

University of Montana

## ScholarWorks at University of Montana

---

Graduate Student Theses, Dissertations, &  
Professional Papers

Graduate School

---

1987

### Using object-oriented techniques in a relation data model

Gregory D. Hume

*The University of Montana*

Follow this and additional works at: <https://scholarworks.umt.edu/etd>

**Let us know how access to this document benefits you.**

---

#### Recommended Citation

Hume, Gregory D., "Using object-oriented techniques in a relation data model" (1987). *Graduate Student Theses, Dissertations, & Professional Papers*. 5096.

<https://scholarworks.umt.edu/etd/5096>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact [scholarworks@mso.umt.edu](mailto:scholarworks@mso.umt.edu).

COPYRIGHT ACT OF 1976

THIS IS AN UNPUBLISHED MANUSCRIPT IN WHICH COPYRIGHT  
SUBSISTS. ANY FURTHER REPRINTING OF ITS CONTENTS MUST BE  
APPROVED BY THE AUTHOR.

MANSFIELD LIBRARY  
UNIVERSITY OF MONTANA  
DATE: 1987

USING OBJECT-ORIENTED TECHNIQUES  
IN A RELATION DATA MODEL

By

Gregory D. Hume

B. A., University of Montana, 1978

Presented in partial fulfillment of the requirements

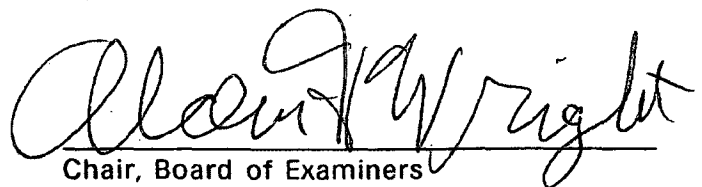
for the degree of

Master of Science

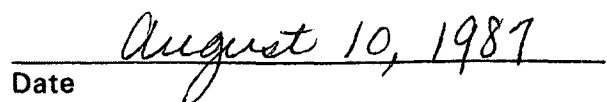
University of Montana

1987

Approved by

  
Chair, Board of Examiners

  
Dean, Graduate School

  
Date

UMI Number: EP40560

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP40560

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

Hume, Gregory D. M.S. July 1987

(Computer Science)

The Application of Object-Oriented Techniques in  
Implementing a Relation Data Model (60 pp.)

Advisor: Dr. Alden H. Wright



Data-base systems are widely used and the demands upon them are becoming greater. Their traditional use has been in business applications where the definitions of the domain field remain fairly static. Currently database systems are being used in scientific applications where the knowledge about the domain field is constantly changing. There is a great need for design strategies that will accommodate such fluctuations in the development of a database system.

The author is currently involved in the development of a Fire Effects Information System (FIRESYS). This project has employed object-oriented techniques in the the design of the database. These techniques were crucial in implementing constant changes in the FIRESYS database structure.

The FIRESYS database structure does not adhere to an established data model. The goal of this paper is to explore the use of object-oriented techniques in one such model, the relation data model.

## Table of Contents

Abstract	ii
Table of Contents	iii
List of Figures	v
Acknowledgments	vi
1. INTRODUCTION	1
1.1. The Subject Area	1
1.2. The FIRESYS Project	3
1.3. The Research Goal	5
2. OBJECT-ORIENTED DESIGN	8
2.1. Introduction	8
2.2. Abstraction	8
2.3. Objects	11
2.4. Message Sending	12
2.5. Inheritance	13
2.6. Objects in FIRESYS	15
2.7. Object-Oriented Design Compared with Traditional Design	16
3. AN OBJECT-ORIENTED DESIGN FOR A RELATION DATA MODEL	19
3.1. The Relational Data Model	19
3.2. Operations on the Relation Data Model	21
3.2.1. Selection	21
3.2.2. Projection	22
3.2.3. Join	23
3.2.4. Division	25
3.3. Justification for Using Object-Oriented Techniques	26
3.3.1. The Design of Objects	30
3.3.1.1. Relation Type	30
3.3.1.2. Domain Type	32
4. IMPLEMENTING A SUBSET OF FIRESYS USING AN OBJECT-ORIENTED RELATIONAL DATA MODEL	33
4.1. The FIRESYS Data Design	33
4.2. Implementing the Plant Species and Wildlife Species Relations	37
4.2.1. Creating the Relations	38
4.2.2. Entering Data	42
4.2.3. Query Operations	45
4.2.4. Changes by the Database Administrator	47

4.2.5. Creating a Data Dictionary	50
4.2.6. Security	51
5. CONCLUSION	53
5.1. Evaluation of Experiment	53
5.1.1. The Semantic Gap	53
5.1.2. User Interface	56
5.1.3. Utility Programs	57
5.2. Concluding Remarks	58
Bibliography	60

## List of Figures

Figure 2-1: FIRESYS Objects	15
Figure 4-1: High Level View of FIRESYS	36



## Acknowledgments

I would like to thank the following:

My advisor, Dr. Alden Wright for his dedication the past four years.

The University of Montana Computer Science Department and faculty members for their academic, financial and personal support.

All the graduate students I have been associated with, particularly those that have worked on FIRESYS.

Employees of the Fire Sciences Laboratory involved with the FIRESYS project.

My family for their patience.

## Chapter 1

### INTRODUCTION

#### 1.1. The Subject Area

The design of data models and the development of database systems is a rapidly evolving topic in computer science. Databases provide users with quick access to information. Information is becoming a more important resource for both business and government. The need for different types of information has grown along with the need for fast access to the information. Databases of the future must provide more than just numbers and facts. They must make information available in a form that can immediately aid in the decision making process.

There are two major components to a database. The first is a logical view of the data, consisting of data items and the relationship between data items. The second component is the implementation scheme to query and modify the actual data being stored. This paper will be primarily concerned with some implementation concerns of one type of data model.

Object-oriented programming is a relatively new programming methodology which has potential applications in the field of implementing databases. The object-oriented approach attaches procedural information to data. Both data and the functions that operate on data are grouped together in an object. The process of attaching procedural information to the data creates programs that are smaller,

less complex and more manageable. Object-oriented programming will be discussed in greater detail in the next chapter.

Traditional, or procedural, programming views procedural information and data as separate entities. A traditional program is analogous to a black box where the details of the program need not be known in order to run the program. Data is fed to this box and then is manipulated in some manner by the box. The result of this process is the desired data.

Several design and programming methodologies have been created so that the development and maintenance of traditional programs are as manageable as possible. The top-down design methodology is the most widely used. Changing aspects of procedural requirements is relatively easy when using top-down techniques. The problem with the top-down and other traditional methodologies is that changing the specifications of the data will often necessitate major changes in the procedural aspects of the design.

Incorporating object-oriented techniques in the development of a database may provide several advantages, particularly in a rapid prototyping environment. The most noticeable advantage may be the ease in which modifications can be made in the logical structure of the data. A change in the structure of the data should, theoretically, not change the programs which manipulate the database. Most changes in procedural types of information would be made right along with the changes in the definition of the data structure.

The complexity of the data model depends somewhat on the domain field. Some fields are very well defined and/or very well understood by the potential

users of the database. Other fields may be only partially understood and experimental in nature. In the case of the latter, the process of creating data models and building a database becomes a learning experience for the potential user. This process may also promote constant changes in the data structures.

## 1.2. The FIRESYS Project

The author is currently involved in a research project funded by the Intermountain Fire Sciences Laboratory in Missoula, Montana. This project is referred to as FIRESYS. This project began in June of 1985 with a development team of five people. The goal of this project is to create an information system to aid land managers in decisions regarding the use of fire.

