

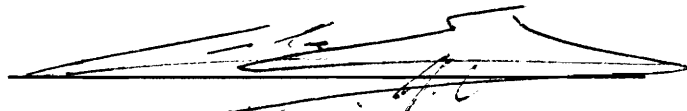
UNDERSTANDING
OBJECT-ORIENTED METHODOLOGY

by
Brian K. Roy

Project and Report submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTERS OF INFORMATION SYSTEMS

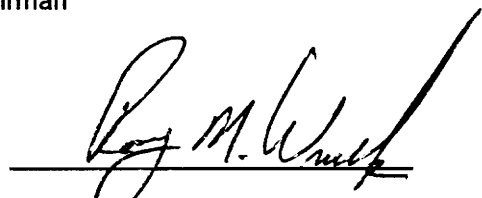
APPROVED:



C. Egyhazy, Chairman



W. Frakes



R. Wnek

December, 1993
Blacksburg, Virginia

C.2

LD
5655
V851
1993
R69
100

ACKNOWLEDGEMENTS

Special thanks to Dr. Csaba Egyhazy for his advise and encouragement, and to Mr. Mike Kinder for taking time to discuss his experiences and knowledge of object-oriented methodology. I am grateful to my employer, the Defense Mapping Agency, for consistently supporting higher education and for provided the challenging work assignments that have contributed to my understanding of this technology.

CONTENTS

List of Figures	v
1.0 Introduction	1
2.0 Project Objectives	1
3.0 Object-Oriented Modeling Concepts	2
3.1 The Object	
3.2 Class and Subclass Structures	
3.3 Inheritance	
3.4 Aggregation and Generalization	
3.5 Encapsulation	
3.6 Message Passing	
3.7 Object Instance	
3.8 Normalization	
4.0 Object-Oriented Data Models	10
4.1 Object Model	
4.2 Dynamic Model	
4.3 Functional Model	
5.0 Object-Oriented Programming Methods	20
5.1 Object-Oriented programming language	
5.2 Using a Non-Object-Oriented programming language	
6.0 Applying Object-Oriented Modeling for Two Types of Applications	32
6.1 Virginia Tech Course Schedule model	
6.2 Geographic Information System model	
7.0 Evaluating the Object Models	45
7.1 Virginia Tech Course Schedule model	
7.2 Geographic Information System model	
8.0 Object-Oriented Databases	55
8.1 Reasons for Object-Oriented databases	
8.2 Reasons for Extending Relational databases	
8.3 Data Definition Language	
8.4 Data Manipulation Language	
8.5 Data Control Language	

9.0 Lessons Learned - Identifying User Requirements by Prototyping 65

- 9.1 Prototyping an Object-Oriented System
- 9.2 Identifying the Objects of the Prototype

10.0 Management Issues Concerning Object-Oriented Model Development 71

- 10.1 Understanding the Goals
- 10.2 Understanding the Readiness (for Object-Oriented Methodology)
- 10.3 Managing the Implementation
- 10.4 Issues Concerning Software Reuse

11.0 Project Summary and Conclusions 77

12.0 References 88

Appendix A - Virginia Tech Object Models

- Course Schedule
- Scenario A
- Scenario B
- System Model

Appendix B - Geographic Information System Models

- Engineering Displays
- Scenario A
- Scenario B-1
- Scenario B-2
- System Model

FIGURES

Figure 3.1	Class and Subclasses	5
Figure 3.2	Inheritance of Attributes and Operations.....	6
Figure 3.3	Example of Aggregation.....	7
Figure 3.4	Encapsulation of Attributes and Operations.....	8
Figure 3.5	Example of a Class Instance.....	9
Figure 4.1	Example of a Class Structure.....	12
Figure 4.2	Example of a Link Attribute.....	13
Figure 4.3	Specifying Multiplicity in an Association.....	14
Figure 4.4	Example of a Role Name.....	14
Figure 4.5	Specifying Constraints in an Object Model.....	15
Figure 4.6	Example of a State Diagram.....	16
Figure 4.7	Constraints in a State Diagram.....	17
Figure 4.8	Specifying Activities in the State Diagram.....	17
Figure 4.9	Inheritance in the Dynamic Model.....	18
Figure 4.10	Concurrency in the Dynamic Model.....	19
Figure 4.11	Part of a Data Flow Diagram.....	20
Figure 5.1	Example of an Association.....	27
Figure 6.1	Major Classes to be Included in the Geographic Information System Model....	39
Figure 7.1	Example of a Message Notation.....	46
Figure 8.1	Object Identifiers Associated with Object Instances.....	60
Figure 8.2	Query Graph Showing Root of Query.....	64
Figure 9.1	Identifying Functions by Creating a User Interface.....	68
Figure 9.2	Establishing a Frame for the Object Model.....	69
Figure 9.3	Example of Tracking Object Links.....	69
Figure 10.1	Software Development Maturity Levels.....	73
Figure 10.2	Changes in Software Development Methodology.....	74
Figure 11.1	Example of a Public Interface Method.....	79
Figure 11.2	Instances of a Class Displayed in an Object Model.....	80
Figure 11.3	Example of a Message Notation.....	81
Figure 11.4	Application Program Accessing Attribute Data.....	84

1.0 Introduction

Recently, many conversations about object-oriented systems have ended with questions such as "What makes a system object-oriented?" Current work responsibilities at the Defense Mapping Agency now include maintaining and enhancing systems created from object-oriented analysis and design. These responsibilities have made learning about object-oriented concepts an important educational experience.

It is hoped that some of the benefits of this technology can be learned by studying reference material on this subject area and then performing an object-oriented analysis of two applications. The first application will be a course registration system which is representative of systems that are primarily transaction oriented. The second application will be a Geographic Information System which is representative of systems that are engineering and computationally intense.

Concepts associated with object-oriented methodology and implementation issues such as object-oriented programming and object-oriented databases will be presented. This information will assist with understanding the benefits and limitations of applying object-oriented methodology and will support object modeling decisions.

2.0 Project Objectives

The following summarizes the primary objectives of this project:

1. To understand the processes associated with object-oriented modeling and to perform an object-oriented analysis of two types of applications: a business application and an engineering application. A summary of the major decisions and lessons learned while creating these object models will be presented.
2. To evaluate and compare the two system object models and to determine the benefits and limitations of applying object-oriented methodology for these applications. The following information will be observed and used to evaluate and compare the object models:

a. The ability of object-oriented constructs and notations to represent the two system applications.

b. The dominant or unique object structures associated with each system application model.

c. The difficulties that may occur when modifying the system object model for the two applications.

3. To understand how an object-oriented system can be implemented using an object-oriented programming and database language. Comparisons between Object-Oriented databases and Relational databases will be presented.

4. To understand some management issues associated with implementing object-oriented methodology.

Note: The static model rather than the dynamic and functional models is the focus of this project because of its use to define the object database schema.

3.0 Object-Oriented Modeling Concepts

Traditional structured development concentrates on modulating the procedures of a program; these programs are allowed to share data and functions without restriction. This results in modules that are dependent on the behavior of other modules; a change to one module may require the dependent modules to be restructured. In an object-oriented system, focus has changed from modulating procedures to:

"combining data and associated procedures into independent objects [1]."

Object-oriented modeling focuses on understanding the problem by organizing related data and procedures into abstract data structures called "Classes". These classes are user defined data types that can be used to model (or define) more complex data structures.

Object-oriented modeling has been used to analyze, design, and implement complex information systems. A characteristic of the object-oriented modeling process is planning and learning up to the final stages of implementation. References have stated that developing an object-oriented system may require more time to model and design; however, the result is a system that may adapt easier to user requests for new capabilities or corrective maintenance -- even as the system is being developed.

At a high level, the major processes of object-oriented methodology are:

- a. building a model of the application domain
(describing the desired system without detail design decisions)
- b. adding detail design characteristics to the model
(deciding on data structures, hardware, etc.)
- c. implementing the design
(coding, etc.)

The details of object-oriented design are not included in this project; concentration will be on modeling the application domain and subsequently discussing object-oriented implementation techniques.

There are many variations of object-orientation methods. The characteristics that are common to each are:

a. Identity:

This characteristic means that each object in the model can be referenced uniquely.

b. Classification:

Classification involves defining attributes and operations and organizing them into classes having similar data structure and behavior.

c. Polymorphism:

Polymorphism exists when the behavior performed by an object depends on its associated object class.

d. Inheritance:

Inheritance is when attributes and operations are automatically obtained from higher level objects in its hierarchical classification structure.

The next sections will explain these and other object-oriented concepts in more detail.

3.1 The Object

The fundamental unit in object-oriented modeling is the "object". An object is something of significant importance to the application that has identity and is distinguishable among other objects.

An object can be one of two types:

- a. a template that defines the structure of an individual or group of instances
- b. actual instances that define specific individual occurrences

In general, the internal structure of an object should not be directly accessible; access to the values of an object is via the procedures that are part of the object. This concept has many advantages which will be discussed in more detail later. Thus, a software system is "object-oriented" if the system is:

"organized as a collection of discrete objects that are related by structure and behavior."

3.2 Class and Subclass Structures

Object-oriented data models organize data into categories called "Classes" which are composed of related data (attributes) and behavior (methods or procedures). Classes are considered objects. The data and procedures within a class are also objects and these objects may contain objects, and so on. A class is an abstraction, not an instance.

Classes may have subclasses that are specializations of a class and represent an "is-a" relationship with its higher level class (Superclass). A subclass is also an abstraction. Sub-

classes are often associated with inheritance (to be discussed later). Classes and subclasses represent data types that the analyst has selected for describing the problem/system. A Class or subclass is an abstraction (like a template) which will be used to create the structures that will hold the instance data of that class type. Following is an example of how a class and subclasses are represented in an object model:

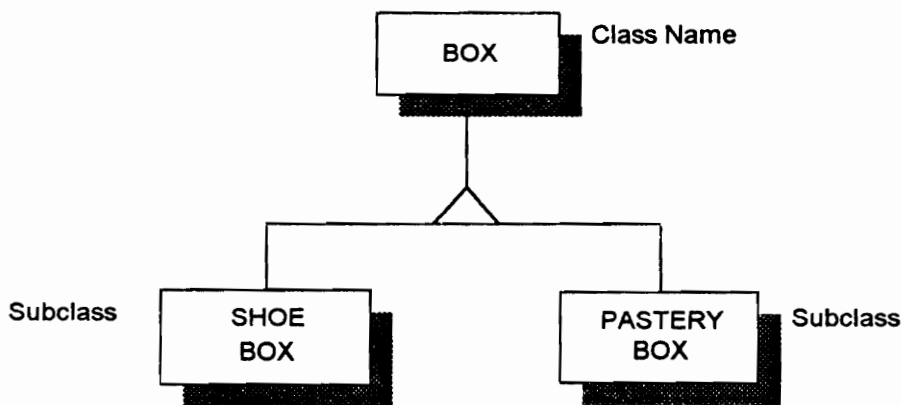


Figure 3.1 Class and Subclasses. "SHOE BOX" and "PASTERY BOX" are subclasses of the class "BOX".

Classes are important because they represent the data structure used to modulate related procedures and data. As stated earlier, traditional development methods focus on modulating procedures.

3.3 Inheritance

Inheritance is the sharing of attributes and operations with other objects in the hierarchical structure. Inheritance may reduce the duplication of information depicted in a conceptual model diagram. The potential reduction of duplicated data and procedures may improve the model appearance (less clutter) and improve the readability and content of the model. Similarly during

implementation (coding), duplication of code may be reduced and the reuse of code increased because of the "up-front" planning and the grouping of related data and procedures.

Inheritance is only applicable to specialization/generalization ("is-a") relationships (i.e., those having a class/subclass structure). Figure 3.2 shows an example of Inheritance.

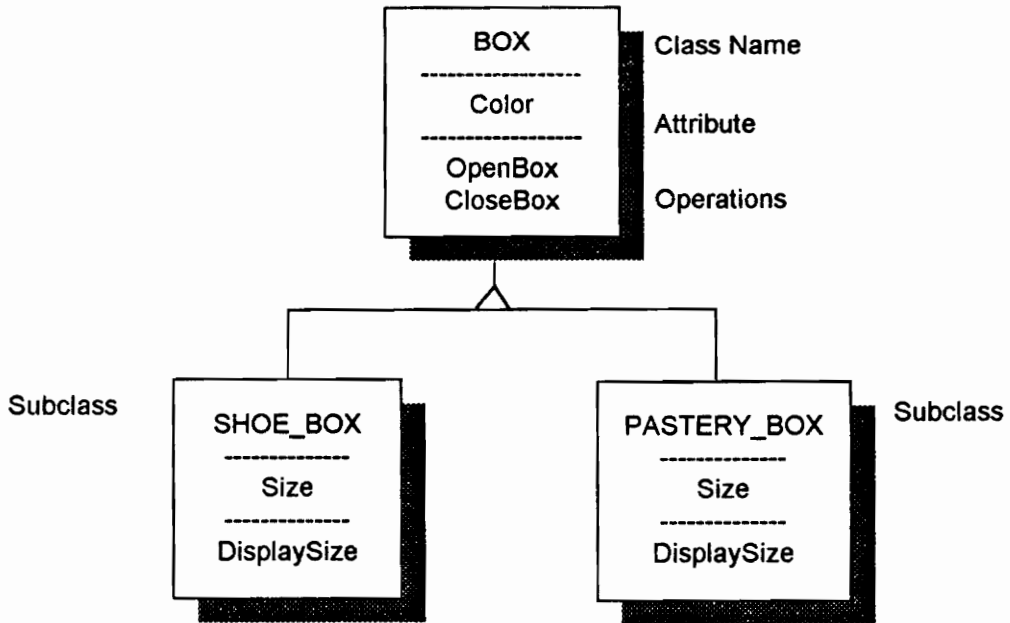


Figure 3.2 Inheritance of Attributes and Operations. The attributes and operations of the class "BOX" are inherited by the subclasses.

3.4 Aggregation and Generalization

Aggregation and Generalization are two modeling constructs that are used to represent certain types of relationships between classes. Their use will have an important influence on the design or the application.

3.4.1 Aggregation

Aggregation exists when an object is created from parts of other objects ("part-of" association). Objects having an aggregation association do not inherit data or procedures; however, through messages, they may access the inherited data and procedures of other object instances. Figure 3.3 shows an example of an aggregation association.

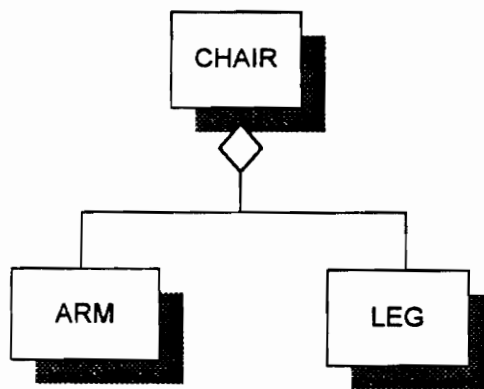


Figure 3.3 Example of Aggregation. Arm and Leg have a "part-of" relationship with Chair.

Aggregation associations are important, for example, during file design for grouping fields in a common file and for determining how objects will be clustered to provide efficient storage and retrieval of instance data.

3.4.2 Generalization

Generalization is the process of creating a data type that is more general so that it encompasses lower level data types. Inheritance may be applied when generalization exists within a model (to be described in more detail later). The class/subclass relationship shown earlier in Figure 3.1 exemplifies a generalization association.

3.5 Encapsulation

Encapsulation is a term for combining data with the procedures that act on this data. Because the data and the procedures are in a common place, we can easily require that access to an object's local data and procedures be through a public interface procedure. Restricting access through a public interface can protect the data of an object from being directly modified by other modules. Figure 3.4 shows an example of a public interface procedure within an object. In this example, all procedures are local except the ComputeTuition procedure.

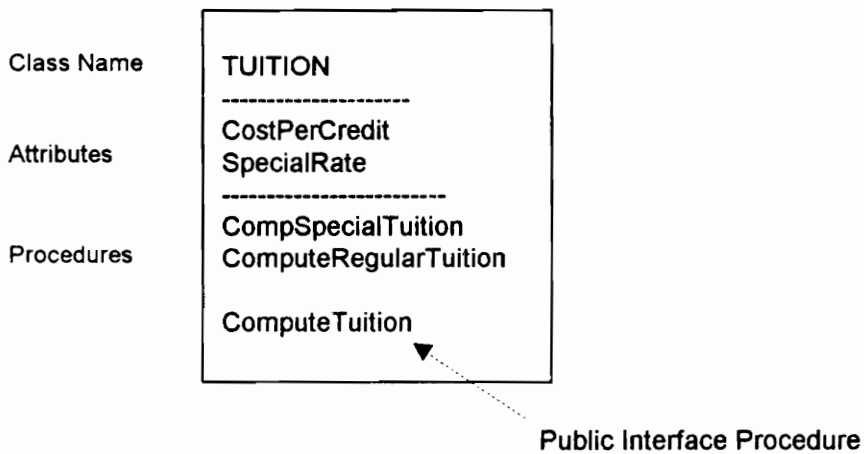


Figure 3.4 Encapsulation of Attributes and Operations. Access to local data and procedures should be through a public interface procedure.

Encapsulation promotes "Information Hiding" which helps control access to local attributes and procedures of an object.

3.6 Message Passing

A message is a "function look-up process" that is used for communicating requests between objects. The message statement that is processed contains the following information:

- a. The class name that contains the function.

b. The function within the class (noted above) that is to be executed.

c. Arguments (qualifiers) that provide specific information that the function will use to qualify its behavior.

Message protocols vary from one object-oriented language to another; however, the concept is the same. Message passing will be discussed in more detail in later sections.

3.7 Object Instance

An instance is an actual occurrence of an object. For example, an instance of the class "PERSON" is shown in Figure 3.5:

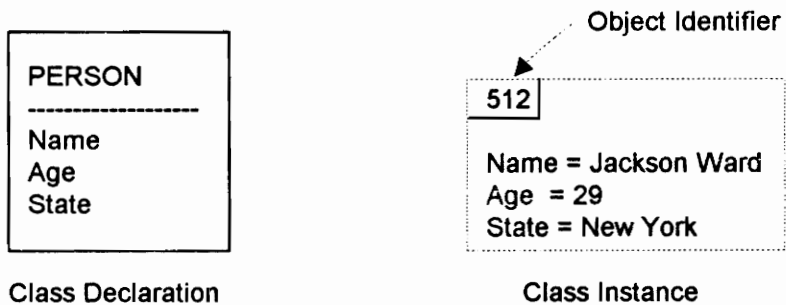


Figure 3.5 Example of a Class Instance.

In the above example the class "PERSON" has the attributes Name, Age, and State. An instance of this class is the actual information (values) of these attributes along with a unique object identifier that refers to the object.

Note : There are additional object-oriented concepts that are useful in object-oriented modeling. These will be discussed as they are encountered during the application modeling.

3.8 Normalization

Little documentation was available on how normalization is applied in an object-oriented environment. Wilkie [10] indicated that by factoring out data into class/subclass relationships, first normal form is obtained. Each attribute in first normal form can take only atomic values. Obtaining second and third normal form in object-oriented systems means ensuring that every attribute in an object depends only on the object ID.

