INHERITANCE PROBLEMS IN OBJECT-ORIENTED DATABASE

DISSERTATION

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

by

Raweewan Auepanwiriyakul, B.S., M.S.
Denton, Texas
May, 1989

This research is concerned with inheritance as used in object-oriented database. More specifically, partial bi-directional inheritance among classes is examined. In partial inheritance, a class can inherit a proper subset of instance variables from another class. Two subclasses of the same superclass do not need to inherit the same proper subset of instance variables from their superclass. Bi-directional partial inheritance allows a class to inherit instance variables from its subclass.

The prototype of an object-oriented database that supports both full and partial bi-directional inheritance among classes was developed on top of an existing relational database management system. The prototype was tested with two database applications. One database application needs full and partial inheritance. The second database application required bi-directional inheritance. The result of this testing suggests both advantages and disadvantages of partial bi-directional inheritance. Future areas of research are also suggested.
ACKNOWLEDGEMENTS

This dissertation is dedicated with love, respect, and gratitude to my parents, Mr. Anant and Mrs. Saukeaw Auepanwiriyakul. They always tell me that education is the only treasure nobody can take away. Because of their love, support and confidence in their daughter, I am able to come this far.

To Professor Kathleen Swigger, I render my deepest gratitude. One of the most intelligent and efficient women I know, she made me feel fortunate to have her as my dissertation advisor.

Appreciation is expressed to all my committee members: Dr. Denis Conrady, Dr. Dana Wyatt, Dr. Robert Brazile, Dr. Roy Jacob, and Dr. Don William. I am grateful to each of them because the completion of this study would have been impossible without their guidance and advice.

Special thanks are also presented to Anto Prijosuesilo who helped me during the prototype’s development. His assistance, friendship and kindness will never be forgotten.

Finally, to Chairuch Vongsawangruessamee, who gave his moral support during my research, I hereby express my sincere gratitude. You do share part of this success.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF ILLUSTRATIONS</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Introductory statement</td>
<td></td>
</tr>
<tr>
<td>Statement of Problem</td>
<td></td>
</tr>
<tr>
<td>Rationale of the study</td>
<td></td>
</tr>
<tr>
<td>Proposed Methodology</td>
<td></td>
</tr>
<tr>
<td>II. REVIEW OF RELATED LITERATURE</td>
<td>16</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>Behavioral and Structural Object-Orientations</td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
</tr>
<tr>
<td>III. DEVELOPMENT</td>
<td>35</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>Basic Concept</td>
<td></td>
</tr>
<tr>
<td>Partial Inheritance</td>
<td></td>
</tr>
<tr>
<td>Bi-Directional Inheritance</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>IV. EVALUATION</td>
<td>53</td>
</tr>
<tr>
<td>First Database Application</td>
<td></td>
</tr>
<tr>
<td>The Reservation System Using the Prototype Database Management System</td>
<td></td>
</tr>
<tr>
<td>The Reservation System Using a Conventional Object-Oriented System</td>
<td></td>
</tr>
<tr>
<td>The Reservation Database Using a Relational Database Management System</td>
<td></td>
</tr>
<tr>
<td>The Orbital Mechanics Tutor Database System</td>
<td></td>
</tr>
<tr>
<td>The Database Design for the Orbital Mechanics Tutor Using the Prototype System</td>
<td></td>
</tr>
<tr>
<td>The Orbital Mechanics Tutor Database Using a Conventional Object-Oriented Database Management System</td>
<td></td>
</tr>
</tbody>
</table>
V. SUMMARY AND FUTURE RESEARCH

Summary
Future Research

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX A</td>
<td>80</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>84</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>86</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>88</td>
</tr>
<tr>
<td>APPENDIX E</td>
<td>91</td>
</tr>
<tr>
<td>APPENDIX F</td>
<td>102</td>
</tr>
<tr>
<td>APPENDIX G</td>
<td>104</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>106</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Subclass-Superclass Hierarchy</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>Subclass-Superclass Inheritance with Inherited Instance Variables</td>
<td>10</td>
</tr>
<tr>
<td>3.</td>
<td>SUBCLASS_TABLE</td>
<td>11</td>
</tr>
<tr>
<td>4.</td>
<td>Subclass-Superclass Inheritance with Inherited Variables</td>
<td>41</td>
</tr>
<tr>
<td>5.</td>
<td>SUBCLASS_TABLE</td>
<td>42</td>
</tr>
<tr>
<td>6.</td>
<td>SUBCLASS_TABLE</td>
<td>45</td>
</tr>
<tr>
<td>7.</td>
<td>Entity-Relationship Diagram</td>
<td>54</td>
</tr>
<tr>
<td>8.</td>
<td>Trace Expert Entities</td>
<td>68</td>
</tr>
</tbody>
</table>
 CHAPTER I

INTRODUCTION

The purpose of this study is to examine the question of inheritances as used in object-oriented database systems. More specifically, a methodology is discussed which supports both full and partial bi-directional inheritance among classes in an object-oriented database.

Introductory Statement

A typical object-oriented database organizes its set of objects into a subclass/superclass hierarchy similar to an IS-A relationship. In such a system, a subclass is a specialization of a superclass and, as such, inherits methods, attributes, and values from the superclass. However, there are numbers of real-world applications that require the designer to use something other than a hierarchical inheritance model that imposes a subclass/superclass structure. These applications require the database designer to specify different types of relations between objects such as LOOK-LIKE, FUNCTION-AS, and SIMILAR-TO, and, as such, require the designer to specify a unique set of inheritance rules. In short, real-world applications demand that the current object-oriented databases be made more flexible and robust.
In order to solve this problem, an object-oriented database system was developed that supports both full and partial inheritance. In order to support such a concept, the system reserves a place within the class definition that allows a user to specify both type and direction of inheritance of variables among the classes. The system also keeps a table of the superclass and subclasses along with their inherited instance variables. Thus, whenever values of these instance variables are changed, the values are propagated throughout the database. The system also supports bi-directional inheritance by allowing the user to specify which instance variables can be inherited from the subclasses.

A prototype of this system was built on top of an existing relational database management system. The prototype supports the creation of a class's definition, the creation of an object of a class, updating a value of the instance variable of an object, retrieving an object, retrieving a value of an object's instance variable, and bi-directional full and bi-directional partial inheritance.

The prototype was tested on two different database applications. One database application requires both partial and full inheritance. The other requires bi-directional partial inheritance. Results of these tests indicate both the advantages and limitations of this approach.
Statement of Problem

A database system is a collection of stored data along with their descriptions, and a hardware/software system that insures reliable and secure management, modification and retrieval of the data [DIT86]. The traditional database management systems use one of the three data models: relational, hierarchical and network. Although all three of these data models have been widely accepted and widely used, they have, nevertheless, proven insufficient and unnatural for certain types of applications. For example, most engineering domains are naturally decomposed into objects describing various machines, shapes, flows, etc. [KEM87]. These applications require a data model that allows the database designer to describe the existing system using a more natural approach. Object-oriented database systems are based on an object-oriented programming paradigm that allows the user to represent one conceptual entity, whatever its complexity and structure, by using only one object. Thus, an object-oriented model can represent an engineering problem in a more accurate and natural way than the traditional database model.

Object-oriented databases can be further distinguished by their structural and behavioral orientation [DIT86]. The behavioral approach defines objects according to their features and the set of operators that act on these objects. Instances of the object type can then be used by calling the
operations to manipulate the objects. The internal structure of the object is known only to the operator.

In contrast, the structural approach allows the designer to define data structures to represent complex entities. The operators, in turn, are written to manipulate these complex entities. The operators are not part of the specific data structures, but rather are defined at the system level. Thus, operators in the structural approach are viewed as more general in nature and use.

Regardless of which approach is used, both object-oriented approaches combine three properties: encapsulation, inheritance and organization. Each object has its private memory (called an instance variable) and in the behavioral approach, local functions. The object-oriented approach organizes this set of encapsulated objects into a subclass/superclass hierarchy. This hierarchy serves to organize similar objects into groups called classes.

However, there are some applications and data descriptions that require a different type of organization than the subclass/superclass relationship. Such applications need inheritance between classes that might be termed LOOK-LIKE, FUNCTION-AS, SOUND-LIKE, etc. Each of these types of relationships requires a different proper subset of instance variables. Further, such relationships may not require the system to support full inheritance of all variables. For example, a LOOK-LIKE relationship
between class A and class B (see figure 1) means that an object in class B looks like an object in class A. As a result, an object in class B should inherit everything in class A that pertains to the appearance relationship.

Furthermore, two such classes of the same superclass should be able to inherit different groups of instance variables of the superclass. For example, class B and class C are subclasses of class A. In figure 1, class A has instance variables that can be grouped into two categories, appearance variables and function variables. Appearance variables are instance variables that describe the appearances of the objects in class A. Function variables are instance variables that describe the functions of the objects in the class.

![Subclass-Superclass Hierarchy](image)

Figure 1: Subclass-Superclass Hierarchy

Objects in class B may need to inherit instance variables about appearance so that the objects in class B will look like objects in class A. On the other hand, objects in class C may need to inherit instance variables
about function so that objects in class C will have the same function as objects in class A.

Although partial inheritance has obvious applications within a database environment, there are currently no object-oriented databases that support this feature. The current object-oriented databases support only full inheritance [BAN88, BAN87, DER86, MAN86, PEN87, WEI88, ZDO85].

A second problem that relates to the inheritance issue concerns the use of bi-directional inheritance with respect to information sharing. As mentioned previously, object-oriented databases support subclass/superclass inheritance. Such a feature is extremely useful if the data can be organized into some type of hierarchy. However, there are many database systems which do not support bi-directional inheritance. For example, a designer may want to set up a database in which the subclass passes variables to the superclass, or a designer may wish to organize the data into a graph-like structure. If the database system allows bi-directional inheritance, then the designer can specify which instance variables of the subclass can send data to the superclass, and how this data will be stored within the superclass.

However, current object-oriented systems only allow designers to specify subclass-superclass inheritance [HUD87, PEN87, MAI85, MAI86, MAN86, BAN87, ZDO85, SMI87]. Bi-
directional inheritance is not currently supported by any of the above systems.

