Inc. 1991.

22. Miguel, Luis. The Design of a CASE Environment Architecture and the Performance Evaluation of Database Designs for Software Documents. University of California, Berkeley. Ph.D. Dissertation. 1992.

23. NATO Communications and Information Systems Agency. NATO Standard for Software Reuse Procedures. vol. 3. 1994.

24. Pietro-Diaz, R. Implementation of a Faceted Classification for Software Reuse. Communication of ACM. vol. 34. no. 5. p. 89-97. May, 1991.

25. Ramamoorthy, C.V., Garg, Vijay, and Prakash, Atul. Support for Reusability in Genesis. IEEE Transactions on Software Engineering. vol. 14. no. 8. p 1145-1153. Aug., 1988.

26. Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick, and Lorensen, William. Object Oriented Modeling and Design. Prentice Hall. 1992.

27. Sebasta, Robert W. Concepts of Programming Languages. The Benjamin Cummings Publishing Company, Inc. 1993.

28. Sommerville, Ian. Software Engineering. Addison-Wesley. 5th Edition. 1995.

29. Sorumgard, Lars Sinert, Sindre, Cuttorm, and Stokke, Frode. Experiences from Application of a Faceted Classification Scheme. Technical Report. Norweigein Institute of Technology. 1994.

30. Steigerwald, Robert Allen. Reusable Software Component Retrieval via Normalized Algebraic Specifications. Ph.D. Dissertation, Naval Postgraduate School. Dec., 1991.

31. Wohlin, Claes and Runeson, Per. Certification of Software Components. IEEE Transactions on Software Engineering. vol. 20. no. 6. p. 494-499. June, 1994.