Attributes of a class having primitive data structures (e.g., integer, char, etc.) should not have any internal information structure that may be divided. Separate attributes should be created when internal information structures exist .

Example: A house address should be divided into component parts so that each specific component can be accessible in current or future applications.

Normalization may require creating new classes to place related attributes. This will minimize object instance size, reduce null information values, and improve reusability of class objects.

4.0 Object-Oriented Data Models

Object-oriented modeling focuses on abstraction, concentrating on the important properties and hiding detail until the implementation stages of the project. The three object models that are needed in an object-oriented analysis are the:

- Object model
- Dynamic model
- Functional model

The Object, Dynamic, and Functional models are all necessary to completely describe a system. The Object model describes the basic units the of application domain. It is the first model to be developed because objects need to be identified before their dynamic and functional

characteristics can be described. The design is improved when the intertwining of these three models is minimized [1]. Following describes each of these models:

4.1 The Object Model

The Object model is a starting point to understanding the problem. This model documents the static structure of the problem domain. The Object model can help identify process flows that are necessary to create the Dynamic and Functional models. The Dynamic and Functional models will be used primarily to develop the application software.

4.1.1 Selecting Object Classes:

Objects classes should be selected to identify areas of the application that have clear boundaries. Objects must be distinguishable and help develop an understanding of the fundamental units of the application domain. The level of abstraction will determine the "boundaries" of the objects that will be implemented into the system.

4.1.2 Attributes and Operations of Objects

The attributes and operations of a class define the properties/characteristics of the class. Each object instance defined in the class will have values for these characteristics. For example, Name, Age, and Weight are attributes that could be used for the class Person. An operation is a procedure or function that is associated with the class; the operations of a class are the primary means of accessing the attribute values of the respective class. Following is an example of a Class and its associated attributes and operations:

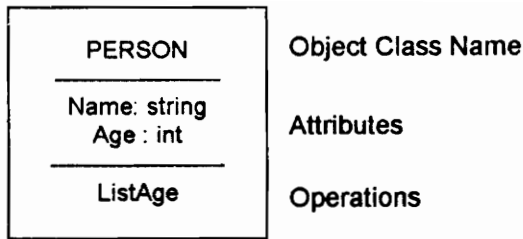


Figure 4.1 Example of a Class Structure.

Each object has an associated object ID that is unique among all objects in the application; however, the object ID is not listed in the model. Displaying the data structure of the attribute (e.g., name is of type string), is optional for an analysis model.

4.1.3. **Methods**

Methods are an implementation of an operation of a particular class. The difference between a method and the operation is subtle: the way a particular operation behaves may be based on the method used to execute the operation, such as the data type of the attributes sent to the operation. For example, if an attribute value received by an operation is of type "integer", the operation may execute a specific variation of the operation; if the attribute value is of type "character", a different variation of the operation may be performed.

When one object wants to execute a method of another object, messages are sent to the remote object. An example of a message format is:

(Selector Receiver [Arg1,Arg2,Arg3])

In the above message format, "Selector" represents the desired operation to execute and "Receiver" represents the object that will receive the message. The "Arg1, Arg2, Arg3" are arguments/qualifiers sent with the message. Because each object had a unique identifier, every defined object can be referenced uniquely by its identifier.

4.1.4 Object Associations

Associations are connections between objects of different classes. Links are connections between specific instances of different object classes. Link attributes are also used when attributes are associated with more than one object (see Figure 4.2).

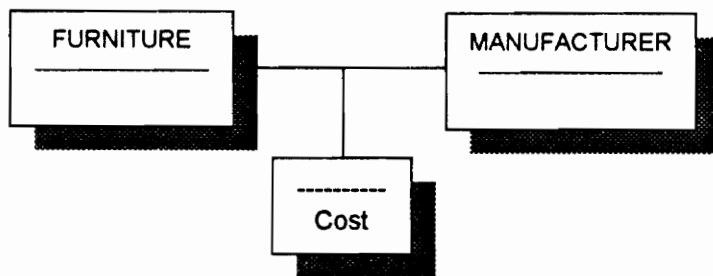


Figure 4.2 Example of a Link Attribute.

4.1.5 Multiplicity:

Multiplicity indicates how many instances of one class may be associated with a single instance of a different class. A notation that can be used to specify multiplicity follows:

- n+ indicates that there is "n" or more instances
- 2,4 indicates that there are 2 or 4 instances
- 3-5 indicates that there are 3 through 5 instances
- n:M indicates there is a "n" to many relationship

Figure 4.3 show an example of how multiplicity can be specified in an association. Some notations use the symbol "o" to indicate "many" if solid or "zero or many" if hollow.

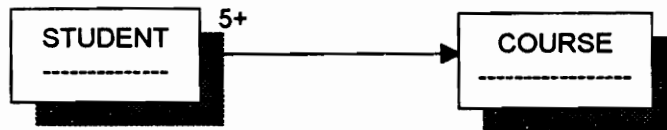


Figure 4.3 Specifying Multiplicity in an Association. The 5+ indicates 5 or more students must be in a particular course.

The most important multiplicity is between "one to many". Underestimating reduces flexibility and overestimating creates more overhead because more information is needed to distinguish between members of an object.

4.1.6 Role Names

Role names provide more specific information about an association. They are particularly useful when distinguishing between two or more objects within a particular class. In the following example, the role name "employee" specifies that association between the classes PERSON and COMPANY involves employees.

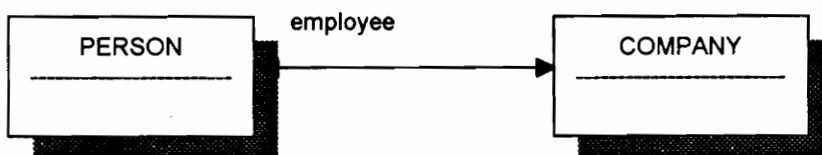


Figure 4.4 Example of a Role Name. The role name "employee" provides the model with more specific information about the association.

4.1.7 Constraints

Constraints identify restrictions or limitations on attributes, operations, or objects. Constraints provide the model with more specific information about the domain of the application. Figure 4.5 shows an example of how a constraint (city size) can be specified.

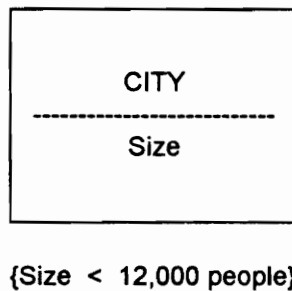


Figure 4.5 Specifying Constraints in an Object Model.
This constraint limits the city size to less than 12,000 people.

4.2 The Dynamic Model

The Dynamic model describes the parts of a system that may change with time or are dependent on the order that operations are executed. Events and states are important to this model. An event is an instantaneous occurrence that changes the state of a system. The state of a system is represented by the attribute values and on-going operations; a state remains the same between any two sequencing events.

For example: A system is in a particular state and an event occurs (such as a button being pushed) resulting in some action by the system. The system responds to the event by taking some action such as displaying a menu. The system is now in a different state.

4.2.1 State Diagram

The Dynamic model consists of state diagrams. The states are shown as nodes and the arcs between the nodes indicate transitions that result from a particular event. Figure 4.6 shows an example of a state diagram. An event "Button Pressed" causes a window to open; an event "Button Released" causes the window to close. Window Closed and Window Open are nodes that represent the state of the system (or part of the system).

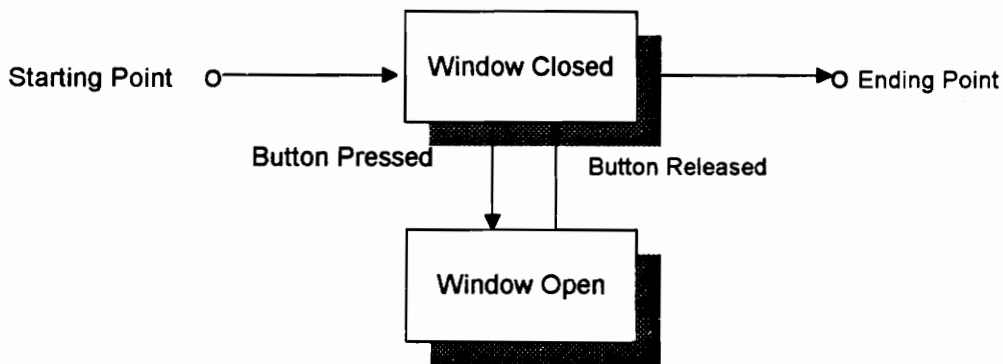


Figure 4.6 Example of a State Diagram.

4.2.2 Conditions

The State diagram can incorporate conditions into the model such as restricting a value to fall within a particular range. These conditions can be annotated by placing them between brackets. The example, Figure 4.7 shows a condition that indicates that if the search retrieves less than 200 records the record search is accepted. Once the condition is met, the state changes.

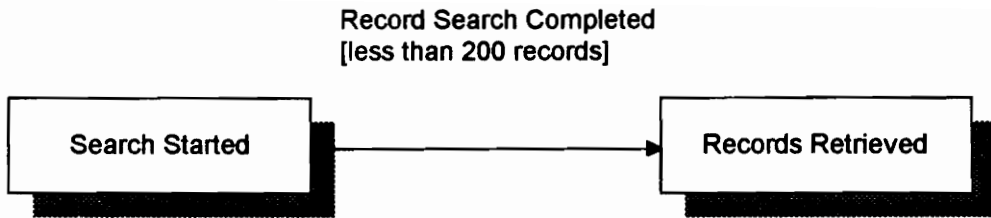


Figure 4.7 Constraints in a State Diagram.

4.2.3 Activities

When an event occurs, such as being told to "stop the ship", an activity (operation) may be initiated such as throwing an anchor overboard. This information can be included in the State diagram by placing a short description of the activity next to the event description. The "/" character separates the event from the activity.

If an activity occurs while in a state, the activity is placed after a "do:" statement within the respective node. In the following example (Figure 4.8), the activity: (yell "anchor overboard") occurs while in the state "Anchor Overboard".

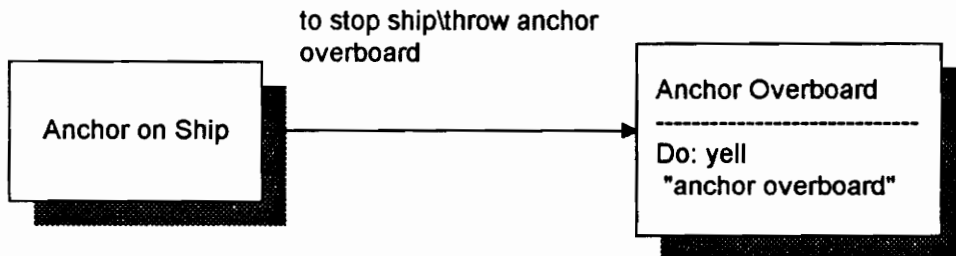


Figure 4.8 Specifying Activities in the State Diagram.

The Dynamic model is tied in with the Functional model by what is described as "Actions" which correspond to functions.

4.2.4 Inheritance

States and events can also be inherited to substates and subevents similarly to the way objects inherit attributes and operations from their superclass objects. The following section provides an example of how inheritance may be incorporated into the Dynamic model.

4.2.5 State and Event Generalization:

Generalization describes the "or" type association which is synonymous with an "is-a" association. Events and states can also have a generalization structure; thus, inheritance may be incorporated into the model. Figure 4.9 shows how the state "Current Menu" is inherited to the substates.

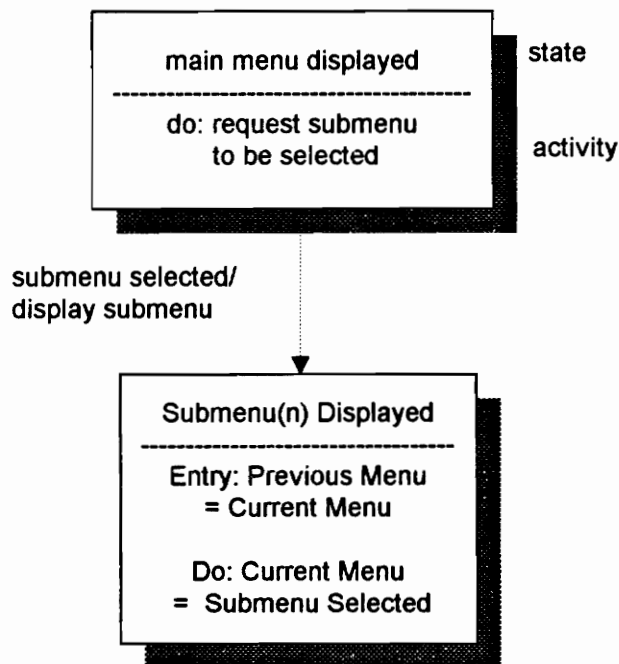


Figure 4.9 Inheritance in the Dynamic Model. The state "Current Menu" is inherited.

Note: Activities not associated with the generalization hierarchy will not inherit the state "Current Menu".

4.2.6 Concurrency

Events, activities, and substates can occur concurrently within a system. A method of representing concurrency in a Dynamic model is to draw arrows to and from the concurrent activities initiated within a node. Figure 4.10 shows how concurrent activities can be represented in a Dynamic model.

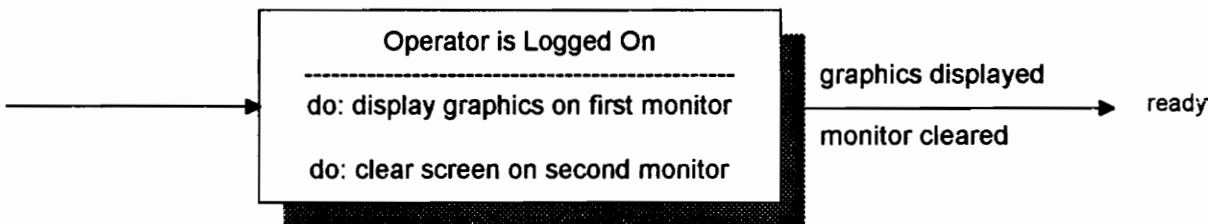


Figure 4.10 Concurrency in the Dynamic Model. The two monitor activities occur simultaneously.

4.3 The Functional Model

The Functional model will model how values are transferred from one process to another and will describe these functions. The Functional model displays functions as nodes in a data flow diagram (and arcs as data flows). This model, like the Object and Dynamic models, is an abstraction; therefore, the details of how to implement these functions are not incorporated into the model.

The Functional model is related to other models in that functions are invoked as actions in the Dynamic model and processes are implemented as methods of operations.

4.3.1 Data Flow Diagram

The data flow diagram shows inputs, outputs, and data stores. The processes are depicted as nodes on the diagram; these nodes represent a transformation of data. Figure 4.11 shows a part of a functional model. The data "map coordinates" and "feature coordinates" are input to the function "Display Data" which is then executed. Once the data has been displayed the message "display menu" is sent to the appropriate object.

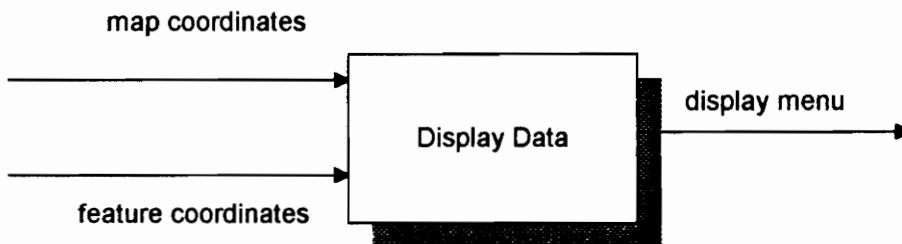


Figure 4.11 Part of a Data Flow Diagram.

5.0 Object-Oriented Programming Methods

The steps involved to implement an object-oriented design are, for the most part, the same using an object-oriented or non-object-oriented language; however, an object-oriented language will automatically create/map the object data structures. In the object-oriented language C++, the object-oriented constructs are preprocessed to convert the C++ constructs into C language statements.

The next sections describe, using basic examples, how object-oriented and non-object oriented languages are used to implement an object-oriented design.

5.1 Using an object-oriented language (C++) to implement an Object model

5.1.1 Implementing a Class:

The declaration of a class consists of two parts: a class head and class body.

a. **Class Head:** The class head is declared in C++ by using the keyword "class" followed by the name of the class.

Example: `class Company`

b. **Class Body:** The body contains the commands that declare the operations and attributes of the class. Note that a class is not an instance but like a template that is used to create an instance structure. Following explains and demonstrates how access to attributes and procedures can be declared in an object-oriented language (C++):

Three levels of access are provided by the C++ language:

1. **Public :** The attributes or procedures in this section can be accessed by any part of the respective application.
2. **Protected:** The attributes or procedures in this section can be accessed only by the class containing the declaration and its subclasses.
3. **Private:** The commands in this section can be accessed only by the class containing the definition.

Example 5.1 (on the following page) demonstrates how these levels of access are used in a class declaration:

```
/* Example */
```

```
class Company
```

```

/* body starts here */
{
    public:
        Company(int Init_totalEmployees)
        float computeHours( );

    protected:
        float hoursPerWeek;
        int totalEmployees;

    private:
        int i,j;
}

```

Example 5.1 Class declaration and three levels of access.

Note: An additional access type not shown above is a special type of access called a "friend" which allows an outside class object direct access to the data and procedures of another class. Direct access to the internal structure of an object violates encapsulation.

The above declaration exemplifies "Information Hiding" or controlling access to data and procedures. Two major benefits of information hiding are:

- Information hiding provides protection and controlled access to a program's data structures.
- Changes may be made to an object's data structure without requiring changes to the programs that access the data; therefore, program maintenance may be easier.

Deleting or changing a public method or interface to this method may cause difficulty because many existing programs may potentially use the method to access the class object data. However, modifying a private method can be performed without requiring external programs to be changed. External programs may be affected by the modification, but these programs should still compile and process. Private methods provide the functionality of the object and are initiated through the public interfaces.

5.1.2 Implementing a Class Instance

The following sections explain how a class instance can be initiated in an object-oriented language.

a. Defining a class operation:

This function includes the statements to execute the method "computeHours" which is part of the class "Company". The function may be placed within the file containing the main program or in a separate data file:

```
Company::computeHours(int totalEmployees)
{
    hoursPerWeek = 40;
    totalHours = totalEmployees * hoursPerWeek;
    return totalHours;
}
```

b. Executing the function:

The function "computeHours" can be executed within a program as shown below. First an instance of the class company is created which also initializes the attribute "totalEmployees", then the "computeHours" function is called.

```
main( )
{

    /* create an instance of the class "Company" */

    Company MyCompany( );

    /* compute the total hours for the instance "MyCompany" */

    total_hours = myCompany::computeHours(totalEmployees)
}
```

5.1.3 Implementing Inheritance

The attributes and operations that a subclass inherits are those declared as public or protected; private attributes or operations are not inherited. Example 5.2 shows how the class "DottedLine" is declared to inherit attributes from the "Line" class.

