THE UNIVERSITY OF CALGARY

AN OBJECT-ORIENTED PROTOTYPE FOR AN ENVIRONMENTAL GIS OF THE CREOSOTE SITE ON THE BOW RIVER, CALGARY

by

CHAO ZHENG

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF GEOMATICS ENGINEERING

CALGARY, ALBERTA

JUNE, 1996

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "An Object-oriented Prototype for an Environmental GIS of the Creosote Site on the Bow River, Calgary" submitted by Chao Zheng in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. J. A. R. Blais
Department of Geomatics Engineering

Dr. R. Li
Department of Geomatics Engineering

Dr. R. C. K. Wong
Department of Civil Engineering

Date: June 14, 1996

ABSTRACT

The relational data model is the most widely applied logical model in current commercial GIS. Due to the increasing complexities of spatial information, this model is begining to show its inadequacies in the areas of data handling efficiency, data semantics, model extensibility and programming interface, etc. Object-oriented modeling, characterized by its concept of class which encapsulates both state and function features, exhibits great potential in modeling complex information. Using object-oriented technology, the positional and other attribute components, as well as fundamental operation functions of a geospatial entity can be modeled in an integrated way as a single object. Groups of classes describing geospatial features at a high level of abstraction can be implemented to form a generic modeling kernel. Support of class inheritance and complex classes offers a natural and efficient mechanism to extend the kernel to specific applications. Thus high efficiency and flexibility in model construction and extension can be obtained. In this project, the use of object-oriented methodology in spatial data modeling has been studied. First, a general object-oriented spatial data model of a high level of abstraction has been designed and implemented as the kernel of a GIS application prototype. An object-oriented analysis and design method, the Booch Method [Booch, 1995], is used for design and Borland C++ is used to implement the spatial data model. Then, an application module is implemented using this prototype to model and manage the DEM data sets obtained in the creosote project [Blais et al. 1995] on the Bow River in Calgary. General conclusions and recommendations are included.

ACKNOWLEDGMENTS

I wish to express my deep gratitude to my supervisor, Dr. J.A.R. Blais, for his consistent guidance, assistance, and encouragement throughout my graduate studies. Appreciation should also go to my fellow graduate students, Mr. K. He and Dr. C. Larouche for their cooperation during my thesis research.

This research has been supported by the University of Calgary, the City of Calgary, Alberta Environment and the Natural Science and Engineering Research Council. These contributions are greatly acknowledged.

I am also grateful to Imperial Oil Ltd. and to Mr. Myron Story for allowing me to finalize the thesis writing while working as a full time employee in Imperial Oil Ltd., Information Services Department.

Dedication

To my parents, my wife, and my new baby !

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Relational Model in GIS

A Geographic or Geospatial Information System (GIS) is a system of computer hardware, software and procedures designed to support the capture, management, analysis and display of spatially referenced data for solving complex planning and management problems. What type of database management mechanism is used to model and manipulate spatial information is the key contributing factor to system operating efficiency and model extensibility in a GIS implementation.

Currently, a relational database management system (RDBMS) is the dominant DBMS used in commercial GISs. RDBMS is known for its maturity and success in handling ordinary thematic information. But as the development of GIS produced spatial information much more complex than ordinary thematic information. RDBMS began to show insufficiency in data handling efficiency, model extensibility, data semantics, program interface, etc. To solve these problems, a number of researchers have proposed an extended relational system with more complex data structures and manipulation functions added into the traditional relational paradigm to handle spatial information. Representative implementations include Postgres system [Rowe and Stonebraker. 1987]

and SIRO-DBMS [Milne 1993, Abel 1989]. These methods, while improving the system, still did not adequately resolve the problems.

1.2 Object Orientation

Another proposed solution is an object-oriented GIS. Object-oriented technology was first discussed in late 1960s by researchers working with the SIMULA language. In 1970s, the widely used object-oriented programming language, Smalltalk language was developed at Xerox PARC, recording a milestone in the development of object orientation [Goldberg 1983]. Interest in Smalltalk was limited, however, due to separate interest in structural programming design at the time. After decades of maturing, object orientation began to exhibit higher efficiency in dealing with the increasing complexity of modern software systems over traditional methodologies, and gradually became popular not only in programming fields but also in database, system analysis and system design fields. Strong driving forces from academy and industry have brought more and more mature object-oriented systems and methodologies in the 1990s.

The power of the object-oriented methodology stems mainly from the fact that it combines recognized software engineering principles which promote code reuse and system extendibility [Jacobson et al. 1992]. Central to object orientation is the concept of class, a software abstraction that includes both state and behavior features, which are represented by data structures and functions, respectively. The inclusion, called encapsulation in object orientation, ensures a class to represent an application entity by

keeping all data pertaining to the application entity bundled together with all the functionality applied to it. Computational models based on class are more direct and more efficient when compared with the separation of the data and functions in conventional strategies. Other important mechanisms contributing to the high modeling efficiency include inheritance, complex object, communication by message passing, etc. More details about these features in the context of GIS applications will be discussed in Chapter 2.

1.3 Object-Oriented Database

Object-oriented database technology combines the representative power and flexibility of object orientation with the capabilities of database management systems. The first generation of object-oriented database dates back to 1986 when G-Base was launched by the French company, Graphael. In 1987, Servio Corp. introduced Gemstone; in 1988, Ontologic developed Vbase, and Symbolics developed Statice. Most of these first-stage systems served limited use in research departments of large companies. The launch of Ontos in 1989 marked the start of the second stage in this field. Third generation products such as Itasca and O2 [Bancilhon et al. 1992] were developed in the 1990s. These products can be defined as database management system (DBMS) with DDL/DML which are object-oriented and computationally complete [Bertino and Martino. 1993].

Generally speaking, object-oriented database is still in its experimental stage, but its development has already had big impacts in fields which need to deal with complex and large data sets of inordinate volume where existing database technologies have shown inadequacies. Examples of these fields include computer-aided software engineering (CASE), computer-aided design (CAD), computer-aided manufacturing (CAM) and GIS.

1.4 Object Orientation in GIS

Object-oriented concepts were introduced in the context of GIS in the 1980s [Egenhofer and Frank. 1987, Worboys et al. 1990], and researchers gradually realized that GIS could benefit greatly from the object-oriented database technology, particularly in terms of architecture, performance, and ease of use. Thus in the past few years, a large amount of research have been devoted to incorporating object-oriented technology in GIS, some of the research aimed at developing an object-oriented GIS (OOGIS) directly, but most relates to building OOGISs based on available OODBMSs. Noteworthy results include Integraph's TIGRIS [Herring 1992, 1987], AccuMap [Lukatela 1989], Smallworld GIS [Newell 1992], Geo2 [David et al. 1993] and the GIS built on ONTOS by CSIRO Division of Information Technology, Australia [Milne 1993]. The Australian group also went further and performed a comparison among ORACLE, SIRO-DBMS (an extended relational DBMS with special GIS capability [Abel 1989] implemented on ORACLE) and their extended ONTOS. An impressive performance gain was reported

[Milne 1993]. Such research, though significant, proved the need for further study into aspects such as more efficient physical models for spatial information, optimal query languages, etc, before a commercially viable object-oriented GIS could result from the experimental systems.