Prescribed fires can be very useful for encouraging the growth of some plant species native to a site and eliminating the growth of others. The proper use of prescribed burning, often in the spring, can also reduce the potential of fires in the hot months of summer. Many factors are involved in predicting the effect of a burn. Some of these are the particular species in the burn area, the severity of the burn, the time of year and the amount of moisture present.

An early potential goal of this project was to develop an expert system that could advise land managers concerning the effects of burning an area and to write a prescription to burn if it was decided that a burn was desirable. The manual process of writing prescriptions is very time consuming. This is due to, among other things, insufficient and inconsistent cataloging of current research. The lack of information on fire effects among land managers and need for assistance in writing prescriptions were the primary motivational factors for this project.

The first immediate goal of the FIRESYS project was to develop a knowledge base. A knowledge base is one of the key components of an expert system. This knowledge base was to contain general information about fire effects, and biological information about plants, animals and communities that could be affected by fire.

During the course of developing FIRESYS some goals changed. The goals of predicting the effects of fire on plants and of writing prescriptions have either been postponed or transferred to other systems. The current goal is to create an information system, or database, that will provide a synthesis of current research to land managers. The programming part of this goal has essentially been reached, although the task of summarizing the knowledge of fire effects and entering it into the database is not complete. There currently is no emphasis on creating a system that can write prescriptions. The change of goals was due to many factors, the most prevalent being:

1. The lack of clear objectives in the beginning of the project.
2. The poor diffusion of existing knowledge in the field of fire effects in general
3. The lack of information available to land managers concerning the effects of prescribed burns.
4. The need early on in the project for presentable prototypes, creating the justification for further funding.
5. The difficulty of creating an expert system.

These factors have created an environment where the process of specifying

system requirements, designing system architecture and developing initial prototypes all occurred simultaneously. During the same period the users (members of the Fire Sciences Laboratory) were continuing to refine their ideas about how information on fire effects might be presented to the land manager. In spite of the constant state of flux, there has been a usable prototype system for inputting data operational since October 1, 1985.

The decision to explore object-oriented principles stemmed from research that suggested that object-oriented techniques were artificial intelligence techniques. Our experience has been that object-oriented techniques gave FIRESYS much flexibility in accommodating a great many changes in the definitions of the FIRESYS data structure. A great influence in our decision to use object-oriented techniques was an article written by Russell Greiner titled "R111: a Representation Language Language". (Greiner, 1980) The essence of this article was that representation languages that are designed for a particular domain are inflexible, hard to modify and impossible to use on another domain. A representation language whose domain is the field of representation languages is flexible, easy to modify and reusable.

### 1.3. The Research Goal

The FIRESYS data structure has not been modeled according to one of the commonly used data models (ie. relational, entity-relationship). This is due to the fact that the initial goal of the project was to build a knowledge base for an expert system rather than to build a database. My hypothesis is that if object-oriented

techniques were used to implement a database using the relational data model, the database would possess the same type of advantages as FIRESYS did. These advantages deal with the simplicity of the designing process and the ease in which modifications are made.

My first objective is to design, using object-oriented techniques, a database using the relational data model. Included in this design will be mechanisms for simple queries of the database, for adding data, for re-design of data structures and for maintaining security and integrity of data. My hypothesis is that the following are potential advantages of using an object-oriented approach:

1. Reducing the semantic gap. How closely can the logical design of a relational database correspond to the design of its implementation?
2. Enforcing constraints. Does the object-oriented approach provide an easy mechanism to enforce constraints?
3. Creation of data dictionaries. A data dictionary defines the logical relationships between the entities of a database. Can the object-oriented approach provide an automated production of a data dictionary?
4. Maintaining security. Can portions of the database be unreadable by some users?

My second objective is to implement a prototype containing a subset of the FIRESYS data structure using the object-oriented relational data model. The programming environment for this prototype will be similar to that of FIRESYS. The computer will be a VAX running the UNIX operating system. The programming language will be LISP.

This prototype will highlight key elements of FIRESYS at an early point in its development. A system to modify the properties of objects will then be designed. Some of the same types of changes made in the evolution of FIRESYS will be implemented in the relational prototype.

My hypothesis is that this new system will be as flexible in accommodating changes as FIRESYS. This flexibility will come from:

1. Data structures that represent real world entities.
2. Logical and physical data independence.
3. Ease in adding and modifying data structures.
4. Simplicity of data input.

The remainder of the paper is outlined as follows:

- \* Chapter 2 is a study of the key aspects of the object-oriented approach to design.
- \* Chapter 3 is a brief description of the relation data model.
- \* Chapter 4 is a summary of the object-oriented, relational data model prototype.
- \* Chapter 5 contains concluding remarks.

There are aspects of using object-oriented techniques in database systems that will not be addressed in this paper. A topic that will not be addressed is the efficiency aspects of using the relational data model. Both the LISP programming language and the relation data model are often criticized for being inefficient. Optimization techniques can be used to minimize inefficient use of memory and to speed up processing.



## Chapter 2

### OBJECT-ORIENTED DESIGN

#### 2.1. Introduction

There has been much written about the object-oriented approach. Many researchers state a list of characteristics that can be attributed to the object-oriented approach. There are several characteristics that most researchers in this field include. These characteristics are:

1. Abstraction
2. Object Identity
3. Message Sending
4. Inheritance

These characteristics are very interrelated. A complete description of any of these involves references to the others. The following sections will highlight important aspects of these characteristics.

#### 2.2. Abstraction

The object-oriented approach is one step in a natural evolution of software development. This constant evolution is striving towards greater abstraction. Abstraction often carries a connotation that it is theoretical and therefore difficult to understand. Abstraction is a process that strives for the opposite. Abstraction

in Computer Science is hiding as many details as possible while concentrating on the essence of the problems at hand.

The successive generations of languages are good examples of milestones in the above mentioned evolution. The first generation of languages are machine code languages. These languages consisted of nothing but zeros and ones. An example of several lines of machine language code might look like:

```
00100111
00100010
01101110
00101010
```

Every digit in every location has a particular meaning. The programmer must constantly be aware of these meanings. There is no abstraction involved with machine languages because there are no non essential details that can be ignored when a solution to a problem is coded.

The second generation of languages are assembly languages. Assembly languages provide instructions that have intuitive meanings, therefore providing some abstraction. Examples of typical assembly languages instructions are inc (increment), tst (test), mov (move), add (addition) and so on. These instructions correspond directly to machine instructions and, therefore, alleviate the programmer from having to be aware of what particular combinations of digits might mean. Using an assembly language instead of a machine language is analogous to being able to listen to letters as opposed to listening to Morse code.

The third generation of languages are high level languages such as FORTRAN, COBOL and Pascal. These languages provide greater abstraction than assembly languages by allowing the developer to ignore some implementation

details. For example, adding numbers in assembly code may involve many lines of code. An expression to add numbers in a high level language requires only one line of code. The low level implementation details of evaluating expressions are ignored when using a high level language.

The concept of an abstract data type is a recent development in the evolution toward greater abstraction. There are two parts to an abstract data type, the operations that can be performed on that type, and the implementation of those operations. In an abstract data type, the syntax and semantics of the operations are specified independently of the implementation of the operations.

A typical example of an abstract data type is a stack. A stack is a list of elements. The semantic essence of a stack is that the last element entered on the list will be the next element taken off the list. Typical operations for this abstract data type are to 'push' an element onto the list and to 'pop' an element off the list. An abstract data type specifies the syntax of its operations. For example, the push operation might require a parameter for a value and a parameter for the name of the stack. If this is the case, the call would look something like:

```
push(item, stack).
```

The process of specifying the exact operations that are allowed on a data structure ensures that the integrity of the data will be preserved, and modifications of the data will never be unintentional. The exact implementation of the data structure is not dependent on the operations that modify the data structure. This scheme follows the predominant principle of abstraction: ignore non essential details and concentrate on essential details. The concept of the abstract data is

the natural precursor to the concept of an object and the object-oriented approach to software development.