Note: The immediate superclass(es) of a particular subclass are listed in the same statement that defines the subclass name. Name conflicts resulting from multiple inheritance will be resolved based on the order that the superclasses are listed in this statement.

```
/* first declare the Line class */
```

```
class Line
{
    protected:
        int LineWidth;
        char LineType;
        int Start_X; Start_Y;    // declare beginning of line attributes
        int End_X; End_Y;      // declare end of line attributes
        void draw_line( );      // declare procedure to draw line

    public:
        int Get_X_Y( );        // method that will get the X,Y coordinates
        char Get_LineWidth( ); // method that reads in the linewidth
        char Get_LineType( ); // method that reads in the linetype
        void DrawLine( );     // function that will call the Get_X_Y Coordinates
                               // method and then call the private draw_line function
}
```

```
/* now declare the "DottedLine" subclass. The class "Line" is declared a superclass of the
Dottedline subclass and will automatically inherit the protected and public attributes and
operations of the "Line" class */
```

```

class DottedLine :: public Line // declare the class "Line" a superclass of DottedLine
{
protected:
    int dot_spacing;           // declare attribute for dot spacing
    void draw_line( );        // modified drawline operation

public:
    int Get_Dot_Spacing( );   // method that reads in dot spacing
    void DrawLine( );        // public interface method
}

```

Example 5.2 Implementing Inheritance.

In the above example, the subclass "DottedLine" inherits the attributes and operations defined in the "Line" superclass. Note that the subclass declaration contains the reference to its superclass. The superclass, in this example, has no reference to the subclass in its declaration.

5.1.4 Implementing Aggregation

Example 5.3 demonstrates how C++ can be used to implement an aggregation association. The two classes used in this example are "Line" and "Point"; the Point class has a "part-of" relationship with the Line class.

/* First declare the "Point" class: */

```

class Point
{
private:
    int x,y; // these attributes are not accessible to other classes
    int a,b; // " " " " " " " " " "

public:
    Point( ) // "inline" function to construct/initialize point to zero
        {x = 0; y = 0;} // (if no arguments are provided)

    Point ( int a, int b) // "inline" function to construct/initilize point to a or b
        {x = a; y = b;} // (if arguments are provided)

    void Move (int a, int b) // "inline" method to move point to value of a and b
        { x = a; y = b;}

    int Get_X_Y( ); // method to get the X and Y coordinates
}

```

```

/* Now declare the "Line" class */

class Line
{
    protected:
        Point p1,p2; // p1 and p2 are declared as variables of type Point

    public:
        Line (Point Start_pt, Point End_pt);
        void drawLine (Point p1, Point p2);
}

```

Example 5.3 Implementing Aggregation.

Examples 5.2 and 5.3 showed how inheritance and aggregation structures can be defined in an object-oriented language. Following is an example of how these definitions can be used to implement a function:

```

main( )
{
    Point pt1(4,2)           // define an instance of point - called pt1
    Point pt2(8,8);         // define another instance of point - called pt2
    Line Myline (pt1, pt2); // define an instance of the Line class
    Myline.drawline( );     // execute the drawline function
}

```

Some important points are demonstrated in the above main program:

1. The first three lines define instances by using previously declared class structures.
2. The last line "Myline.drawline ()" does not require arguments. The reasons are:
 - a. When the instance "Myline" was created, the two points (pt1 and pt2), were initialized.

- b. The "drawline" function has automatic access to the internal instance variables of its associated class, even if the class function is physically located outside of the class.

5.3 Implementing Association

Basic association is used when a class object needs to reference data and methods of a different class object. Some object-oriented systems require that the methods of the associated class be explicitly stated in the calling class declaration. For example, Intergraph's object-manager system requires that a function be declared as "imported" in the calling class. In C++, public associations do not need to be explicitly declared in the calling class. However, to access an internal function of an associated class, the respective method must be declared as a "friend" function within the calling class. Declaring a private or protected method as "friend" violates encapsulation of those methods.

In Figure 5.1, the attribute "CompanyName" of the Employee class is associated with the attribute "Name" of the Company class.

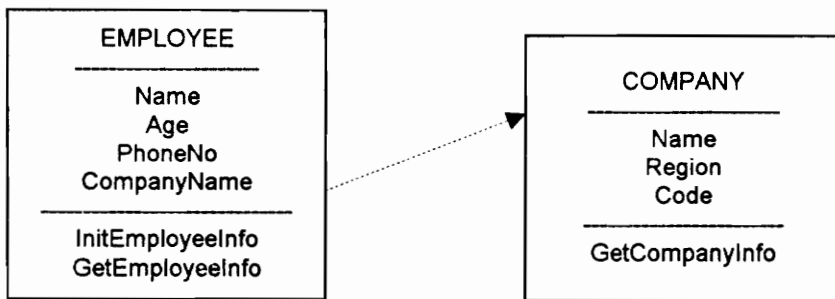


Figure 5.1 Example of an Association.

The C++ statements below show how this association can be implemented. The "friend" statement allows the Employee class to directly reference the "Name" attribute in the Company

class. Note: Allowing direct access violates encapsulation; however, this violation is controlled by restricting access to specific classes.

```
class Employee : Public Company;  
{  
protected:  
    char * Name;  
    int Age;  
    char * CompanyName; // reference to the company name  
public:  
    InitEmployeeInfo;  
    GetEmployeeInfo;  
}  
  
class Company  
{  
protected:  
    char[30] Name;  
    char[30] Region;  
    int Code;  
public  
    GetCompanyInfo;
```

5.2 Using a Non-Object Oriented language (C) to implement an Object-Model

Implementing an object-oriented design using a non-object-oriented language is similar in many ways to that of an object-oriented language. Object-oriented languages typically have a preprocessor that converts object-oriented statements into the basic programming language. For example: C++ uses a preprocessor to convert C++ object-oriented statements to straight C code.

5.2.1 Implementing a Class

To declare a class, the attributes and the references to functions are mapped to a record. In the language C, this is performed as shown in the following example:

```
struct Button
{
    int SizeX;
    int SizeY;
    char name[30];
    void (* funct1) ( );
    void (* funct2) ( );
}
```

The above example declares the structure of the class. SizeX, SizeY, and name[30] are attributes, and funct1 and funct2 are references to the functions.

To define an instance of the class, the specific name of instance is required:

```
struct Button * button_1;
```

In this above example, "button_1" is defined as a reference to an instance of the class "Button". Storage is created for the reference (pointer). To create the storage for the structure and to assign it to the "button_1" pointer, the following statements are used:

```
button_1 = ( struct buf *) malloc(sizeof(struct buf));
```

The attributes or operations can be accessed by referencing the desired structure element. For example, the value of the attribute "SizeX" can be accessed by referring to the respective record field as shown below:

```
button_1 ->SizeX;
```

5.2.2 Implementing Inheritance

Inheritance in "C" can be implemented by first declaring the structure of the superclass and then embedding this declaration in the subclass declaration. Following is an example of how this can be performed.

```
/* first declare the structure of the superclass "ShoeBox" */
```

```
struct ShoeBox  
{  
    float Length;  
    float Width;  
    float Height;  
}
```

```
/* now declare the subclass "DressShoeBox" which will inherit the attributes from the  
ShoeBox superclass structure. An additional attribute "color" has been included */
```

```
struct DressShoeBox  
{  
    struct ShoeBox;  
    color; // additional attribute – attributes from ShoeBox are inherited  
}
```

To define a class/subclass structure that uses inheritance, the following can be performed:

```
/* declare "myShoeBox" to be a pointer to a structure of type "DressShoeBox".
```

```
struct DressShoeBox * myShoeBox;
```

```
/* allocate space for a new instance and assign the address of the beginning of this  
space to the pointer variable "myShoeBox" */
```

```
myShoeBox = (struct DressShoeBox *) malloc (sizeof (struct DressShoeBox));
```

To access attributes or operations in a function, the following can be performed:

```
/* this initializes the color of the instance to blue */
```

```
myShoeBox->color = blue ;
```

5.2.3 Implementing Association

To implement an association between two objects (e.g., Object1 and Object2), a field within the Object1 structure can be used to refer to the Object2 structure, and a field within the Object2 structure can be used to refer to the Object1 structure. In an aggregation association, the structure of a single object instance may contain many fields that reference different object structures types.

/ Following is the declaration of the "Arm" and "Leg" class structure. Each contains three fields, (two attributes and one function) */*

```
struct Arm
{
float armSize;
int armCount;
int ( * count_arms) ( );
}
```

```
struct Leg
{
float legSize;
int legCount;
int ( * count_legs) ( );
}
```

/ Following declares the "Chair" class structure which is an aggregate of the Arm and Leg object structures. */*

```
struct Chair
{
struct * Arm;
struct * Leg;
}
```

To define (create an instance of) an association, a pointer in each object instance must be created and space allocated for each instance. Following is an example:

```
/* define a pointer to a structure of type "Chair" */
```

```
struct Chair * new_chair;
```

```
/* Following allocates space for a new instance and assigns the address of the beginning of this space to the pointer variable "new_chair" */
```

```
new_chair = (struct Chair * ) malloc (sizeof (struct Chair));
```

```
/* Following is how the attributes or operations of an aggregate structure can be referenced:  
This statement initializes the size of an instance of armLength */
```

```
new_chair->Arm->armSize = 25.0 ;
```

6.0 Applying Object-Oriented Modeling for Two Types of Applications

Identifying the two applications:

Two applications will be used to demonstrate and analyze object-oriented modeling:

1. Virginia Tech Course Registration System:

The Virginia Tech course registration system was selected because the application represents a typical business/data processing application. One of the primary questions of interest is: "Can this type of application be modeled using object-oriented methodology?"

2. Geographic Information System (Surveying and Civil Engineering)

The Geographic Information System application was chosen because the application is engineering and graphics related which is significantly different from the business oriented class schedule application.

Processes to complete the models:

Following are the processes that will be used to complete the two models:

1. Select candidate object class names.
2. Identify the attributes and operations for each class.
3. Arrange objects into a class structure (i.e., generalization hierarchy, aggregation, association, and composite structures).
4. Normalize the model.
5. Isolate parts of the system that may change.
6. Iterate to correct or refine the model.

6.1 Virginia Tech Class Schedule

Scope of Model:

The scope of the model is to create an object-oriented model that can be used to subsequently design and implement a Virginia Tech course schedule application and database. The model must support queries that will provide the information to support the registration processes. The application and database will provide information associated with courses, students, instructors, and other administrative activities related to a particular semester.

Creating the Object Model:

1. Identifying the Class Objects

After studying the course schedule and asking some questions at the registrar's desk, the following class objects were identified:

- Campus
- College
- Department
- Instructor
- Student
- Course
- Address & Phone
- Payment Plan
- Important Dates
- General Info.
- Tuition_Fee_Refund
- Course Limits

The class object names were similar to the headings used on the Virginia Tech course schedule. The Virginia Tech course schedule printout is located in Appendix "A".

2. Identifying Attributes and Operations:

a. Selecting Attributes:

The attributes for most of the class objects were obtained directly from an existing course schedule. The student and instructor attributes were selected based on registration data that is likely to be entered or queried. The attributes of a class are the internal data that will be used by the functions associated with the class. This implies that attributes should be selected with operations in mind.

Attributes selected for the Virginia Tech course schedule are shown in the model diagrams in Appendix "A".

b. Selecting Operations:

The operations initially identified were mostly generic (e.g., print, display, compute, etc.). To identify more specific operations, as well as attributes, a couple of scenarios were developed and analyzed. The class information, identified as a result of creating and analyzing these scenario models, was transferred to the overall Virginia Tech "System Model". Therefore, classes involved in the scenarios will generally contain more detailed information than those which were not involved in the scenarios. Following describes the scenarios created for the Virginia Tech system:

Scenario (A):

A student walks into the desk and asks if a course is still available. If the course is still available, the student asks how many vacancies exist (for estimating how soon he must decide to enroll in the course).

Processes:

- a. Operator logs into terminal and main menu appears.
- b. Operator selects "Course Information" from the main menu.
- c. Operator enters the course number and marks the fields that correspond to the information to be queried.
- d. A query is sent out to the database to retrieve the information and display it to the registrar.

The Virginia Tech Scenario (A) model is displayed in Appendix "A".

Comments on Scenario Contribution:

This scenario assisted with understanding and modeling the following: a) the user interface, b) data that may be incorporated into a database, and c) assignment of procedures to objects. Modeling this scenario helped recognize that many procedures can be assigned as "local" to the object. These local procedures are accessible via "public" interface procedures. This scenario helped to identify some new classes, such as those associated with the user interface, and provided detailed process flow information.

Scenario (B):

All instructors need a list of students registered in their respective classes. The instructor also would like the classroom number, and the course times.

Processes:

- a. Operator logs into terminal and menu is initialized.
- b. Operator selects Instructor_Info_Menu.
- c. Operator selects an option that requests a list of students for each course the instructor(s) teaches.
- d. A query is performed to obtain the desired information.
- e. The information is displayed or printed.

The Virginia Tech Scenario (B) model is displayed in Appendix "A".

Comments on Scenario Contributions:

This scenario provided more information on how the database objects would interact. The message interaction between the Student and Instructor classes can be done within the database if the database management system has this capability. This is an important capability of object-oriented databases which will be discussed later in more detail.

The model also refined the attributes and methods that were initially identified. The scenario provided a structured way to identify the various processes involved, and the specific attributes and methods needed to perform these functions.

3. Arrange Objects into a Class Structure:

The objects were arranged into a hierarchy. This initially caused some difficulty because some aggregation relationships were misrepresented as class/subclass relationships. For example, the association between Campus, College, and Department classes was initially misrepresented as a class/subclass hierarchy; however, the association is strictly aggregation. A College is "part of" of Campus but "is not" a Campus; similarly, a Department is "part of" a College but "is not" a College. Therefore, inheritance does not apply to the Campus, College, and Department class structures.

An aggregation association was initially used to represent the associations between Department and the Course, Instructor, and Student classes. As will be discussed in the conclusions, an aggregation association between these classes is questionable.

Class associations were arranged based on the requirements of one class to access methods and data of another class. Some classes use the methods and data of other classes to perform its functions. The accurate arrangement of object classes is important for subsequent design decisions. For example, objects that are closely associated may be grouped for efficient storage and retrieval.

4. Normalize the Model

Normalization was implemented as the model was being developed. In some cases additional classes were created to apply these rules. For example, the Address_&_Phone class was created to minimize the duplication of address information and to place address components into separate attributes.

Normalization was not applied in cases where the cost of efficiency appeared to exceed that of the cost of normalization. The department head information of the Department class, shown in the Virginia Tech "System Model" in Appendix A, exemplifies this point. The department head information may have been used as a basis for creating a new class such as "Department Heads"; however, this attribute data was retained in the Department class for the following reasons:

- the department head attribute data is closely tied to the Department class
- the department head attribute data will frequently be used in conjunction with the Department class
- little is expected to be gained by creating a new class

An important observation from this model is:

"Understanding the application is important for deciding how and when to apply Normalization rules. "

5. Isolate Parts of the System that may Change

Parts of a system can be made more independent by defining class structure(s) that connect different parts so that each part is protection from changes to their associated parts. This may be important, for example, when "public interfaces", hardware devices, and other components of a system are expected to change.

Isolating parts of a system was not a primary concern in this project; however, classes to support isolation could be placed in the Virginia Tech model at the following locations:

- between the user interface and other object classes, and
- between the application objects and the database objects

Most modern systems have structures to provide portability and protection against component modifications.

6. Iterate to Correct and Refine the Model

To complete these models, many iterations were required. In section 9.0, details will be provided that explain a technique called "Rapid Prototyping" which uses an interactive method to developing an object-oriented system. Developing an object-oriented system through a series of iterations is a method typically used when system requirements are not well defined.

6.2 Geographic Information System Model

Scope of the model:

This model represents a land information system that a surveying and civil engineering consulting firm would benefit from. This land information system could be used for engineering tasks such as designing subdivisions, performing property surveys, designing sewer or water lines, and locating powerlines. Figure 6.1 shows the primary types of information that will be included in the model:

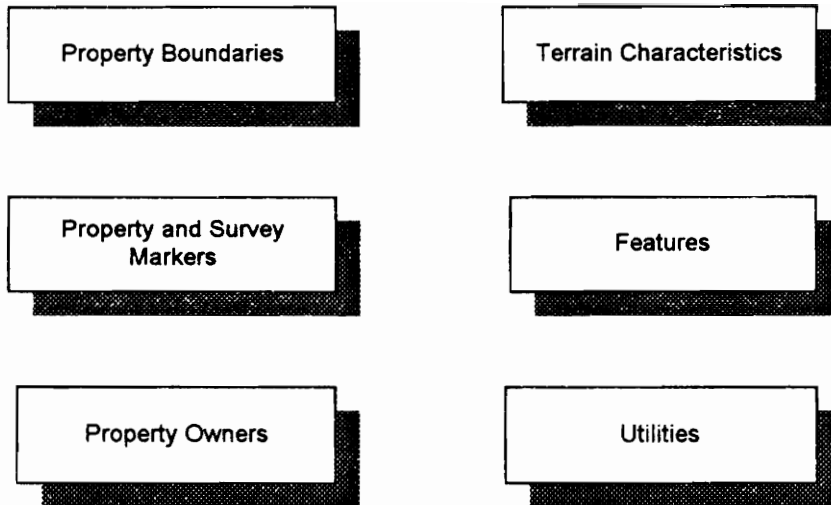


Figure 6.1 Major Classes to be Included in the Geographic Information System Model.

Initial Processing Capabilities:

After drawing some of the graphic displays that a surveying and civil engineering firm is expected to need (see Engineering Displays, Appendix "B"), the following capabilities were selected to be included in the GIS model:

- a. The engineer should be able to display various layers of graphical information. For example, different types of utilities could be displayed individually or combined.
- b. Previously completed maps are to be revisable without restarting from scratch.
- c. The system should be able to display the owner of a piece of property and its adjacent owners. Lot identification information should also be displayed.
- d. Contours are to be computed and saved for instantaneous redisplay.
- e. Engineers/surveyors often are required to verify that a house is a minimum distance from the property lines, flood plane, etc.; therefore, the system is to compute these offsets.

Creating the Object-Model:

1. Identifying Class Objects:

Following are the class objects that were initially identified:

- WaterLine ◦ PowerLine ◦ SewerLine ◦ Map ◦ Owner
- Utility ◦ Terrain_Item ◦ WaterBody ◦ Elevation ◦ Feature
- Marker ◦ Road ◦ Property ◦ Structure ◦ Wetland
- Treeline ◦ SoilType

Many of these class objects have specialization or generalization relationships with each other that demonstrate a natural hierarchy structure (e.g., the class/subclass relation of the Utility class with the WaterLine, SewerLine, and PowerLine subclasses).

The classes were selected based on the level of abstraction required for the application. Many additional classes could have been incorporated into this model to provide more detailed information, as would be expected in a commercial GIS system; however, to reduce the scale of the project, only a subset was chosen.