Rationale of the Study

The purpose of this study is to examine the problems of inheritance as it relates to object-oriented databases. The inheritance feature of object-oriented databases should be expanded to include both partial and bi-directional inheritance. The rationale for including partial inheritance is that a subclass entity often does not need to inherit every instance variable from a superclass. Indeed, the subclass only needs to inherit a subset of the superclass' instance variables and does not need to access the rest of the superclass' instance variables. If an object-oriented database supports partial inheritance, memory can be saved because fewer instance variables will be inherited by the subclass and thus, fewer instance variables will be copied into the subclass's memory location. Additional memory can be saved if the same type of storage system is used in cases that require multiple inheritance. For example, given a database with the superclasses TRUCK and TOY, and subclass TOY-TRUCK, it is natural to assume that class TOY-TRUCK will inherit only appearance variables from class TRUCK and function variables from the class TOY. Memory is saved if class TOY-TRUCK does not inherit the rest of class TRUCK's instance variables. Further memory is
saved if class TOY-TRUCK does not inherit the rest of class TOY's instance variables.

A second reason for implementing a database system with partial inheritance is to give a subclass more precise meaning. The instance variables that do not apply to a subclass will not be inherited by the subclass, and the subclass will not be able to access those instance variables. For example, class TOY-DUCK is a subclass of class DUCK, but it only needs to inherit instance variables that make the object in class TOY-DUCK look like the object in class DUCK. TOY-DUCK should not inherit other instance variables about a living animal such as GIVE-BIRTH, EATING etc. Thus, when one examines the TOY-DUCK object, it will only contain the variables that give it meaning.

A system should also support bi-directional inheritance because there are many applications that require a subclass to pass back some of its instance variables to its superclass. In turn, the superclass needs to retain the information about the subclass so that it can be accessed by the superclass at a later time. For example, a tutoring system is divided into several submodules. The top level of the system needs to keep track of different types of error messages that are produced by the submodules. Each submodule has to determine a certain kind of error. If it detects an error, the error message must be sent to the top level. After the student has made a certain number of
errors, the system displays the most common error to the student. This tutoring system needs bi-directional partial inheritance.

In order to model the database applications that require the designer to use something other than a hierarchical inheritance model, an object-oriented database system should support both bi-directional full and bi-directional partial inheritance.

Methodology

Given the above rationale, a prototype object-oriented database was developed that supports both full and partial inheritance by providing a place within the class's definition so that a class can specify which instance variables it needs to inherit from its superclass. In this manner, a subclass inherits the definitions of the instance variables, or the definitions and the values of the instance variables. The system also keeps a record of the superclass, its subclasses and their inherited instance variables, especially the subclass that inherits the definitions and the values of the instance variables. Whenever the values of these instance variables are changed, the values are propagated throughout the hierarchy. In order to support this, there is a table in the system, called SUBCLASS_TABLE, that keeps track of the subclasses and instance variables that are inherited by each subclass.
The system also uses a flag to indicate the types of inheritance.

For example, in figure 2, class A has three instance variables, Ax, Ay, and Az. Both the definitions and the values for Ax and Az are defined in the class' definition. Only the definition of Ay is defined in class A. Any subclasses that need to inherit Ax, Az, or both inherit both their definitions and their values. Any subclasses that need to inherit Ay inherit only its definition.

![Class A diagram](image)

Class A
Ax=10, Ay and Az=yes are instance variables

Class B
Ax is inherited instance variable

Class C
Ay and Az=yes are inherited instance variables

Figure 2: Subclass-Superclass Inheritance with Inherited Instance Variables

Class B and class C are the subclasses of class A. If class B needs to inherit Ax, class B specifies in its definition that it needs to inherit Ax. The system then looks to see whether Ax has its value at the class level. If so, class B inherits both the definition and the value of Ax. The system then adds class A, class B, Ax, and 1 to its SUBCLASS_TABLE in its private memory. Now, when class C
needs to inherit Ay and Az, it also specifies this in its definition. However, class C inherits both the definition and the value of Az, and only the definition of Ay. The system adds class A, class C, Az, and 1 to its SUBCLASS_TABLE. It also adds class A, class C, Ay, and -1 in the SUBCLASS_TABLE. At this point, the SUBCLASS_TABLE looks like figure 3.

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclass</th>
<th>instance variables</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>Class B</td>
<td>Ax</td>
<td>1</td>
</tr>
<tr>
<td>Class A</td>
<td>Class C</td>
<td>Az</td>
<td>1</td>
</tr>
<tr>
<td>Class C</td>
<td>Class C</td>
<td>Ay</td>
<td>-1</td>
</tr>
</tbody>
</table>

Figure 3: SUBCLASS_TABLE

If there is a change in the value of Ax in class A, the system looks at the subclass table to see whether Ax is inherited by any subclasses. If this is the case, then Ax is changed to the new value in every subclass.

If class D is another subclass of class A, and it does not specify which instance variables it needs to inherit, it inherits all instance variables of class A. If any subclasses do not need to inherit any instance variables from the superclass, the keyword NONE is specified.

The system provides a place in the class’s definition which allows a class to specify its superclass and inherited instance variables. The system also provides a place within a class’s definition that allows a class to specify which
instance variables it needs to inherit from its subclass. When an object of a class is created, the inherited instance variables are not initialized. The instance variables receive their values when the subclass sends them to the superclass which has the corresponding definitions. Every object in this superclass receives the values. Therefore, the system supports bi-directional inheritance.

The above prototype was constructed on top of an existing relational database. The prototype supports the creation of a class’s definition, the creation of an object of a class, updating the value of the instance variable of an object, retrieving the object, retrieving a value of an object’s instance variable, and bi-directional full and bi-directional partial inheritance.

After the prototype was constructed, database applications that require both full and partial inheritance were created as examples. They were tested to determine whether the prototype system worked as expected. The test plan consisted of:

1. Retrieving an object to determine whether its instance variables correspond with its class definition and its inheritance.

2. Updating a value of an inherited instance variable of an object in a superclass. If a subclass inherited both the definition and the value of the instance variable, the value of the inherited instance variable in
the subclass was also changed. If a subclass inherited only the definition of the instance variable, its value in the subclass was not changed.

3. Having a superclass inherit instance variables from its subclass. If a subclass invoked a special method to send values to the superclass, then the values of inherited instance variables in the superclass were updated.

The results of the prototype operation indicate that it works properly and that there are many advantages in using both full and partial bi-directional inheritance among classes in an object-oriented database systems. These advantages are discussed in detail in chapter V. The results also indicate that this system provides more flexibility for the user than conventional object-oriented database systems. Bi-directional partial inheritance is a feature that should be included in object-oriented database systems.

The remainder of this paper is structured as follows. In Chapter II, the related literature is reviewed. Chapter III describes the prototype of an object-oriented database system which supports bi-directional partial inheritance. Chapter IV provides an evaluation of the prototype. Finally, Chapter V discusses advantages of bi-directional partial inheritance and states the conclusion of this research.
CHAPTER BIBLIOGRAPHY


CHAPTER II

REVIEW OF RELATED LITERATURE

A significant amount of literature about object-oriented database systems already exists. Therefore, this review will concentrate on the literature relevant to existing object-oriented database systems and to the inheritance problem as it relates to these systems.

The first section presents an introduction to object-oriented database systems. It describes the differences between object-oriented database systems and more conventional database systems, and provides a rationale for selecting an object-oriented database approach over existing database models. Object-oriented database systems are further divided into two categories, behavioral and structural object-oriented systems. Consequently, the second section describes the differences between the behavioral and structural approach in object-oriented design. The last section describes different inheritance problems and possible solutions to these problems. The inheritance problems discussed in this section include name conflicts which occur within multiple inheritance and incomplete inheritance.
Introduction

The concept of an object and a class was first introduced into programming language in 1962 with the programming language Simula [NYG86]. Simula referred to classes as named activities and objects as processes. This concept of a class/object data structure later formed the basis of several object-oriented programming languages such as Smalltalk, Flavors, LOOPS, and object-Lisp [WOE86]. Following the emergence of object-oriented programming languages, several proposals were made for using the object-oriented concept within a database environment. Examples of such systems include GemStone [COP84, MAI85, MAI86, PEN87], Iris [DER85, DER86, FIS87, LYN86], ORION [BAN87, CHO87, KIM87], PROBE Data Model (PDM) [MAN86], ENCORE [SMI87, ZDO85], Object Oriented Programming System (OOPS) [SCH88], Cactis [HUD86, HUD87], and Statice [WEI88].

An object-oriented system, regardless of whether it is a programming language or a database system, represents conceptual entities as objects. An object can represent anything from a simple number to a series of complex entities. Each object consists of its own private memory and contains the values of its instance variables.

The behavior of an object is encapsulated in procedures called "methods." Methods are procedures or code that manipulate other objects and return values of instance variables. Methods are part of the object itself and are
not visible from outside the object. Objects communicate with each other by sending messages to themselves and other objects. Messages, together with any arguments attached to the messages, present the public interface of an object. For each message, there is a corresponding method that executes the message. Similar objects are grouped into a class. All objects of the same class have the same instance variables and methods. Objects that belong to a class are called instances of that class. A class can have none, one, or more than one superclasses. A class inherits every instance variable and method from its superclass. It can have additional instance variables and methods. This subclass/superclass relationship is represented by a class hierarchy [CHO87]. Some object-oriented database systems support multiple inheritance; that is, a subclass can have multiple superclasses. A class hierarchy is also represented by a lattice. For those systems that do not support multiple inheritance, a class hierarchy is represented as a tree.

The idea of a class/subclass data structure, coupled with inheritance of attributes and values, appears to be one solution to many of the problems that have plagued more traditional database models. Several authors [SCH88, COP84, SMI87, RAM88, HUD87] have attempted to catalog these problems, and a brief review of these criticisms are listed as follows:
1. Most traditional database systems (hierarchical, relational, and network) do not support sophisticated data types that describe the structure of a complex object in both a natural and efficient manner. Traditional database systems supply the database manager with a fixed number of simple data types such as integer, real, and character. Traditional database systems lack the ability to define new, more complicated data types.

2. Traditional database management systems cannot express complex objects. An object is often divided into several database objects. The data structuring capabilities of existing techniques do not adequately support the complexities and variations that occur in real data.