### 2.3. Objects

An object in the object-oriented scheme is the representation of any entity that can be perceived. Employees, missiles, state governments, grocery stores are all real world entities and software has been written to maintain information on all these entities. The object-oriented approach treats these entities as 'objects'. With this attitude, a software design can have a close correlation to a real world situation.

As with an abstract data type, exactly how a object is implemented in a software environment does not need to be known outside of that object's environment. The essence of an object to the outside environment is its identity. If the object's identity is known, the object can be accessed. The mechanisms for communicating with an object will be discussed in the subsequent sections. The details of an object, such as its unique properties or the data structures used to represent the object, are hidden from the environment external to the object.

The potential advantages of using this notion of an object are the simplicity of design of data structures and the independence that these designs have from implementation concerns. A high level design of software, using the object-oriented approach, concentrates on the high level objects and their important properties. A high level design that is less technical in nature is more understandable by end users and software developers.

## 2.4. Message Sending

There are basically two types of properties that an object can possess. One type is factual information and the other type is procedural information. Factual information is the typical type of data that can be associated with entities. For example, an employee has a name, a social security number, an address and so on.

An example of an employee possessing procedural information can be the algorithm used to calculate that employee's pay. This algorithm can be embedded within an employee object and hidden from the environment outside of the employee object. When a property embedded in an object is procedural in nature, it is called a method.

The mechanism used to communicate with objects is called message sending. To access an employee's address, a message is sent to that employee object requesting its address. To access the employee's amount of pay, a message is sent to that employee object requesting its amount of pay. In this latter case, there is no factual information available, only a method that is capable of calculating the amount of pay. This situation would cause the method to be activated. The result would be an amount of pay which then would be returned to the sender.

Message sending is the only mechanism used to access the properties of an object. Message sending supports another principle of abstraction, that of information hiding. Information cannot be accessed or altered except through a standardized set of messages. The message sending system permits objects to have the knowledge of how to access information from other objects, but no

further knowledge about these other objects. This protocol ensures the integrity of data within an object. Data can only be accessed or altered through specified means and never in an unintentional or accidental way. This is analogous to abstract data type principles where there are specified operations and the details of implementation are hidden.

## 2.5. Inheritance

The concepts of inheritance is crucial to the object-oriented scheme. Consider the above employee example to illustrate this concepts. If a company has hundreds of employees it would not make sense to embed the same pay algorithm in hundreds of objects. The solution to this problem is to consider 'employee' a class of objects. The pay algorithm can then be attached to that class of object. Each individual employee is an object that can inherit the pay algorithm from the class of objects called employee.

A class of objects is itself an object. An object that is a member of a class of objects is an instance of the class of objects. An instance of an object is capable of inheriting properties from that object.

The previous example illustrates that an individual employee object can inherit a property, the pay algorithm, from the employee class of objects. The individual employee object may also maintain its own properties. The best examples would be a name and a social security number. Every employee has his/her own name and a unique social security number. Some individual employee objects may also possess their own pay algorithms. A practical example of this would be when some employees receive a commission in addition to a salary.

The message scheme can be used to implement inheritance. When a message is sent to an object and that object does not know how to respond, the object re-sends the message to the object that it is an instance of. This first object is not concerned with how the message is processed, it only expects a value to be returned. When it receives this value, it then returns it to the original sender.

All objects handle message passing in this manner. When an object re-sends a message to its parent class of objects, it does not know if that object had to re-send the message to its parent class of objects. Inheritance may occur through many levels of classes of objects.

The first advantage of this inheritance scheme is to reduce the amount of code that needs to be maintained. Using the employee example, it could be that a large majority of employees are paid in the same manner. That pay algorithm can then be stored in one place, inside the object that is the class of employee objects.

A second advantage of the inheritance scheme is the ease in which modification can be made. For example, when an employee is given special incentives along with his/her salary, a separate pay algorithm can be placed within the object representing that individual employee. This modification will have no effect on the rest of the system.

## 2.6. Objects in FIRESYS

This section provides a short description of how the object-oriented approach was employed in FIRESYS. Consider figure 2-1 which contains several FIRESYS objects.

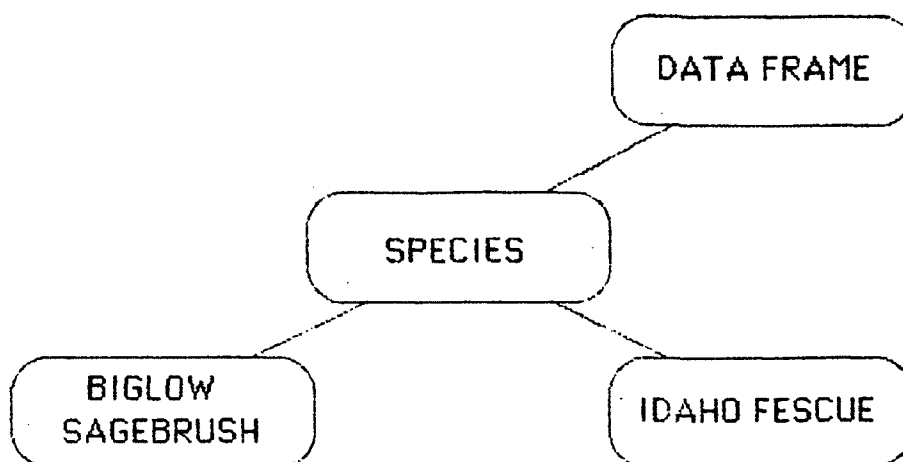


Figure 2-1: FIRESYS Objects

Both the Bigelow sagebrush object and the Idaho fescue object are instances of a class of objects called SPECIES. Each of these two instances maintain some unique properties. They have their own names, their own geographic locations where they grow and so on. These two instances also inherit properties from their parent class of objects, the SPECIES object. Many of the properties that the two instances inherit deal with the implementation of FIRESYS. For example, both



inherit the same prefix used in generating a symbol that is used by the system to identify objects. This prefix is "SPECIES".

The object SPECIES is an instance of the class of objects called DATA FRAMES. The SPECIES object maintains some properties unique to it. An example is the above mentioned prefix. The SPECIES object also inherits properties from its class of objects called DATA FRAMES. One such property is a display routine. The contents of almost all instances of DATA FRAMES are displayed on the screen in the same manner.

## 2.7. Object-Oriented Design Compared with Traditional Design

The concept of an object containing such information is a radical departure from the more traditional approach, called top down design. This methodology decomposes a problem into hierarchy of sub-problems. The first emphasis of this methodology is on the processing, the second emphasis is on the structure of data. Another approach often used in business applications, called data structure design, takes the opposite approach. The data structures are designed first, then the sub-programs that will operate on the data structures are designed.

The point is that traditional approaches to design have treated data and procedures as separate entities. The object-oriented approach treats data and procedure as the same type of entity. They are both properties that can be contained within an object.

An argument against the use of traditional methodologies is that they do not provide a high degree of abstraction for both procedures and data. For example,

the design of the hierarchy of modules in the top-down approach provides a high degree of abstraction for the procedures that need to be performed. A high degree of abstraction of the data structures involved is not incorporated into this hierarchy of modules.

An example from the FIRESYS project is used to further illustrate the difference of the two approaches. Various types of data are required to be displayed on the terminal in different forms. For example, When the references to information are displayed, they need to be numbered and listed on separate lines. When text-like information is displayed, the screen must be cleared if it will not fit at the current position on the screen.