1.5 Research Objectives

The purpose of this study is to investigate the applicability of object orientation in GIS. Considering that most research on object-oriented GIS is aimed at a general purpose system, it might also be very useful to develop an object-oriented prototype which contains general spatial data modeling components together with basic supporting functions. For some environmental or other applications which only need to deal with certain types of data and only need certain functions, these can be developed with special data types and corresponding functions implemented using the prototype at a more affordable size. Moreover, this prototype can also be a starting point for a general object-oriented GIS with more sophisticated storage, indexing support and more comprehensive function designs.

The basic idea is first, using object-oriented analysis and design (OOA and OOD) methods, to model the geometric and feature information of spatial entities in an integrated way as a single class (object), with the fundamental supporting functions embedded. A generic kernel is implemented in Borland C++ 4.5, composed of a group of classes describing geographic features at a high level of abstraction. Then, using this

modeling fundamental, an application is implemented to handle a set of Digital Elevation Model (DEM) data obtained from the creosote project [Blais et al. 1995] on the Bow River, Calgary. The implementation includes data conversion and storage, information retrieval and browsing, etc. Based on these experiments, conclusions and recommendations on applying object-oriented technology, such as OOA, OOD and object-oriented programming, can be formulated.

1.6 Thesis Outline

The background, principle, methodology, design and implementation of building an object-oriented prototype for an environmental GIS of the creosote site on the Bow River will be discussed in the following chapters.

Chapter 2 discusses the background of applying object orientation in GIS. The contents include: analysis of the limitation of the traditional logical model in spatial information modeling, discussion of GIS architectures, fundamentals of object-oriented technology and its advantages over traditional strategies in GIS applications.

Chapter 3 introduces basic ideas of an object-oriented analysis and design methodology, the Booch method, followed by how to use this method to analyze, design and how to use Borland C++ to implement an object-oriented spatial information modeling prototype. Discussions of what should be contained in an object-oriented spatial information modeling prototype is also included.

Chapter 4 begins with some background on the creosote project and the data set obtained in this project. Then the discussion concentrates on how to use the modeling prototype to build up an application system to handle the DEM data set.

Chapter 5 includes the main conclusions and recommendations obtained from this project.

Chapter 2

SPATIAL DATA AND SPATIAL DATA MODELING

Currently, most of the commercial Geographic Information Systems use a Relational Database Management System (RDBMS), such as Oracle or Ingres, as the data manager. The RDBMS is known for its maturity and success in handling ordinary thematic information. As the spatial information is much more complex than the ordinary thematic information, higher requirements for a data handling mechanism are necessary.

2.1 Characteristics of Spatial Information

When a spatial object is studied, besides its thematic features, several other dimensions can be of great concern as well [Worboys 1994b]. These may include spatial, graphical, temporal and textual/numeric dimensions (Figure 2.1). All these features need to be considered as a whole while modeling spatial information.

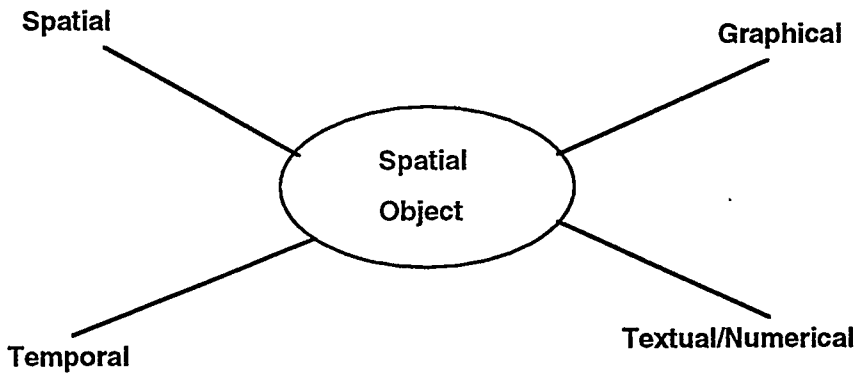

Figure 2.1 Dimensions of spatial object.

2.1.1 Spatial domain

The specification of a spatial object depends upon the real-world space in which the object is situated. In GIS applications, the absolute and relative positional information of spatial entities are the most fundamental concern. The variations of other aspects are studied with respect to position so as to be within a common reference space. Thus the spatial dimension has always been the major focus of activity for GIS research. Among all the position related features, topological relationships are the most important ones. Because modeling topological information, such as enclosure and adjacency, requires more powerful model construction capability which is beyond what is provided by traditional models, it is one of the most challenging constraints for traditional logical models. The problem is mainly due to the limitation of the data semantics and model extension ability supported by the traditional models [Lee 1990]. This will be illustrated in the following section with more detailed analysis.

2.1.2 Graphical domain

Graphical domain mainly means the representation form of spatial objects in the cartography or visualization aspect of a GIS [Worboys 1994b]. The graphical domain focuses on a spatial objects' existence in presentation, while spatial domain is mainly concerned about spatial objects' existence in an application. This distinction was not clearly made until recently when more sophisticated visualization functions were applied and more advanced techniques were demanded in GIS. Graphical domain modeling, as

well as the transformation from the spatial domain to graphical domain are the main concern in this field [Veldon et al. 1990].

2.1.3 Temporality

Temporality is an inherent aspect of spatial information. Time in information systems is measured along at least two separate axes. One is database time, the time when transactions take place within an information system. Another is event time, which refers to when the events actually occur in the application domain. Currently, according to which methods and abilities to represent temporal information, there are four types of temporal information organizations: **static**, **static rollback**, **historical** and true **temporal database** [Snodgrass 1992]. **Static systems** support neither database nor event time; **static rollback systems** support database time; historical systems support only event time; while **temporal systems** support both database time and event time [Snodgrass 1992]. Only recently has the research towards a true temporal system been done even in general Database Management Systems (DBMS). This is mainly because of the inadequacy of the technological support, in terms of both hardware and software [Worboys 1994a]. Now more and more researchers in the GIS field are aiming at a GIS able to support temporal as well as spatial aspects of geographical information. The speed and capacity of hardware, along with the software that is now becoming available, making temporal information systems possible. But efficient temporal information logical modeling is still the "bottle neck" [Muller 1993, Tansel et al. 1993] in such systems.

It is quite common that any set of spatial information can have all of the above mentioned features and thus its modeling requires consideration of all the domains. For example, a model of a national park may have a polygon representing its boundary in the real world (spatial), polygons, points and arcs representing its cartographic form at differing levels of generalization (graphical), times when it was created in the real world and in the system (temporal), and attributes describing its area and name (textual/numeric). Modeling approaches must be able to represent all of them in an integrated format.