3. Operational semantics of complex objects cannot be expressible. In a traditional database management system, a user cannot include operations of the object in the database; he can only specify them in his application program.

4. The existing database systems do not support a hierarchy of types and inheritance of properties along a hierarchy.

5. Existing database systems force the user to specify constraint checking in the application program, not in the update command. Since every data item can have constraints attached to it, the update command should include constraint checking to preserve integrity.
6. Traditional database systems do not support a data manipulation language that is capable of arbitrary computation on database objects. The user usually must learn two languages: the data manipulation language and a general-purpose language such as C or PASCAL. These two languages may have some different data structures and programming paradigms.

7. Most traditional database systems lack efficient functions for complex object storage and retrieval. Because complex objects are stored as a series of database objects, a number of queries are needed to retrieve a single object from the database. Similarly, a number of queries are needed to store a single object.

Object-oriented database systems can overcome many of the limitations listed above. Because object-oriented database systems allow the user to specify new data types and operations to those data types, the limitations listed above under items 1, 3, 5 and 7 are satisfied. Also, object-oriented database systems allow the user to represent any type of entity. Such systems support class hierarchy, particularly the subclass/superclass relationship. Not only do object-oriented database systems overcome most of the problems associated with traditional database systems, they also provide many additional features [MAI86, MAI86a, DER86, FIS87]. Some of these features are summarized as follows:
1. They allow the designer to represent complex design entities in a more direct manner than the traditional database management systems.

2. They enhance efficiency. Because all operations are performed in the database, data does not have to be transferred from the database to the application program. Moreover, a single method corresponds to a number of database queries and updates. Thus, communication between database and application programs are reduced.

3. They increase data dependency. Application programs access data in object-oriented database systems by sending messages. Therefore, unnecessary details are removed from the application program. If the structure of the database changes, the application program does not need to be changed as long as the messages are not changed.

4. They facilitate information hiding. Application programmers do not have to worry about the details of retrieving data, finding their data, and checking constraints, because these items are coded into the operations in the database management system.

5. By storing an operation in the database, a system administrator can ensure that there exists only one copy of that operation. Every application program can be forced to use the same set of operations. Thus, application programs that send the same message with similar arguments will get the same result.
6. They insure integrity of the data. By restricting database access to sending messages to specify objects, a database administrator can reduce the risk of intentional or unintentional corruption or disclosure of data by an application program. The system can also check the consistency of the data in the same operation.

7. They allow the designer to specify object identity to distinguish the object from all other objects. In the relational data model, the properties of an entity must be sufficient to distinguish it from other entities. However, properties of an entity may not be sufficient to identify it in the real world.

As a result of the advantages listed above, many authors have proposed database systems that use object-oriented concepts in their data model. These systems are intended to meet the needs of several new database applications such as office information system, knowledge-based systems, CAD/CAM applications, geographic information systems, and hardware and software design systems [FIS87, MAN86, CH087].

Behavioral and Structural Object-Orientations

Object-oriented database systems can be further divided into two categories, structural and behavioral object-orientations [DIT86]. The behavioral object-oriented database systems are based on either the concept of object-oriented programming language or the functional data model.
The behavioral approach defines objects according to their features and a set of operations (methods) that act upon these objects. Instances of the object type can be accessed by sending a message to the object. One message corresponds to one operation. The internal structure of the object is known only to the operations. The interface between objects is the set of messages that manipulate that object.

Examples of behavioral object-oriented database systems include GemStone [MAI86, PEN87, COP84, MAI85], Iris [DER85, LYN86, DER86, FIS87], PDM [MAN86], ORION [CHO87, KIM87, BAN87], ENCORE [SMI87, ZDO85], and OOPS [SCH88].

One group of behavioral object-oriented databases uses a functional data model to describe its application [DER85, LYN86, DER86, FIS87, MAN86]. For example, in the DAPLEX system [SHI81] an instance variable is a function that maps a class or an object to a value. Iris [DER85, DER86, LYN86, FIS87] and PDM [MAN86] use a similar model, but allow the user to define methods using only a set of predefined functions and instance variables. Methods in the Iris and PDM systems do not belong to a particular class or object, but rather take one or more classes (or objects) as their arguments.

Other behavioral object-oriented databases borrow concepts from the object-oriented programming paradigm, particularly in their use of abstract data types coupled with the idea of "persistent" objects [MAI86, PEN87, COP84,
A system called ORION [CHO87, KIM87, BAN87] also includes the concept of shared-value and default-valued instance variables. A shared-value instance variable is, as the name implies, a variable that takes on the same value as another object. This concept is similar to the idea of a class variable as it is defined in object-oriented programming languages. The default-valued instance variable allows the user to specify a default value for any unassigned variable.

Behavioral object-oriented databases usually specify operations (methods) in two ways. One approach is to develop a special purpose database language that will allow the user to send messages to objects. These languages are usually compiled and stored in the database itself. The internal form is later interpreted in the same way that compiled queries are stored and executed in a relational database. One of the main advantages of this type of approach is that the language can be tailored to the database management system. Unfortunately, one of the disadvantages of using this approach is that the user will have to learn two languages: a database language that is used to query the database and a general-purpose programming language that is used to write the application programs. Moreover, the database manager must also write a compiler or interpreter for the database language. Behavioral object-oriented systems using this approach are GemStone [MAI86,
PEN87, COP84, MAI85], PDM [MAN86], Iris [DER85, LYN86, DER86] and OOPS [SCH88].

The second way to specify methods is through the use of an existing programming language. Advantages to using this approach are that the user needs to learn only one language, and the database manager does not have to write a separate compiler. A behavioral object-oriented system that uses this approach is ORION [CHO87, KIM87, BAN87].

Whenever an operation is defined in an object-oriented database, there is a question of who "owns" the operation. Behavioral object-oriented systems generally use one of two approaches to assign ownership of methods. Ownership can be associated with a class (type) with the assumption that each object belongs to a specific class. This approach can also be thought of as a form of data abstraction and has been used by object-oriented systems such as ORION, GemStone, OOPS, and ENCORE.

The second approach for determining ownership of methods assumes that the methods belong to no one. Objects and classes are merely tokens that are passed like parameters to operations. Operations can act upon multiple objects or classes. Behavioral object-oriented database systems using this approach are Iris and PDM.

The structural object-oriented database systems are database systems that use a data model that allows the user to define data structures to represent entities of any
complexity. The data model often includes generic operators to deal with complex objects. These operators are provided by the system and do not belong to any particular class of objects. Every object in the database can use these predefined operators. This concept is in contrast to the behavioral object-orientation approach in that the behavioral approach allows users to specify the class along with the operators (methods) that manipulate the objects of that class. Some examples of the structural object-oriented database systems include Postgres [STO86, ROW86, KEM87], Statice [WEI88], Object-Oriented Data Model (OODM) [ZHA88], and Cactis [HUD86, HUD87].

Postgres [STO86, ROW86, KEM87] is a structural object-oriented database system built on top of the relational database system called INGRES. Postgres supports complex objects by using an extensible type system for defining new columns for relationship, new operations on these columns, and new access methods.

Statice [WEI88] is a structural object-oriented database system based on DAPLEX. Under this system, whenever a user creates a class and specifies its attributes and their types, the system automatically creates an accessor function for each attribute. An accessor function takes an object as its argument and returns the value of an attribute of that object. Statice also provides a set of
system functions that can create and update every object in every class.

The Entity-Relationship model (E-R model) of Chen [CHE76], the Relational Model/Tasmania (RM/T) of Codd [COD79], and the Semantic Data Model (SDM) of McLeod and Hammer [HAM78] have directed the development of a system called Object-Oriented Data Model (OODM) [ZHA88]. This system consists of two parts, a conceptual schema and data operations. The conceptual schema represents classes, objects and relationships between these classes. Data operations, which are provided by the system, include functions for defining schema, creating the database, and manipulating objects.

Cactis [HUD86, HUD87] is based on a principle called active semantics and can support complex functionally-defined data. Under Cactis, attributes of an object can be intrinsic or derived. Derived attributes have an attribution rule attached to them, while intrinsic attributes do not. These rules allow attributes to be derived from other attributes within a given instance and from the values contained in related instances. Cactis provides a number of data manipulation primitives such as creating and deleting class objects, retrieving and replacing attribute values, etc. Each object in Cactis is designed to respond to a standard predefined set of messages. After a user creates classes, Cactis provides the
actual methods which respond to these messages for each class in the system. All objects respond to the same set of messages, but objects of different classes will respond to the same message in different ways.

There are several approaches that have been used to develop a structural object-oriented database systems. These approaches are based on the relational database model, the functional data model, and active semantics.

Inheritance

The idea that classes should inherit variables and values from higher-ordered (super) classes was first introduced by the language Simula [MEY86]. From its first inception, inheritance has been widely used in other object-oriented programming languages and database systems. The basic idea is that a new class may be defined as an extension of a previously defined class.

In most object-oriented database systems, objects are grouped into classes by similarity. Classes are then arranged in a class hierarchy. A class hierarchy is a hierarchy of classes in which an edge between a pair of nodes represents the IS-A relationship; that is, the lower-level node is a specialization of the higher-level node. For each pair of classes in a class hierarchy, the higher-level class is called a superclass of the lower-level class, and the lower-level class is called a subclass of the higher-level class. The instance variables and methods that
are specified for a class are inherited by all its subclasses. Additional properties may be specified for each of the subclasses. If a class has more than one superclass, then it is said to have multiple inheritance.

There have been several proposals and implementations of the inheritance property in different object-oriented programming languages. Versions of multiple inheritance were described in [BAN87, SNY86, MIN87], and incomplete inheritance in [SNY86].

One of the major research issues relating to the question of inheritance concerns multiple inheritance and the problem of name conflicts. Since a class may inherit instance variables from two or more superclasses, inherited instance variables may contain name conflicts between the superclass and subclasses. In most systems, name conflicts between a class and its superclasses are resolved by giving precedence to the definition within the class over that in the superclasses. Name conflicts among superclasses usually are resolved by giving priority to the first superclass in the list of superclasses of a given class. This solution is used by ORION [BAN87]. An alternative conflict resolution strategy is found in Extended Smalltalk [SNY86] which gives an error message whenever the system notices a name conflict. A third solution is to first flatten the inheritance graph to a linear chain, eliminate duplicates, and then treat the result as a single inheritance. The
systems that use this strategy are Flavors and CommonLoops [SNY86]. Name conflicts between a class and its superclass can also occur in single inheritance systems and are resolved by using the same techniques described above.