In the traditional top down approach, one of several things may occur when something, for example a list of references, needs to be displayed on the screen. One possibility is that a high level display subprogram would be called and the data would be sent to that subprogram as a parameter. This subprogram would then make decisions, due to the fact that the parameter is a list of references, to activate the appropriate sub module within the display subprogram.

Another possibility is that there is a control structure involved before the call to a display sub-program. This control structure would determine that the data is a list of references and then make the call to the appropriate sub-program. Either possibility includes a greater degree of complexity due to a lesser degree of abstraction.

In FIRESYS, when an object like a list of references is to be displayed, a message is sent to that object. There is no control structure before the call and

no decisions that have to be made according to the type of parameters after the call is made. The details of how the object is displayed are hidden within the object and hidden from the outside environment. The only added complexity deals with the inheritance mechanism and this mechanism is standard for all objects.

## Chapter 3

### AN OBJECT-ORIENTED DESIGN FOR A RELATION DATA MODEL

#### 3.1. The Relational Data Model

The relational data model is a scheme for defining the logical relationships of various information. The details of implementing a database are ignored in the relations data model. The primary component of this model is the relation. A mathematical definition of a relation is a subset of a cross product of sets of attribute values. (Smith 1987/ p. 305)

The relational data model puts all information in table form. The advantage of this is that it is easy for the user of a system to understand the logical view of the data. The following is a simple example. It is important to stress that this is a logical view of the data.

NAME	ID	SEX	AGE	TITLE
Johnson, Pete	34782	M	42	Manager
Billings, Sara	34551	F	29	Clerk
Jones, Phil	44021	M	34	Janitor
Fraizer, Susan	34618	F	22	Clerk

The notation for specifying the above relation is:

EMPLOYEE FILE (NAME, ID, SEX, AGE, TITLE)

All relations are tables that have the following properties:

1. Each entry in a table represents one data item; there are no repeating groups.
2. They are column homogeneous; that is, in any column all items are of the same kind.
3. Each column is assigned a distinct name.
4. All rows are distinct; duplicate rows are not allowed.
5. Both the rows and the columns can be viewed in any sequence at any time without affecting either the information content or the semantics of any function using the table. (Martin 1976/ p. 96)

In the relational data model, columns are referred to as domains and rows are referred to as tuples. A relational database is composed of one or more relations. Every relation contains a primary key which is used to uniquely identify the a tuple. A primary key is composed of one or more of the domains of the relation. Each tuple must be uniquely identified by its primary key. Consider the example above. The primary key is the ID domain. Every other domain lends itself to the possibility of common values for different tuples

Let us assume that the name of the above relation is EMPLOYEE FILE. The primary key is ID (underlined). The ordering of the domains is not important. There are four tuples and five domains in this relation.

## 3.2. Operations on the Relation Data Model

There are four basic operations performed on relations when a relational database is used for query purposes. These are selection, projection, join and division. These operations are presented in very simple form. Any operations required in a query could be performed by combinations of these four operations.

The result of using any combination of these four operations will be a new relation. A new relation created by any operation will be referred to as temporary relation. Their lifetime consists of the duration of a query session. They are not stored on a storage device (disk or tape) for later use. Relations that are stored on such devices for later retrieval will be referred to as permanent relations.

### 3.2.1. Selection

The selection operation selects certain tuples from a relation based on the value of one domain in a tuple. The selected tuples then form a new relation. This new relation has the same set of domains and the same primary key as the original relation. The tuples in the new relation are a subset of the tuples of the original relation.

Suppose we wish to view all the tuples of EMPLOYEE FILE where the employees are older than thirty. The following is the notation used for the selection operation:

```
EMPLOYEE FILE2 <-- Select EMPLOYEE FILE  
                   Where (AGE > 30)
```

The resulting relation, EMPLOYEE FILE2 (NAME, ID, SEX, AGE, TITLE), would look like:

NAME	ID	SEX	AGE	TITLE
Johnson, Pete	34782	M	42	Manager
Jones, Phil	44021	M	34	Janitor

The actual implementation of the selection operation could allow for multiple conditions or boolean combinations of conditions. If the implementation does not allow for this, the desired result could be achieved through successive calls to the selection operation.

### 3.2.2. Projection

The selection operation can be thought of as processing a relation by tuples. Some of the tuples of the first relation may not be included in the resulting relation. The projection operation can be thought of as processing a relation by domains. The projection operation creates a relation that has fewer domains than the original relation.

If we wish to view all the possible titles and sexes of EMPLOYEE FILE, the notation of the projection operation is:

```
EMPLOYEE FILE3 <-- Project EMPLOYEE FILE
                  On (TITLE,SEX)
```

The resulting relation, EMPLOYEE FILE3 (TITLE,SEX), would look like:

SEX	TITLE
M	Manager
F	Clerk
M	Janitor

The projection operation often forces all of the domains to be the primary key. If the primary key(s) are not included in the projection operation then no subset of domains can guarantee the uniqueness of each tuple. The exception is when the original relation's primary key is included in the projection. In this case, the resulting relation would have the same number of tuples as the original tuple.

In EMPLOYEE FILE there were two female clerks. In EMPLOYEE FILE3 there is just one tuple for female clerks. This is consistent with the fourth rule in the definition of a relation. Every tuple, or row, must be unique.

### 3.2.3. Join



Selection and projection are monadic operations. They are performed on only one relation. Join is a dyadic operation. It is performed on two relations. A join concatenates tuples from different relations if their common domains have equal values. Consider the relation: EMPLOYEE RELIGION (NAME, ID, RELIGION) :

NAME	ID	RELIGION
Johnson, Pete	34782	Catholic
Gil, Russell	34979	Protestant
Billings, Sara	34551	Jewish
Fraizer, Susan	34618	Protestant

The operation:

EMPLOYEE FILE3 <-- Join EMPLOYEE FILE, EMPLOYEE RELIGION

would look like:

NAME	ID	SEX	AGE	TITLE	RELIGION
Johnson, Pete	34782	M	42	Manager	Catholic
Billings, Sara	34551	F	29	Clerk	Jewish
Fraizer, Susan	34618	F	22	Clerk	Protestant

In this example the two relations have the same primary keys. This ensures that there are no duplicate tuples. This is not always the case in the join operation. Just as in the projection operation, the join operation must discard duplicate tuples. The resulting primary key(s) of a join is the combination of the primary key(s) of the original two relations.

### 3.2.4. Division

Division is another dyadic operation. Division discards all domains from one relation that are common domains with another relation.

Let us suppose that EMPLOYEE FILE4 looked like:

SEX	AGE	SALARY
M	44	37,000
F	31	16,500
M	44	12,125
M	27	14,000
M	29	22,250
F	31	27,400

Now let us suppose that there was an EMPLOYEE FILE5 that looked like:

ID	SALARY
99981	37,000
25987	16,500
43761	12,125
38982	14,000
26991	22,250
23741	27,400

At this point, if we wish to give the command:

```
EMPLOYEE FILE6 <-- DIVIDE EMPLOYEE FILE4 by EMPLOYEE FILE5
```

The relation would look like:

SEX	AGE
M	44
F	31
M	27
M	29

Division is somewhat similar to projection. The difference is that division requires another file to determine the domains to discard. Like projection and join, division must discard duplicate tuples.

### 3.3. Justification for Using Object-Oriented Techniques

The relational data model has been used the past two decades and object-oriented techniques have received much attention in the past decade. There is, however, very little research on using object-oriented techniques with the relational data model. This section contains a justification for combining these two concepts and a preliminary object-oriented design of a relation data model.

