THE OBJECT-ORIENTED DATABASE EDITOR

THESIS

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

MASTER OF SCIENCE

By

Sidney M. Coats, B.S.
Denton, Texas
December, 1989
Because of an interest in object-oriented database systems, designers have created systems to store and manipulate specific sets of abstract data types that belong to the real world environment they represent. Unfortunately, the advantage of these systems is also a disadvantage since no single object-oriented database system can be used for all applications.

This paper describes an object-oriented database management system called the Object-oriented Database Editor (ODE) which overcomes this disadvantage by allowing designers to create and execute an object-oriented database that represents any type of environment and then to store it and simulate that environment. As conditions within the environment change, the designer can use ODE to alter that environment without loss of data. ODE provides a flexible environment for the user; it is efficient; and it can run on a personal computer.
TABLE OF CONTENTS

LIST OF ILLUSTRATIONS ........................................ iv

Introduction ....................................................... 1
Existing Database Systems
Object-oriented Database Systems
Definitions
Related Work

A Description of ODE .............................................. 9
Comparison to Existing Object-oriented
Database Systems
ODE Work Space
Sample Database System Using ODE

Implementation .................................................... 16
Data Storage
User Interface
Efficiency Considerations

Current Status .................................................... 25

Conclusion ......................................................... 26

APPENDIX .......................................................... 28

REFERENCES ...................................................... 33


LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Properties of a Database</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>Class Structure</td>
<td>14</td>
</tr>
<tr>
<td>3.</td>
<td>ODE's Logical Storage Structure</td>
<td>20</td>
</tr>
</tbody>
</table>
Introduction

Existing Database Systems

Historically, research on database systems has tended to emphasize problems related to large volumes of homogeneous data typically found in business applications. As a result, most of the commercially available database management systems rely on data models (network, hierarchical, and relational) that support simple entity structures; these types are most useful for business applications [DITT86]. However, in recent years, a number of new applications have emerged that require database systems that can support very large, complex entities. For example, computer aided design (CAD) applications require database systems that can manipulate large amounts of information concerning design specifications for different types of physical devices. Similarly, graphical interface design tools need to store and manipulate complex icons, pictures, and text. Both of these applications require systems that can represent, store, and manipulate large, complex physical or structural entities as single entities. Unfortunately, traditional database management systems do not allow this type of manipulation. Whenever a database designer wants to represent a complex entity and is using one of the traditional database management systems, he must separate the entities into several sub entities and then
store each individual sub entity as a single record or tuple. To recall or manipulate the complex entity, the designer must submit several database transactions in order to retrieve each of the sub entities. In addition to being cumbersome to use, difficult to manipulate, and inefficient, most traditional database management systems' data models do not necessarily reflect the logical structure of the data. As objects become more and more complex, the logical structure of the data becomes more divorced from the physical structure imposed by the database management system [DITT86].

Object-oriented Database Systems

An obvious solution to the above problem is to provide a database management system that allows users to describe data both as a single entity and as individual parts. Such a system might allow users to define abstract data types, along with a set of special operators that manipulate the abstract items. For example, the graphical user interface application might have a specific data type called icon and a set of operators that manipulate icons such as stretch, shrink, and shape. In turn, each of these operators could be implemented as a function that performed the actual database manipulations on the icons' sub entities. The ability to describe and manipulate complex entities as a single entity is also known as object-oriented programming.
By combining the concept of object-oriented programming with the idea of a database management system, database designers have created the concept of object-oriented database management systems [PETE87]. These database systems are capable of storing and manipulating complex entities as single objects within a database framework. The database management system automatically processes the sub entities, thus freeing the application programmer from a tedious task [MCCA87].

Definitions

The concept of an object and a class was first introduced to computer science through the programming language called Simula [DAHL66]. Following the emergence of object-oriented programming languages [STEF86, GOLD83], several proposals were made for integrating the object-oriented paradigm within the database environment. Examples of such systems include those of Maier, Stonebraker, and Zdonik [MAIE86, STON86, ZDON85]. Thus, like their programming language counterpart, most object-oriented database management systems contain the concepts of class, methods, and inheritance. A detailed discussion of each of these concepts follows.