2. Identify Attributes and Operations

a. Selecting Attributes:

The attributes selected included many of the display characteristics that the particular feature would exhibit such as Color, LineWidth, LineType, MapScale, etc. As with the Virginia Tech model, attributes were selected based on their association with the functionality of the class.

Attributes selected for this GIS model are shown in the System Model diagram in Appendix "B". The attributes were refined when specific process scenarios were modeled. These scenarios will be discussed in the following sections.

b. Selecting Operations:

To decide what operations were important for the application, the Engineering Displays noted earlier were used to identify an initial set of operations for each class. In addition, two specific scenarios were developed and analyzed. The GIS "System Model" was then updated to include changes resulting from the scenarios. Following are the scenarios created for the Geographic Information System:

Scenario (A):

An engineer would like to display the water line utilities on a piece of property so that the field crew does not accidentally damage these utilities when staking out the property corners and road location for the subdivision. The information must be printed.

Processes:

1. Engineer logs on to GIS system and GUI screen appears
2. Engineer selects the "Utilities" menu button
3. Engineer selects the Water Line utility button
4. Engineer is prompted for the area of interest
5. Engineer enters the Parcel ID and the utility information is searched.
6. Engineer selects "Print Information"
7. Utility information is printed.

The GIS Scenario (A) model is displayed in Appendix "B".

Comments on Scenario Contributions:

The scenario was an important contribution to understanding and developing the user interface and understanding the process flows of the application. The user interface was the primary method of navigating through various optional functions.

Inheritance was applied for many of the classes associated with this scenario. The scenarios help to identify which attributes and operations should be inherited. For example, utility routines, such as "Get Area of Interest" and "Search Area of Interest" were inherited to each utility subclass. Because this was the first scenario modeled for the GIS application, the scenario helped considerably with understanding the user interface and the process flows, and with refining the class attributes and operations. The Property and Water_Line classes were identified as database object classes.

Scenario (B-1):

A land surveyor/engineer must return to a piece of property to measure the position of a newly built house on the respective property. To measure the location of the house with respect to the property lines, the surveyor/engineer needs to know the location of the property corners and previously measured survey working points. A printout is needed so that this information can be used in the field.

Processes:

Surveyor Prepares for a Field Survey:

1. Surveyor logs into GIS and GUI is displayed.
2. Surveyor selects "Property Information".
3. Surveyor enters "Parcel ID" if known (if not, the general location of the property).
4. A query for the marker information is performed.
5. A printout of the property corners and survey working points is created.

The GIS Scenario (B-1) model is displayed in Appendix "B".

Comments on Scenario Contributions:

This scenario contributed similarly to the previous GIS scenario reviewed. An additional database object class was identified (Marker). This process flow and associated operations were easier to model because the user interface and search operations were similar to that of the already constructed GIS Scenario (A) model. Many attributes and operations for the Marker class were identified or modified.

Scenario (B-2):

After the surveyor returns from the field, the position (measurements) of the house location is entered and the offsets to the property boundaries are computed. The offsets are computed to ensure the house is a proper distance from the property lines.

Processes:

Surveyor Returns from the Field:

1. Surveyor logs into the GIS system and GUI is displayed.
2. Surveyor enters new house survey information.
3. Surveyor requests that house offsets are computed.
4. Offsets are displayed.
5. Surveyor prints a certification map that states the house is certified to be a proper distance from the property lines.

Note: The GIS could have a database that contains the minimum offset distances for a particular jurisdiction (e.g., county, city, etc.). The minimum offset information could be accessed to support step #4 and #5.

The GIS Scenario (B-2) model is displayed in Appendix "B".

Comments on Scenario Contributions:

This scenario required more involved processing and database interactions; the database classes involved in this scenario are the Structure, Property, and Map classes. This scenario provided a good understanding of processes for this function.

This scenario helped with identifying methods which could be encapsulated. For example, the PrintStructure method can be defined as a "private" method and the PrintOrDisplayStructure method can be used as the public interface to the PrintStructure method. The benefits of encapsulating a method are similar to those of encapsulating attributes (e.g., allowing changes to be made internally with minimal impact to external accesses to the object).

This scenario brought an awareness of the applicability of implementing an "active" object-oriented database. This will be discussed in more detail in later sections.

3. Arrange Objects into a Class Structure

Most of the classes were arranged into three hierarchical classes/subclass structures. The roots of these three class hierarchies are Utility, Feature, and Structure classes. Each of these hierarchies have aggregation associations with the class "Map" and "Property" classes. For example, features are "part of" a Map. This type of association will allow efficient storage and retrieval of map and property information and allow these associations to be "clustered" within a GIS database. Arrows were drawn from the Map and Property classes directly to the subclasses of these three hierarchies which allow individual subclass information to be accessed and presented.

Aggregation and generalization/specialization structures were dominant for this type of application; all class instance data, except for the user interface classes, are candidates for database storage. From this observation, this application is well suited for an active object-oriented database management system.

4. Normalize the Model

The normalization processes were applied similarly to that of the Virginia Tech models. Again, strict normalization rules were not always applied because of strong associations of the attributes with particular classes and for efficiency concerns. For example, the "Projection" attribute of the Map class could have been used to create a separate projections class. Creating a separate class for projection information may be appropriate for more advanced systems that use many different types of map projections.

5. Isolate Parts of the System

Similar to the Virginia Tech model, isolating various parts of the system is important for ensuring that future changes to components of the system, such as the user interface, do not require major modifications to other components of the system. In this GIS application, the two major components are the user interface and the database system. Thus, these two components may incorporate a logical separation by adding an intermediate class that provides independence and portability.

6. Iterate to Correct or Refine the Model

The process of creating the GIS model involved refining the model as more specific information about the processes was learned. The scenarios provided more detail about the attributes and operations, and other important capabilities were identified as the models were being developed. These additional capabilities were included in the model.

7.0 Evaluating the Object Models

Following are the criteria used to compare the benefits/limitations of using an object-oriented model for the Virginia Tech and GIS applications:

- Ease in using modeling constructs, notation, etc.
- Ability of model to represent system.
- Difficulty in modifying the model.
- Difficulty in using the model as a programming tool.

7.1 Virginia Tech Model

7.1.1 Ease in Using Modeling Constructs, Notation, etc.

The object model was easy to create using the constructs and notations available. Most of the difficulties were minor. For example,

1. None of the model notations researched provided a clear methodology for extending models that are too large to fit on a single sheet. The only recommendation provided was to place class hierarchies on separate sheets of paper.

2. Notations did not exist for distinguishing between internal and public methods or attributes within a class. Therefore, a notation was developed for this purpose when depicting message passing; this is shown in Figure 7.1. This notation was applied in the GIS Scenario (B-2) model previously described.

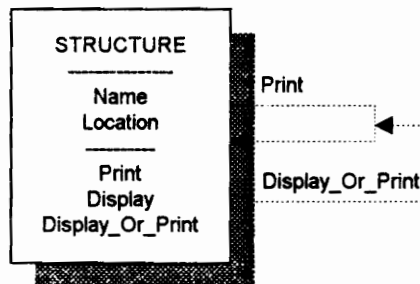


Figure 7.1 Example of a Message Notation. The public method "Display_Or_Print" calls the internal method "Print".

3. Many model notations did not use arrows to indicate the direction of associations. Arrows provided more information and direction to the model and made the models easier to follow.

4. Some object analysis notations provided more capabilities than others which allowed

object analysis to be more closely related to design. For example, KIM[2] allows lines to be drawn from a class attribute to another class. These lines represent references to external class objects, which is closely associated with design.

In summary, the notations researched provided an adequate set of constructs to accurately create an object model of the Virginia Tech system. Only minor problems were noticed.

7.1.2 Ability of Model to Represent System

The object-oriented approach to modeling the Virginia Tech course registration system was not difficult to apply once the modeling process theory was understood. For this application the following characteristics were noticed:

- a. Inheritance was not used or needed to model the system; all relationships were primarily associations and composite associations. Thus, some of the unique characteristics of object-oriented modeling were not applicable or required for this application.
- b. The aggregation structure of the Campus, College, and Department resulted in a clear and clean graphical model that described the associations well.
- c. The design showed a use for "long" data fields for text description information corresponding to various sections of the course schedule. The attributes of the General_Info class is one structure that could not be easily represented using a relational system. Most of the other information associated with this model could be represented using a relational system.

7.1.3 Difficulty in Modifying the Model

The effect that modifying the object-oriented model has on an application, and the modifications that are allowed (taxonomy), depend on the object management system being used. The following schema changes associated with object-oriented systems:

1. Changing the definition of a class
 - changing a name of an attribute or method
 - adding or dropping an attribute or method

2. Changing the class-hierarchy structure

- adding or dropping a class
- changing a superclass/subclass relationship

7.1.3.1 Changes to Definition of Class

For each type of change, an example will be provided to evaluate the impact of the change.

a. Changing the name of an attribute or operation:

Example: Changing the name of the attribute "Time" in the Course class to "Course_Time".

When changing the name "Time" to "Course_Time", the methods of the Course class would have to be updated to reference the new attribute name; external calls to the object would not be affected. However, recent work experienced has shown that:

"Although encapsulation protects against such internal name changes, changing the name of an attribute could make it more difficult to read the application code. "

This is because programmers working on separate object modules typically use the same variable names although it may not be necessary.

Example: Changing the name of the method "GetTime" in the Course class to "GetCourseTime".

If the GetTime method is defined as a "public" method, all calls to this method will be affected. This exemplifies the importance of encapsulating methods as well as attributes. If the method is defined as internal to the object, changing the name will only affect the methods in the respective object.

b. Adding or Dropping an Attribute or Method

Example: Adding the attribute "Prerequisites" and method "GetPrerequisites" to the Course class.

Adding an attribute or method will have an impact on the structure of new instances of that class. Additionally, if the definition of the attribute's length is modified, the applications that send a message to the class object may not be compatible with the new signature requirements.

Existing instances may or may not be affected depending on the object management system. The ORION system, for example, avoids updating existing instances when the schema change does not require the instance data to change its class. If database updates are required, the effect of adding an attribute (e.g., Prerequisites) could be substantial depending on the number of existing instances. Because the Virginia Tech model did not require that inheritance be applied, the effects of adding an attribute to a class are primarily local to the respective class.

Example: Dropping the attribute "CourseNo" and the method "GetCourseNo" from the Course class.

Dropping an attribute or method may impact the application when the attribute or method is inherited into lower level classes of the class/subclass hierarchy. This exemplifies a violation of encapsulation of subclass objects. Methods that access the dropped attributes will need to be updated. If these methods are not encapsulated and if the signature of the method changes, then external methods may also be affected. If multiple inheritance is allowed, a dropped attribute or method may be substituted by that of the same attribute and method defined in a different superclass. Because the Virginia Tech model did not require the use of inheritance, the effects of dropping an attribute are primarily local to the respective class.

7.1.3.2 Changing a Class Hierarchy Structure

a. Adding or Dropping a Class

Example: Adding a class "Plan_of_Study"

The impact of adding a class is primarily dependent on whether the class is associated with a class/subclass hierarchy. This is because inherited attributes and methods from the new class may change the structure of its subclasses. The inherited attributes and methods from the new

class may also conflict with subclasses in the hierarchy, depending on the class hierarchy rules defined for the object management system. Thus, adding a class "Plan_Of_Study" would probably not have a major impact unless it was implemented into a class/subclass relationship with an existing class. Adding a class is considered a "soft" change because it does not require attributes to change classes. Therefore, immediate updates to existing instances may not be required.

Example: Dropping the College Class

Dropping the class "College" could result in a significant impact to any objects that send messages to this class or the subclasses that inherit information from this class. These messages must be identified and removed or redirected. However, because the college class is not a superclass of other classes, dropping this class will only require that the class itself be removed and an association between Campus and Department be defined. Dropping a class is considered a "soft" change. Depending on the object management system, immediate updates to existing instances may not be required.

b. Changing a Superclass/Class Relationship

This change is not applicable to this model. Changing the superclass/class relationship will be discussed in the GIS model (Section 7.2.3.2).

7.1.4 Difficulty to Use as a Programming Tool

The object model is used primarily for developing specifications and subsequent design of the system as compared to use as a programming tool. The model is not detailed enough to begin writing software because important issues such as the process speed requirements, message structure, attribute types, and other design decisions have not been made. However, the model is important to the programmer for the following reasons:

The object model provides:

- information about the fundamental structure of the system
- information about the attribute names and methods names

7.2 Geographic Information System Model

7.2.1 Ease in Using Modeling Constructs, Notation, etc.

The object model, similarly to the Virginia Tech model, was easy to create using the modeling constructs provided. However, the inheritance constructs were applied extensively to this model. Some of the problems/benefits that were noticed are:

- Even with a minimal set of methods and attributes identified for the object model, it was difficult to arrange the hierarchy structures and hierarchy levels because of the space requirements. Using separate sheets for each class hierarchy may improve the readability of the model.
- The arrows were important for improving the readability of the model as well as showing the direction of the associations.
- Associations with the User Interface may be difficult to depict. This is because the User Interface is associated with most of the classes; the numerous lines that are required to depict these associations may become dominant and obscure the model. Therefore, the User Interface associations may need to be separated from the other classes and a technique developed to represent their respective associations.

Note: The complete User Interface was not modeled for the Virginia Tech or the GIS systems.

7.2.2 Ability of Model to Represent System

Following are some observations acquired while constructing the GIS model:

- This object-oriented modeling approach provided a method to represent a relatively complex system such that the system's structure could be easily understood.
- Inheritance was extensively used to represent the feature, terrain, and utility information. Aggregation relationships were used when a map product was generated.

- The class objects provided clear boundaries of information; each class was selected to provide information that may be presented individually or may be combined with other layers of information. Direct paths from the map to individual subclasses demonstrate this requirement.
- The GIS model more extensively used the capabilities provided in an object-oriented environment.

7.2.3 Difficulty in Modifying the Model

7.2.3.1 Changes to Definition of Class

For each type of change, an example will be provided to evaluate the impact of the change. Most of the examples will involve inheritance because this characteristic was not applicable in the Virginia Tech model.

a. Changing the name of an attribute or operation:

Example: Changing the name of the attribute "LineWidth" to "TerrainLineWidth" in the "Terrain Item" class.

Changing the name of the "LineWidth" attribute in the superclass "Terrain_Item" will only affect methods that access the "LineWidth" attribute directly (or via the methods defined for the class). If the signature of the methods that access the data remain the same, then the change should have a minimal effect, requiring corrections only to the method that accesses the attribute.

In this GIS model, this attribute is locally defined by each of the subclasses of the "Terrain_Item" superclass. Thus, changing the name in the superclass does not affect the instances of the subclasses. However, all new instances will now inherit the newly named attribute. This change is a "soft" change which will not require existing instance data to be immediately updated to reflect the schema change.

Example: Changing the name of the method "GetLineWidth" to "GetTerrainLineWidth"

Because the "GetLineWidth" is inherited to all the subclasses, changing the name will affect all methods that call this method. If the "GetLineWidth" method is encapsulated, then the changes could be made internal to the "Terrain_Item" class without any affect to external calls. This is another example of the importance of encapsulation of methods. This change is also a "soft" change which will not require existing instance data to be immediately updated to reflect the schema change.

b. Adding or Dropping an Attribute or Method

Example: Adding the attribute "ContourSpacing" and the method "GetContourSpacing" to the "Elevation" class.

As mentioned earlier, a new attribute or method may affect all new instances in its class and those which inherit the respective object characteristics. When an inherited attribute conflicts with a locally defined attribute, the locally defined attribute will take precedence. Similarly, an inherited method can be "overridden" which indicates that the locally defined method will be used rather than the inherited method which has the same name. In this example, adding the "ContourSpacing" and "GetContourSpacing" to the "Elevation" class is not expected to impact existing instances. This type of change is a "soft" change.

Example: Dropping the attribute "LineType" and the method "GetLineType" from the "Terrain_Item" class.

Dropping attributes and methods affects all internal and public methods in the "Terrain_Item" class and subclasses which access these data and procedures. Therefore, when inheritance structures are involved, schema changes can be expensive to maintain. The problems extend to external classes if these attributes and methods are not encapsulated.

7.2.3.2 Changing a Class Hierarchy Structure

a. Adding or Dropping a Class

Example: Adding a class "Map_Item"

If a class called "Map_Item" was added and Feature, Terrain_Item, and the Utility became subclasses of Map_Item, then all new instances would inherit the attributes and methods of the "Map_Item" class. The attributes and operations of the new class may conflict with existing attributes and methods. Therefore, the impact could be extensive if the process is not carefully thought out. This change is also a "soft" change.

Example: Dropping a class "Wetland"

Dropping a class affects all the associated classes that access the methods of the dropped class. Dropping a class may cause significant maintenance requirements. This change is a "soft" change and does not require instance data to be updated immediately to reflect the schema changes.

b. Changing Superclass/Subclass Relationship

Example: Hypothetically, if the "Map_Item" class is redefined as a superclass of the "Terrain Item" class and the "Terrain Item" class is removed.

Removing the Terrain Item class requires the subclass to have new superclasses, and object-identifiers of instance data may need to be updated. Thus, new instances of the subclass may now have different inherited attributes and methods. Rearranging a superclass/subclass structure may significantly impact the system.

7.2.4 Difficulty in Using as a Programming Tool

The benefits of using the GIS as a programming tool are the same as those stated earlier for the Virginia Tech model. In addition, the following observations were made:

- Because other GIS models can be expected to contain more complex and higher levels of inheritance structures, the object model may be very helpful to the programmer for tracking which attributes and operations are inherited.

- Because the associated classes are specified in class definitions, the object model is an important document for providing this information to the programmer.

8.0 Object-Oriented Databases

Most programs today, in a multiuser system, share data by using a database management system (DBMS). A DBMS is used to handle simultaneous access to the shared data and to efficiently retrieve various requests for stored data. Using a database management system also promotes independence of procedures by allowing the DBMS to keep track of the complex data structures needed to store and retrieve data rather than placing them in the individual procedures of the application.

An object-oriented database is a database that is defined and manipulated in a way that is consistent with an object-oriented data model. An object-oriented database system organizes data and behavior into classes as previously described and may provide many characteristics that are common to current database systems.

The purpose of this section is to provide an understanding of the primary characteristics of object-oriented databases and how they are associated with object-oriented modeling. Comparisons are made to relational databases to help with understanding the benefits or limitations of a database characteristic.

8.1 Reasons for Object-Oriented Databases

Although a relational database can be applied to an object-oriented design, object-oriented databases provide some unique benefits:

1. Object-oriented languages and databases may be designed with similar language constructs.

The integration of the host and database language may reduce the complexity and learning time for a programmer. Following explains how a host language can access data from a relational database and object-oriented database.

a. Accessing a Relational database using a Host Language:

Host application programs that interact with a relational database require embedding SQL commands within the application program. The programming and database languages

are separate languages. Following are examples of how embedded SQL commands can be implemented for single record and multiple record queries.

Single record query:

```
EXEC SQL SELECT EMPLOYEE_NAME
        INTO : E_NAME
        FROM EMPLOYEES
        WHERE EMPLOYEE_ID = 004_003_2222
```

The above example returns the name of the employee that has the employee ID of 004_003_2222. The name retrieved is copied into the host variable E_NAME.