As stressed earlier in this chapter, the advantage of using the relational data model is that it presents a logical view of the data in a manner that the end user can understand. The end user does not need to know anything about the implementation of the database. The end user does need to work with the database developer to create the logical definitions but never with any considerations toward the internal representations of the definitions.

The process of the end user and the developer working together to produce the exact logical specifications of the database is traditionally called the analysis phase of the software lifecycle. This is traditionally the first phase of the software lifecycle. The end product is a document that contains a set of precise specifications. In the case of a database, this specification document, or a portion of it, is called a data dictionary. A data dictionary contains the logical definitions of the inter-relationships of the data. For example, a data dictionary would specify the domains and primary keys of a relation.

A data dictionary does not address any implementation concerns. These concerns are generally addressed in the second phase, called the design phase. Some of these concerns deal with what machine to use and what programming language to use. A more important implementation concern for this discussion is what data structures should be used to represent the semantics of the real world situation as outlined in the data dictionary. The conclusion of this design phase will produce a second view of the world to be modeled. These two views are:

1. A logical view of the world to be modeled as defined in a data dictionary.
2. A technical view of the data structures used to represent the logical view of the world to be modeled. This technical view has two main components:
  - a. Storage of the data structures in the computer.
  - b. User interface to the data structures.

These two views pose several questions. How closely do these two views

correspond to each other? Is it possible that all of the details of a real world object can be captured in a data structure object?

How important the answers to these questions are depends, to a degree, on the complexity of the data. If the data structure objects do not correlate closely with the real world objects in a simple database yet all the requirements of the system are met, then the answers to these questions are irrelevant. This situation may be almost impossible if there are changes made in the logical view of the data or if the logical view of the data is complex. Complexity may arise from a large number of relations or from aspects not covered by the relation model. Examples of such aspects might be constraints on domain values and security privileges on portions of the data base.

The ability to design data structure objects that capture all the essential qualities of real world objects will provide several advantages. The first is the simplicity of the design process. The developer has already employed a particular methodology to specify the logical view of the data. It would be less time consuming to re-use the previous strategies and techniques in the creation of a design of the data structures than it would be to use a separate approach.

Another advantage would be that the end user can be more involved in the total development of the system. In the typical software lifecycle, the end user's participation is suspended at the end of the analysis phase. The developer then performs the design and coding phases without the end user. It is often not until the testing phase that the end user resumes involvement in the development of the system. This is somewhat of an oversimplified situation but often is the case

because the end user does not understand the technical aspects of the design phase. If the design of the data structure objects were as simple as the definition of the real world objects then the end user could participate to a greater degree.

The ability to make future modifications to the logical view of the data will depend greatly on the simplicity of the data structures used. If the details of a real world object are not all encapsulated in a corresponding data structure then modifications to the logical view of a real world object will involve more than just modifications to a corresponding data structure.

Object-oriented techniques have been acclaimed for being able to fully represent real world entities. The process of using these techniques in designing a relational database would reduce the gap between the logical view of the data and the structures used to represent them.

The term user needs to be addressed at this point. There are several types of users when referring to a data base. One is the end user that will be allowed to access the database through a query language. A second type of user is a data entry person. A third type of user is a database administrator. This person is responsible for maintaining the data dictionary and for maintaining the database software. Unless specifically stated otherwise, the term user will refer to the later definition for the remainder of this chapter. One of the primary justifications for using object-oriented techniques is to provide easy to use tools for a database administrator.

### 3.3.1. The Design of Objects

There are three primary objects, or classes of objects, to consider: relations, domains and tuples. The primary components of an instance of a relation are a list of domains, a primary key and a list of tuples. A tuple object contains actual data values. A tuple is designed as an instance of a relation type class of objects so that a tuple can inherit its list of domains from the relation object. Therefore, this preliminary high level design will address the relation and domain object but not tuple object.

#### 3.3.1.1. Relation Type

The first object to design in this relational data model is the relation. The normal process of query creates many temporary relations. Both relations that are permanently stored and relations that are created for temporary use will usually share some common characteristics. The following are the methods and properties of a typical relation object. These methods are invisible to the user.

1. DISPLAY-RELATION - This method provides a mechanism for displaying a relation on a screen. It is very likely that many instances of relations will not inherit this method since they will have their own DISPLAY-RELATION method. Some instances of relations will require customized display mechanisms. This method will repeatedly call DISPLAY-TUPLE.
2. DISPLAY-TUPLE - This method provides a mechanism for displaying a tuple that belongs to the relation. This method will call DISPLAY-DOMAIN.
3. SELECTION - Described above.

4. PROJECTION – Described above.
5. JOIN – Described above.
6. DIVISION – Described above.

The following methods of a relation object are accessible to the user through a user interface. This user interface is designed for use by a database administrator who is responsible for the maintenance of the data.

1. USER-CREATE-RELATION – This method allows the user to add a permanent relation. The primary purpose of this domain is to allow the user to specify the domains and the primary key(s).
2. USER-DELETE-RELATION – This method allows the user to delete a permanent relation.
3. USER-MODIFY-RELATION – This method allows the user to add or delete domains from the domain list of a relation, or modify characteristics of a relation.

The following methods allow a data entry person to enter or delete data:

1. CREATE-TUPLE – This method allows the user to input data into a tuple of a permanent relation. This method will access the domain list of the relation and call USER-ADD-DOMAIN-VALUE for each domain on the domain list.
2. DELETE-TUPLE – This method allows the user to delete a tuple in a permanent relation.

The following are properties (but not methods) of a relation object:



1. **PRINT-NAME** - This contains a short phrase used to logically identify a relation when it is displayed.
2. **PARENT-OBJECT** - This contains the identification of the class of objects that this relation is an instance of.
3. **DOMAINS** - This contains the list of domains that this relation contains.
4. **TUPLES** - This contains a list identifying the tuples that are currently maintained by this relation.
5. **PRIMARY-KEY** - This contains a list of domains that comprise the logical identification of a tuple.

#### 3.3.1.2. Domain Type

A domain can be more traditionally referred to as a type. A domain has certain constraints as to what its legal values are. Many of the properties of a domain object deal with these constraints. The following are some of the methods of a typical domain object:

1. **IS-LEGAL** - This is a method that checks if a value meets the restrictions placed on its domain. If the domain is of some string type, the restrictions may deal with size. If the value is numerical, the restrictions may deal with a maximum or minimum value.
2. **ADD-VALUE** - This method is called when a data entry person is adding data. This method will call **IS-LEGAL**.
3. **DISPLAY-DOMAIN** - This method will provide information on how to display its domain on the screen.

## Chapter 4

### IMPLEMENTING A SUBSET OF FIRESYS USING AN OBJECT-ORIENTED RELATIONAL DATA MODEL

#### 4.1. The FIRESYS Data Design

The Firesys design is based on a frame system. This is because the original goal was an expert system. The original design was based on research in a frame based knowledge representation scheme. Many of the differences between a frame based knowledge representation scheme and a database are little more than terminology.

A frame is analogous to a tuple in that a frame contains related values. These values are stored in slots. A slot is analogous to a domain in that a slot contains a value, and there are usually constraints on the values of a particular slot type. There are only five types of slots in FIRESYS. These are:

1. Atom. This term was borrowed from lisp. It means that the value of this slot may contain a single value. This value could be a number, a word or a short phrase.
2. List. This term was also borrowed from lisp. It means a collection of atoms.

3. Text. A text slot may be of any size. It may contain any combination of prose or tables. An important consideration of this slot is that, although very readable and informative to the end user, the system may not make any inferences based on the value of this type of slot.
4. Header. This slot contains no value. It is use to group other slots into a logical category.
5. Generated Frame Pointer. This slot links the parent frame to its subordinate frame.
6. Generated Frame Pointer List. This slot is similar to the generated frame pointer slot except that there may be any number of subordinate frames linked to the parent frame through this slot.