All data within the database must belong to one or more classes. Every class has a name and a list of variables called properties. The property name and the
values for the property are defined within the class [KIM87]. For example, the graphical interface database might contain a class called Icon. It, in turn, would have Location, Rotation, Size, and Components properties. The Location property might also belong to the Coordinate class which, in turn, would contain values for the X, Y, and Z properties. Similarly, the Size property might belong to the Volume class and contain property values for Length, Width, and Height; see Figure 1. Every database entity must be assigned to at least one class. Whenever the user assigns an entity to a class, he creates an instance or object of that class. Thus, the graphical interface database might have a number of different instances of Icons such as Sphere, Cube, and Square.

* Basic object type built into ODE.

Figure 1. Properties of a Database
The database system manipulates the data by invoking various operators, called methods, which are associated with each class. These methods perform the usual database functions such as accessing, deleting, and modifying data [PETE87]. Similar to the programming environment, methods are defined for each class. Each class has its own set of methods that are valid only for that class. If the designer does not create a specific method for a class, then that operation cannot be performed on any member of the class [COX86]. For example, the graphical interface database Read-rotation and Read-components both retrieve specific information about a particular icon. The Icon class would also need a series of methods that would assign the corresponding information to each icon.

One of the ways in which different objects share information is through a mechanism called inheritance. Inheritance allows an object in one class to obtain information from another class [STEI87]. For example, the graphical interface database might have an object in the Line class that wants to inherit the Location value stored in the Icon class. In order to inherit this value, the classes must be related to each other in some manner. (Although inheritance from a hierarchy is not reciprocal, the designer may create user-defined reciprocity.) Thus, the Line class must be related to the Icon class. Relationships may be defined between objects and classes,
among objects of the same class, or among objects of different classes. These relationships may take the form of one to one, one to many, many to one, or many to many, types of relationships [ULLM89].

Related Work

Initially, much of the work in the area of object-oriented database dealt with specific applications. One of the early object-oriented database management systems was PROBE, a system developed by the Computer Corporation of America. Designed as a knowledge-oriented database management system, PROBE's primary purpose was to maintain and execute mathematical functions on stored data. PROBE uses a data model (PDM) that consists of two basic types of data objects: entities and functions. Entities are grouped into one or more collections of data called entity types which serve as classes in PDM. The entity type defines the functions which may be applied to different entities. Properties of entities, relationships between entities, and operations on entities are represented in PDM by functions. PDM defines a function as a relationship between collections of entities and scalar values [MANO86].

Another early system was Telesophy (TSL), which was designed by Bell Communications Research for the purpose of providing access to all of Bell's on-line information. As
such, TSL defines only one class called Information Unit (IU). Each IU object contains a unique identifier number, descriptive information, and a list of related IUs [CAPL87].

It soon became apparent that there was not a single set of data abstractions that could be used to represent all object-oriented database applications. A different object-oriented database system was required for each new application. Rather than produce a separate object-oriented database management system for each new application, researchers began to look at the possibility of producing tools for generating systems for the different applications. Such tools would allow a user to describe different data types and the operators on those data types [CARE86].

In 1986, The Computer Science and Engineering Department at the University of Connecticut developed the Data Model Compiler (DMC) [MARY86], which was designed to generate an object-oriented database management system. Given a description of objects, the relationships among the various objects, and the methods that operate on the objects, this tool automatically produces an object-oriented database management system for the described database [MARY86]. The DMC project focuses on the problem of defining different types of relationships in an object-oriented database and, as such, argues that it is the
relationships among objects that really distinguish object-oriented databases from other types of database systems [MARY86].

The Computer Science Department at the University of Wisconsin developed a second type of object-oriented management tool [CARE86]. EXODUS is an extensible database system that facilitates the development of application specific object-oriented database systems by providing users with the ability to define data types, called objects, within the system. These objects are stored as record structures of basic object types. Access to these objects is provided by user defined methods that are written in the EXODUS database programming language, which is an extension of the C programming language [CARE86].