The problem of incomplete inheritance is examined by Snyder [SNY86], who suggests that one solution to the problem is to exclude inherited operations (methods) from the subclass’s own external interface. This technique might be both reasonable and useful if inheritance is viewed as an implementation technique as in object-oriented programming languages. According to Snyder, a class can inherit some or all methods from another class (superclass). Snyder also suggests that the way to implement stack and deque is to define a class called stack which inherits from the class called deque, but excludes the extra methods that are not applicable for stack. Stack is not a specialization of deque because it does not have all the deque methods; it just has some methods that are already defined for deque.

The system that provides incomplete inheritance is CommonObjects [SNY86]. Although Snyder’s system of incomplete inheritance deals with partial inherited methods, he does not explain whether he allows partial inherited instance variables.

Summary

As this section indicates, object-oriented databases have many advantages over more traditional database systems.
Because of their greater flexibility, object-oriented databases are more suitable for certain classes of problems.

However, before one begins to design an object-oriented database, he must first make certain kinds of decisions. First, the designer must decide whether to implement the object-oriented database using a behavioral or structural approach. If the designer decides to use a behavioral approach, then he must further decide whether to use a special-purpose database language or an existing programming language to specify methods. Also, the designer must answer the question of who "owns" an operation. The final question involves the use of multiple inheritance and the resolution of name conflicts. If the designer selects multiple inheritance, then he must also decide on a method to resolve conflicts between two inherited variables, and between two inherited methods.

The object-oriented database system described in the following chapters adopts a behavioral approach while using an existing programming language to specify methods. In the prototype system, methods belong to a class. Further, the system allows multiple inheritance with error messages displayed whenever name conflicts occur. This object-oriented database also supports both full and partial bidirectional inheritance among classes.
CHAPTER BIBLIOGRAPHY


CHAPTER III

DEVELOPMENT

This chapter describes the prototype of an object-oriented database system which supports bi-directional partial inheritance. The chapter discusses the overall basic concepts of the system, explains the use of bi-directional partial inheritance, and gives a detailed description of the user interface.

Basic Concept

The object-oriented database described in this chapter is based on the behavioral approach of object-oriented database design which, in turn, is based on concepts proposed by developers of object-oriented programming languages. According to these principles, each object is defined according to its instance variables and the methods that act upon it. Objects are further grouped into classes and subclasses in order to facilitate efficiency and conceptual simplicity. All objects that belong to the same class are described by the same instance variables.

There are three types of instance variables that are usually defined in object-oriented databases: shared-value, default-valued, and normal instance variables. Shared-value instance variables, sometimes referred to as class
variables, were first introduced in the ORION [BAN87] system which proposed that all objects of the same class assume specific values for shared-value instance variables.

Objects of a class that do not have specified values are assigned values by the system. These latter type of variables are called default-valued instance variables and are considered an extension of the shared-value type of instance variable because they allow objects to take on the value of a default value, if no other value has been specified. At the same time, default-valued instance variables allow the user to specify exception values. For example, most people have two legs. However, if a person has an accident and one of his legs gets cut off, then that person has only one leg. If the value for the Leg instance variable is defined as a shared-value instance variable, then the value for this variable is equal to two and can never be changed or updated. However, if Leg is defined as a default-valued instance variable, then its value can be changed to ONE for this special case.

Finally, normal instance variables are instance variables that have only their definitions specified in the class's definition. Because shared-value types are really a special case of default-valued instance variables, the object-oriented database system described in this paper supports only default-valued and normal instance variables.
Normally, methods are "owned" by either classes, objects, or no one. If a single object owns the methods, then the system is forced to maintain large amounts of redundant information. On the other hand, if no one owns the methods, then objects that belong to the same class may not behave in the same way, since an object's behavior is determined by an object's methods. Therefore, as a compromise, this database system specifies that classes own methods. A single method corresponds to one message. An object can communicate to another object by sending it a message. All objects in a single class respond to the same messages.

In order to define a method, the database designer is given the choice of either using a special database language or an existing programming languages such as C. The prototype was written using the C programming language. One method is equivalent to one function. As a result, the database manager does not have to write a compiler or an interpreter for the database language. Moreover, a user does not have to learn two languages.

Normally, an object-oriented database management system allows the designer to group similar objects into classes. This particular feature tends to reduce the amount of necessary storage as well as eliminates duplicate information. The concept of class hierarchy extends this feature even further. In most object-oriented database
systems, a class hierarchy is one in which an edge between a node and a child is represented by an IS-A relationship. IS-A relationships are defined as relationships in which the child node is a specialization of the parent node (and the parent node is a generalization of the child node). The parent node is called the superclass of the child. The child node is called a subclass of the parent node. Instance variables and methods specified at the class level are inherited by all the classes at the subclass level. In addition, instance variables and methods may be specified for each class at the subclass level. This phenomenon is called full inheritance.

Some object-oriented database systems limit the class size to that of a single superclass with single inheritance. Additionally, a class hierarchy may be limited to a tree structure. Most object-oriented database systems have relaxed this restriction and now allow the designer to specify more than one superclass. This phenomenon is commonly referred to as multiple inheritance [BAN87, SNY86, MIN86] and means that a subclass may have more than one superclass. Unfortunately, multiple inheritance may also introduce conflicting names into the database. There are two types of name conflicts that can occur under this approach; conflicts between a class and its superclass and conflicts between the superclasses of a single class. In the object-oriented database system described in this paper,
an error message is displayed to the user whenever the
tool encounters a duplicate names for instance variables,
methods or classes. The error message is displayed while
the database designer is creating the class.

Partial Inheritance

Currently, object-oriented database systems [HUD87,
PEN87, MAI85, MAI86, MAN86, BAN87, ZDO85, SMI87, SNY86] support only full inheritance. Full inheritance means that a subclass inherits all instance variables and methods from its superclasses. The relationship between subclass and superclass is restricted to an IS-A relationship. An IS-A relationship implies full inheritance. Unfortunately, there are some relationships that do not fall under the category of an IS-A relationship. Examples of such relationships include LOOK-LIKE, FUNCTION-AS, etc. With these types of relationships, the class should inherit only a proper subset of instance variables from another class. The solution, of course, is to relax the subclass/superclass restriction and allow the class to inherit a proper subset of the superclass's instance variables. This feature is called partial inheritance. As an additional feature of partial inheritance is that two subclasses of the same superclass do not have to inherit the same proper subset of instance variables from their superclasses. For example, class B, and class C might be subclasses of class A. The relationship between class A and B might be a LOOK-LIKE
relationship, whereas the relationship between class A and class C is a FUNCTION-AS relationship. In the LOOK-LIKE relationship, class B inherits only the instance variables that pertain to a description of the object, while in the FUNCTION-AS relationship class C inherits the object's functions. Such an example shows that the relationships between classes should be extended beyond the IS-A relationship category.

The database system described in this paper allows partial inheritance of instance variables. Whenever a class is created, the system designates a place within the class's definition that specifies which superclasses and instance variables are inherited. If the relationship between the subclass and superclass is an IS-A relationship and needs full inheritance, then the user may specify the keyword "all." If the user does not want the subclass to inherit any instance variables from its superclass, then the user specifies the keyword "none." The user can also specify which subset of instance variables will be inherited from the superclass.

In the prototype system, a subclass can inherit only definitions of the instance variables, or the definition and the values of the instance variables. If the inherited instance variable is a default-valued instance variable, a subclass inherits both the definition and the value of that instance variable. If the inherited instance variable is a
normal instance variable, a subclass inherits only the definition of that instance variable. The system keeps a table called SUBCLASS_TABLE that contains four columns: superclass, subclass, inherited instance variable, and flag. The flag is set to 1 if the inherited instance variable is a default-valued instance variable. If the inherited instance variable is a normal instance variable, the flag is set to -1.

Figure 4: Subclass-Superclass Inheritance with Inherited Instance Variables

In figure 4, class A has three instance variables, Ax, Ay and Az. Ax and Ay are default-valued instance variables. Az is a normal instance variable. Both the definitions and the values of Ax and Ay are defined in the class's definition. Only the definition of Az is defined in class A. Any subclasses that need to inherit Ax, Ay, or both Ax and Ay, inherit both their definitions and their values.
Any subclasses that need to inherit Az inherit only Az's definition.

Class B and class C are subclasses of class A. Class B specifies in its class's definition that it needs to inherit Ax. The system looks to see whether Ax is a default-valued or normal instance variable. If it is a default-valued instance variable, the system adds class A, class B, Ax and 1 to its SUBCLASS_TABLE. Class C specifies in its class's definition that it needs to inherit Ay and Az. The system adds class A, class C, Ay and 1 to its SUBCLASS_TABLE. It also adds class A, class C, Az and -1 to the SUBCLASS_TABLE. At this point, the SUBCLASS_TABLE looks like figure 5.

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclass</th>
<th>Instance variable</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>Class B</td>
<td>Ax</td>
<td>1</td>
</tr>
<tr>
<td>Class A</td>
<td>Class C</td>
<td>Ay</td>
<td>1</td>
</tr>
<tr>
<td>Class A</td>
<td>Class C</td>
<td>Az</td>
<td>-1</td>
</tr>
</tbody>
</table>

Figure 5: SUBCLASS_TABLE

If a message is sent to class A to change the value of Ax (which is a default-valued instance variable), the system looks at the SUBCLASS_TABLE to see whether Ax is inherited by any subclasses. If this is the case, then the system automatically changes the value of Ax in the specified class and all its subclasses. However, if the object is designated as an exceptional object, then the value stored for Ax at the object level may not be the same value stored
for Ax at the class level (default value). For example, if the value stored for Ax at the class level is 2, but the value for Ax at the object level is 3, then the system notes that this is an exceptional object and does not propagate changes. This will also be true for any exceptional objects designated in the subclasses. Before the system changes default-valued instance variable, it looks to see whether the value of the object's instance variable is the same as the value in the class's definition. If the two values are the same, the system makes the change, else the system makes no changes.