2.2 Traditional Logical Model

As an important component, the DBMS designed for GIS serves two major functions: to provide efficient management and flexible manipulation of data. An efficient spatial database minimizes storage and maximizes processing speeds, while a flexible one provides complete support mechanisms to express the user's view of how objects are organized in the real world. In this sense, flexibility is measured by the expressive power of the data model. At the core of the spatial database are data structures for spatial entities and relationships. Data structures that are flexible and efficient tend to be complex and difficult for the general user to understand and manipulate. It is sometimes necessary for the database designer to hide the actual data structure, called the physical model, with a logical model which highlights flexibility and ease of use. The

user interacts directly with the logical model, and the database management system will automatically map the operations to the physical model.

These logical models have actually been implemented in existing database management systems and are also called database models, or implementation models. The evolution of logical models has been through the hierarchical model, the network model and the now popular relational model [Vossen 1991]. Currently, the relational model is the most popularly applied one in GIS, but studies show that all these three models have drawbacks in handling spatial information.

2.2.1 Hierarchical and Network Models

In the hierarchical model, the underlying logical structure is hierarchical or, more formally, a tree. The tree structure is recursively defined as a collection of nodes *T*. The collection can be empty, or consist of a distinguished node *r*, called the *root*, and zero or more (sub)trees *T1, T2, ..., Tk*, each of whose roots are connected by a direct *edge* to *r* [Smith et al. 1987]. In the tree, nodes represent record types and the edges represent the parent-child relationships between the ancestor and descendant nodes, respectively. Figure 2.2 shows a hierarchy among five record types.

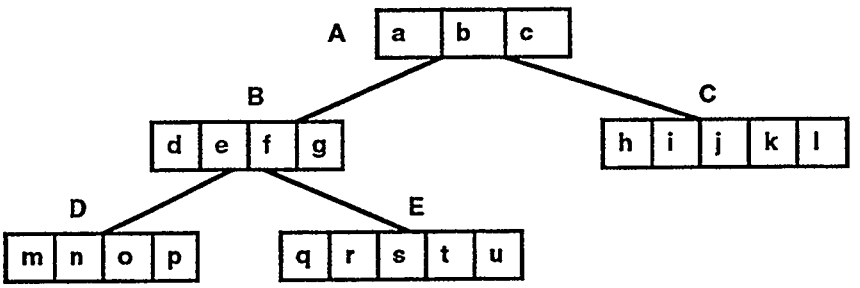

Figure 2.2 A hierarchical schema among five record types.

The tree structure implies an N-to-one, written N:1, mapping from children to parent instances. Thus one-to-one and many-to-many strongly connected relationships, which prevail in spatial information, can be represented directly by hierarchies. The advantages of this model include data access of natural tree traversal type can be straightforward and fast, and the system is easy to extend. However, one has to duplicate trees to represent a many-to-many relationship, which is required quite often in topological information modeling, and huge redundancies can be caused. Thus this model is not practical for spatial data modeling.

The network data model can be described, in the graph theoretical sense, as a graph having no cycles (with the exception of self links), and a collection of record types connected by a set of links to reflect the relationships between record types [Smith and Barnes. 1987]. Figure 2.3 depicts a network schema.

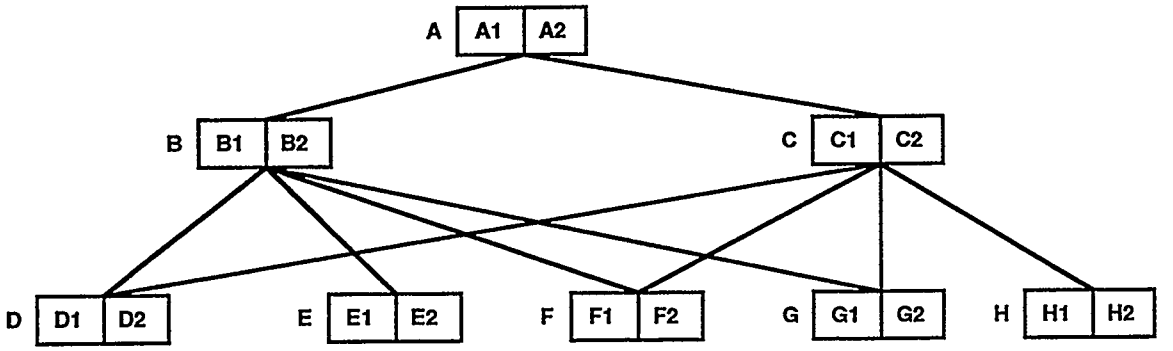

Figure 2.3 A network schema among eight record types.

The network model is more flexible than the hierarchical model. Using sets, the network model can represent diverse types of associations among record types. It can implement M:N associations without large redundancies. But in its implementation, this model uses a lot of pointers to maintain the direct relationships between record types. Besides the storage space needed for the pointers, the system of pointers is very complex and difficult to maintain, which makes data manipulation very complicated [Smith and Barnes. 1987].

2.2.2 Relational Model and Limitation

A relational DBMS presents to the user a logical model of the database in terms of tables, or relations. Part of the flexibility of the relational model stems from the organization of data into homogeneous units called tuples forming rows of tables. Simplicity and data independence are the major features of a relational DBMS, but can also cause deficiencies in the following sense for spatial information modeling:

Manipulation Efficiency

Efficiency refers to the speed by which data can be processed. The primary reason why a relational database is slow is due to the requirement that all relations must be in the first Normal Formula (1NF): the domains of attributes must only include atomic values (simple, indivisible) and the value of any attribute in a tuple must be a single value taken from the domain of that attribute [Smith and Barnes. 1987]. The 1NF forces a set (such as a spaghetti consisting of several arcs) to be decomposed into its elements (arcs) before being stored in tables. The 1NF assumption is fundamental to relational models, and the separation of related data into different tables is dictated by rules for good design on relational databases, mainly for integrity control. As a result, some complete geometric information can be retrieved only by addressing multiple tables. This needs to involve a *JOIN* operation which physically combines tuples from different relations through their common values. The operations demand a lot of storage space and computation time, thus slow down the process.

Data Semantics

Data semantics refer to the data model's capability to express the meanings of the data attributes and relationships. Such ability is particularly important for providing integrity constraints and efficient database browsing. A data model rich in semantics must distinguish between different types of relationships of classification, generalization and aggregation [Lee 1990]. The result of classification is a class of similar types.

Generalization is the process of generalizing several types with common properties into a more abstract super type. This process hides the differences between several classes and highlights their similarities. An aggregation can model composite entities from their components.

The different types of relationships required in GIS exceed the above three. But the relational model cannot distinguish all of them because it provides only two constructs for representing relationships, one within a table and the other across tables through common values. This leads to a phenomenon called semantic overloading [Hull and King. 1987] indicating that a single construct has to support several types of relationships thus causing an ambiguity in meaning.

Model Extension

A data model that provides a limited number of data types and cannot support the creation of new types by the users lacks model extension capabilities. In a relational database, only several primitive data types such as integers, real numbers and character strings are supported. Above them are the relations, the only type-like construction a user can define. But relations are not true types and cannot be used in the same way as the built-in types. For example, a relation can reference an integer number but cannot reference another relation.