Multiple record query:

When more than one row (tuple) is retrieved, a data construct called a "cursor" is required. A cursor is a buffer area used to store the data retrieved from the query. This buffer area can subsequently be accessed by the host language. A cursor is required because the host language accesses records one at a time as compared to SQL queries which may generate data sets that contain many rows of data. Following is an example of how a cursor is defined and used when a query is executed:

/ Following is the cursor definition. The cursor name is C_NAME for this example */*

```
EXEC SQL DECLARE C_NAME CURSOR FOR
        SELECT EMPLOYEE_NAME, EMPLOYEE_AGE
        FROM EMPLOYEE
        WHERE (EMPLOYEE_AGE > 25) AND (EMPLOYEE_AGE < 30);
```

```
/* begin main program */
```

```
main( )
```

```
{
```

```
/* The following OPEN command executes the query previously defined for the cursor. After  
the data is retrieved, the cursor pointer is set to the beginning of the resulting table */
```

```
EXEC SQL OPEN C_NAME;
```

```
/* The following FETCH command reads the attribute data from one row and puts it  
into the host variables "e_name" and "e_age" */
```

```
EXEC SQL FETCH C_NAME INTO : e_name; e_age;
```

```
.
```

```
.
```

```
.
```

```
}
```

b. Accessing an Object-Oriented database using a host language:

Object-oriented languages will integrate the host and database commands into one language having common constructs. Following is an example of how a single and multirecord query can be implemented in an object-oriented language:

```
/* Define a class and its associated attributes and operations. This definition is for a  
class that will be part of the object-oriented database */
```

```
(make-class Employees
```

```
  :superclass
```

```
  :attributes
```

```
    (name
```

```
      :domain character
```

```
    age
```

```
      :domain integer)
```

```
  :methods
```

```
    (DisplayEmployeeInfo name age))
```

```
/* Retrieve the names and ages of all employees whose age is between 25 and 30 */
```

```
main( )  
{  
  (select Employee name, age  
   where ((age > 25) and (age < 30)))  
}
```

Note: The application can now use the results of the query in the application.

From this above example, the database constructs used to define the structure of the database and to provided access to the database are similar to that of the common object-oriented programming constructs. The database and programming language are integrated into a common language.

2. Complex applications are difficult to implement using Relational systems.

a. Applications that require nested structures, long data files (such as a large bitmap), or complex searching (such as artificial intelligence applications) are not easily supported in a relational system.

Following is an example of how a nested query can be implemented using a relational database language (SQL) to find "the manager of an employee's manager".

```
EXEC SELECT NAME  
FROM EMPLOYEE_MANAGERS:EM  
WHERE (EM:NAME = NAME);  
  (SELECT NAME  
   FROM EMPLOYEE_MANAGERS:EM  
   WHERE EM:NAME = JACKSON)
```

The same retrieval of data using an object-oriented database language follows. A hierarchy structure can be traversed until the query is satisfied.

```
(select Employee (recurse Manager) (Name = "Jackson"))
```

8.2 Reasons for Extending Relational Databases

Relational database systems are being enhanced to include some of the special capabilities of object-oriented systems. These databases are called Extended Relational Databases. Because a detailed study of Extended Relation database systems is outside the scope of this project, a summary follows which describes the primary reasons for their development:

The reasons why Extended Relation databases may be favored over current object-oriented databases are:

- Technology is developed based on a mathematical theory.
(Relational Algebra, Relational Calculus)
- Relational databases are established and have been effective.
(i.e., standards exist)
- Relational databases have proven security features. Object-oriented systems may not have fully developed or tested security features.
- Learning object-oriented modeling and design requires training and experience.
- Converting an existing system to an object-oriented system may be costly.

8.3 Data Definition Language

Referencing the Data Definition Language described by KIM[2], defining an object-oriented database is performed similarly to the way a program defines a class structure. Following is an example of how a class definition for an object-oriented database can be implemented:

```
(make-class Classname [: superclasses ListofSuperclasses]
                  [: attributes ListofAttributes]
                  [: methods ListofMethodsSpecs])

(AttributeName [:domain DomainSpec]
              [:inherit from Superclass])
```

8.4 Data Manipulation Language

8.4.1 Navigation method

After the database schema has been defined, information can be stored, retrieved, or updated using database commands. Conventional databases reference data (navigate) by value; object-oriented database reference data by object identifiers.

Object identifiers are system generated unique references that are used to identify object instances.

In the ORION system described by KIM [2], an object identifier consists of a combination of the class identifier and an instance identifier; the combination provides a unique reference to the associated instance. For example, in Figure 8.1, the class PERSON may have an identifier 100 and the subclass "ENGINEER" may have an identifier of 200, and an instance of engineer, "Mary Payne", may have an identifier of 500. Thus, to refer to this instance, the navigation path from the class "PERSON" is 100 200 500. However, the object identifier can be uniquely referenced using the identifier 200 500 which is the combination of the class and instance identifier.

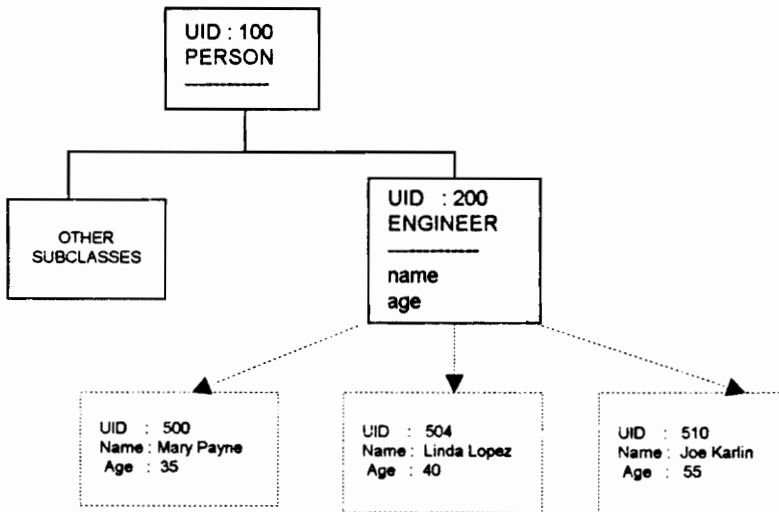


Figure 8.1 Object Identifiers Associated with Object Instances.

An object instance is stored in a structure; the object identifier is stored in a field in this structure. Another field within this structure may contain the attributes of the class which may refer to other structures, as in a composite hierarchy.

When a search query is performed, the object identifiers of the instance data are returned from the Query. For example, in Figure 8.1, to obtain all instances of engineers that have the name "Mary Payne" and age "35", the following query may be submitted:

```
(Select Engineer
  where name = "Mary Payne"
    and age = 35)
```

A list of object-identifiers (or the actual object instances) are returned to the application. Object identifiers can be used to limit the search on subsequent queries prior to retrieving the actual instance data. Including the class "PERSON" in the query is not necessary because the target class "Engineer" contains the search space of interest.

The indexing method on the Unique Identifier (UID) can be implemented as one index for a hierarchy of classes or as separate indexes for each class attribute. The explanation provided by KIM [2], indicates that using one index to store the UIDs requires generally less search time and less storage requirements.

Navigation Method (Advantages and Disadvantages):

Following summaries the advantages and disadvantages of navigating by using object identifiers in an object-oriented system versus using attribute values in a relational system.

a. Navigation by Object-Identifiers:

Advantages:

° More efficient and convenient because object identifiers are automatically generated. This is important in a distributed environment.

◦ Navigation performance may be better because object identifiers can be generated to facilitate object look-up.

◦ Using object identifiers allows objects to be identified without requiring the actual object to be immediately fetched. The object instances can be fetched as needed.

Disadvantages:

◦ When a object identifier is a combination of a class identifier and instance identifier as noted above, changing a class may require significant overhead to update the class identifiers.

◦ Ulman [3] has stated that an object navigation method reduces the ability of a system to be "declarative", meaning that there is less flexibility in the ordering of commands that are used to obtain a desired set of data. However, he indicated that in the future, knowledge-based systems are expected to provide object-oriented capability with declarativeness.

b. Navigation by Values Within a Record:

In a relational system, navigation is performed by referencing the values of attributes. To navigate from one table to another in a relational system, a "foreign" key is used. A foreign key is an attribute of a table that takes on values of the primary key of another table.

Advantages:

◦ Provides easier support for declarative operations

◦ Easier support for ad-hoc queries

Disadvantages:

◦ The programmer is required to generate their own primary keys.

◦ Primary keys can be inefficient to store as compared to object identifiers.

Object-identifiers are unique system generated values; primary keys are user defined and typically longer to obtain meaning.

8.4.2 Query Language:

This section will explain how an object-oriented language can be used to perform the basic operations used in a relational system. A relational system has the following basic operations:

- Selection
- Projection
- Join

Following describes how these three operations can be performed using an object-oriented Data Manipulation Language:

Single Operand Queries:

Single operand queries (Selection and Projection) in an object-oriented system are comparable to operations performed on a single relation in a relational system. However, in object-oriented systems, these types of operations are associated with one class or a class hierarchy rooted at a class.

"The root of a class for a Single Operand Query is the highest class that includes all of the objects involved in the query." (see figure 8.2)

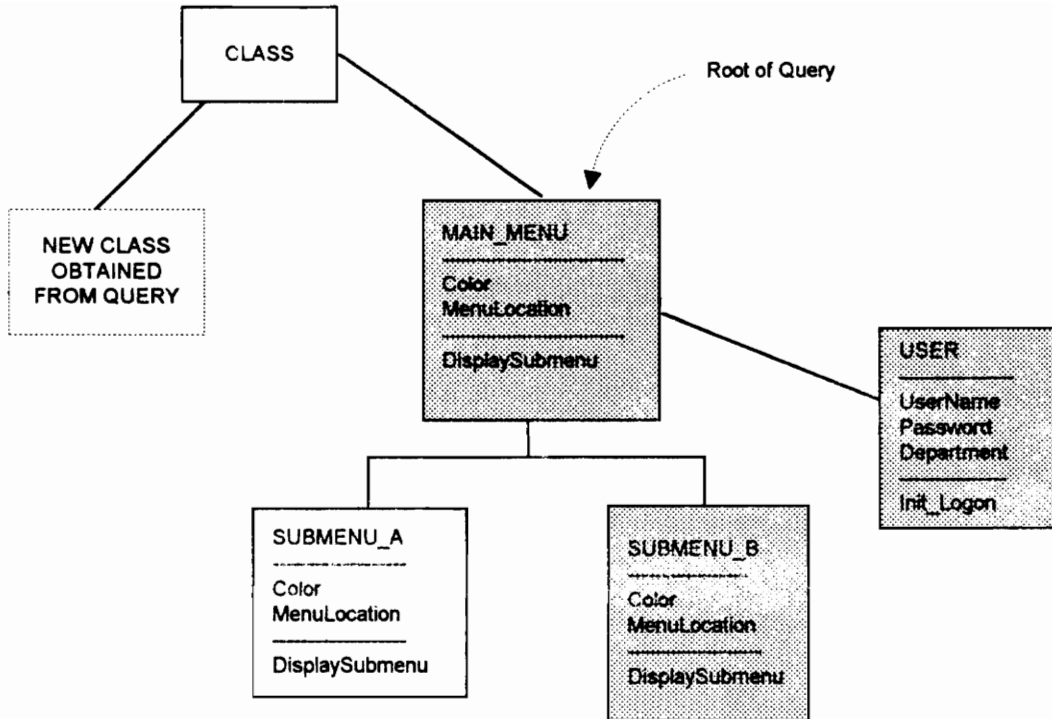


Figure 8.2 Query Graph Showing Root of Query.

The data returned from the query, depending on the database system, can be:

- a set of instances that is subsequently stored in a new class hierarchy in the object system; or
- a list of object identifiers that can be retrieved as desired by the application.

Both of these methods result in a new class hierarchy that can be accessed using regular host language constructs.

Following is an example of how a Selection can be performed using an object-oriented database programming language. This query returns all the attribute data associated with the object instances (or object identifiers) for users who have the UserID "J24100".

```
(select USER
      where UserID = "J24100")
```

Following is an example of how a Projection is performed. This query returns only the user-names associated with the object instances that have the userID "J24100". The Username is an attribute of the "USER" class.

```
(select Username from USER
      where UserID = "J24100")
```

Multiple Operand Queries:

Following is an example of how a Join operation can be performed. This query returns a list of part names that are associated with the classes PART and VEHICLE.

```
(select PartName from PARTS
      where PartName is-in VEHICLE)
```

8.5 Data Control Language

The Data Control Language deals with other database functions such as data integrity, data management, etc. These responsibilities are performed similarly to that of relational systems. For example, the ODBMS assigns ownership to objects similarly to the way a relational system assigns ownership to tables. Access privileges to database information can be established using the database language. Data control responsibilities also include locking and unlocking database objects to ensure integrity during concurrent data processing.

9.0 Lessons Learned - Identifying User Requirements by Prototyping

A problem with determining what specific methods and attributes should be included in each class and how these classes are related was encountered during the early stages of the Virginia Tech and GIS model development. A couple specific scenarios were used to help obtain this

information and to demonstrate the interactions (message passing) between objects. As identified by Dr. Csaba Egyhazy,

"You do not want to specify classes, attributes, and methods prior to understanding how the system is to function."

Without understanding how the application processes are to improve the current system, there is no direction on how to proceed. The following observation was made:

"Creating a User Interface and simulating processes flows by using scenarios are two techniques used for identifying user requirements. These two techniques are primary methods for developing a prototype of a system in an Object-Oriented environment."

Prototyping is a method that can be used to help identify the user requirements prior to building the complete system according to fixed specifications. The following section describes how a system can be analyzed and specified using the prototyping process:

9.1 Prototyping an Object-Oriented System

Prototyping is an analysis and design technique used primarily when the client cannot define the requirements of the system; prototyping allows the problem to be defined as the system is being developed. In a traditional system development environment, changes to software in response to a design change are called "engineering changes". Engineering changes may cause significant delays and modifications to a system. In object-oriented prototyping, an environment is established where the client is allowed to see and interact with a "skeleton" system and provide feedback to improve it. Thus,

"Users and developers can identify or modify system requirements prior to developing the complete system."

An assumption is that "the more structured a design process becomes, the less problems will occur." Documentation [5] has indicated that clients are generally unsure of the system requirements and will recognize important requirements as the system is being built; in some cases, precise requirements cannot be specified unless the system is built. Prototyping

provides a flexible way to assist the client with identifying user requirements in the initial stages of the analysis and design.

Three major steps in developing an object-oriented system using a prototype are:

1. Use the iterative technique of prototyping until the system demonstrates that it will meet the user requirements.
2. Use the resulting prototype system to develop the specifications of the complete system and formally describe the system.
3. Follow traditional structured/management methodologies to build the system.

In summary, prototyping is used to determine the system requirements, and formal traditional processes are used to build the system to these requirements.

9.2 Identifying the Objects of the Prototype

Before a prototype can be developed, we must first identify the problem to solve (formally called target recognition) and a solution to this problem must be possible. Second, we must identify how the current system operates and examine the parts associated with the processes being developed.

A method of identifying the process flows and other program characteristics is to define a user interface; the user interface definition is usually a direct representation of the underlying functionality of the software. An example is the user interface defined for the GIS model (shown in Figure 9.1).

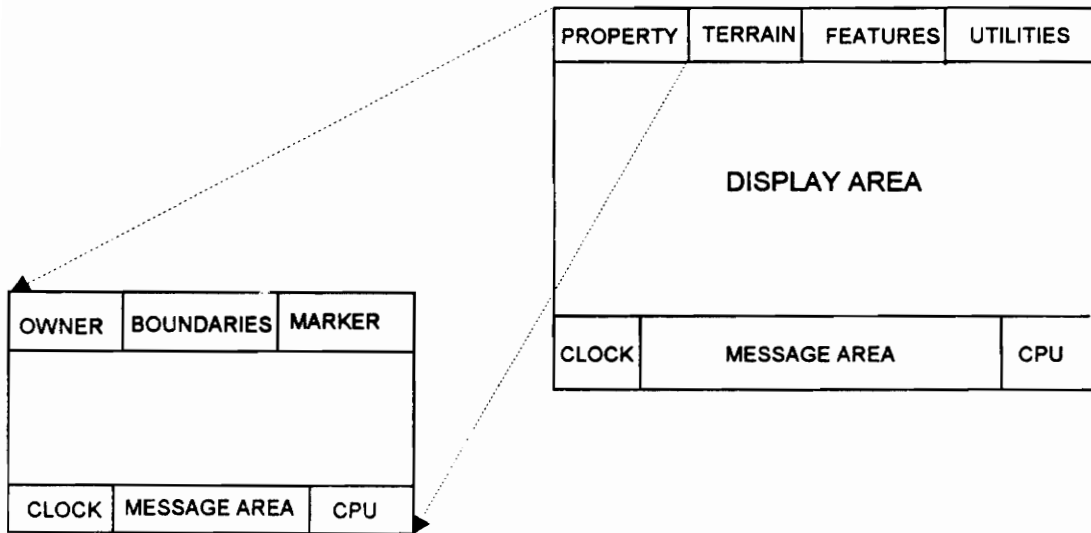


Figure 9.1 Identifying Functions by Creating a User Interface.

The user interface is the focus of the prototype because it directly interacts with the user and is a means of generating feedback prior to building the complete system. The user interface provides a method of navigating through the functionality of the system. The prototype user interface can be used to display, for example, samples of: return values from a function, a graphics display, or a database query return set. The data can be displayed to the user to simulate the real functions "without requiring the functions actually be performed or fully implemented.

From the user interface, a developer can identify characteristics such as classes and their attributes, associations/navigation routes between functions, data field privileges, etc. Although the user interface is a good place to start, a more in-depth search for system components will be necessary. The following steps provide a general description of how this process can be performed:

1. Select the most obvious base class. A base class is a class that is not a subclass.
2. Select subsequent classes that provide the largest number of new requirements.

Note: The classes identified in steps #1 and #2 are very important because they are the frame of the system. Subsequent classes/characteristics of the system will add to this frame (see Figure 9.2).

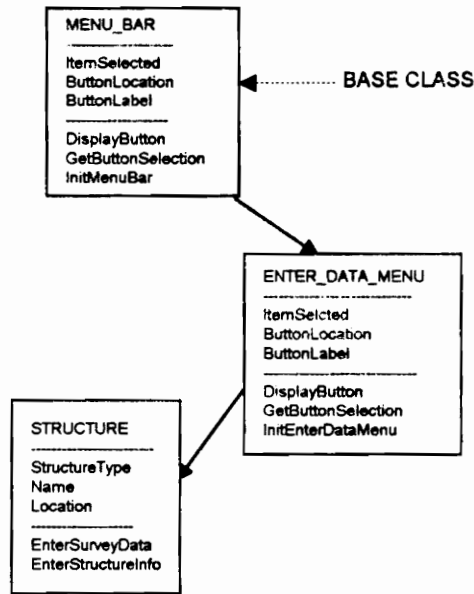


Figure 9.2 Establishing a Frame for the Object Model.