The system also determines which superclass's methods are inherited by its subclasses. If a method refers to only inherited instance variables, the method is inherited by its subclass. For example, figure 4 shows that class A has three instance variables, Ax, Ay and Az. Class A has two methods, M1 and M2. M1 refers to Ax and Ay while M2 refers to only Az. Further, class B specifies class A as its superclass and Ax as its inherited instance variable. Since M1 refers to Ax and Ay, but Ay is not inherited by class B, class B cannot inherit M1 from class A. Since M2 refers to only Az, but Az is not inherited by class B, class B cannot inherit M2. Since class C inherits Ay and Az, class C inherits M2.

The prototype system provides a place in the class's definition which allows a class to specify its superclass
and inherited instance variables. A class can have multiple superclasses. All of its superclasses must be defined. An error message is displayed whenever 1) a superclass is not defined, 2) a name conflict occurs, or 3) an instance variable specified as an inherited instance variable in a subclass’s definition does not belong to the specified superclass. The system also allows the user to change the value of any default-valued instance variable. The system automatically propagates the new value throughout the database.

Bi-Directional Inheritance

The prototype system also supports bi-directional inheritance. The superclass can specify which instance variable it wishes to inherit from its subclass. When invoked, the subclass can send its value to the superclass. After all the classes are defined, the sub/super inherited instance variables are defined. During this interaction, the user must specify three things: the superclass name, subclass name, and inherited instance variables. An error message is displayed to the user, if the system detects a name mismatch.

The system also keeps a record of which superclass inherits which subclass’s instance variables in its SUBCLASS_TABLE. For example, class A is a superclass of class B. If class A specifies that it needs to inherit B’s
from class B, the system adds class A, class B, Bx, and 0 in the SUBCLASS_TABLE. The SUBCLASS_TABLE looks like figure 6.

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclass</th>
<th>Instance variable</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>Class B</td>
<td>Bx</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6: SUBCLASS_TABLE

Flag in the SUBCLASS_TABLE is used to indicate whether or not the inherited instance variable is a default-valued instance variable, and whether the inherited instance variable is inherited by a superclass from its subclass. If Flag = 1, the inherited instance variable is a default-valued instance variable. If Flag = -1, the inherited instance variable is not a default-valued instance variable. If Flag = 0, the inherited instance variable is inherited by a superclass.

The system also provides a special method that, when invoked, transmits a value from the subclass to a superclass. This special method checks whether the message’s sender is the subclass of the message’s receiver and whether the specified instance variable is the superclass’s inherited instance variable. If either of these conditions is not true, then an error message is displayed to the user.

When the user specifies that a superclass should inherit instance variables from its subclass, the user must
indicate which variables will be inherited upward. Therefore, the system supports bi-directional partial inheritance.

Implementation

The object-oriented database prototype was developed on an IBM PC/XT under IBM PC-DOS and uses Cdata, a relational database management system [STE87]. Cdata is written in C and provides the designer with two ways to specify the database's schema. The designer can use the C language to specify the database's schema or use Cdata Data Definition Language (DDL) which uses its own compiler. Cdata also provides the designer with a Data Manipulation Language (DML) and a set of utility functions and programs that support the Cdata database environment.

For reasons previously stated, the prototype database was written in Turbo C, version 2.0. The prototype database was designed for an interactive environment and, as such, provides an interface which allows the designer to create classes and objects, and to send messages. The designer creates classes by specifying a class name. The system then prompts the user for its superclass and inherited instance variables. The system continues to prompt the user until he specifies "end." If the class does not have a superclass, the user enters a dash ("-") to indicate that there is no superclass (see Appendix A). After each entry, the system adds this information to its SUBCLASS_TABLE.
The system continues by prompting the user for the class’s instance variables. The user must specify the variable’s name, type, length and default value, if any. The type attribute refers to the standard data types such as integer, real, and character (see Appendix B).

After all the classes are defined, the system asks the user to indicate which superclass will inherit which instance variables from its subclass. These inheritance properties must be entered one at a time (see Appendix C). The system adds this information to its SUBCLASS_TABLE (see Appendix D).

Finally, the system asks the user to specify the filename which contains the list of the classes’ methods. The methods themselves are written in the C language and are designed so that one method is equivalent to one function in C. The naming convention for method’s names is

<class_name>.<method_name>.

For example, "car.price" is a method called "price" which belongs to the class called "car." This convention is used for methods that its corresponding message will be sent to an object. If the message that corresponds to the method goes to a class rather than an object, then the method’s name is in the form of

<class_name>.class.<method_name>.

For example, "car.class.change" is a method called "change" which belongs to the class called "car." This
convention is used for methods that its corresponding message will be sent to a class.

Within each method, the designer sends a message to other objects in the form of function calls in C. Messages are specified in the form of

\[ \text{<receiver>..<message_name(parameter)>}. \]

For example, the message "a.price()" can be interpreted as a message that will be sent to the class or object called "a" to invoke method named "price" with no parameters being passed to "price" (see Appendix E).

The method's interpreter acts like a preprocessor of the C language. It translates all methods and messages to normal C language statements.

After all classes are created, the system prompts the user with a menu which allows him to either create objects or send messages. The user creates objects by entering the class's name. The system then displays the instance variable names along with the default value (see Appendix F). The user then enters the value for each instance variable. This process continues until all objects have been created. When an object is created, the system generates an unique object identifier for the object.

The user sends messages by specifying the receiver, name, and any arguments of the message (see Appendix G). The system provides the user with eight primitive methods (or messages): display_object, create_object, first_object,
next_object, remove_object, change_class_att, change_att_value, and send_value_up.

Display_object methods are used to retrieve a particular object. Create_object methods send messages to classes telling them to add an object to the class. The user has two ways to create an object: using the create_object method or creating objects via the menu. First_object methods send messages only to classes to retrieve the first object of the class. Next_object methods send messages to classes to retrieve the next object of the class. As the name implies, remove_object methods delete an object from its class.

The change_class_att methods are used to change the value of a default-valued instance variable. These methods act by changing the default value in every object in the class, except those objects that do not have the same value as the variable at the class level (exceptional objects). The change_att_value methods change a value of an instance variable of an object. Finally, the send_value_up message is sent by a class to its superclass in order to change values at the superclass level.

The designer can use any of these primitive methods within other user-defined methods. Further, the format for invoking the primitive methods is the same as invoking other user-defined methods.
The interface for the prototype database system uses an interactive style that consists of a combination of windows and menus. Different colors differentiate between top level and lower level commands. The only restriction imposed upon the user is that he must create classes before he creates any objects or sends messages.
CHAPTER BIBLIOGRAPHY


CHAPTER IV

EVALUATION

This chapter describes two database applications that were implemented using the prototype database management system. This chapter also compares the prototype database design to a more conventional object-oriented design and a relational database design.

First Database Application

The first database application deals with the problem of assigning people to space within a university. Simply stated, the database must assign classrooms to faculty members and staff throughout the university. A faculty or staff member can request a specific room within a specific building, or he can have the system automatically generate an assignment. The database consists of university, colleges, departments, faculty, capital_item, building, room, office, class_room and reservation entities. Figure 7 shows the entities and the relationships between the entities.

Container is an entity that contains other entities. It has a single property which is a List of the entities that are being contained.
Capital_item refers to any item within the University that is labelled as a capital expense. Capital_items involve the expenditure of money. Therefore, Cost is a property of Capital_item.

University has both Univ_name and Address. Both Buildings and Colleges belong to University. Therefore, there are relationships between University and Building, and between University and College. In the prototype system, every College must belong to a University.
Building is a subclass of both Container and Capital_item. Thus, it inherits a Cost property from Capital_item and a list of contents from Container. In this particular application, Building inherits a Univ_name property from University. Also, since most buildings have rooms, the system defines a relationship between Room and Building that represents where the rooms are located. Thus, Building has a Location property which can contain either a building name or building address.

Room is a subclass of Container and inherits a List of contents. Because rooms are located within buildings, Room inherits a Location property from Building. All rooms must be located in one building. All rooms have a property called Room_number assigned to them.

Office is a subclass of Room. Since Room inherits a List of contents from Container and a Location from Building, Office also inherits a List of contents from Container, and a Location and Room_number from Room. The relationship between Faculty and Office means that an office is assigned to a particular faculty or staff member.

Class_room is a subclass of Room and inherits Location and Room_number. Class_room also has a relationship with Reservation that shows that a particular room has been reserved as a classroom for a specific time.

College has two properties: College_name and Degree_offered. College inherits a Univ_name from
University. College also has a relationship with Department that shows which departments "belong to" or "are in" different colleges. In the prototype system, a department can belong to only one college.

Department has a Dept_name property. Faculty are usually assigned to different departments; thus, there is a relationship between Department and Faculty in the prototype system. A faculty member must be assigned to a department. The Department entity inherits Univ_name and College_name from College.

The Faculty entity has a Social_security_number, Name, Salary, and Position property. It inherits a Dept_name and Univ_name from Department. Its relationship with Reservation represents the assignment of a particular faculty member to a particular reservation. Because a faculty member occupies an Office, Faculty also inherits Room_number and Location from Office.

Reservation has Start_time, End_time, and Date properties. Reservation inherits a Name property from Faculty. It also inherits a Room_number and Location from Class_room. The instance of Reservation is created when someone makes a request to reserve either a specific classroom or any classroom. The request cannot conflict with the scheduling of any other room at a designated time.

Therefore, in order to design a schema for the object-oriented database system, the designer does not adhere to
traditional relational or hierarchical methods. This application has used for both full and partial inheritance. Building is a subclass of Capital_item and Container and, as such, inherits everything from both entities. Office also needs to inherit everything (full inheritance) from its superclass. On the other hand, Class_room requires only a few properties from its superclass such as Room_number and Location. While both Office and Class_room are considered subclasses of the same superclass, Office requires full inheritance from Room while Class_room requires only partial inheritance from Room. Not only are subclass/superclass relationships supported by this model, but other types of relationships also can be specified. For example, there is a relationship between Building and Room, but it is not a subclass/superclass relationship. The relationship between Building and Room can best be supported by what is called partial inheritance in which Room inherits only the Location property from Building. Thus, the prototype database system is able to support both full and partial inheritance. Further, the prototype system is able to support subclass/superclass relationship as well as other types of relationships.