This deficiency is directly related to the 1NF assumption because 1NF does not allow nested relations. So the modeling for complex objects can be very complicated. For an example, aggregations can only be expressed in the form of SQL query such as " *SELECT lakes, path FROM National_Park where Park_name='Banff'* ". But this scheme usually requires the user to possess complete knowledge of the relations in order to form the queries, which is often a difficult task for general users.

Program Interface

Another problem faced by relational DBMSs is impedance mismatch [Bancihon 1988], that is, the difference between the type system of the programming language and the type system of DBMS Data Definition Language (DDL) and Data Manipulation Language (DML). When the user of a database is a piece of software which accesses and manipulates data through a program interface provided by the DBMS developer, the gap between the two systems can hamper smooth interaction. The discrepancies are caused by two major reasons.

First, a programming language is procedure oriented, whereas the DDL and DML are entity oriented. One system is designed for defining the behavior of entities through procedures while the other is for recording the state of entities through the use of data. As a result, programming languages in general lack the capability to maintain data items residing in permanent storage. Conversely, database languages such as SQL cannot be used for general programming. There are pre-compilers that allow a procedural language

to embed SQL commands in its source code, such as Pro*C in ORACLE, which enable the program to access a relational database. But this method only produces a hybrid environment, however, and does not truly integrate the two approaches.

Secondly, a programming language (before Object-Oriented programming) supports more general and primitive data types such as integers, whereas a database language supports richer and more complex data types such as date type or data set. This reflects their differences in the degree of specialization and, as a result, likely produces problems during the transfer of data from one environment to another. For example, the transfer of records from a program to relation tables often involves considerable changes in data structure.

2.3 GIS Architectures

As the above analysis shows, general purpose DBMSs do not have direct support for geometric attribute types (e.g., point, polylines, polygons) and operators (e.g., distance, intersection, circumference, area). Multidimensional access methods and index mechanisms are not directly supported either. It is impossible to store geographic data in a natural manner, or to pose queries such as: *" Select all lakes with an area of 1,000 square metres that are located within 1 kilometre from a path "*. So currently, extra modules have been added into GIS to enhance its capabilities to efficiently handle spatial information. This has led to three main different types of GIS: dual architectures, layered architectures, and integrated architectures [Vijlbrief and van Oosterom. 1992].

2.2.1 Dual architectures

The most common and straightforward type of commercial GIS architecture is the dual one. The basic idea of this architecture is that besides embedding a relational DBMS to handle the thematic information, another separate subsystem is included to store and retrieve geometric information. These dual architectures are easy to implement but not efficient in terms of performance. The thematic and geometric components of a certain spatial object are stored in two separate subsystems and linked by a common identifier. In order to retrieve an object, the two subsystems have to be queried and the answer has to be composed. Figure 2.4 illustrates one dual GIS architecture. Typical examples of GIS with dual architectures are ARC/INFO of ESRI and MGE of Intergraph.

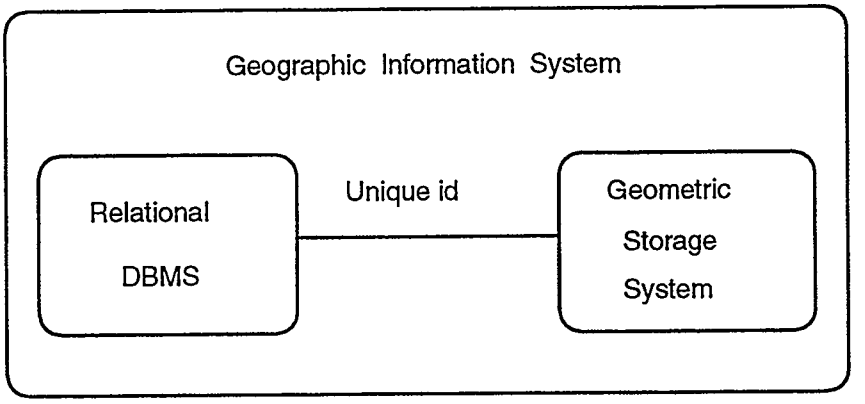

Figure 2.4 Dual GIS architecture.

Because the dual architecture has direct support of a standard DBMS, the storage and retrieval of attribute data can be very efficient. However, this architecture has some severe drawbacks directly caused by the existence of two different storage mechanisms.

First, the integrity constraints of the system can be violated. For example, an entity's geometric information can still exist in the geometry storage subsystem while its attribute information has been deleted from the relational DBMS. Second, query optimization is impossible to the extent of the whole system. Third, the currency control is difficult, because the two storage managers have their own locking protocols.

2.2.2 Layered architectures

Because the drawbacks of dual architectures are directly caused by the coexistence of two different kinds of data managing mechanisms, another kind of architecture, layered architectures were proposed in GIS. Instead of two data managers, only relational data models are adapted in GIS to store the spatial data [van Oosterom and van Den Bos. 1989]. In order to fit the complexity of spatial data into the relational model, a spatial support layer is added on top of the standard relational database (Figure 2.5). The responsibilities of this layer include:

1. Converting the spatial information into the elementary data types of RDBMS.

2. Translating spatial information queries into standard SQL queries.

3. Implementing spatial indexes. These indexes are usually implemented by means of auxiliary relations that contain the required index data.

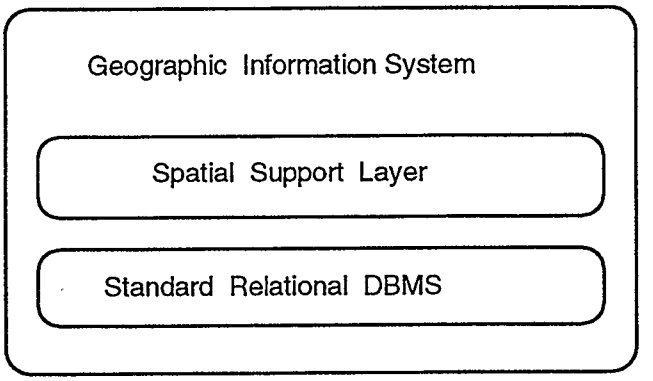

Figure 2.5 Layered GIS architecture.

In this way, the support for transaction semantics and integrity constraints is restored and users can be freed from formulating difficult queries by the help of the additional layer. However, since the coherent geographic information has to be broken into its most primitive parts to be stored in separate tables, retrieval of the original geographic entities has to be done by joining relations, which may greatly effect the efficiency of the system because joining operations are the most time and space consuming operations in RDBMS. Another drawback of this methodology is that the spatial indexes are usually implemented by means of auxiliary relations which contain the required index data. This can speed up spatial access, but the queries become even more complicated due to the additional use of the auxiliary relations. This indirect implementation of an access method is less sufficient than a direct implementation in the DBMS kernel. System 9 [Unisys 1994] from Unisys and SIRO-DBMS [Abel 1989] from CSRIO Australia are characteristic examples of layered architecture systems.