3. If you cannot decide the sequence of adding objects, select the one with the largest number of associated objects. These objects may provide the most information about the system.
4. Keep track of the association links in a table format (see Figure 9.3).

<u>FIELD</u>	<u>OBJECT</u>	<u>LINKED TO</u>
ItemSelected	ENTER_DATA_MENU	From MENU_BAR
ItemSelected	FEATURE_MENU	From ENTER_DATA_MENU

Figure 9.3 Example of Tracking Object Links.

5. Continue until all of the core objects have been identified.
6. Implement a part of the system as a prototype.
7. Obtain feedback and reiterate steps 1-6 until the system requirements are understood.
8. Write the design specifications.
9. Fully implement the design.

In reference to item #6 above, the parts of the system that should be implemented are those that are necessary to help understand the user requirements. The prototyping process stops when enough of the functionalities of the system have been identified so that the design specifications can be created. At this point the prototyping process is "frozen" and the system is built to specifications using traditional management techniques.

Other Notes Concerning the Prototyping Process:

- Many requirements are fixed, such as the requirement to use an existing database system. The developer must build the system around these requirements.
- Object-Oriented Methodology does not reduce the problems caused by a poor or incorrect design. Modifying an object-oriented design can be as burdensome as a conventional design.
- At some point in the prototype development, the interface will no longer be enough to demonstrate the progress of the prototype to the user; the user must be provided with some concrete output data from the system. The answer is to:

"Focus on design areas that will be unique to this system and avoid developing the functions that are common to previous developed systems."

Care must also be taken that the prototyping process does not use up most of the funding that is needed to develop the real system from the prototype specifications.

- Quick changes may be necessary when developing a prototype; thus, a powerful prototyping language (such as smalltalk) should be used in the prototype development.

10.0 Management Issues Concerning Object Model Development

To obtain the benefits from object-oriented analysis and design, the process must be managed effectively. A book "The Decline and Fall of the American Programmer" [5] provided information concerning these issues. Following are some important management considerations that should be understood *prior* to changing to a different development methodology.

10.1 Understanding the Goals

The goals that justify a change to a company's method of development or managing the development are:

- Increased productivity
- Increased quality

Thus, how does object-oriented analysis and design meet these goals? What is really needed to obtain these two goals? Edward Yourdon [5] recommended that companies be aware of the following:

a. Programming Languages:

- A new language (such as an object-oriented programming language) is not easily converted to; it is an expensive and time-consuming process to learn.
- A better programming language may not improve quality and productivity unless the management processes are also improved.

b. Better People:

- Better languages and tools will not improve quality and productivity if a company's hiring practices do not obtain highly capable people who receive ongoing training, can be motivated, have an adequate work environment, and facilitate the creation of effective teams.

c. Automated Tools:

◦ Automated tools can expect a decrease in productivity up to the first 18 months because of the learning, training, and management requirements. In addition, CASE tools typically lag behind methodologies, in some cases by several years.

◦ In structured programming, Computer-Aided Software Engineering (CASE) tools assist with managing the development and with converting graphical models into textual output. However, the process of creating the graphical models historically has demanded as much time as writing the programs.

d. Prototyping:

◦ Prototyping does not eliminate the need for formal analysis and design work on large, complex projects.

e. Organization Readiness

◦ The organization should analyze its readiness to change to a new development methodology (to be discussed later).

f. Software Reusability:

◦ Highly effective organizations (world class) are achieving 60 to 70 percent software reusability[5] which may only be achieved if engineers are motivated and a management structure established to incorporate software reuse.

g. Software Reengineering:

◦ Contrary to common belief, the greatest gains in productivity may be achieved in maintaining software rather than developing new systems.

Most of the recommendations mentioned above will require organizational behavior changes (i.e., management and technology must work together for significant improvements to be made).

10.2 Understanding the Readiness (for Object-Oriented Methodology)

Introducing a new technology or methodology in the middle of a financial crises or in the middle of project which has a tight deadline may cause the project or company to fail. It is important to understand the readiness of the organization's ability to incorporate a new technology.

"The level of maturity of the software development organization is important for knowing the next steps to take to improve the organization."

The Software Engineering Institute has identified five levels of organizational maturity:

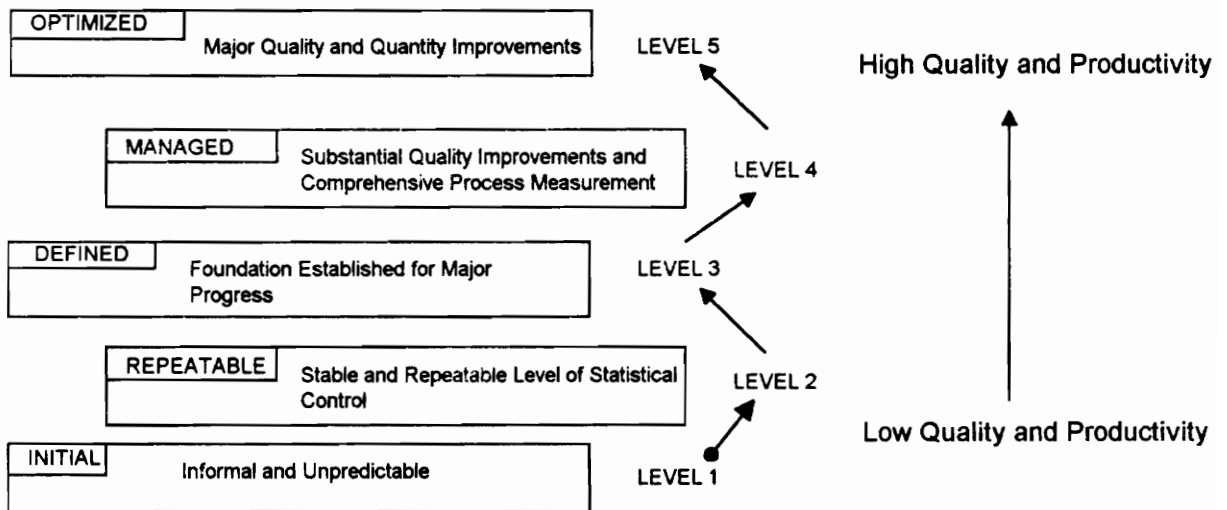


Figure 10.1 Software Development Maturity Levels.

To implement object-oriented methodology, the maturity level of the respective organization should be understood. The reason is because new methodologies that are used in a disorganized and poorly managed manner, will not produce increases in productivity and quality.

Documentation [5] has stated that an organization, which is considering implementing a new technology or methodology, should be aware of:

- Skipping maturity levels is not usually possible.
- There may be considerable time and expense to move from one maturity level to the next.
- Complete conversion to a new technology or new methodology at the lower levels (level 1 and 2) should be avoided; recommendation is to implement new technology or methodology as a pilot program or as an experimentation.

10.3 Managing the Implementation (of Object-Oriented Methodology)

To retaining productivity it is important not only to progress gradually to the next level of organizational maturity, but to progress gradually to the next level of development methodology. Figure 10.2 shows three levels of development methodologies:

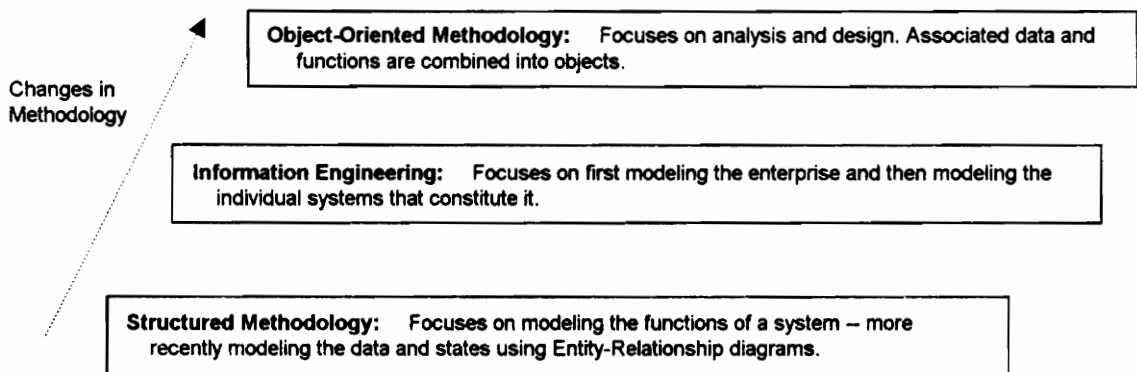


Figure 10.2 Changes in Software Development Methodology.

It may be important to establish software process group(s) for evaluating a new methodology and subsequently manage its implementation. Management structures will be needed, for example, to establish and managing a reusable library and to ensure the library is continually improved with each project.

10.4 Issues Concerning Software Reuse

10.4.1 Case Repository

Software tools are available that assist with managing software reuse. Modern CASE Repository software provide an organization with the tools to create, modify, and maintain a repository system. Following is an example of the information that can be stored for each item (object) in the repository:

- Name of item
- Description
- Type, alias, and item information
- Parents of the object
- Rules for using the object
- When it was created and last updated
- Status and version number
- Where it is physically stored

The capabilities of CASE Repository software varies from manufacturer to manufacturer. Not only are individual objects candidate for reuse; items such as complete programs, models, diagrams, etc., are also potential candidates for reusability.

10.4.2 Management Requirements

To seriously implement "reusability of software" into an organization, the organization must establish a management structure to facilitate the processes involved. The organization must:

- Invest time to create and document reusable components.
- Invest time to perform higher levels of testing for these components.

- Invest in the maintenance of these libraries and in software to access the required components.
- Recognize and reward employees for reusing components.
- Establish separate groups that are responsible solely for identifying, testing, and maintaining of library components.
- Implement tight control of library components (e.g., what is put into the library).
- Implement tight configuration management of library components.

Reusability of project work may be one of the most important benefits of object-oriented methodology.

"When reusability is properly implemented and managed, productivity increases of 50 to 200 percent can be expected." [5]

11.0 Project Summary and Conclusions

The following summarizes the major observations and contributions that have resulted from this project:

1. Understanding the processes associated with object-oriented modeling and performing an object-oriented analysis of two types of applications.

Applying object-oriented modeling was time consuming and difficult because of in-experience and lack of understanding of the processes involved. Following are the key observations and lessons learned while creating the object models:

a. Object modeling is not limited to special applications; it can be effective, it appears, to model most applications.

The characteristics of object-oriented methodology, such as classification, encapsulation, and polymorphism, have benefits that should improve the flexibility and organization of various types of software systems. The power in object-oriented modeling may be its ability to define abstract data types which allow the analyst to focus more on the "level of the real-world system" as compared to the level of programming languages. Once a type has been defined, it can be used as a building block for more advanced types.

b. Prototyping is an effective way of learning about the application and may be one of the prime methods for helping the user and developer understand the requirements of the system.

Prototyping was important for identifying the attributes and operations of the object models. This was demonstrated in the Virginia Tech and GIS models when scenarios were determined necessary for accurately identifying the attributes, operations, and the process flows associated with the object model.

When developing a system prototype, it is important to select scenarios that represent each major functionality of the system and which include the root classes of class hierarchies and class composition hierarchies (base classes). This will minimize the need to modify class

structures or abstract new class structures when additional capabilities are included in the prototype.

Prototyping provides a flexible development environment that allows the system to be evaluated and refined up to a point that the requirements can be identified. At this point, requirements are defined and the system can be implemented using traditional structured methods.

c. Class names can be accurately identified when the analyst understands fairly well how the system will function.

Most of the class names used in the models prior to using the scenarios were accurately identified. This may be a result of prior knowledge of the primary functions of each system. However, the attributes and operations of the system were not accurately defined without creating scenarios.

d. The decision to implement an Aggregation versus a Generalization or Association structure should be based on the independence and equality of object status.

While modeling the Virginia Tech system, a problem occurred with deciding whether to model Students, Instructors, and Courses as a "part-of" relationship with the Department class. An aggregation relationship was chosen; however, this may have been implemented as straight associations. This is supported by the following statement which was documented for a similar situation:

"A company is not an aggregation of its employees, since a company and a person are independent objects of equal status. " [1]

The decision to use aggregation to model the Student, Instructor, and Course classes in the Virginia Tech model is debatable. A close coupling exists between these respective classes which may support the decision to use an aggregation relationship.

The physical structure of a "part-of" relationship causes the classes to be nested within the aggregation class (i.e., a structure within a structure). The effect of incorrectly modeling a non-aggregate relationship as an aggregate relationship is that the functions are less independent within the application. The resulting nested structure can become large and more complex, and the management of this structure by the operating system may be handled

differently. For example, the operating system may unnecessarily swap the nested structure in and out of memory because of its tightly coupled "part-of" relationship.

e. Methods as well as attributes should be encapsulated internally.

Most referenced documentation failed to mention the importance of encapsulating methods. The following important concept was observed:

"Methods can be protected internally and still be accessible indirectly to the public. This is done by adding a public interface that calls the internal method."

For example, in Figure 11.1, the internal methods AddFeature or DeleteFeature are accessed through a public interface called "Add_Or_DelFeature(arg1,arg2)" where arg1 is a flag indicating whether to add or delete a feature object.

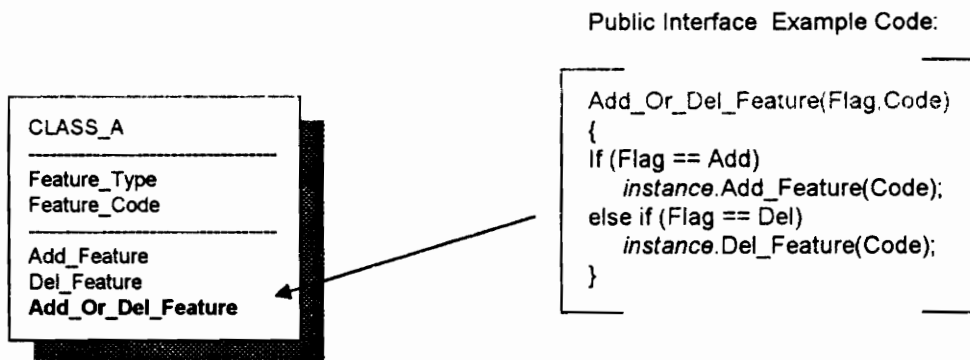


Figure 11.1 Example of a Public Interface Method.

The cost of adding a public interface method to call an internal method is the time required to look up the specified internal method identified as an argument in the message.

f. The model may include instance data if this information provides important information to the designer.

A question arose concerning the need to list the college names in the Virginia Tech object model. College names are instances of the class "College"; however, if the individual colleges have significant differences in class structure, these colleges could be subclassed. To provide more information in the object-oriented analysis model, the individual colleges could be listed as instances of the College class. This may be favorable because of the small number of colleges (see Figure 11.2).

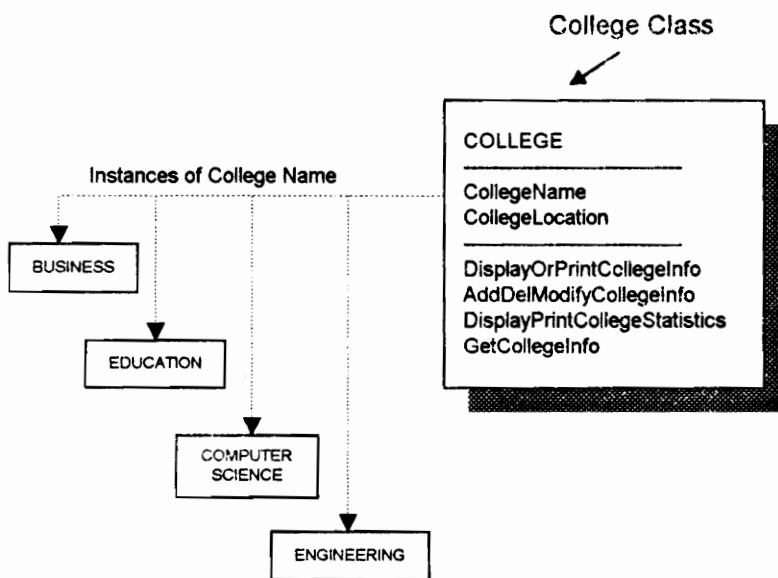


Figure 11.2 Instances of a Class Displayed in an Object Model

g. The purpose of analysis is to identify and define the user requirements.

The separation between object-oriented analysis and object-oriented design was better understood after reading a book on Rapid Prototyping[4]. This reference indicated that the primary purpose of analysis is to identify and define the user requirements. From the analysis results, the system specifications can be created. Analysis is for describing and modeling the

problem and proposed solution; however, the designer will control the details of the variables, will decide the best data structures to use, and will make decisions about such items as the system architecture and database management system.

2. Evaluating and comparing the Virginia Tech and GIS object models and determining the benefits and limitations of applying object-oriented methodology.

a. The ability of object-oriented constructs and notations to represent the two system applications.

"The special constructs of object-oriented modeling were particularly useful for reducing the complexity of both the Virginia Tech and GIS models."

The object model was easy to create using the constructs and notations available. Most of the difficulties were minor. However, a clear notation for extending models to adjacent sheets was not available. These models demonstrated a need for graphics/development tools that will allow the programmer to efficiently see different parts of a model and make changes to the model.

A notation was developed for distinguishing between public and internal methods when message passing is overlaid on the object model. The example below shows a public interface "Display_Or_Print" sending a message to the internal method "Print". This notation was applied in a GIS scenario model (see Figure 11.3).

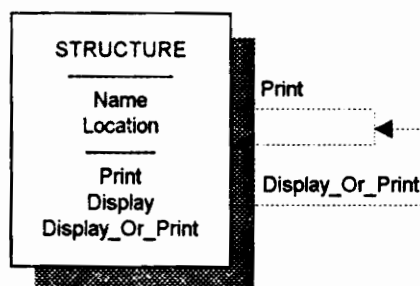


Figure 11.3 Example of a Message Notation.

b. The dominant or unique object structures associated with each system application model.

"The Virginia Tech model applied association and aggregation relationships. The Geographic Information System model applied inheritance, aggregation, and association relationships."

The Geographic Information System and Graphical User Interface applications applied class hierarchy structures having layers of specialized information. Inheritance was applied extensively in these applications. The GIS system also applied aggregation and association when combining various types of map information.

The Virginia Tech model did not require the use of inheritance; most of the class structures had association relationships. However, this system demonstrated a need for "long" data fields for storing textual information (such as rules and regulations). Object-oriented databases are important for storing this type of information.

c. The difficulties that may occur when modifying the system object model for the two applications.

"Schema changes associated with GIS systems may be more difficult to maintain as compared to applications such as the Virginia Tech course schedule system."

Schema changes to a class definition in a class/subclass hierarchy may affect other classes within the hierarchy structure and require database updates. Although the changes demonstrated in previous sections allowed existing instances to coexist with the new definition, some schema changes associated with class/subclass hierarchies may require immediate updates to the object identifiers of existing instances.

3. Understanding how an object-oriented system can be implemented using an object-oriented programming language and database language.

a. Understanding the differences between aggregation and inheritance structures at a low level (coding level) was the key to understanding the potential impact to a system if inheritance is misidentified as aggregation or vice versa.