The Reservation System Using the Prototype Database Management System

In figure 7, each entity is represented by a class. There are eleven classes in the database: Capital_item,
Whenever the system creates a class, it adds an object identifier property to the class. All the objects in the classes are identified by a unique object identifier. The designer can use the object identifier instead of the variable name whenever he wants to specify full inheritance in a relationship that is not an IS-A relationship. For example, the designer can use Class_room’s unique identifier number instead of the instance variable names Room_number and Location when he specifies that Reservation inherits Class_room’s instance variables.

For each class, the designer must specify the class’s own instance variables and all instance variables that are inherited from the superclass. If the designer wants a class to inherit all the instance variables from its superclass, then he specifies the keyword “all” after the prompt “inherited instance variables.” Thus, the database described in figure 7 would be defined as follows.

```
Capital_item class:
   Related with: none.
   Instance variables: Cost.
   Methods: none.

Container class:
   Related with: none.
   Instance variables: List.
   Methods: none.

University class:
   Related with: none.
   Instance variables: Univ_name and Address.
```
Methods: none.

College class:
Related with: University.
Inherited instance variables: Univ_name.
Instance variables: College_name and Degree_offered.
Methods: none.

Department class:
Related with: College.
Inherited instance variables: College_name, Univ_name.
Instance variables: Dept_name.
Methods: none.

Building class:
Related with: Capital.
Inherited instance variables: all.
: Container.
Inherited instance variables: all.
: University
Inherited instance variables: Univ_name
Instance variables: Location.
Methods: none.

Room class:
Related with: Container.
Inherited instance variables: all.
: Building.
Inherited instance variables: Location.
Instance variables: Room_number.
Methods: Get_room() - get Room_number of a particular object.
: Get_location() - get Location of a particular object
: Get_id(Room_number, Location) - get object identifier for a specified Room_number and Location.

Office class:
Related with: Room
Inherited instance variables: all.
Instance variables: none.
Methods: none.

Class_room class:
Related with: Room.
Inherited instance variables: Room_number.
: Location.
Instance variables: none.
Methods: none.
Faculty class:
Related with: Office.
  Inherited instance variables: object_id.
    : Department.
  Inherited instance variables: Dept_name,
    : Univ_name.
Instance variables: SSN,
    : Name,
    : Salary and
    : Position.
Methods: none.

Reservation class:
Related with: Faculty.
  Inherited instance variables: Name.
    : Class_room.
  Inherited instance variable: object_id.
Instance variables: Start_time,
    : End_time, and
    : Date.
Methods: Reserve (Name, Start_time, End_time,
    Date, Room_num, Location).

List contains a list of object identifiers of other objects that Container contains. The Reserve method attached to the Reservation class assigns a particular room to a particular faculty name. The Reserve method also checks whether the Start_time is less than End_time. If the user does not specify a Room_number or a Location, then the system generates its own solution by assigning the individual to any room which is unoccupied at the particular time. The user can also indicate a specific building without identifying a room number, and the system will find an available room.

Within the Room class, the Get_room () method refers to only Room_number. The Get_location () method refers to only Location. Finally, Get_id (Room_num, Location) method
refers to only Room_number and Location. Class_room and Office inherit all methods defined for Room, because Office has full inheritance from Room and Class_room inherits both Room_number and Location from Room.

A designer can modify the database by invoking one of the special change methods provided by the system. For example, if the user specified NTSU as the value of Univ_name, then the College, Department and Faculty classes inherited both the definition and the value of Univ_name instance variable from the University class. If the user changes the value of University's Univ_name to UNT by invoking the change_class_att method, the change_class_att method changes the value of Univ_name in University class and all other classes that inherit Univ_name.

In the initial design, Class_room inherited only definitions of Room_number and Location from Room. If the user changes the value of the instance variables for the Room class, the changes do not effect the values of Class_room's Room_number and Location.

The Reservation System Using a Conventional Object-Oriented System

Conventional object-oriented systems [HUD87, PEN87, MAI85, MAI86, MAN86, BAN87, ZDO85, SMI87] support only full inheritance, which means that the user can define only subclass/superclass relationships. The relationship between subclass and superclass is usually interpreted as an IS-A
relationship. As a result, there are a number of revisions that must be made to the reservation system in order to accommodate this more restrictive definition. These changes include the following:

1. The inheritance property between Class_room and Room cannot be limited to partial inheritance. Class_room must inherit all Room’s instance variables, including the list of contents inherited from the Container class.

2. Because the system does not support partial inheritance, the Room class cannot inherit a Location property from the Building class. Room would have to define its own Location instance variable. The designer would then have to duplicate all the methods to insure that the meaning and constraints of both Building’s Location and Room’s Location are the same. Similarly, both Building and College must maintain duplicate entries for the Univ_name instance variable. The Department class needs to add Univ_name and College_name to its instance variables. Finally, the Faculty class needs to define Dept_name and Univ_name as instance variables. Any methods that refer to any of these instance variables would need to be duplicated in each of the classes. For example, if there is a method that manipulates the Location instance variable stored in Building, this method must be redefined for Room.
3. The Reservation and Faculty classes cannot inherit an object identifier from Class_room and Office. The user must either add Room_number and Location instance variables to Reservation and Faculty classes, or add a single instance variable for the object identifier to both classes. If the user selects the first option, then he is also responsible for maintaining the integrity of the database due to the problem of redundancy.

If the system cannot support partial inheritance, then the user becomes responsible for maintaining all the relationships in the database that are not subclass/superclass relationships. He must also maintain the meaning of those relationships and the translation of those meanings to the various parts of the system. The designer must design a subsystem that automatically maintains the set of duplicate entries for the various classes. This subsystem would be responsible for maintaining the integrity of the database. It is obvious that such a situation is contrary to the intent of an object-oriented database and hinders, rather than facilitates, database design.

The Reservation Database Using a Relational Database Management System

In a relational database, each class is equivalent to one or more relations. However, one or more of the attributes of the relation must be assigned as a key. All
other attributes can be uniquely identified by the key. In order to conform to these criteria, the Reservation system must be defined as follows.

University has Univ_name and Address as attributes. Every university has an unique Univ_name. The University relation can be defined as follow:

University (Univ_name, Address)

College has College_name and Degree_offered as its attributes. One university has several colleges. A college of a university offers several degrees such as BA, BS, etc. A college of different universities might offer different degrees. The College relation can be defined as follow:

College (College_name, Univ_name, Degree_offered)

Department has Dept_name as its attribute. A department can be in only one college of a university. However, two different universities might has a department assigned to two different colleges. The Department relation is defined as follow:

Department (Dept_name, College_name, Univ_name)

Every Faculty member has a Social_security_number (SSN), Name, Salary, and Position as attributes. A faculty can belong to only one department. A department can have several faculties. The Faculty relation can be defined as follow:

Faculty (SSN, Name, Salary, Position, Dept_name, Univ_name)
Every building has a Cost, Location, and List of its contents as attributes. Content is another entity and has its own attributes. It is assumed that a building contains classrooms and offices. Each building has a unique location. A building can only be owned by a university.

Room has a Room_number as its attribute. Two rooms in different buildings can have the same room number.

Office has a Room_number and a List of contents. A faculty member is contained within an office. An office can be occupied by several faculty members. Building, Office, and Class_room relations can be defined as:

Building (Location, Cost, Univ_name)
Room (Location, Room_number)
Office (SSN, Location, Room_number)

The values of the Location and Room_number attributes are a subset of the Room relation.

Reservation has a Start_time, an End_time, and a Date associated with it. A faculty member can make several reservations for the same classroom at different times or for different classrooms at the same time. A classroom can be reserved by one or more faculty members. A classroom cannot be reserved at the same time on the same day by two or more people. The Reservation relation can be defined as:

Reservation (Location, Room_number, Start_time, Date, End_time, Name)
The values of the Location and Room_number attributes are a difference of those values of Room relation and Office relation.

The schema of reservation database is:

University (Univ_name, Address)

College (College_name, Univ_name, Degree_offered)

Department (Dept_name, College_name, Univ_name)

Faculty (SSN, Name, Salary, Position, Dept_name, Univ_name)

Building (Location, Cost, Univ_name)

Room (Location, Room_number)

Office (SSN, Location, Room_number)

Reservation (Location, Room_number, Start_time, Date, End_time, Name)

The methods designed for the object-oriented database system would be implemented by using a database manipulation language or a general purpose language.

In relational database example, the user is responsible for maintaining the values of Location and Room_number attributes in the Reservation relation. These values for Location and Room_number are obtained via set difference of Room relation and Office relation. The object-oriented database system is responsible for propagating the new value of a default-valued instance variable throughout the database if there is a change to it. However, in a relational database, the user is responsible for maintaining
any changes made to a default-valued instance variable. Thus, the user has more responsibilities in a relational database environment than in an object-oriented database environment.

The Orbital Mechanics Tutor Database System

The second database application defines tutoring knowledge for the subject of Orbital Mechanics. The purpose of this particular system is to teach students about the relationship between orbital elements (numbers) and the shape of a ground track (words) [SWI87]. Students are given an elaborate computer display that allows them to manipulate various orbital elements and generate ground tracks. Everytime the student manipulates the numbers for the orbital element, they are suppose to predict, in words, the shape of the ground track. Students are free to explore the orbital mechanics world at their own pace. It is only after several incorrect predictions, that the system intervenes and offers assistance. Therefore, the database for this particular application contains both facts and rules about the relationships between orbital elements and shape descriptors. The system also contains the rules for deriving the shape descriptors, counters for correct and incorrect answers, and lists of probable error messages. Figure 8 contains the structure of this system.
It should be noted that the student receives an error message only after repeated, failed attempts. Thus, the system retreats to the background and accumulates evidence before it offers its "best" advice. In order to accommodate this strategy, the database was designed in the following manner. The lowest level entities store facts, rules, counters for correct answers, counters for incorrect
answers, and the error messages associated with each incorrect answer. As the student enters incorrect predictions, the system increments the incorrect counter for a particular shape descriptor. However, other incorrect counters are also being incremented at the same time. Whenever an incorrect counter reaches a certain threshold, the system sends an error message associated with the shape descriptor to the next level entity. Again, the system determines whether there is sufficient evidence to display a particular error message. The error message that wins the competition is sent to the next level, and the cycle continues. Finally, a single error message is passed to the final entity which displays it to the student. The top level entity prints the error message with the highest incorrect counter value.