Although very useful for describing object-oriented database systems, neither DMC nor EXODUS provides all the elements that are required for an effective database generator. EXODUS, for example, does not automatically create and maintain relationships between the objects. The user is forced to create and maintain his own set of relationships and inheritance structures. Although DMC maintains its own relationships, it does not allow the user to change the environment. Because it produces a separate executable system for each described environment, the user must re-generate the entire system whenever he makes a single change to the database. Further, all of the
data in the old system must be transferred or copied to the new system.

Because of these and other problems, this research undertook to develop an object-oriented database tool that could operate within a more flexible environment. It was the intent of this research to provide a tool that is capable of maintaining relationships and objects within a changing environment.

A Description Of ODE

Comparison to Existing Object-oriented Database Systems

Similar to DMC, EXODUS, and other object-oriented database tools, ODE generates an object-oriented database given a description of objects, the relationship among objects, and the methods for each object. Unlike EXODUS, ODE automatically creates and maintains all the relationships within the same database. Unlike DMC, ODE can be used to both create the database system and drive the system. The reason ODE can accomplish these tasks is that it creates a database system that is wholly contained in data. This particular feature gives ODE several advantages over other types of database generators. First, because it is as a single tool, it can be used to control and describe the database. Second, by providing a management system that is completely data driven, ODE can be used to change or modify the system without loss of
current data. Such a feature gives the designer greater flexibility over his current environment.

ODE Work Space

The object-oriented database described in this paper is based on concepts proposed by developers of object-oriented programming languages [BENN87]. According to this paradigm, objects are defined according to their instance variables and the methods that act upon them. 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. A more thorough description of each of these features is given below.

Wegner describes a class as a template from which other objects may be created [WEGN87]. ODE defines a class in a similar manner and uses classes to represent real world entities. Classes contain properties that describe objects [WEGN87]. A real world setting is defined in terms of classes, just as variables within a programming language are defined in terms of data types. Therefore, a class within ODE is similar to a type definition.

ODE defines an instance of a specific class as an object. Each object consists of its own private memory and contains the values of its instance variables. Each object has a domain of instance variables that can be
specified only at the class level. ODE defines objects as data.

ODE defines the operators that manipulate data as methods. Methods are used to examine or alter the data. Each class contains its own set of methods. Thus, the only methods that can operate on an object are those that are defined for the object's class. A single method can be defined in several classes. ODE defines methods as database operators.

As previously stated, the ability to specify different types of relationships among the objects and the inheritance capability of those relationships is the major advantage of object-oriented databases over other traditional databases. Most object-oriented database systems support the ISA relationship. An ISA relationship is defined as a relationship 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. In an ISA relationship, the instance variables and methods specified at the class level are inherited by all the classes at the subclass level. In addition, new instance variables and methods may be specified for each class at the subclass level. This phenomenon is called full inheritance and is supported in ODE automatically whenever the user specifies a
relationship between two objects.

However, there are some relationships that do not fit into an ISA relationship. Examples of such relationships might be LOOKS-LIKE or FUNCTION-LIKE. In these types of relationships, the class should inherit only a subset of the variables from another class. This feature is called partial inheritance and is supported in ODE through user defined relationships. A user may define a relationship in ODE by name and also specify the properties that may be inherited by members of the relationship. The user may then assign the relationship between two entities within the database. For example, a user can specify the relationship LOOKS-LIKE and state that members of the relationship will inherit the properties of Height, Width, and Color. He could then create entity A and entity B and relate the two entities through the LOOKS-LIKE relationship. Entity A could inherit Height, Width, and Color from entity B, but nothing more. In this manner, ODE supports partial inheritance. Entity A might inherit a Height, Width, or Color value from some other relationship. The value that was ultimately inherited would depend upon which relationship was checked first. The system automatically checks the ISA relationships first, followed by the user defined relationships in the order that they are assigned to the entity.
Sample Database System Using ODE

This section describes a database application that was implemented using ODE. This database application defines a university registration system. Simply stated, the database assigns students to various classes which, in turn, are assigned to various classrooms throughout the campus. There are three different classes in this database system: the Student, Subject, and Room classes. The Student class represents an individual student and contains instance variables for Courses taken and Courses needed. Courses taken refers to the list of courses that the student has successfully completed. The Courses needed variable refers to the list of courses that the student needs to take in order to graduate. The Student class also contains an instance variable called Current schedule that defines the list of courses in which the student is currently enrolled. All of these instance variables have values that are themselves objects that are defined in other other classes in the registration system.