The last two slots mentioned are crucial in implementing the data as a tree like structure. These slots are the equivalent of arcs in tree terminology. The overall scheme of the FIRESYS tree is as follows:

1. The the higher the node (frame) is in the tree, meaning the closer to the root, the more general in nature the information is.
2. The lower the node in the tree, the more specific the information is.

The root node in the FIRESYS tree structure is called the SUPERIOR frame. There are various slots in this frame that deal primarily with internal system maintenance. In fact, this frame is totally invisible to the end user.

The SUPERIOR frame contains three slots of type generated frame pointer list. These slots are:

1. Ecosystems. A list of ecosystems. Currently FIRESYS contains one ecosystem; the sagebrush ecosystem.
2. Plant Species List. A list of plant species. Currently FIRESYS contains approximately one hundred plant species.
3. Wildlife Species List. A list of wildlife species. Currently FIRESYS contains approximately ten wildlife species.

An ecosystem may contain any number of cover types. These are implemented with a slot of type generated frame pointer list within the ecosystem frames. A cover type's primary key, in a relational data model scheme, is the combination of names of the cover type and its superior ecosystem.

There are frames subordinate to the plant species and wildlife species frames. These subordinate frames contain more specific information. Unlike the cover types within the ecosystems, the only real purpose for most of these subordinate frames is to partition data into logical groups. The primary Key for such subordinate frames is the inherited species (plant or wildlife).

Figure 4-1 shows a subset of the high level logical view of the FIRESYS tree like data structure. The actual FIRESYS design does not follow a recognized data model such as the relational model. There are several reasons for this. The project evolved to the point where a customized information system was the goal. Many of the normal type queries (queries that would require the projection, selection, join and division operations) that would be performed on a database were not required on this information system. Access to information must follow, almost exactly, the tree like structure of the data.

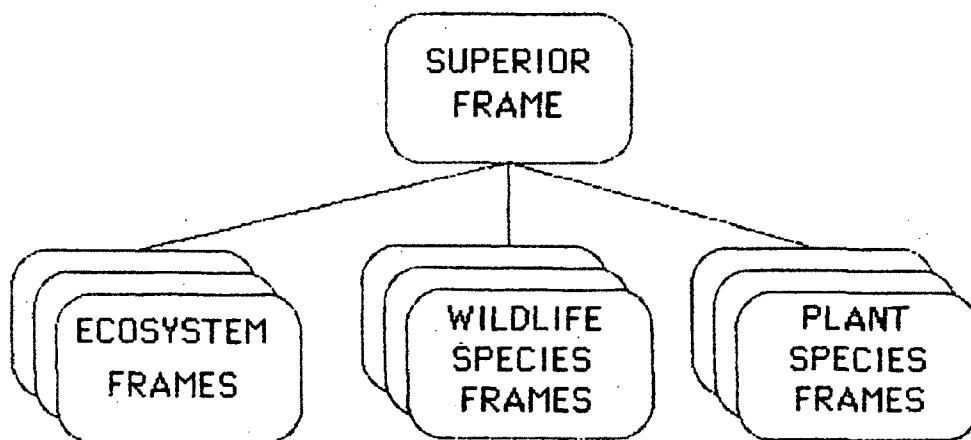


Figure 4-1: High Level View of FIRESYS

Another reason the FIRESYS design does not follow a recognized data model and does not incorporate many of the usual type queries is its heavy dependence on textual information. The FIRESYS queries were designed primarily to lead the user through a library of textual information. Queries that use the projection selection, join and division operations rely on comparing actual values. Although comparisons can be made on textual information, only certain comparisons are easy to implement. For example, a simple comparison might involve a search for a key word. Comparing textual information for semantic meaning is very difficult. There is current research on making such comparisons on textual information, but the current technology is insufficient.

## 4.2. Implementing the Plant Species and Wildlife Species Relations

The purpose of implementing a subset of the FIRESYS database using an object-oriented relational data model is to explore possible advantages and disadvantages of using this approach. The prototype implementation is incomplete in that it does not model the entire FIRESYS database and there is not a polished user interface.

As noted in the previous section, textual information may be of little value for making many traditional type queries. Therefore, the following design omits text slots. Without the text slots there is less need to partition more specific information into sub-frames. For the purposes of this implementation, the following list are the domains for the plant species relation. Included with each domain is the class of domain objects that it is an instance of. This could also be an entry into a data dictionary.

SPECIES (atom-type)

ABBREVIATION (atom-type)

LIFE FORM (atom-type)

BLM PHYSIOGRAPHIC REGIONS (list-type)

ECOSYSTEMS (list-type)

STATES (list-type)

ADMINISTRATIVE UNITS (list-type)

The following is a data dictionary entry for the implementation of the wildlife species relation:

WILDLIFE SPECIES (atom-type)  
 ABBREVIATION (atom-type)  
 CLASS (atom-type)  
 ECOSYSTEMS (list-type)  
 STATES (list-type)  
 ADMINISTRATIVE UNITS (list-type)  
 BLM PHYSIOGRAPHIC REGIONS (list-type)

#### 4.2.1. Creating the Relations

The first step in this implementation is to create the instances of the above two relations. This is the responsibility of a database administrator. Through an interface program, a message is sent to the relation-type object. This object is the class of objects that all relation objects are instances of. The message is to create a new relation. This activates the appropriate method within the relation-type object. This method interacts with the database administrator. The following is an example of that interaction for the plant species relation:

=====

Please enter the print name  
 for the new relation:           Plant Species

You are to enter the names of the primary keys,  
 after each entry you will be prompted for a domain

type. Enter a period to conclude.

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a primary key domain: .

The primary key has been entered. You are to enter the names of the non-primary key domains; after each entry you will be prompted for a domain type. Enter a period to conclude.

Enter a domain: Abbreviation

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a domain: Life Form

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a domain: Ecosystems



Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a domain: States

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a domain: Administrative Units

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a domain: BLM Physiographic Regions

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 3

Enter a domain: .

Plant Species relation has been added.

=====

In this case, the add relation method allows the database administrator to add domains that are one of three types already in the system, positive integers, lists and atoms. A utility can be provided to allow the database administrator to add a new domain type. It may be that the number of types of domains remains static and such a utility might not be very important.

It is very important that the database administrator be able to attach special properties to specific domains. For example, there are only certain values allowed for the domain Life Form. These are:

1. Tree
2. Shrub
3. Graminoid
4. Forb

Life Form is an instance of the atom type domain and that the atom type domain object has its own method to determine if a value is legal. The database administrator can attach a method to the Life Form domain object that ensures that only one of the above values are allowed. Utility programs can be provided to make such a task easy.

#### 4.2.2. Entering Data

Assuming that both relations have been created the next step is to enter data. This is the responsibility of a data entry person, not the database administrator. A user friendly interface program communicates with the user. If the user chooses to add a new plant species an appropriate message would be sent to the plant species relation. The following is an example of that interaction:

=====

Entering for Species Name

Enter : Fectuca Idahoensis

=====

Entering for Abbreviation

Enter : FEID

=====

Enter one of the following numbers  
representing a value for Life Form

- 1 : Tree
- 2 : Shrub
- 3 : Graminoid
- 4 : Forb

ENTER-OPTION: 3

=====

Adding value for :  
BLM PHYSIOGRAPHIC REGIONS

Enter each item when prompted.

Enter a period to terminate list :

Enter : Northern Rocky Mountain

Enter : Wyoming Basin

Enter : .