The Orbital Mechanics tutor requires upward inheritance. The top entities, shown in figure 8, inherit error messages and counters that are stored in the entities at the lower levels.

The Database Design for the Orbital Mechanics Tutor Using the Prototype System

Each entity in the Orbital Mechanics tutor is defined as a class. The lowest level classes store facts, an incorrect_counter, a correct_counter and error_message instance variables. The parent classes (upper level entities) inherit error_messages and incorrect_counters from
their children (lower level entities). The lower class entities also have methods which are responsible for incrementing the incorrect_counter and correct_counter values. When the incorrect_counter reaches a certain threshold, the error_message associated with the counter is sent to the parent class using a method called send_value_up. The class at the top level has a method that determines which error message should be printed.

Example definitions of some of the classes are as follows:

Polar class:
   Superclass : Inclination.
   Inherited instance variables : none.
   Instance variables : Facts,  
   : Incorrect_counter, 
   : Correct_counter, and
   : Error_message.
   Methods : Check (Conditions) - increments Correct_counter if Conditions match Facts. If this is not the case, Incorrect_counter is incremented. Compare Incorrect_counter value to a threshold. If it is more than the threshold, then the method invokes send_value_up method to send Error_message and Incorrect_counter to its superclass.

Inclination class:
   Superclasses : OrbitExpert.
   Inherited instance variables : none.
   Instance variables : none.
   Methods : Send () - invokes send_value_up to send Error_messages and Incorrect_counters that has the Highest value of Incorrect_counter
   Inherited upward : Error_message and : Incorrect_counter.
Expert class:
Superclass: none.
Instance variables: none.
Methods: Print() - prints the value of
Error_message that has the highest
value of Incorrect_counter.
Inherited upward: Error_message and
: Incorrect_counter.

Eccentricity inherits three pairs of Error_messages and Incorrect_counters. Inclination inherits five pairs, etc.
The top level entity, Expert, inherits twenty six pairs of Error_messages and Incorrect_counters from its subclasses.

The Orbital Mechanics Tutor Database Using a Conventional Object-Oriented Database Management System

Conventional object-oriented databases do not support upward inheritance. Therefore, if the Orbital Mechanics tutor were to be implemented using a conventional object-oriented database, then the database designer must make the following modifications. First, all upper level classes need to duplicate the counter and error_message instance variables that are currently defined at the lowest class level. At the higher level classes, the designer would also have to define a special method that would pass up the values for error messages, counters, and code. The instance variable labelled code would be used to store a pointer from the shape descriptor to the higher level classes so that the error messages could be associate with the appropriate shape descriptors. In summary, the database designer would be responsible for writing a system that would maintain the
relationships between the shape descriptors and their appropriate error_messages. A second alternative would be to collapse the data structure into a single layer that would allow messages to pass between all the entities. Of course, this situation would make the data model rather oblique and difficult to maintain.

The bi-directional partial inheritance property allows the system to reuse information that has previously been defined. Because of this, the database designer has more flexibility and can create data models that reflect the real world.
CHAPTER BIBLIOGRAPHY


CHAPTER V

SUMMARY AND FUTURE RESEARCH

This chapter summarizes the findings of this dissertation. More specifically, it lists the advantages and disadvantages of bi-directional partial inheritance. It also includes a discussion of future research.

Bi-directional partial inheritance offers several advantages that can be listed as follows.

1. Bi-directional partial inheritance provides the designer with greater flexibility than full inheritance. The bi-directional partial inheritance allows the designer to describe any type of relationship that might occur in an object-oriented database. Given that the designer may want to show different types of relationships, then he should be given a database management system that allows him to describe his data in a natural way. An object-oriented database that features bi-directional partial inheritance gives the designer the flexibility of describing different relationships without the problem of introducing redundancy. As illustrated in the sample databases described in chapter 4, the database designer does not have to redefine the instance variables and methods everytime he introduces different types of relationships. Bi-directional partial
inheritance reduces the amount of redundant information in the database.

2. Bi-directional partial inheritance allows the designer to represent a number of different types of applications in a natural way. The Orbital Mechanics tutor is an example of an application in which the superclass needs information from its subclass (a COME-FROM feature). A previous implementation of the Orbital Mechanics tutor required a design that appears both awkward and unnatural. The entire tree was duplicated in a reverse order.

3. Partial inheritance makes the database easier to maintain. If the system must keep two copies of the same information, then the user will be forced to maintain the meaning of these two copies. If a variable gets changed in one of the copies, then the user is responsible for changing the value of the second copy. As a result, default-valued instance variable will need special attention. If one of the copies contains a default-valued instance variables, then there may be some question as to whether the second copy gets changed.

4. Partial inheritance allows for contradiction between the subclass/superclass. For example, both "birds fly", and "penguins and ostriches are birds" are true. Yet, "penguins and ostriches are flightless" [BOR88]. Partial inheritance allows a subclass to inherit a proper subset of its superclass' instance variables. Moreover, two
subclasses of the same superclass do not need to inherit the same set of instance variables. Therefore, the class Bird can have Fly as one of its properties. The Penguins and Ostriches classes do not have to inherit the Fly property. All other subclasses of Bird can have full inheritance.

5. More memory can be saved in a system that supports partial inheritance because fewer instance variables are inherited by the subclass, and thus, fewer instance variables are copied into the subclass's memory location.

In short, this approach offers flexibility, ease to maintain, and memory saving. Furthermore, it allows the designer to represent a number of different types of applications, and allows for contradiction between the class and subclass.

One of the disadvantages of the bi-directional partial inheritance approach is that it introduces a new type of complexity. Furthermore, this approach does not offer the advantages found in other conventional, full inheritance object-oriented database system where there is an obvious and transparent representation. The IS-A relationship provides a very consistent view of the database. Everyone who looks at the design is clear about what the relationships mean. However, once the designer introduces different types of relationships, the database no longer has a consistent view. Maintenance also become more difficult
as the complexity increases. In short, flexibility exists at the cost of complexity.

Future Research

This dissertation has introduced a number of areas of research that can extend the work on bi-directional and partial inheritance. For example, the problem of representing both composite objects and incomplete inheritance should be investigated. Composite objects are objects that have other objects as subparts. For example, a car object may have a door object as a PART-OF it. If a car does not exist, the doors that belong to that car cannot exist. The prototype database system could be extended in order to represent composite objects by allowing the class to inherit the subclass's object identifier. The system would also have to include a method that would verify that the object that is PART-OF another object does not exist without the main object. The system will also have to provide a delete method that will delete an object and any objects that are part of the main object. Finally, the prototype system might be extended to include incomplete inheritance by either allowing the user to specify the method the subclass needs to inherit or allowing the system to determine inheritance based on the inherited instance variables.
This dissertation shows that bi-directional and partial inheritance is a feature that should be included in object-oriented database. It also shows that bi-directional and partial inheritance has many advantages over conventional object-oriented approaches. Future research should try to enhance the features suggested here and focus on implementation and design constraints.
CHAPTER BIBLIOGRAPHY

APPENDIX A

CLASS DEFINITION WINDOWS
Specifying class' name.

Creating Classes

class : university

No superclass.

Creating Classes

class

Related Class

Relating with Class : -
Specifying superclass and inherited instance variable.

Creating Classes

class Related Class

Relating with Class: university
inherited attributes: univ_name

Specifying superclass and keyword "all."

Creating Classes

class Related Class

Relating with Class: university
inherited attributes: all
Specifying superclass and keyword "none."

Creating Classes

<table>
<thead>
<tr>
<th>class</th>
<th>Related Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relating with Class : university
inherited attributes : none

Specifying keyword "end" to end this section.

Creating Classes

<table>
<thead>
<tr>
<th>class</th>
<th>Related Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relating with Class : end
inherited attributes :
APPENDIX B

CLASS ATTRIBUTE WINDOWS
Specifying default-valued instance variables.

Creating Classes

- Its own attributes -

<table>
<thead>
<tr>
<th>attribute :</th>
<th>univ_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>type :</td>
<td>char</td>
</tr>
<tr>
<td>length :</td>
<td>20</td>
</tr>
<tr>
<td>value :</td>
<td>UNT</td>
</tr>
</tbody>
</table>

Specifying normal instance variable.

Creating Classes

- Its own attributes -

<table>
<thead>
<tr>
<th>attribute :</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>type :</td>
<td>char</td>
</tr>
<tr>
<td>length :</td>
<td>20</td>
</tr>
<tr>
<td>value :</td>
<td>-</td>
</tr>
</tbody>
</table>
Specifying upward inheritance.

---

**Creating Classes**

---

Inherit from its subclasses

Superclass wants to inherit some attributes from its subclasses

**Superclass name:** expert
**Subclass name:** polar
**Inherited attributes:** error

---

Ending this section.