The Subject class is used to describe all the courses that a student can take at this particular university. The Subject class contains an instance variable called Prerequisite which describes a list of subjects that the student must take prior to enrolling in a specific course. It also contains the instance variable, Price, which represents the amount of money that the student must pay
in order to take a course.

Finally, the Room class refers to the actual classrooms where the courses will be taught. The Room class contains an instance variable called Capacity which represents the number of students that can fit into a particular room. It also contains an instance variable that defines a list of potential courses that can be assigned to a particular classroom. Finally, the Room class contains an instance variable that specifies the actual course that is assigned to the classroom. Figure 2 shows the class structure for the room registration system.

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>Add_subject</td>
<td></td>
</tr>
<tr>
<td>Courses_taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Courses_needed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current_schedule</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subject</td>
<td>Add_prerequisite</td>
<td>Equivalence</td>
</tr>
<tr>
<td>Prerequisite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>Delete_prerequisite</td>
<td></td>
</tr>
<tr>
<td>Room</td>
<td>Add_student</td>
<td></td>
</tr>
<tr>
<td>Capacity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Course_list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current_subject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete_student</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign_subject</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Class Structure
Because this is a simple system, there is only one type of relationship that needs to be specified for the classes which is the Equivalence relationship between subjects. The Equivalence relationship allows the system to recognize when the student has completed the prerequisites for a course. For example, suppose a student wants to take an engineering course, but the prerequisite for this course is engineering calculus. Because the student has completed science calculus, and because there is an Equivalence relationship between science calculus and engineering calculus, the system will recognize that the student has satisfied the prerequisites for this particular course and will allow him to enroll in the course.

Finally, there are a number of methods that need to be defined for each of the classes in the registration system. The Student class requires an Add_subject method that adds the various courses to the student's schedule. The Add_Prerequisite, Delete_Prerequisite, and Check_Prerequisite methods maintain the appropriate information concerning the prerequisites for each course. These three methods are associated with the Subject class. The Room class contains the Add_Student, Delete_student, and Assign_subject methods which are used to assign the appropriate class to a room and to limit the enrollment in a particular class. All three classes have one additional method that is responsible for creating new instances of
the classes.

After performing the initial setup, the database administrator can begin to assign students to various courses. In addition, he can alter the subjects that will be taught in various classrooms, the prerequisites for any course, or the equivalence relationships between two different subjects. In short, the database administrator can modify the database so that it conforms to the constraints that are current for each semester.

Although the registration system is a rather primitive example of a database application, it illustrates the flexibility of the object-oriented model. It also shows how the data model can be used to simulate the real world.

**Implementation**

**Data Storage**

I will now examine ODE's implementation by looking at how the database stores the data items and then by presenting a detailed description of the overall design.

Theoretically, all data should be stored in a way that allows the programmer to manipulate and access objects as though they were in virtual memory [KIM88]. ODE stores its classes, objects, relationships, and methods in a file that has characteristics similar to that of random access memory. Blocks of memory from this ram file can be allocated in any size--from one byte up to two gigabytes.
However, none of the entities should ever need the maximum amount of memory since they store their instance variables as a linked list of pointers to actual instance variable locations. The instance variable locations, in turn, contain other sets of pointers to the actual values. For example, the registration system would store the values for Courses_taken, Courses_needed and Current_schedule as three ram file pointers to the actual locations where instance variable information is stored. There is only one entity that might exceed the two gigabyte limitation and that is the entity of type "string." Theoretically, the system can point to a string value that exceeds the maximum length allowed by the system. However, this situation is highly unlikely. Thus, for all practical purposes, the size of a single entity in ODE is unbounded.