Aggregation involves structures within structures (i.e., one structure contains pointers to other structures). Inheritance involves sharing common space within the class hierarchy. Aggregation requires following a pointer from the aggregate structure to the structures of the aggregate elements. Therefore, more overhead can be expected in an aggregate association. An inheritance structure may be faster and necessary for certain applications where overhead must be minimized.

b. Encapsulation is applied to minimize the impact of changes to attribute data or changes to the methods within an object. However, inheritance structures may result in a violation of encapsulation.

The application programmer must be disciplined to structure his programs so future model changes will not impact existing programs. This is particularly true when class/subclass object structures are being accessed. The following advice was provided by an experienced application programmer:

"When an application accesses inherited data, the message to retrieve the data should be sent to the lowest level class of interest in the inheritance hierarchy. This ensures that when modifications are made to any class or subclass in the hierarchy, these changes will be accessed by the application without needing to change the application."

Figure 11.4 exemplifies this concept. A new subclass "VEHICLE_A" is added that overrides the attribute "Body_Width". The applications which access the higher level class "VEHICLE" will now be invalid because they do not access the new attribute value. This assumes that the subclass "VEHICLE_A_1" is the subclass of interest.

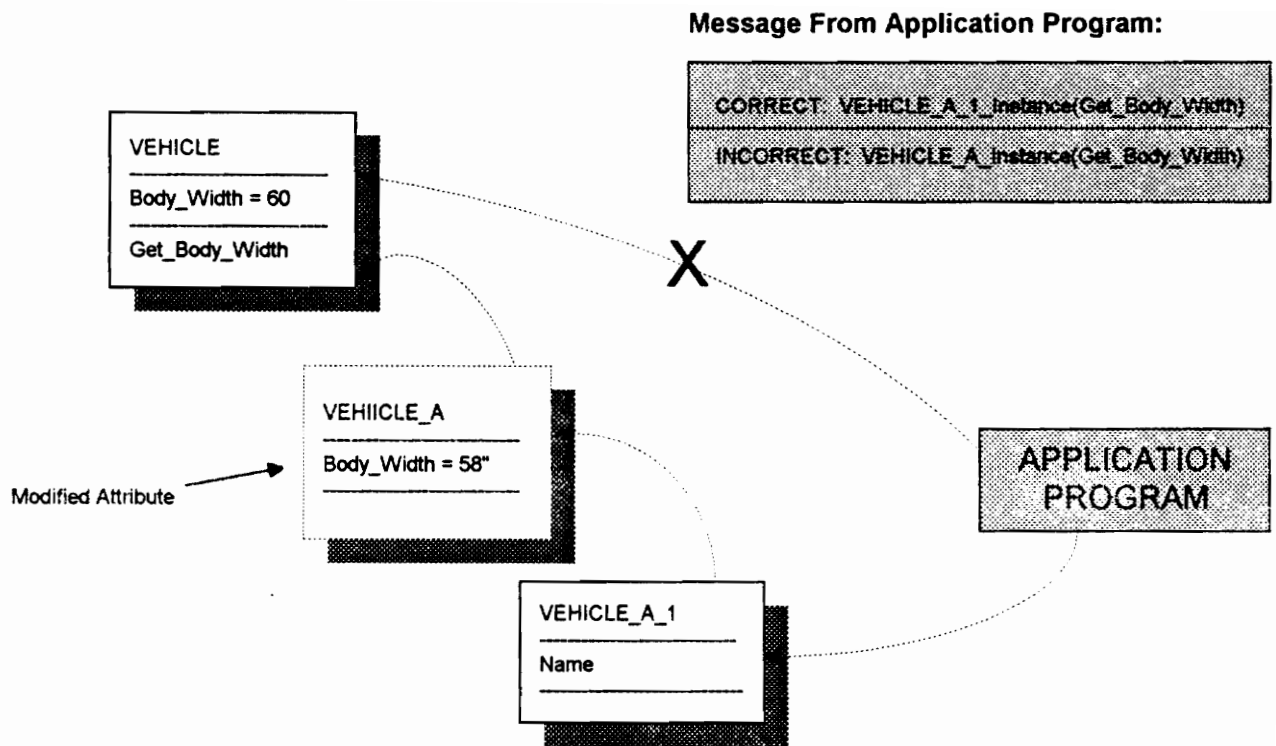


Figure 11.4 Application Program Accessing Attribute Data.

c. **Understanding the Object-Management System is important for understanding the impact of a modeling decision.**

The decision to implement one type of relationship versus another will have an effect on the relative physical storage of the instance data associated with the application. Various object management systems may use different object storage and retrieval methods; therefore, it is important to understand the characteristics of the object-management system that will be used.

d. **Object-oriented databases provide many features that are effective for complex systems having various data types and data sizes.**

The decision to use an object-oriented database versus a relational database may be primarily influenced by the application to be implemented. Object-oriented database are particularly useful for:

- storing large and variable size data (e.g., CAD, CASE, and Multimedia applications).
- for storing "active" procedures which may be executed, for example, when a field is accessed.
- for retaining and using objects from previous schema or instance versions.
- for reducing the "impedence mismatch" between the application programming language and the database programming language.

Object-oriented databases can be a "passive" system, which primarily store data and procedures, or can be an "active" system that acts upon itself by executing procedures within the database system. The Virginia Tech and the GIS models demonstrated a need for interaction among database objects. Therefore, implementing an active database may be of significant importance for these application types.

Object-oriented database systems provide a means to map stored objects directly into the object format used by the application; however, data retrieved from the database must be converted from disk storage format to memory format.

Some of the primary concerns with implementing an object-oriented database are:

- maturity level of the object-oriented database software
(e.g., new and unproven technology)
- better tools are needed to support and manage these systems
- obtaining personnel that are skilled in object-oriented methodology
- conversion costs from a relational system to an object-oriented system

4. Understanding management issues associated with implementing Object-Oriented methodology.

a. Object-Oriented methodology does not replace inadequate planning or structured management techniques.

Object-oriented methods require significant time at the beginning of a project for analyzing the problem. This issue was discussed with an experienced software engineer. He referenced a major software contracting company that discontinued all object-oriented development because of an unsuccessful attempt to implement this technology. The reason for the failure was because the company:

- failed to allocate the necessary time for the planning stage of the project
- implemented a major project without developing the necessary skills to efficiently develop the project with this new technology

The engineer stated the following:

"When less than appropriate time is allocated for analysis and design of an object-oriented system, the result is software that grows rapidly and can become too large for the system that was intended to run the application."

This statement indicates that object-oriented methodology does not substitute or replace inadequate planning and structured management techniques. Object-oriented methodology focuses on planning with the expectation that future systems or enhancement applications will benefit from and build upon this system.

b. To fully benefit from Object-Oriented methodology, management structures may need to be developed.

Managing the development of an object-oriented system requires that the organization which is implementing this methodology recognize: a) the benefits and limitations of the technology, b) that significant time is required to develop the necessary skills, and c) that new organizational structures may need to be created.

The maturity of the organization and its current level of technology must be advanced enough to handle the various management and implementation processes associated with object-oriented methodology.

c. The ability to maintain, adapt, and reuse components of an existing system may provide the competitive edge for many companies that must invest in complex and expensive software development.

Companies are recognizing that one of their biggest assets are existing programs. Therefore, reuse and management of previous project work is of significant importance. Object-oriented systems and tools are directed towards this effort.

One Final Note:

This project was an important experience in this graduate program. It provided the background to support object-oriented programming responsibilities. Some implementation issues have not been addressed in detail; however, overall, the goals of this project were attained.

12.0 References

1. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, "Object-Oriented Modeling and Design," Prentice Hall, Inc., 1991.
2. Won Kim, "Introduction to Object-Oriented Databases," The MIT Press, Inc., 1992.
3. Jeffrey D. Ullman, "Database and Knowledge - Base Systems," Computer Science Press, Inc., 1988.
4. Mark Mullun, "Rapid Prototyping for Object-Oriented Systems," Addison-Wesley Publishing Co., Inc., 1990.
5. Edward Yourdon, "Decline & Fall of the American Programmer," Yourdon Press, Inc., 1993.
6. David A. Taylor, "Object-Oriented Technology," Addison-Wesley Publishing Co., Inc., 1990.
7. C. J. Date, "An Introduction to Database Systems," Addison-Wesley Publishing Co., Inc., 1981.
8. Stanley B. Lippman, "C++ Primer," Addison-Wesley Publishing Co., Inc., 1991.
9. Roderic G. Cattell, "Object Data Management," Addison-Wesley Publishing Co., Inc., 1992.
10. George Wilkie, "Object-Oriented Software Engineering," Addison-Wesley Publishing Co., Inc., 1993.

APPENDIX "A"
VIRGINIA TECH OBJECT MODELS

INDEX COURSE CL. NO. DEPT. NO. HRS. COURSE TITLE INSTRUCTOR DAY TIME

COLLEGE OF ARTS & SCIENCES

COMPUTER SCIENCE AND APPLICATIONS/ MASTERS OF INFORMATION SYSTEMS (M.I.S.)

Students in the Master of Information Systems program (MIS) should check with a CI advisor if they have questions concerning the courses for which they should register. All courses reflected in the Long Range Course Offering schedule, as depicted in the MIS brochure, are being offered and course information reflected under the appropriate instructor.

Master of Science in Computer Science and Applications
 Master of Science in Information Systems
 Dr. William French, Program Director
 Telephone: (703) 686-6228
 Room 308

COMPUTER SCIENCE

684	CI	494	3	Principles of Computer Architecture and Operating Systems	Hubert	TH	6:30-9:15pm
681*	CI	388	1	Tutorial of Computer Science (P/F Only)	Strom	M	6:30-8:30pm
681B	CI	388	1	Tutorial of Computer Science (P/F Only)	Strom	M	8:30-10:30pm
681C	CI	388	1	Tutorial of Computer Science (P/F Only)	Strom	M	10:30-12:30pm
682	CI	320	3	Special Seminars	W		6:30-9:15pm
683	CI	321	3	Special Seminars	W		6:30-9:15pm
684	CI	322	3	Special Seminars	W		6:30-9:15pm
685	CI	323	3	Special Seminars	W		6:30-9:15pm
686	CI	324	3	Special Seminars	W		6:30-9:15pm
687	CI	325	3	Special Seminars	W		6:30-9:15pm
688	CI	326	3	Special Seminars	W		6:30-9:15pm
689	CI	327	3	Special Seminars	W		6:30-9:15pm
690	CI	328	3	Special Seminars	W		6:30-9:15pm
691	CI	329	3	Special Seminars	W		6:30-9:15pm
692	CI	330	3	Special Seminars	W		6:30-9:15pm
693	CI	331	3	Special Seminars	W		6:30-9:15pm
694	CI	332	3	Special Seminars	W		6:30-9:15pm
695	CI	333	3	Special Seminars	W		6:30-9:15pm
696	CI	334	3	Special Seminars	W		6:30-9:15pm
697	CI	335	3	Special Seminars	W		6:30-9:15pm
698	CI	336	3	Special Seminars	W		6:30-9:15pm
699	CI	337	3	Special Seminars	W		6:30-9:15pm
700	CI	338	3	Special Seminars	W		6:30-9:15pm
701	CI	339	3	Special Seminars	W		6:30-9:15pm
702	CI	340	3	Special Seminars	W		6:30-9:15pm
703	CI	341	3	Special Seminars	W		6:30-9:15pm
704	CI	342	3	Special Seminars	W		6:30-9:15pm
705	CI	343	3	Special Seminars	W		6:30-9:15pm
706	CI	344	3	Special Seminars	W		6:30-9:15pm
707	CI	345	3	Special Seminars	W		6:30-9:15pm
708	CI	346	3	Special Seminars	W		6:30-9:15pm
709	CI	347	3	Special Seminars	W		6:30-9:15pm
710	CI	348	3	Special Seminars	W		6:30-9:15pm
711	CI	349	3	Special Seminars	W		6:30-9:15pm
712	CI	350	3	Special Seminars	W		6:30-9:15pm

ECONOMICS

Master of Arts in Economics
 Dr. David Macgregor, Program Director
 Telephone: (703) 686-6228
 Room 308

697	ECON	350	3	Survey of Economic Principles	Thomson	TH	6:30-9:15pm
698	ECON	351	3	Intermediate Theory	Thomson	TH	6:30-9:15pm
699	ECON	352	3	Advanced Theory and Practice	Thomson	TH	6:30-9:15pm
700	ECON	353	3	Topics in Applied Economic Analysis	Thomson	TH	6:30-9:15pm
701	ECON	354	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
702	ECON	355	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
703	ECON	356	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
704	ECON	357	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
705	ECON	358	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
706	ECON	359	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
707	ECON	360	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
708	ECON	361	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
709	ECON	362	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
710	ECON	363	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
711	ECON	364	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm
712	ECON	365	3	Advanced Topics in Economic Analysis	Thomson	TH	6:30-9:15pm

STATISTICS

688	STAT	496	3	Theoretical Statistics	Strom	W	6:30-9:15pm
689	STAT	497	3	Applied Statistics	Strom	W	6:30-9:15pm
690	STAT	498	3	Advanced Statistical Methods	Strom	W	6:30-9:15pm

MATHEMATICS

683	MATH	366	3	Elementary Analysis and Calculus (Cross-list with CI 366 623)	Olson	T	6:30-9:15pm
-----	------	-----	---	--	-------	---	-------------

COLLEGE OF BUSINESS

Master of Business Administration
 Dr. Purcell Chubb, Program Director
 Telephone: (703) 686-6228
 Room 308

ACCOUNTING

682	ACCT	386	3	Fundamentals of Accounting	Perkins	T	6:30-9:15pm
683	ACCT	387	3	Fundamentals of Accounting	Perkins	T	6:30-9:15pm
684	ACCT	388	3	Intermediate Accounting	Perkins	T	6:30-9:15pm
685	ACCT	389	3	Advanced Accounting	Perkins	T	6:30-9:15pm

FINANCE

686	FIN	390	3	Financial Principles I	Medley	T	6:30-9:15pm
687	FIN	391	3	Financial Principles II	Medley	T	6:30-9:15pm
688	FIN	392	3	Financial Analysis and Security	Medley	T	6:30-9:15pm
689	FIN	393	3	Financial Institutions, Markets, and Money	Medley	T	6:30-9:15pm
690	FIN	394	3	Advanced Topics in Finance	Medley	T	6:30-9:15pm

MANAGEMENT

691	MGT	395	3	Social, Legal and Ethical Dimensions of Management	Perkins	W	6:30-9:15pm
692	MGT	396	3	Administrative and Organizational Behavior	Perkins	W	6:30-9:15pm
693	MGT	397	3	Human Resources Management	Perkins	W	6:30-9:15pm
694	MGT	398	3	International Management	Perkins	W	6:30-9:15pm
695	MGT	399	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
696	MGT	400	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
697	MGT	401	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
698	MGT	402	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
699	MGT	403	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
700	MGT	404	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
701	MGT	405	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
702	MGT	406	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
703	MGT	407	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
704	MGT	408	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
705	MGT	409	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
706	MGT	410	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
707	MGT	411	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
708	MGT	412	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
709	MGT	413	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
710	MGT	414	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
711	MGT	415	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm
712	MGT	416	3	Advanced Topics in Management	Perkins	W	6:30-9:15pm

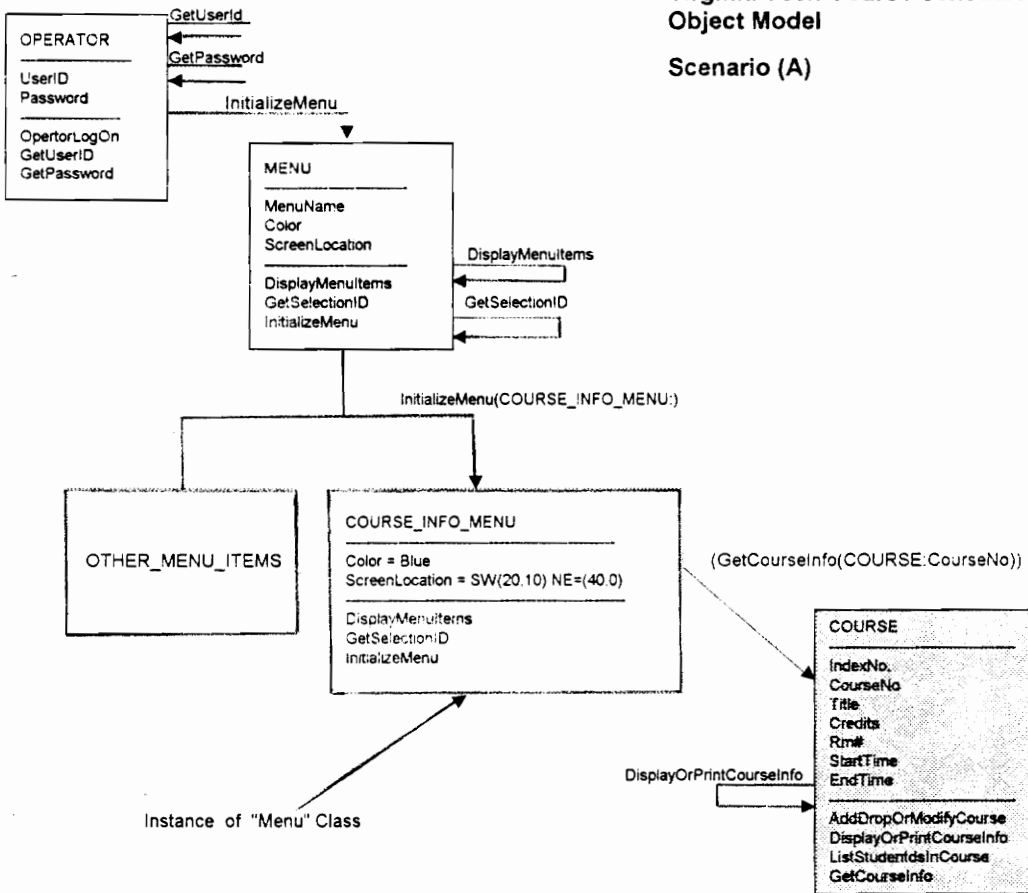
MANAGEMENT SCIENCE

680	MSC	416	3	Management Science	Strom	TH	6:30-9:15pm
681	MSC	417	3	Operations Management: Production and Inventory Control	Strom	TH	6:30-9:15pm
682	MSC	418	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
683	MSC	419	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
684	MSC	420	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
685	MSC	421	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
686	MSC	422	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
687	MSC	423	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
688	MSC	424	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
689	MSC	425	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
690	MSC	426	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
691	MSC	427	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
692	MSC	428	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
693	MSC	429	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
694	MSC	430	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
695	MSC	431	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
696	MSC	432	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
697	MSC	433	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
698	MSC	434	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
699	MSC	435	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
700	MSC	436	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
701	MSC	437	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
702	MSC	438	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
703	MSC	439	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
704	MSC	440	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
705	MSC	441	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
706	MSC	442	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
707	MSC	443	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
708	MSC	444	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
709	MSC	445	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
710	MSC	446	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
711	MSC	447	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm
712	MSC	448	3	Advanced Topics in Management Science	Strom	TH	6:30-9:15pm