2.2.3 Integrated architectures

The inconvenience and inefficiency mapping from the complex spatial objects to traditional data types is a fundamental problem of GIS architectures. To avoid this, now more and more researchers are working on integrated architecture GIS, a kind of system with direct support for more attribute types and access methods. Based on these supports, users may extend the DBMS with their own basic abstract data types. Of course, for this purpose, extra efforts are needed to implement additional data types and access methods within the DBMS environment, which can soon become quite complicated. However, once this task has been performed, the advantages of this approach are great. The implementation of the data model becomes easy due to the availability of appropriate geometric types. The formulation of spatial queries becomes straightforward and efficient because of the direct support of extensible query language by means of adding more spatial operators such as distance, area, and intersection.

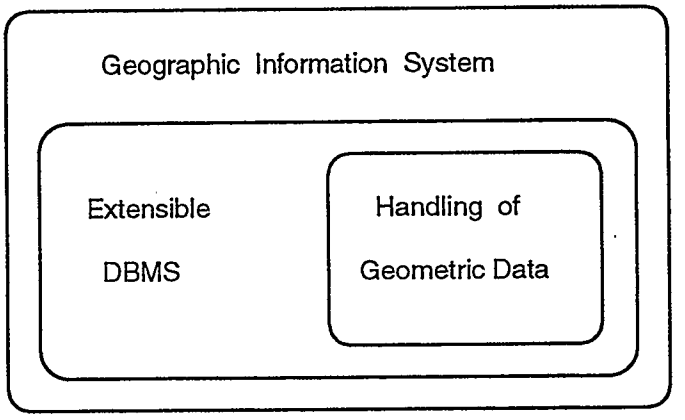

Figure 2.6 Integrated GIS architecture.

The development of integrated GIS architectures mainly depends on the availability of open DB's. According to the above analysis, this architecture cannot be feasibly based on a standard relational DBMS. The object-oriented paradigm, which is characterized by its support of users own defined types, makes a more open system. It has shown great advantages in model extension and flexibility over traditional paradigms. Most of research activities in this stream are focused on adapting the object-oriented model in GIS, trying to achieve working object-oriented GIS systems. Characteristic examples of integrated GIS architecture are TIGRIS [Herring 1987] from Intergraph and the research oriented system GEO++ [Vijlbrief and van Oosterom. 1992].

2.4 Object-Oriented Models

With the integration of database technology with the object-oriented paradigm, a new type of DBMS, the object-oriented DBMS (OODBMS) began to emerge in 1986 (G-Base). Unlike the relational system that is characterized by its maturity based on the sound systematic theory and firm mathematical foundation [Maier 1982], the object-oriented model has no standards yet, implying no common model as reference, no formal foundation for the concepts. However, different object-oriented models or systems share similarities in the fundamental supporting concepts and features, such as class, object, encapsulation, inheritance, etc [Garvey and Jackson. 1989]. In this section, fundamental object orientation concepts and features will be described, together with the features of an

object-oriented database model. Then the advantages of object-oriented systems for spatial information modeling will be summarized.

2.4.1 Object orientation and object-oriented logical models

Object-oriented methodology originated in the computer programming field. Many of the ideas come from the SIMULA language, but this method only became popular later as a result of the introduction of Smalltalk. The key to object-oriented programming is to consider a program as being composed of independent objects, grouped into classes, which communicate with each other by means of messages. The concepts of class, object, encapsulation, polymorphism and inheritance are the fundamental elements of object orientation.

For the integration of database technology with the object-oriented paradigm, current OODBMSs are still at an experimental stage because of technical and commercial complications. Though the available OODBMS show different features due to the lack of standard object-oriented database specifications, they also exhibit a type of common database model which consists of certain generally accepted concepts with combined features of both object-orientation and database system. This collection of concepts and features can serve as the core model and identify the main differences in comparison to the traditional models.

Class and Object

As the basic modeling unit, an object models a real world entity of interest in an application. It encapsulates the entity state and behavior through data structures and functions. The state is represented by the values of the object's attributes, whereas behavior is defined by the methods acting on the state of the object upon invocation of corresponding operations [Jacobson et al. 1992]. An object is an instance of a class type. A class describes a group of similar objects. It names and types the common components of the data structure of each object in the class and declares the behavior that can be applied to them.

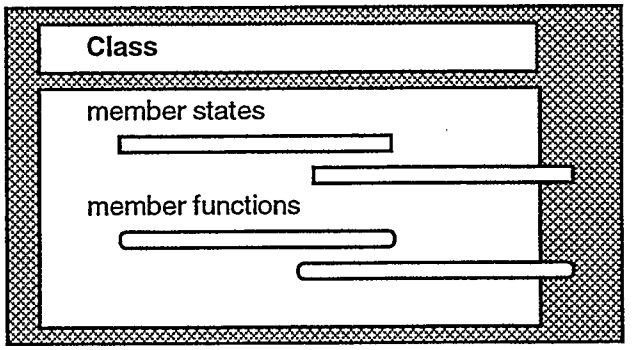

Figure 2.7 A class encapsulates both state and behavior features.

Objects and Identity

In an object-oriented DBMS, each object is identified by a single OID (Object Identifier). The identity of an object has an existence independent of the values of the

object attributes. By using the OIDs, objects can communicate with other objects and general object networks can be built.

An important concept of the relational model is the key concept, an attribute or set of attributes whose values identify unequivocally each tuple in the set of all those tuples belonging to the same relation. A key consists of the value of one or more attributes and can be modified, whereas an OID is independent of the state of the object. Two objects are different if they have different OIDs, even when their attributes have the same values. Moreover, a key is unique within a relation, whereas the OID is designed to be unique within the entire database. By using OIDs one can define heterogeneous collections of objects which even belong to different classes. Indeed, a collection consists of a set of OIDs which identify the objects belonging to the collection. These OIDs are independent of the class to which the objects belong.

Researchers have investigated different approaches to constructing OIDs in order to have OIDs with richer semantics and be more efficient for information retrieval. One example is that an OID consists of a pair - 'class identifier, instance identifier, system generated number' - where the first is the identifier of the class to which the object belongs, the second identifies the object within the class, and the third one is system generated serial number. When an operation is invoked on an object, the system can extract the class identifier from the OID which then determines the method for executing

the operation. Usually this number is stored in a LookUp Table (LUT), and maintained by the system to keep track of objects.

Encapsulation

Encapsulation is one of the most beneficial concepts in the context of object orientation. Encapsulation combines data structures and functionality into objects. This mechanism ensures that an object represents an application entity naturally and efficiently by keeping all data pertaining to the application entity bundled together with all the functionality that applies to it [Coad and Yourdon. 1991].

Encapsulation also supports information hiding, that is, internal aspects of objects are hidden with specification for which features of an object to be accessible. The users only need to know what to perform instead of how to perform. Access to data and code has to be granted explicitly and achieved through sending messages. A message refers to a request to perform a method for an object. It results in the invocation of the method.

Encapsulation in OODBMS

Encapsulation in programming languages means that an object consists of an interface and an implementation. The interface is the specification of the set of operations which can be invoked on the object and is its only visible part. The implementation contains the data, i.e. the representation or state of the objects and the methods which provide, in whatever programming language, the implementation of each operation.