ODE's overall data structure is a series of trees. There are three main trees: a class tree, a methods tree, and a relationship tree. Pointers to the roots of these three trees are contained in a known location within the database, providing the only external entry point into the database. A description of the entities contained in each of the trees is summarized below.

The class tree nodes contain a list of the instance variables associated with the class, a list of all subclasses associated with the class, a list of all the objects associated with the class, a list of all possible
relationships that the objects of the class can have with one another, and a list of the methods associated with the class. The class's instance variables are stored as a linked list, whereas the class's list of subclasses, objects, relationships, and methods are stored as B trees. Each node in the instance variable list contains a type definition name and a pointer to the actual value. Each node in the object tree contains the object's name, a list of the object's instance variables, and a list of the relationships that the object has with other objects.

Similar to the class data structure, the instance variables for each object are stored as a linked list, and the object's list of relationships is stored as a B tree. Each node in the relationship and method trees, as well as the object's relationship tree, contains a reference (by name) to a user defined relationship as well as the name of the object or class to which the owning class or object is related. In this manner, the relationships and methods can be shared among the classes without having to duplicate information.

The relationship tree contains all of the user defined relationships that exist for a particular database. Each node within the tree contains the name of the relationship. Because ODE supports partial inheritance, each node in the tree also contains a list of the attributes that will be inherited by members of the
relationship. In those cases where an object can inherit the same property from two or more relationships, the system uses the order in which relationships are assigned to classes and objects to determine the order of inheritance. The ISA relationship always receives priority over any other type of user defined relationship.

The methods tree contains a list of pointers to operations that are supported in the database. Each node in the methods tree stores the name of the method along with its source code. The source code is stored as a linked list. Although this method of storage appears to be inefficient in terms of execution time, it facilitates rapid prototyping. All methods must be written in a subset of the C programming language, a language which also contains a few object-oriented constructs.

Because of the way that ODE stores its data, both physically and logically, the system can transfer database entities from permanent to transient storage very efficiently. Every time the system brings in additional data, it changes the file pointers to memory pointers. In order to avoid the problem of bringing in the data all at once, the memory pointers maintain a location variable that indicates whether the pointers are going to memory or a file. Thus, paging is unnecessary, and the entire object can be brought into memory in a single process. A
diagram of ODE's logical storage is shown in Figure 3.
Using ODE

ODE was written in MicroSoft C 3.0 and was developed on an 80286 class machine. ODE was designed for an interactive environment and, as such, provides an interface that allows the designer to create classes and objects, and to send messages. The database designer creates classes by responding to prompts that appear on a menu. Through this interface, the designer can create classes, subclasses, instances, methods, and relationships. When creating a class, the user is prompted for the name of the class and all of its properties. The designer has the opportunity to assign a default value to each of the properties of the class. For example, the designer might add the Radius property to the Circle class. The designer could then assign a default value to Radius which would be inherited by any of Circle's subclasses or instances. Finally, the designer is able to assign the names of relationships and methods that are valid for the class. At any time during the iteration, the designer may edit a class or assign and delete methods, relationships, or properties.

When creating subclasses or objects, the designer is asked to give the name of the class that "owns" the entity he is creating. He is then prompted for the name of the sub entity he is creating. The subclasses are entered in a manner similar to the class definition. In the case of
objects, however, the designer is not prompted for methods or new properties since both of these items are explicitly defined by the owning class. The designer has the opportunity to assign new values to the existing properties within the object. He also has the ability to assign relationships between the object and other existing objects within the database.

When creating a relationship, the designer is prompted for the name of the relationship and a list of the properties that belong to members of that relationship. Each object or class which references a relationship contains the name of the relationship along with the name of the class or object that belongs to the relationship.

When creating a method, the designer is prompted for the name of the method. He is then allowed to enter the source code for the method. The method must be written in ODE source code which is then automatically compiled and stored in the method tree. The method is stored in both source and object form. The source code version is used during the editing process while the object version is used during execution.