MARKETING

681	MKTG	416	3	Marketing Principles and Strategy	Strom	M	6:30-9:15pm
682	MKTG	417	3	Research for Marketing Decisions	Strom	M	6:30-9:15pm
683	MKTG	418	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
684	MKTG	419	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
685	MKTG	420	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
686	MKTG	421	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
687	MKTG	422	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
688	MKTG	423	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
689	MKTG	424	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
690	MKTG	425	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
691	MKTG	426	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
692	MKTG	427	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
693	MKTG	428	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
694	MKTG	429	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
695	MKTG	430	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
696	MKTG	431	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
697	MKTG	432	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
698	MKTG	433	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
699	MKTG	434	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
700	MKTG	435	3	Advanced Topics in Marketing	Strom	M	6:30-9:15pm
701	MKTG	436	3	Advanced Topics in Marketing	Strom		

Virginia Tech Course Schedule
Object Model
Scenario (A)



DATABASE QUERY:

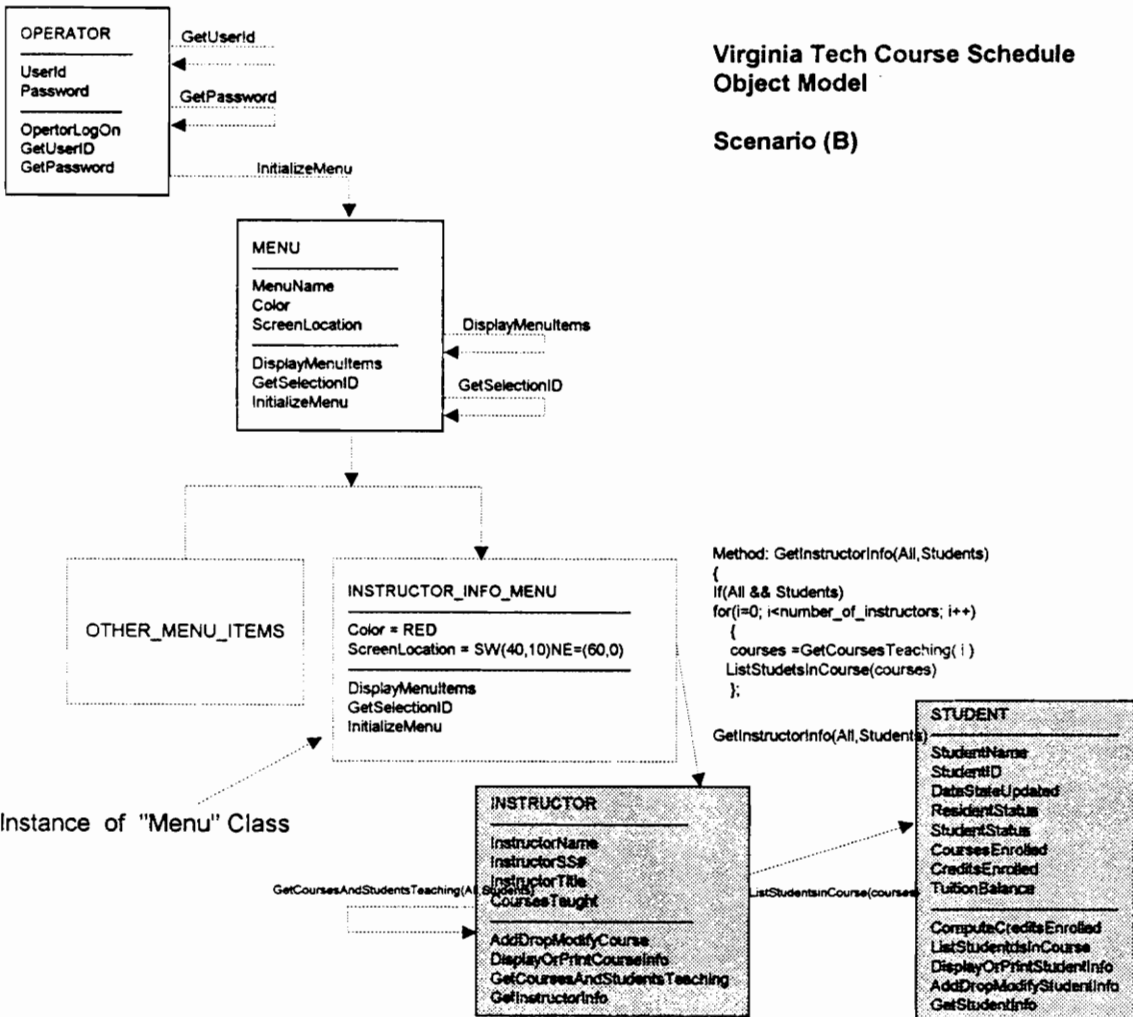
```
(Select IndexNo, Title, Credits, Rm#, StartTime, EndTime
from Course
where CourseNo=1234)
```

Processing Description:

1. Operator Logs On
2. Message sent to OPERATOR Class to Get Userid
3. Message sent to OPERATOR Class to Get Password
4. Message sent to MENU Class to Initialize Menu
5. Message sent to MENU Class to Display Menu Items
6. Message sent to MENU Class to Get the Menu Selection
7. Message sent to Initialize Course Info Menu.
8. Message sent COURSE Class to Get Course Information
9. Message sent to COURSE Class to Display or Print Course Information

Virginia Tech Course Schedule Object Model

Scenario (B)



DATABASE QUERY:

```

(Select StudentName from STUDENT
 where Course =
 (Select CoursesTaught from INSTRUCTOR
  where InstructorName = 'ALL')
    
```

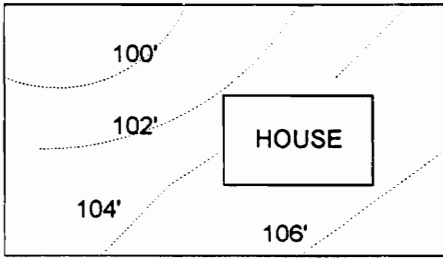
Processing Description:

1. Operator Logs On
2. Message sent to OPERATOR Class to Get UserID
3. Message sent to OPERATOR Class to Get Password
4. Message sent to MENU Class to Initialize Menu
5. Message sent to MENU Class to Display Menu Items
6. Message sent to MENU Class to Get Menu Selection
7. Message sent to INSTRUCTOR class to Get Instructor Info
8. Message sent to INSTRUCTOR class to Get Courses taught for an Instructor
9. Message sent to STUDENT class to List the students in course
10. Steps 8 & 9 are repeated.

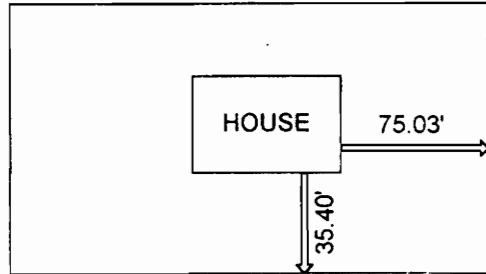
APPENDIX "B"

**GEOGRAPHIC INFORMATION SYSTEM
OBJECT MODELS**

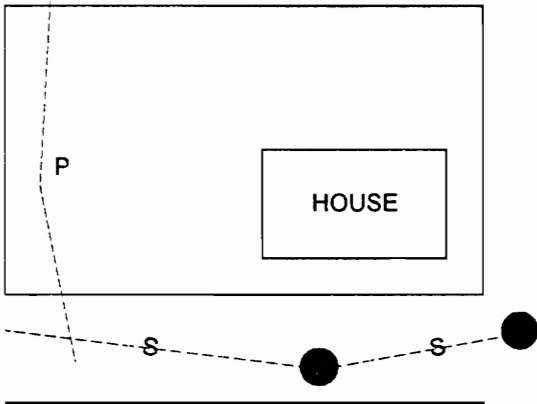
COMPUTE AND DISPLAY CONTOURS



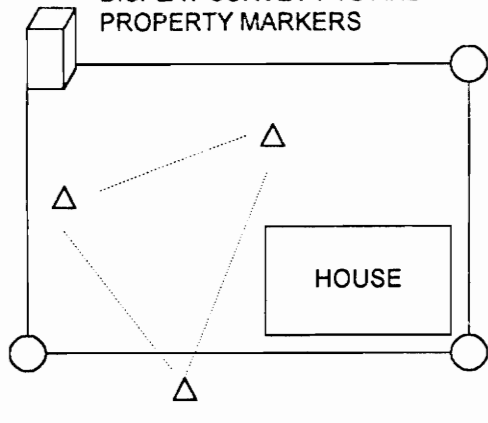
COMPUTE AND DISPLAY OFFSETS



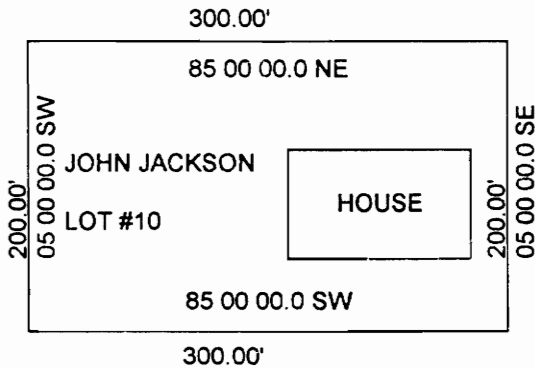
DISPLAY POWER AND SEWER LINES



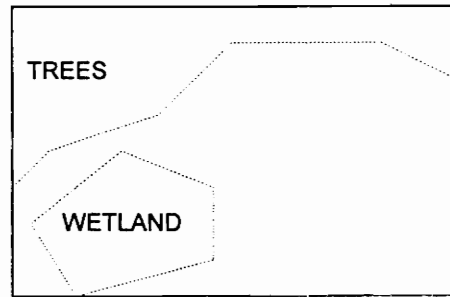
DISPLAY SURVEY PTS AND PROPERTY MARKERS



DISPLAY PROPERTY BOUNDARIES AND OWNER



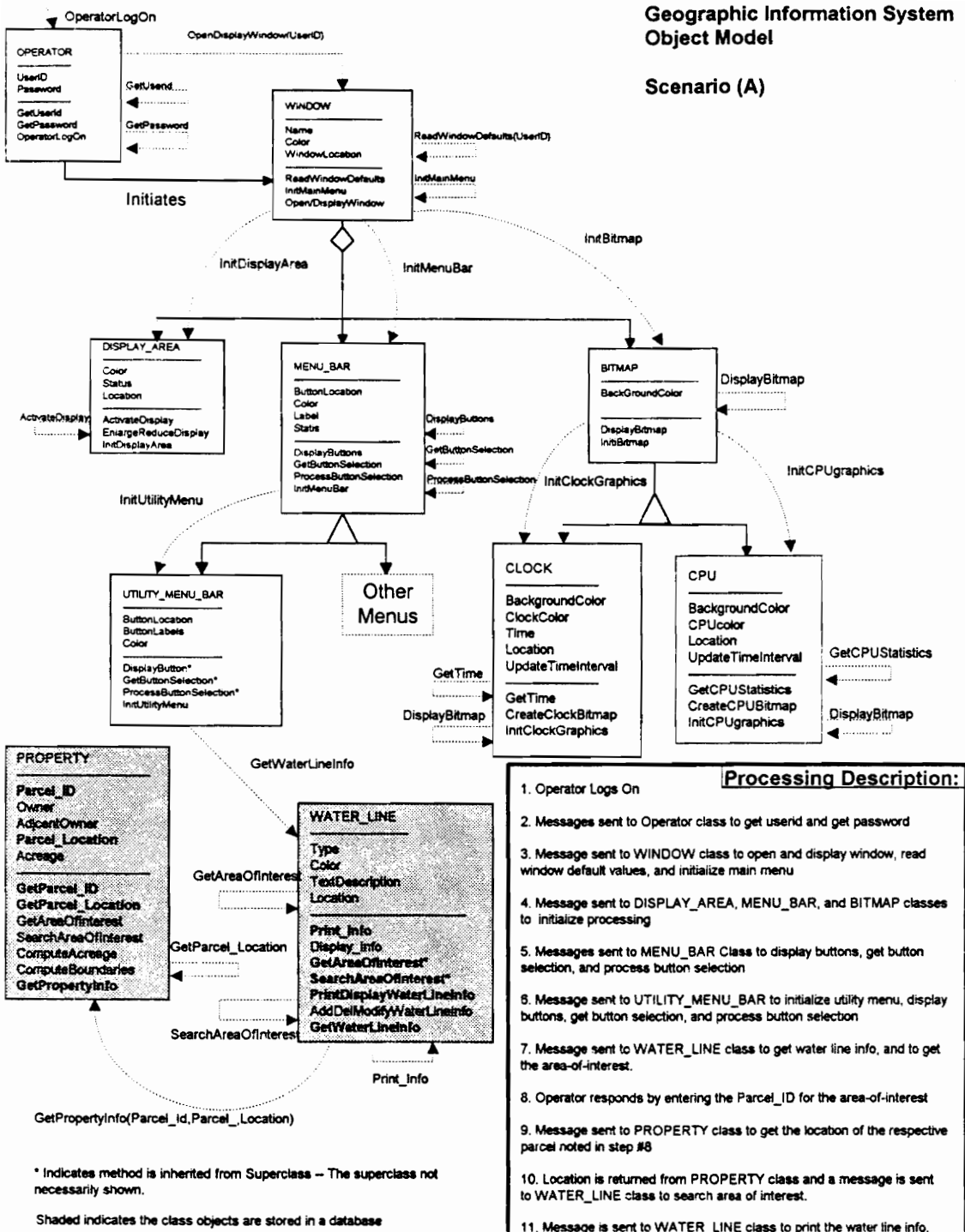
DISPLAY TREELINES AND WETLANDS



ENGINEERING DISPLAYS

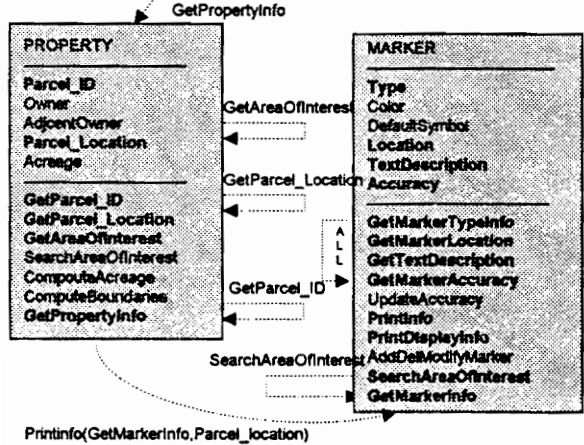
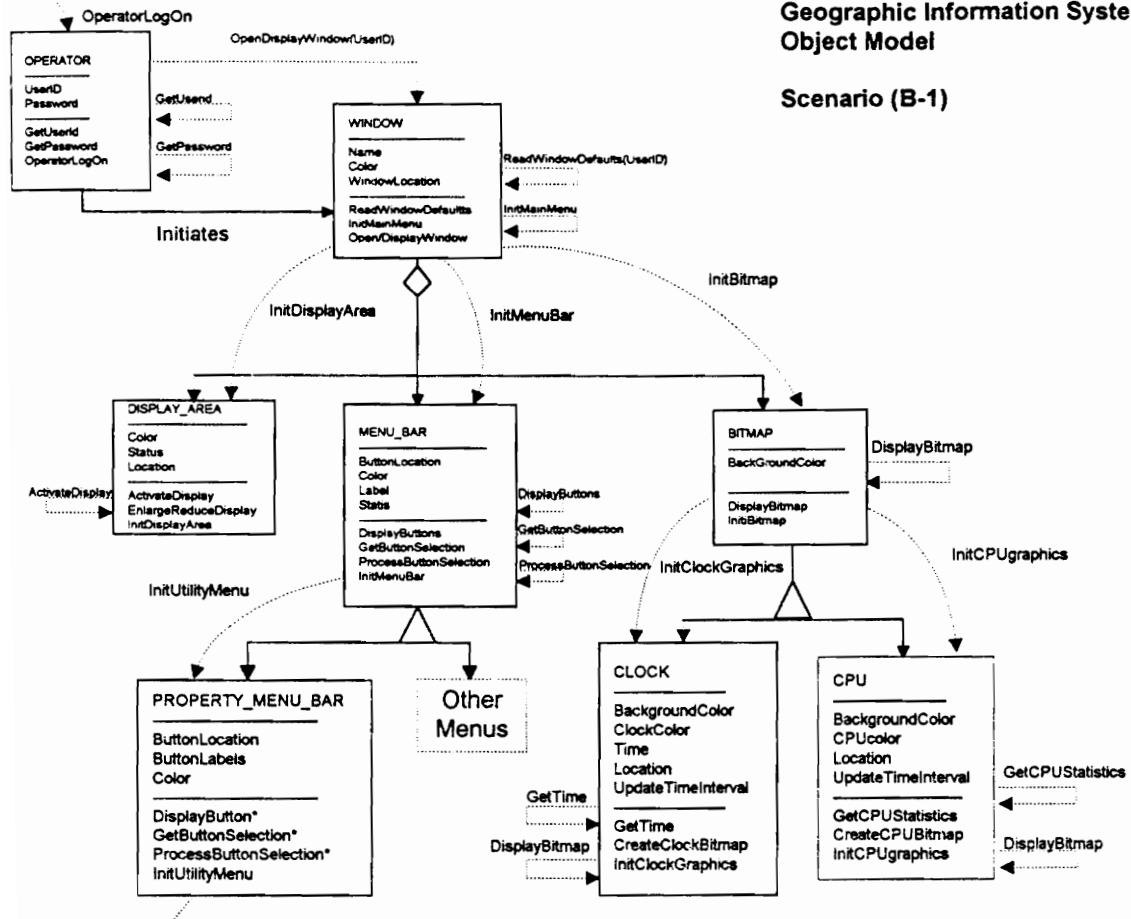
Geographic Information System Object Model

Scenario (A)



Geographic Information System Object Model

Scenario (B-1)



Processing Description:

1. Operator Logs On
2. Messages sent to Operator class to get userid and get password
3. Messages sent to WINDOW class to open and display window, read window defaults, and initialize main window
4. Message sent to the DISPLAY_AREA, MENU_BAR, and BITMAP classes to initialize processing.
5. Messages sent to MENU_BAR Class to display buttons, get button selection, and process button selection
6. Message sent to PROPERTY_MENU_BAR to initialize PROPERTY_MENU_BAR menu, display buttons, get button selection, and process button selection
7. Message sent to PROPERTY class to get property information, to get the area-of-interest, and to get the Parcel_ID.
8. Operator responds by entering the Parcel_ID for the area-of-interest
9. Message sent to PROPERTY class to get the location of the respective parcel identified in step #8.
10. Location is returned from PROPERTY class and a message is sent to the MARKER class to get and print marker information.

* Indicates method is inherited from Superclass – The superclass not necessarily shown.
 Shaded indicates the class objects are stored in a database