But this principle is not applied very strictly in database since it is not clear whether the structure is part of the interface or not, whereas in programming languages the data structure is clearly part of the implementation and is not visible. Query management needs direct access to objects' attributes, and this makes violating encapsulation almost obligatory. In fact, in databases, it should not be considered a disadvantage to know which attributes and references an object consists of. Queries are very often expressed in terms of predicates for the values of the attributes. Therefore, object-oriented DBMSs (OODBMSs) should allow direct access to attributes supplying 'system-defined' operations which read and modify these attributes. These operations are provided as part of the system (and are not defined by the user) and they are implemented by the system in a highly efficient manner and at a low level. There are two advantages, described below, of being able to access or modify directly the attributes of an object [Bertino and Martino. 1993]

- It avoids the users having to implement a considerable amount of methods which have the sole purpose of reading and writing the various attributes of the objects.

- It increases the efficiency of the applications, in that direct access to the attributes of objects is implemented as system-provided operations.

For the contrasting requirements, various OODBMSs provide different solutions. Some systems, provide 'system-defined' methods for reading and writing the attributes of an object. These methods are implemented efficiently and at low level by the system.

However, these methods can be redefined by the user (overriding). Other systems, such as O2, allow the user to state which attributes and methods are visible in the object's interface and which can be invoked from outside. Finally, in other systems, all attributes can be accessed directly, both while reading and writing, and all methods can be invoked.

Message Passing

The execution of an object-oriented procedure or function occurs through message passing (see Figure 2.8), instead of arguments passing in traditional systems [Jacobson et al. 1992]. A message usually includes the information about sender, receiver and the functions to be invoked. When received by an object, a message is matched with one of that object's methods or attributes, and then a method would start executing. The execution of a method can involve sending messages to other objects, and execution spreads through the system as messages are passed from one object to the next. When a method finishes executing, a value (object) may or may not be returned as control is passed back to the object that sent the message. The message serves as the interface between objects.

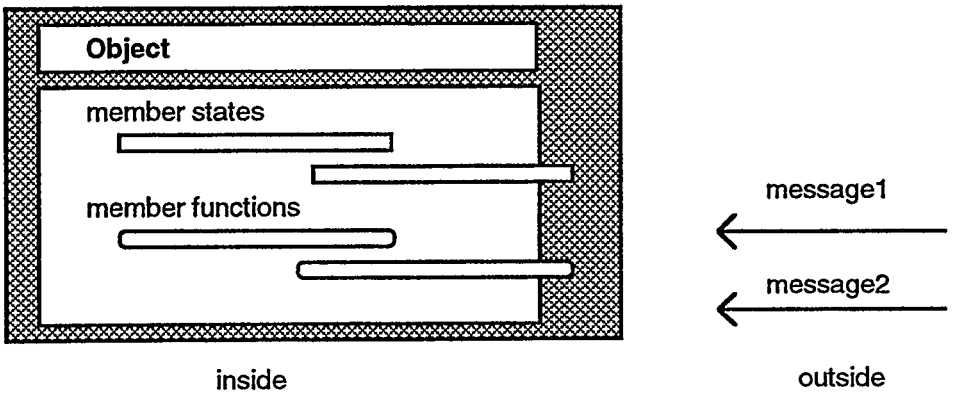

Figure 2.8 Object with messages coming.

Inheritance

One of the most important features supported by the object-oriented methodology is inheritance: the ability to derive new classes from existing ones [Jacobson et al. 1992]. New classes can be constructed by extracting some features from existing classes and adding some new features, thus common specifications and elements can be reused. Inheritance is the mechanism that contributes most to the productivity increase attained with object-oriented systems. The class that the other classes inherit features from (both state and behavior) is called the superclass, while the inheriting class is called the subclass.

There are two types of inheritance. Multiple inheritance allows a subclass to share features of several incompatible superclasses, while single inheritance only involves one superclass for a subclass. Single inheritance and multiple inheritance can be represented by tree structure and network structure, respectively (Figure 2.9 and 2.10)

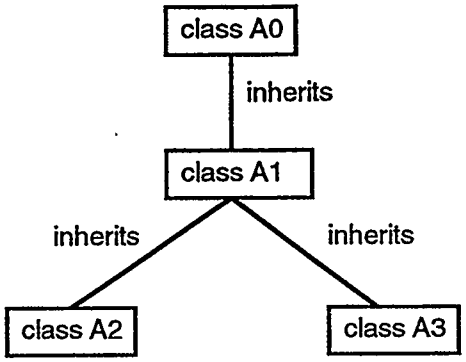

Figure 2.9 Single inheritance hierarchy.

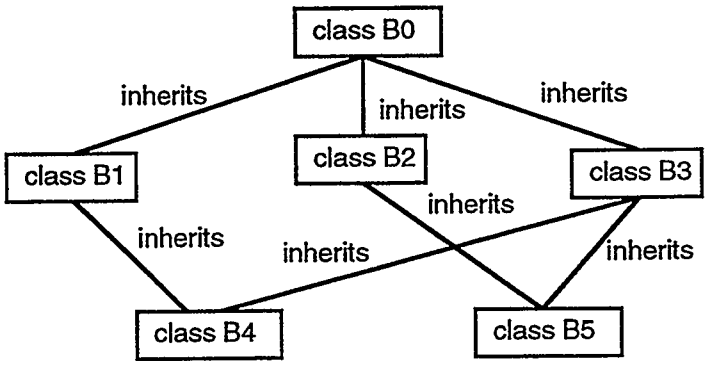

Figure 2.10 Multiple inheritance hierarchy.

Polymorphism

Polymorphism is another key concept, which means that the sender of a message does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class [Jacobson et al. 1992]. A message can be interpreted in different ways, depending on the receiver's class type. It is the instance which receives the message that determines its interpretation, not the transmitting instance. Thus one operation can be

implemented in different ways in different classes. Conceptually, it is based on the assumption that the type (or class) of an instance needs not match the type (or class) of the object that the variable refers to.

Polymorphism allows the specification of modules at higher or more abstract levels. The user only needs to know that another instance can perform a certain behavior, not which class the instance belongs to and thus not which operation actually is to be performed. This is a powerful tool for more flexible systems.

Complex objects

Complex object refers to the mechanism that the member attribute can be another object or a set of objects. Compared with the 1NF constraints in the relational model, the attributes of an object can be other objects, both primitive and non-primitive ones. This characteristic enables arbitrarily complex objects to be defined in terms of other objects.

Completeness

Most of the OODBMS have very basic supporting manipulation language, leaving a big part of complex manipulation functions to be implemented directly by the programming language. Great flexibility and completeness can be obtained from this combined support.

2.4.2 Advantages of object-oriented data model

The object-oriented model has advantages in the areas of modeling and manipulation efficiencies, data semantics and model extension capabilities over the relational model, showing great potential in GIS applications.