=====

Adding value for :  
States

Enter each item when prompted.  
Enter a period to terminate list :

Enter : Idaho

Enter : Montana

Enter : Wyoming

Enter : .

=====

Adding value for :  
Ecosystems

Enter each item when prompted.  
Enter a period to terminate list :

Enter : Sagebrush

Enter : .

=====

Adding value for :  
 Administrative Units

Enter each item when prompted.  
 Enter a period to terminate list :

Enter : Yellowstone

Enter : .

=====

Fectuca Idahoensis has been added.

=====

The Life Form domain object is an instance of list-type domain object just as several of the other domains are. However, the method used to interact with the user for data input was different than the other instances of list-type domain objects. This is an example of an object not inheriting a method from its parent class of object.

The user also has options to view and/or modify existing species tuples. A method is attached to the appropriate relation object (a tuple is an instance of a relation) that allows the user to select a domain to modify. A message would be sent to the appropriate domain that would activate a modification method.

### 4.2.3. Query Operations

A complication involved with performing traditional query operations on the FIRESYS database was the heavy dependence on textual information. The above data dictionary does not include textual information, therefore the problem of making traditional type queries on textual information will not be discussed.

Another complication is the treatment of list-type domains. For example, a typical query might be: What are all the plant species in the Yellowstone National Park Administrative Unit? This would require the selection operation. The Administration Unit domain is an instance of the list-type domain type. A tuple (a particular plant species) might have several values for this domain. The complication is that a different form of comparison, a set membership, must be used to compare a single value with a list of values than would be used to compare two single values, which uses equality for comparison.

The selection method (operation) is embedded in the relation-type object; the class of objects that all relations are instances of. How should this method determine what form of comparison it should use? The answer that it needs to send a message to the domain requesting the appropriate equality operation. In the above example, the selection method sends a message to the Administrative Unit object requesting its equality method. The Administrative Unit object's equality method, inherited from the list-type domain object, searches a list of values for a given value.

The prototype implementation is capable of performing the selection operation and the previous mentioned query: what are all the plant species in the

Yellowstone National Park Administrative Unit? The selection operation can be performed on domains that are of positive-integer-type, atom-type and list-type. The former two domains use equality for comparison and the latter domain uses set membership for comparison. Although the query language has not been written, more complex queries can be made with a combination of operations. For example, consider the following query: What are all the plant species found in Montana or in Wyoming. The following steps can be taken:

1. Perform the selection operation on the plant species relation. The States domain is compared with Montana. The result is a temporary relation with domains for species and state.
2. Perform the selection operation on the plant species relation. The states domain is compared with Wyoming. The result is a second temporary relation with domains for species and state.
3. Perform the divide operation on the two relations, dividing by the states domain. The result is a relation that contains only one domain, that of species. This relation is displayed to the user.

The problem of using selection on list-type domains is moderately simple to solve. Similar situations in other operations become more complex. For example, the join operation concatenates tuples of separate relations if their common domains have equal values. In this case a list of values is being compared with a list of values. Should they be considered equal if each list contains one common value. Do the two lists have to share all the same values? If so, should the values be in the same order? These questions are answerable and the solution of how to

implement equality is similar to the solution used for the selection operation. Each domain is responsible for determination of equality, whether the equality is comparing a value with a list of values or comparing a list of values with a list of values.

#### 4.2.4. Changes by the Database Administrator

Modifications to the data model and specification of the system occurred constantly in FIRESYS. The most frequent changes in the data model was the addition, elimination or change in name of slots in a frame. The equivalent situation in this prototype is the modifications of domains within a relation.

Some tools have been implemented in this prototype to assist the database administrator in making such changes. These tools adhere to object-oriented principles and are embedded within objects. The following is an example of the interaction with the database administrator. A new domain is being added to an existing relation.



=====

Adding a domain for the relation : Plant Species Type

Please enter the new Domain Name :

References

Choose a domain type

- 1 : positive-integer-type-domain
- 2 : list-type
- 3 : atom-type

ENTER-OPTION: 2

=====

This interaction was the result of a message being sent to the plant species type relation. Existing tuples of this relation would currently have no values for the references domain. It would be the responsibility of the data entry people to add such information.

Another common change is in the specifications of the display of information. In the above example, the new references domain would be displayed as all other domains that are instances of the list-type domain. If there were a requirement that references are numbered when listed, this could be accomplished by attaching a display method to the new references domain object. This method would not allow the reference domain object to inherit the list-type domain object display method.

In this case, tools can be made to assist the database administrator, but the

database administrator would be responsible for writing some code. The following segment of code is the display method used by the list-type domain object:

```
(display-domain
 (lambda (tuple domain row column)
  (do
   ((d-list (get-object-value tuple domain) (cdr d-list))
    (currrow row (1+ currrow)))
   ((null d-list))
   (ptgoto currrow column)
   (printit (car d-list)))
  (length-common (get-object-value tuple domain))))
```

The following segment of code is the new display method to be used by the references domain object:

```
(display-domain
 (lambda (tuple domain row column)
  (do
   ((d-list (get-object-value tuple domain) (cdr d-list))
    (count 1 (1+ count))
    (currrow row (1+ currrow)))
   ((null d-list))
   (ptgoto currrow column)
   (printit
    (string-append
     (princ-to-string count) COLON (car d-list))))
  (length-common (get-object-value tuple domain))))
```

There are only two modifications made in the second algorithm. There is a counter variable, called count, the gets incremented for every value to be printed. This variable is attached, with a colon, to each value as it is printed.

#### 4.2.5. Creating a Data Dictionary

A data dictionary is an excellent tool for communication and is a valuable aid for all levels of users. Data entry people may need to refer to a data dictionary to ensure the accuracy of their work. The database administrator uses it to confirm consistency and integrity. Those that commission the development of the database refer to it while working with the developers.

The normal course of development is that software is designed from, among other things, a data dictionary. During the course of development requirements may change. A valuable software utility would be one that generates a data dictionary from the current database. This would provide all level of users with a current data dictionary thus avoiding the problem of people using outdated documentation.

The object-oriented scheme lends itself very easily to such a utility. A data dictionary utility has been written for the prototype implementation. The algorithm is as follows:

1. For every object that is an instance of the relation-type object:
  - a. Print a list containing all the primary keys and the name of the domain object that the primary key is an instance of.
  - b. Print a list containing all the other domains and the name of the domain object that the primary key is an instance of.

The data dictionary listings of plant species and wildlife species presented in this section were the output of the data dictionary utility. The requirements of a

data dictionary may vary. A similar scheme can be used to create a data dictionary that provides more information on domains.

#### 4.2.6. Security

There are many aspects to security of a database system. Some of these aspects, such as protection from fire or vandalism, have nothing to do with a software design methodology. Important forms of security that must be integrated in the design process are the ability to restrict access to portions of the data, control who can enter or modify data and control who may alter the definition of the data.

Some of these constraints might be easily handled by a system's operating system or other means. For example, the utility programs and files associated with the definitions of the data might be made accessible to the database administrator only. Data entry people might be required to enter a password that the database administrator would have control of.

The ability to protect portions of the data from a query user should be incorporated into the software design. An example of this in FIRESYS are the slots containing information on when the frame was last modified and by whom. This information is important to the data entry people and the database administrator but of no value to the end user.

The equivalent situation in this prototype implementation is a relation that has some domains that are not to be readable by the query users. There are two possible approaches to solve this problem. One approach is to make the relation object responsible for knowing which of its domains should be readable by the

query user. This could be implemented by attaching a property to each relation object containing the list of domains that are readable by the query user. When a tuple is to be displayed, only the domains on this list are displayed.