---

**Creating Classes**

---

Inherit from its subclasses

Superclass wants to inherit some attributes from its subclasses

**Superclass name:** end
**Subclass name:**
**Inherited attributes:**
APPENDIX D

SUBCLASS TABLES
First database application, subclass table

UNIVERSITY COLLEGE UNIV_NAME 1
COLLEGE DEPARTMENT COL_NAME -1
COLLEGE DEPARTMENT UNIV_NAME 1
UNIVERSITY BUILDING UNIV_NAME 1
CAPITAL_ITEM BUILDING COST -1
CONTAINER BUILDING LIST -1
BUILDING ROOM LOCATION -1
CONTAINER ROOM LIST -1
ROOM OFFICE LOCATION -1
ROOM OFFICE LIST -1
ROOM OFFICE ROOM_NUM -1
ROOM CLASS_ROOM ROOM_NUM -1
ROOM CLASS_ROOM LOCATION -1
OFFICE FACULTY OFFICE_KEY -1
DEPARTMENT FACULTY DEPT_NAME -1
DEPARTMENT FACULTY UNIV_NAME 1
FACULTY RESERVATION NAME -1
CLASS_ROOM RESERVATION CLASS_ROOM_KEY -1
Second database application, subclass table

EXPERT POLAR ERROR1 0
EXPERT NODES ERROR2 0
EXPERT PEAKNORTH ERROR3 0
APPENDIX E

CLASS METHODS
Specifying filename contains class methods

Creating Classes

filename contains methods: c_meths.c

Processing....
First database application, class methods

```c
int class_room.get_room()
{
    /* get room number of a particular object */
    static int room;
    room = room_num;
    return (room);
}

char *class_room.get_loc()
{
    /* get a location of a particular object */
    static char loc[26];
    strcpy(loc,location);
    return(loc);
}

int class_room.class.get_id(int room_in, char *loc_in)
{
    /* get object identifier for a particular room */

    static int objid;
    int room;
    char loc[25];

    objid = first_object("class_room");
    room = objid.get_room();
    strcpy(loc,objid.get_loc());
    if ((room_in == room) && (strcmp(loc,loc_in) == 0))
        return(objid);
    else
    {
        objid = next_object("class_room");
        while (objid != -1)
        {
            room = objid.get_room();
            strcpy(loc,objid.get_loc());
            if ((room_in == room) &&
                (strcmp(loc,loc_in) == 0))
                return(objid);
            else
                objid = next_object("class_room");
        }
    }
}
```
return (0);
}

int class_room.class.find (char *loc, int room)
{ /* determine whether or not a particular room exist */
    int objid;
    int room_in;
    char loc_in[25];

    objid = first_object("class_room");
    room_in = objid.get_room();
    strcpy(loc_in,objid.get_loc());
    if ((room == room_in) && (strcmp(loc,loc_in) == 0))
        return (1);
    else {
        objid = next_object("class_room");
        while (objid != -1)
        {
            room_in = objid.get_room();
            strcpy(loc_in,objid.get_loc());
            if ((room == room_in) && (strcmp(loc,loc_in) == 0))
                return (1);
            else
                objid = next_object("class_room");
        }
        return(0);
    }
}

int reservation.set(int  sttime, int endtime, char *re_date, int id)
{ /* Check whether a reservation can be made to a particular room */
    char in_date[10];
    static Int ret;

    ret = -1;
    strcpy(in_date,re_date);
    move(in_date,8);
    if (strcmp(in_date,date) == 0)
    { if (id == class_room_key)
        { if (sttime > end_time)
ret = 0; 
else 
    if (endtime < start_time) 
        ret = 0;
    
else 
    ret = 0;
}
else 
ret =0;
return (ret);
}

void reservation.class.reserve (char *name, int sttime, 
    int endtime,char *re_date, 
    int room, char *loca)
{
    /* this method tries to make a reservation to a particular 
    room or to any room */

    char par[80];
    int objid;
    int r_obj;
    char str[80];
    int done;
    int finish;
    int tempf;
    char temp2[25];
    char temp[25];
    char class[25];
    char loc[25];

    strcpy(loc,loca);
    move(loc,20);
    strcpy(str,name);
    move (str,20);
    strcpy(par,str);
    strcat(par,"\"");
    itoa(sttime,str,10);
    int_value(str,5);
    strcat(par,str);
    strcat(par,"\","");
    itoa(endtime,str,10);
    int_value(str,5);
    strcat(par,str);
    strcat(par,"\","");
    strcpy(str,re_date);
    move(str,8);
    strcat(par,str);
    strcat(par,"\","});
/* User specifies either room number or location but not both */
if ((room == 0) || (*loc == '0'))
{
    objid = first_object("class_room");
    finish = 0;
    while ((objid != -1) && (finish == 0))
    {
        done = 1;
    }
/* User specifies room number */
if (room != 0)
{
    while ((done == 1) && (objid != -1))
    {
        tempf = objid.get_room();
        if (tempf == room)
            done = 0;
        else
            objid = next_object("class_room");
    }
}
else
{
/* User specifies location */
if (*loc != '0')
{
    while ((done == 1) && (objid != -1))
    {
        strcpy(temp2, objid.get_loc());
        if (strcmp(temp2, loc) == 0)
            done = 0;
        else
            objid = next_object("class_room");
    }
}
if (objid != -1)
{
    r_obj = first_object("reservation");
    done = 1;
    while ((done == 1) && (r_obj != -1))
    {
        if (r_obj.set(sttime, endtime, re_date, objid) == 0)
            r_obj = next_object("reservation");
        else
            done = 0;
    }
if (r_obj == -1)
{
    itoa(objid,str,10);
    int_value(str,6);
    strcat(par,str);
    create_object("reservation",
                  "name,start_time,end_time,date,
                  class_room_key", par);

    finish = 1;
}
else
    objid = next_object("class_room");
}
}
go toxy (2,7);
if (objid == -1)
cprintf("Cannot make a reservation");
else
    cprintf("Reserved");
else
{
    /* User specify both room number and location */

    gotoxy (2,7);
    strcpy(class,"class_room");
    if (class.find(loc,room) == 1)
    {
        objid = class.get_id(room,loc);
        r_obj = first_object("reservation");
        done = 1;
        while ((done == 1) && (r_obj != -1))
        {
            if (r_obj.set(sttime,endtime,re_date,objid) == 0)
                r_obj = next_object("reservation");
            else
                done = 0;
        }
        if (r_obj == -1)
        {
            itoa (objid,str,10);
            int_value(str,6);
            strcat(par,str);
            create_object("reservation",
                          "name,start_time,end_time,date,
                          class_room_key", par);
            cprintf("Reserved");
        }
        else
cprintf("Cannot make a reservation");
}
else
  cprintf("This room number %d at %s does not exist", room, loc);
Second database application, class methods

```c
void polar.p_check1(char *cond)
{
    char str[80];
    char message[80];
    int objid2;
    int counter;

    if (strcmp(cond, fact1) == 0)
        cor1_counter ++;
    else
    {
        ++incor1_counter;
        if (incor1_counter >= 1)
        {
            counter = incor1_counter;
            itoa(counter, str, 10);
            strcpy(message, error1);
            strcpy((message+18), str);
            objid2 = first_object("expert");
            send_value_up(objid, objid2, "error1", message);
        }
    }
}

void polar.class.p_check(char *condition)
{
    int objid;
    char cond[25];

    objid = first_object("polar");
    strcpy(cond, condition);
    move(cond, 15);
    objid.p_check1(cond);
}

void nodes.n_check2(char *cond)
{
    char str[80];
    char message[80];
    int objid2;
    int counter;

    if (strcmp(cond, fact2) == 0)
        cor2_counter ++;
    else
    {
        ++incor2_counter;
        if (incor2_counter >= 1)
        {
```
counter = incor2_counter;
itoa(counter,str,10);
strcpy(message,error2);
strcpy((message+18),str);
objid2 = first_object("expert");
send_value_up(objid,objid2,"error2",message);
}

void nodes.class.n_check(char *condition)
{
    int objid;
    char cond[25];

    objid = first_object("nodes");
    strcpy(cond,condition);
    move(cond,15);
    objid.n_check2(cond);
}

void peaknorth.pe_check3(char *cond)
{
    char str[80];
    char message[80];
    int objid2;
    int counter;

    if (strcmp(cond,fact3) == 0)
    
        cor3_counter ++;
    else
    {
        ++incor3_counter;
        if (incor3_counter >= 1)
        {
            counter = incor3_counter;
            itoa(counter,str,10);
            strcpy(message,error3);
            strcpy((message+18),str);
            objid2 = first_object("expert");
            send_value_up(objid,objid2,"error3",message);
        }
    }
}

void peaknorth.class.pe_check(char *condition)
{
    int objid;
    char cond[25];

    objid = first_object("peaknorth");
    strcpy(cond,condition);
move(cond,15);
objc.pe_check3(cond);
}

void expert.ex_prt()
{
    char mess[30];
    int c,e1;
    char count[10];

    strcpy(count,(error1+18));
    c = atoi(count);
    strncpy(mess,error1,15);
    strcpy(count,(error2+18));
    e1 = atoi(count);
    if (c < e1)
    {
        c = e1;
        strncpy(mess,error2,15);
    }
    else
    {
        strcpy(count,(error3+18));
        e1 = atoi(count);
        if (c < e1)
        {
            c = e1;
            strncpy(mess,error3,15);
        }
    }
goxy(2,7);
cprintf("Error message = %15s",mess);
}

void expert.class.ex_print()
{
    int objid;

    objid = first_object("expert");
    objid.ex_prt();
}
APPENDIX F

CREATE OBJECT WINDOW
Create object template.

--- university ---

UNIV NAME
UNT
ADDRESS
APPENDIX G

MESSAGE WINDOWS
Message window template.

```
<table>
<thead>
<tr>
<th>Sending message</th>
</tr>
</thead>
<tbody>
<tr>
<td>receiver :</td>
</tr>
<tr>
<td>(object id or class name)</td>
</tr>
<tr>
<td>message name :</td>
</tr>
<tr>
<td>argument :</td>
</tr>
</tbody>
</table>
```

Specifying receiver, message's name, and argument.

```
<table>
<thead>
<tr>
<th>Sending message</th>
</tr>
</thead>
<tbody>
<tr>
<td>receiver :</td>
</tr>
<tr>
<td>university</td>
</tr>
<tr>
<td>(object id or class name)</td>
</tr>
<tr>
<td>message name :</td>
</tr>
<tr>
<td>in_first_object</td>
</tr>
<tr>
<td>argument :</td>
</tr>
<tr>
<td>none</td>
</tr>
</tbody>
</table>
```

Specifying "end" to end this section.

```
<table>
<thead>
<tr>
<th>Sending message</th>
</tr>
</thead>
<tbody>
<tr>
<td>receiver :</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>(object id or class name)</td>
</tr>
<tr>
<td>message name :</td>
</tr>
<tr>
<td>argument :</td>
</tr>
</tbody>
</table>
```
BIBLIOGRAPHY

Books


Articles


Kim, W., et al., "Composite Object Support in an Object-Oriented Database System," *Proceedings of ACM*