Once the designer has created all of the entities within the database, he can modify them as often as he likes. Any changes that are made to a single entity which may affect other entities within the database are automatically taken care of by the system.
Through this same interface, users can create new instances of classes and subclasses. They may also create new relationships and methods for the entities they create. Finally, they may invoke methods within the database in one of the following ways: (1) by using the direct command mode in the user interface, (2) through external software, (3) or through other user-defined methods. In the future, the external software will be supported by run-time libraries that will be able to link the database to other C programs. Thus, the database can be used programmatically from C source code.

Efficiency Considerations

Although object-oriented databases are receiving increasing attention, they have yet to be accepted by a majority of the database users. One of the reasons that object-oriented databases are not more widely accepted is that they are viewed to be inefficient, both in terms of storage and access of objects [DUHL88]. Because of the complex nature of some of the relationships, the system is often forced to access individual objects by traversing a complicated hierarchy. Also, as the number of methods increases, and as the number of objects increases, the size of the overall database tends to get quite large since each object and method must be stored in its own class. Fortunately, ODE has several features that alleviate some
of these inefficiencies.

Access time in ODE is significantly reduced because the user can obtain all the information about an object at one time. Although the user can retrieve an attribute's value by submitting a query using the attribute's name, he can also obtain this information by going directly to the class or instance and retrieving all of the attribute's values at one time. To do this, the user must be aware of the internal record structure of the desired entity. For example, suppose the user has created a class called Rectangle and wants to know the height and width of the rectangle. He may obtain this information by submitting two queries which require two separate database accesses. A second method allows the user to retrieve all the values associated with Rectangle. To employ this method, the user must know: (1) that Rectangle has two attributes associated with it, (2) that both Height and Width have a length of two bytes, and (3) that the value of Height comes before the value of Width. The user would then request the value of Rectangle and supply a four byte buffer in which to return all the values.

Therefore, ODE provides both an object-oriented access to information as well as a more efficient access to the same information. This feature may be likened to the high and low level file access methods available in the C programming language. The high level access method is
much easier for the user to understand and is more portable. However, the low level access method requires fewer lines of code and runs much faster. Users are given a choice as to which method they prefer.

Another feature that makes ODE faster is its use of B trees as a common data structure. As mentioned previously, ODE stores everything in one of three trees. Each node in the tree contains a pointer to the data it represents as well as pointers to the next items in the tree. By storing all (or as many as possible) of the nodes of a single tree together, ODE is able to access an enormous amount of information using a minimal number of disk accesses. Whenever an object is written and deleted during normal database transactions, the system simply updates the trees. Whenever the user leaves the database, the system organizes the trees into a single record and writes out the entire database to disk.

Current Status

The database modules completed are the physical ramfile processor, the logical tree I/O processor, the class, object, relationship, and method entity handlers, and the inheritance logic modules. The database modules not completed include the database user interface and the method language processor. These two modules will remain incomplete until the rest of the database is finalized.
However, the grammar for the method language can be found in the appendix.

In its current form, ODE is not usable by end users. It can, however, be linked with application specific C modules to perform various database tasks. It has been tested by linking it with test drivers to verify the functionality of its individual modules.

Conclusion

I have come to several conclusions over the course of this project. I have concluded that object-oriented databases provide the ability to prototype complex real world situations with much more speed and ease than do relational database systems. On the other hand, relational database systems are more efficient in both speed and storage than object-oriented database systems. To be truly useful, I feel that a database system must provide a mixture of efficiency and ease of use. Some applications do not depend on extremely fast database access. For those applications, it is more important to provide an easy interface to the database where the database system does most of the work to manipulate complex objects for the application. However, some applications depend on extremely fast database I/O. For those applications, the database should allow the application to get complex objects with one database access and then process them...
internally to the application. ODE allows both forms of data access for both forms of applications. Finally, I have approached the ODE project from the object-oriented world. As such, I have had to learn much about database management. Much of ODE's performance can be enhanced by using good database engineering techniques. For example, I currently read an entire object into memory when a user accesses it. However, some objects may be quite large, requiring some method of reading in portions of an object. Another good database engineering technique is to store as much of an object and the things associated with it on a single disk sector as possible. This greatly enhances the access time of the database. I am continuing to research database engineering methodologies which will make ODE more efficient and usable in a commercial environment.
BNF 'for' ODE Method Language