An alternate approach is to make each domain object responsible for knowing if its instances are to be displayed. When a relation's tuple is to be displayed, each domain would be checked to see if it is to be protected from the end user.

Each of these solutions have merit and perhaps a combination of these solutions would be optimal. Using the FIRESYS example, all of the slots that pertained to modification dates and entry person were uniformly inaccessible by the end user. In this prototype implementation, the domain objects could be responsible for protecting the data. If a domain object was to be unreadable in some relations and readable in others, then the particular relation could be responsible for controlling access.

## Chapter 5

### CONCLUSION

#### 5.1. Evaluation of Experiment

The objectives of this paper have been to explore the possibility of combining a moderately new approach to design with an established model used in databases. This paper has reviewed the important aspects of the object-oriented approach to design and has identified some of the key elements of the relational data model. Some of the considerations of using this design approach to implement a database using the relational data model have been discussed. The development of the FIRESYS project has been outlined. An attempt to implement portions of FIRESYS using an object-oriented relational data model has been made.

The overriding hypothesis of this paper has been that using the object-oriented approach in designing a relational data model would provide several advantages. The following sections will summarize the results of this experiment.

##### 5.1.1. The Semantic Gap

The development of a database produces two worlds to be modeled. The first world to be modeled is the representation of the domain field. The second world to be modeled is the collection of domain field entities as represented within the database. The variations between these two models is the semantic gap.

A traditional problem in software design is to encapsulate into the design of

a real world entity all the important characteristics of the real world entity. If some of the important characteristics of a real world entity are not encapsulated within the software representation of that entity, then those characteristics must be represented elsewhere in the software design. This situation poses several potential problems, particularly with respect to future modifications. Modifications to the model of a real world entity might result in a modification of a software component that is not encapsulated in the software representation of the real world entity. This may produce an undesirable effect in other components of the database.

The objects, or software representations of real world entities, in the prototype implementation did successively encapsulate all the essential characteristics of the real world entities. These characteristics fall into four categories:

1. Names of properties that contain real world data values. For example, the Abbreviation for the Fescue Idahoensis tuple has the value FEID.
2. Names of properties that contain internal system values. For example, relations and domains objects have a Print Name value that is used when being displayed on a terminal. Another example is that almost every object has a parent object, the object that it is an instance of.
3. Procedural information that performs operations that are invisible to the user. Examples of these are the selection, projection, join and division operations.

4. Procedural information that provides an interface with the interactive user. This includes interfaces to the query user, data entry user and the database administrator user.

Changes to properties that fall in each of these categories have been made in the prototype implementation. None of these changes have caused any effect in any other portion of the system. In this prototype, a change in a real world data value occurs in a tuple object. A tuple object has no instances and therefore the change cannot effect any other portion of the system.

Changes to properties that fall in the latter three categories do potentially cause the same change to be made in the instances of the object. This is due to the inheritance of an object-oriented system and is a desired effect. Aside from the inheritance mechanism, changes made to properties of the latter three characteristics produce no effects to other software components.

One important note is that the object-oriented relational data model discussed in this paper is very simple in many respects. One feature that it does not address is the possibility that an object is an instance of more than one object. This situation would add to the complexity in ensuring that a change in one component of a software system would not adversely effect another component.

This experiment has shown that using object-oriented techniques in implementing a relational data model database does greatly reduce the semantic gap. Due to this fact, a variety of modifications to the database can be made easily without effecting the reliability of the system.

A reduced semantic gap has other potential advantages. The process of



creating a software design from the specifications of a database system might become less time consuming. In fact, the process of creating a software design might become part of the process of creating specifications, thus creating an excellent rapid prototyping environment. This, however, cannot be confirmed from this experiment.

### 5.1.2. User Interface

There are three levels of users in a database: query user, data entry user and database administrator user. A complete, high level interface language has not been written for any of these levels of users in the prototype implementation. The primary interfaces implemented were encapsulated within various objects as methods. These methods invokes many screen handling functions that were borrowed from the FIRESYS input/output, or IO, package.

The ease of writing methods that interfaced with a user was due to the modularity of the object-oriented scheme and the modularity of the IO package. The methods did not need to know the details of the IO, only the names and purposes of a small number of interface functions.

An interesting point to consider is to extend the object-oriented design so that the terminal screen is considered an object and the existing interface functions would become methods. This would require that the standard message sending protocol be used to communicate with the terminal screen.

The task of writing a high level user interface language is not trivial. The properties of information hiding, encapsulation and message sending inherent in the the object-oriented approach would make the task somewhat easier.

### 5.1.3. Utility Programs

There was one utility program created in this prototype implementation, a data dictionary program. The high level algorithm for this utility is extremely simple. This is primarily due to the reduction of the semantic gap. All the information that needs to be known about an object is contained within that object.

There are a great number of utilities that can be written that would use a similar algorithm:

1. Visit every object.
  - a. If that object has a particular characteristic, do something.

Examples of such utilities are:

1. Find all the domain objects whose legal values are numerical values.
2. Find all domain objects that are instances of a particular object and do not inherit a particular method.
3. Find all relations with multiple primary keys.

It is obvious that efficiency of visiting every object is not good, but that is a worthwhile tradeoff for the simplicity of the algorithm. The object-oriented design provides an easy mechanism for the database administrator to extract information about the database. The simplicity of creating such tools can assist the database administrator in more effectively controlling all aspects of the database.

## 5.2. Concluding Remarks

It is the opinion of the author that there are two primary results of this study. First is that using object-oriented techniques in a relational data model permits the creation of a non domain specific database. This database can be used and re-used for a variety of domains. A more complete implementation than this experiment could provide a user interface language that is not domain specific.

The second result of this study is that the object-oriented scheme provides a natural way of developing tools for the database administrator to use in maintaining a database. These tools can be particularly valuable where the domain field is constantly evolving.

There are many aspects of databases that this study *did not address*. The speed of processing queries and the use of internal memory and external storage devices are important factors that must be considered in developing a database. The handling of more than one user making updates to the database is a tedious problem to solve. This paper makes no conclusions on these topics.

Some interesting questions arise from this study. If the object-oriented scheme easily accommodates changes in the domain field, in what other ways does the data model become extensible? For example, how easily can the data base be converted to a knowledge base to be used by an expert system?

In summary, it does appear that object-oriented techniques provide the same types of advantages in developing a database using the relational data model as they have in other software systems. In particular, the results of the prototype

experiment were comparable to the FIRESYS project. Using object-oriented techniques is a Relational Data Model in particular and in databases in general is an important topic for further study.

## Bibliography

- Dittrich, Klaus R. "Object-Oriented Database Systems: the Notion and the Issues" Proceedings of the 1986 International Workshop on Object-Oriented Database Systems, p. 2-4, 1986
- Greiner, Russell "RLL-1: A Representation Language Language." Stanford Heuristic Programming Project, HPP-80-9 (working Paper), Computer Science Department, Stanford University, Stanford California, October 1980
- Kersten, Martin L. and Schippers, Frans H. "Toward an Object-Centered Database Language." Proceedings of the 1986 International Workshop on Object-Oriented Database Systems, p. 104-112, 1986
- Martin, James Principles of Data-Base Management. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976
- Mctavish, Bruce J. "Analyzing a Frame Based Information System Using the Relational and Entity-Relationship Data Models." Thesis, University of Montana, Missoula, Montana, 1986
- Mitchell, James A. "Object-Oriented Programming, Lisp Flavors and their Application to a Fire Effects Information System." Thesis, University of Montana, Missoula, Montana, 1986
- Smith, Peter D. and Barnes, G. Michael Files and Data Bases. Addison Wesley, Reading, Massachusetts, 1987.