Modeling Efficiency and Extendibility

All real world entities have both state and behavior features, thus conventional methods require extra efforts to combine the separated primitive data types and algorithms to model the entities. In relational systems, the state attributes of spatial entities are represented by tables while their behaviors are implemented by SQL programs or vendor-supplied subroutine calls outside the DBMS. The class, as an integration of both states and behavior information, enables the object-oriented logical model to represent entities more naturally and directly. Moreover, the complex objects and classes mechanism provides the aggregation support which is lacking in relational models. This enables the object-oriented system to model complex entities by simply composing the fundamental class types instead of going into very basic elements, which is very beneficial for modeling geometric entities. The modeling efficiency can be improved greatly, and system extension can have increased flexibility.

Another object-oriented concept, inheritance, further enhances the systems' representation capabilities because it reflects the nature of the relationships between real

world entities. An inheritance hierarchy can clearly show the similarities shared by the concerned entities, and the sharing implies reusing of code which results in a dramatic increase in programming productivity. Hence users only need to consider the specialties of subclasses while modeling them. Building or extending an application based on well defined fundamental classes become more efficient and easier.

Manipulation Efficiency

Because of the direct mapping from real world entities to classes and objects, the retrieval of complex objects is more straightforward in an object-oriented system, compared with relational systems where queries usually involves complicated SQL commands to locate the elements and reconstruct the geometric information of spatial entities from the decomposed information. With some data access method encapsulated in objects, certain information retrieval can be carried out by a single message directly and easily.

Data Semantics

Based on the support of class, complex class types and inheritance, the object-oriented system can distinguish between classification, generalization and aggregation, thus enrich its data semantics.

Modularity

This object-oriented system also supports the concept of modularity [Coad and Yourdon. 1992]. Here modularity does not mean groups of similar and closely related functions in a traditional solution of a problem, but the general aspect of decomposition of a complex problems into collections of smaller units (modules) for easier handling. The object-oriented paradigm provides a natural way to modularize an application where the class serves as the unit for a module.

Object Identity

In an object-oriented system, unique identifications are generated for all objects by the system, independent of address or data value, and can survive updating and database reorganization. They are useful both for data access and for maintaining relationships among the objects. The essential advantage of using OIDs over keys as the object identification is that the risk of changing the uniqueness of the object identification can be completely avoided while updating the attributes. Also, since OIDs are implemented by the system, the applications programmer does not have to concern himself with selecting the appropriate keys for the various classes of objects.

Smoother Program Interface

A smoother program interface can be achieved through the similarities between object-oriented programming systems and OODBMSs. The most important one is the generic object concept which helps not only encapsulation but also data modeling. For

the first time, this concept makes models of real objects basic building blocks of a program. This provides both the programmer and database user with a common perspective and programming fundamental. Thus the semantics and programming gaps between the two systems has been reduced.

In summary, the inadequacy of traditional logical models to handle increased complex spatial information efficiently, coupled with the emerging trend in the database field towards object-oriented database systems converges towards the utility of developing object-oriented model for GIS application.

Chapter 3

SPATIAL DATA MODELING

USING AN OBJECT-ORIENTED APPROACH

As stated in Chapter 2, a logical model is actually a high-level interface between users and the physical model to make the latter more flexible and easier for understanding and implementing. To construct a system with object-oriented modeling capability, the first step is building up the prototype for a physical model. An object-oriented software development method is used to build up the prototype. This chapter describes the method to analyze, design and implement the prototype. The design and implementation will concentrate on the general spatial data model and related supporting functions.

3.1 System Analysis and Design

To develop a software package, the selection of analysis and design method is important. A software development method is a standardized means of presenting and communicating the requirements of a system and the design decisions, which provide an effective means of delivering those requirements to developers and users. This technology emerged as the solution to the 'software crisis' in 1970s, and has benefited from the phase of data-processing oriented methods to object-oriented method phase. The data-processing method distinguishes between processing and data, where processing, in

principle, is active and has behavior, and data are passive holders of information which are affected by functions. Software systems are typically broken down into groups of processings, whereas data are sent between the processings. Examples include SADT (Structured Analysis and Design Technique) [Ross 1985], SA/SD (Structured Analysis and Structured Design) [Yourdon 1979]. From the late 1980s, the Object-Oriented method began to dominate this field.

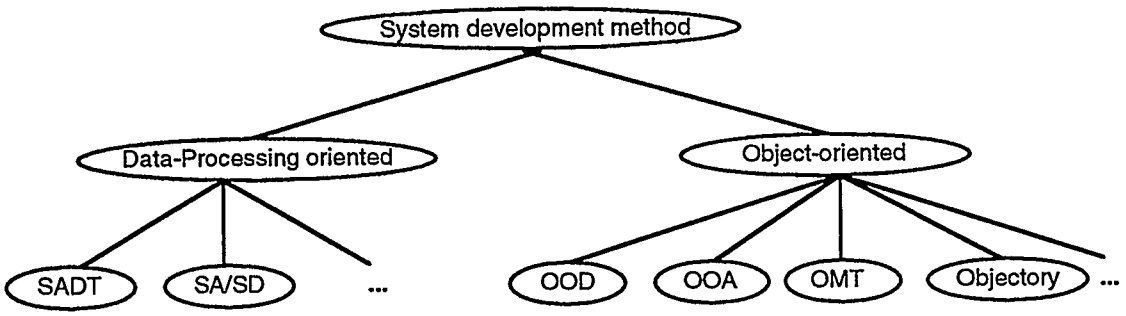

Figure 3.1 Data-processing oriented and object-oriented system development methods

In this project, an object-oriented method will be used to analyze, design and implement the system, a decision based not only because of its popularity, but also for the consistency of the development procedure within the planned system itself.

3.1.1 Object-oriented analysis and design

The object-oriented analysis and design method is the integration of the development of information system analysis and design methodologies and object-

oriented programming languages. In an object-oriented method, the basic unit of design is the object, with both state and behavior encapsulated, compared with the separation of the data and functions in conventional methods. An object can correspond directly to a recognizable real world entity or an abstraction based on that entity. This allows the software to obtain an almost one-to-one mapping with the real world. Compared with the fact that in traditional methods, a large amount of effort is spent applying sophisticated algorithms to the very basic and limited data types to construct complex models of the dynamically changing world, thus object orientation improves the modeling capability greatly. It also implements the sound concepts of information hiding, coupling, and cohesion.

Compared to systems designed using traditional software development methods, these designs

- are more adaptable as the world (or the designer's understanding of the world) changes
- provide units that fit the environment rather than a specific system, and apply to any system dealing with that part of the environment
- make the system easier to understand, and thus to maintain, by both customers and software developers

There are several object-oriented development methods available. The early pioneers include Booch, whose first version of his Object-Oriented Design (OOD or the Booch Method) emerged in 1983; Jacobson, whose famous Objectory method was formulated in 1985. Other popular methods are the Object-Oriented Analysis (OOA) by Coad and Jourdon [Coad et al. 1991], Object Modeling Technique (OMT) by Rumbaugh et al., which is based on entity-relationship modeling [Chen 1976] with extension to modeling classes, inheritance and behavior [Rumbaugh et al. 1991]. The OMT and the Booch method have been combined to form a more powerful method in 1995.