Method  \rightarrow\ Typename MethodName '=('MethodParms ')'} ParmDefs FunctionBody

MethodName  \rightarrow\ IDENTIFIER

MethodParms  \rightarrow\ MethodParms ',' IDENTIFIER

MethodParms  \rightarrow\ MethodParms ',' Typename IDENTIFIER

MethodParms  \rightarrow\ Typename IDENTIFIER

MethodParms  \rightarrow\ IDENTIFIER

MethodParms  \rightarrow\

ParmDefs  \rightarrow\ Parmdefs Typename NameList ';'

Parmdefs  \rightarrow\ Typename NameList ';

Parmdefs  \rightarrow\

NameList  \rightarrow\ NameList ',' IDENTIFIER

NameList  \rightarrow\ IDENTIFIER

Typename  \rightarrow\TypeID

TypeID  \rightarrow\ 'char'

TypeID  \rightarrow\ 'int'

TypeID  \rightarrow\ 'boolean'

TypeID  \rightarrow\ 'float'

TypeID  \rightarrow\ 'string'

TypeID  \rightarrow\ IDENTIFIER

FunctionBody  \rightarrow\ '{' InternalDefs Statementlist

InternalDefs  \rightarrow\ Parmdefs

Statementlist  \rightarrow\ Statementlist Statement

Statementlist  \rightarrow\ Statement
Statementlist ->
Statement -> FunctionBody
Statement -> AssignmentStatement
Statement -> LoopStatement
Statement -> ConditionalStatement
Statement -> Expression ';
Statement -> ';
AssignmentStatement -> Variable EQUAL Expression
LoopStatement -> 'for'Statement
LoopStatement -> 'while'Statement
LoopStatement -> DoStatement
ConditionalStatement -> IfStatement
ConditionalStatement -> SwitchStatement
Expression -> Addop Expression
Expression -> Expression Addop Term
Expression -> Term
Term -> Term Multop Factor
Term -> Factor
Factor -> '(Expression ')
Factor -> ConstantVal
Factor -> Variable
Factor -> FunctionCall
Variable -> IDENTIFIER
FunctionCall -> FunctionName '(' ArgList ')' ';
FunctionName -> IDENTIFIER
ArgList -> NameList
ArgList ->
ForStatement -> 'for' '([' ExpressionList ']' ';' ExpressionList ';' ExpressionList '])' Statement
ExpressionList -> ExpressionList ',' Expression
ExpressionList -> Expression
ExpressionList ->
WhileStatement -> 'while' '([' Expression '])' Statement
DoStatement -> 'do' FunctionBody 'while' '([' Expression '])' '('; Statement
IfStatement -> 'if' '([' Expression '])' Statement ConditionalElse
ConditionalElse -> 'else' Statement
ConditionalElse ->
SwitchStatement -> 'switch' '([' Expression '])' '{' Caselist '}'
Caselist -> Caselist 'case' Expression COLON Statementlist
Caselist -> 'case' Expression COLON Statementlist
Caselist ->
Addop -> '+'
Addop -> '-'
Addop -> '++'
Addop -> '--'
Addop -> '||'
Addop -> '|
Addop -> '!'
Addop -> '=='
Addop -> '!='
Multop -> '*'
Multop -> '/'
Multop -> '&&'
Multop -> '&'
Multop -> '^'
ConstantVal -> STRINGCONST
ConstantVal -> INTCONST
ConstantVal -> FLOATCONST
ConstantVal -> CHARCONST
ConstantVal -> BOOLCONST

All productions with no right hand side represent an epsilon production.
REFERENCES


[CAPL87] Caplinger, M. "An Information System Based on Distributed Objects," in Proc. OOPSLA Conference, October 1987, Orlando, FL.