The Booch method was selected in this project because of its popularity. Another reason is that this method has an implementation software tool called Rational Object-oriented Software Engineering (ROSE) available with interface in Borland C++, the programming language used in this research. ROSE can generate frames of C++ code from the analysis deliverables directly thus easing implementation.

3.1.2 Overview of the Booch Method

The Booch method is an object-oriented method based on proven heuristics for developing quality software. It provides a model to support solid analysis and design, and allows the developers to enhance, correct, and maintain the same consistent model from the beginning of analysis through coding and implementation.

As shown in Figure 3.2, the Booch method consists of five iterative steps:

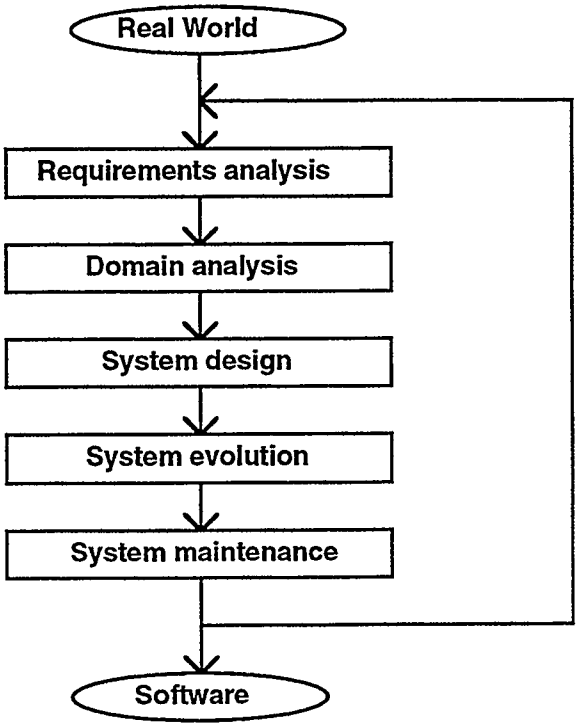

Figure 3.2 Major steps of the Booch method.

- Requirements analysis, which provides the identification of the functionality of the system.
- Domain analysis, which provides the key logical structure of the system.
- System design, which provides the key physical structure of the system, maps the logical structure to it, and leads to working executable releases.
- System evolution, which provides the growth and change on the implementation through successive refinement, ultimately leading to the production.

- System maintenance, which provides the postdelivery evolution.

Each step has deliverables which construct and document the progress from the understanding of the problem to the solution to the problem. With the deliverables specifying the design more and more clearly and closer to the computational model, the frame of code can be obtained.

As this research is focused on prototype analysis, design and implementation, discussion will concentrate on the first three steps.

3.1.3 An iterative approach

Traditional software development methods used to insist on a rigid series of steps. Classically, first the developer discovered aspects about the user requirements, then the general design of the problem solution, and so on. Subsquent steps are always based on the assumed correctness and completeness of the previous steps, a view more often than not in conflict with the natural way of thinking in human. The difficulties and high cost for correction of early stage analysis and design has been one of the biggest problems faced by the traditional methodology. The Booch method allows for the reality that the development of a system is an iterative process--previous work must always be added to or refined as the results of that work are used in the next stage of development, which is more close to nature of human's thinking.

This iterative approach retains the classic steps: developers study the user requirements first and then map them to design. However, as developers continually integrate their analysis results into one underlying model, they can easily move back and forth between analysis and design to refine their study. In fact, the iterative approach allows developers to analyze a little, design a little, and then implement a little. In practice, the method specifically encourages early implementation of pieces of the system to aid in the requirements analysis process. Then the developer cycles back and goes through the procedure again, only for better understanding and design of the system. All of analysis, design, and implementation are accomplished, but in a series of cycles rather than three large leaps.

3.2 Requirements Analysis for Spatial Data Modeling Prototype

3.2.1 Requirements analysis

Requirements analysis is the process of determining what the a system is expected to do. It is a high-level stage to identify the key functions the system is to perform, to define the scope of the domain that the system will support. The first step analysis results provide the fundament for further development.

Requirements analysis essentially forms a conclusive understanding of what kind of functions the system is going to provide. The understanding is not fixed, instead,

it is changed often as the development cycle continues. It, however, does serve as a starting point and central reference for what the system is supposed to do.

The understanding of the requirements usually is put in forms of two deliverables: system charter, which outlines the responsibilities of the system, and system function statement, which outlines the key use cases of the system.

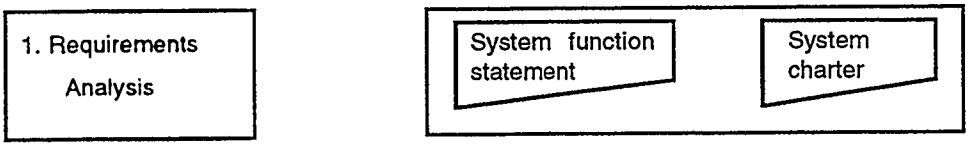

Figure 3.3 Requirements analysis step and the deliverables

3.2.2 Use case analysis

Use case is defined as "a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events." [Jacobson et al. 1992]. For example, in a GIS system, to display the DTM of a certain interested area according to a user's request is a use case. The functionality of a system can be considered as the whole collection of use cases. In most of the object-oriented analysis and design methods, identifying use cases in an application is the first step to specify the complete functionality of an application system. The basic idea is that, at this stage, the scenarios that are fundamental to the system's operation are first enumerated. The functions of the application can be primitively

outlined by these scenarios. Analysis then proceeds by closer studies of each scenario, using the traditional storyboarding technique. As the study walks through each scenario, objects which participate in the scenario are identified. Then the responsibilities of each object, and how those objects collaborate with other objects, in terms of the operations each invokes upon the other, could be discovered. At this point, the system development is ready to enter the domain analysis phase, whose duty is mainly to deepen and refine the analysis results of this phase. The use case driven analysis ideas will be applied in the whole analysis and design, through to the implementation procedures. Moreover, further specified scenarios also serve as the basis of system tests.

3.2.3 Requirement of the prototype

In this project, the main purpose is to set up an object-oriented GIS prototype for modeling the digital terrain information. First, a general object-oriented spatial data model is implemented as the core of prototype, then the data model will be adapted to construct an application module to handle a set of DEM data. To reach these goals, the support in the following fields is critical.

Two general types of modeling components

In terms of GIS applications, aspects of the real world can be viewed as a collection of entities with combined features in different domains, such as spatial, graphical, temporal and textual/numeric domains. Interactions between entities exist and contribute to the changes on entities' features. A GIS modeling prototype should have

direct support for entities and the relationships between them as the fundamental model components. The entities can range from a national park to a transportation network, and relationships can include topological relationship and cause-result relationships. Further analysis techniques such as abstraction and classification will be applied in the following design and lead to concise and efficient groups of modeling components. The components can be implemented as class types, objects or attributes while using an object-oriented programming language.

Support of modeling mechanisms

As the research is an investigation in object-orientation's applicability in GIS, the core data model is designed to serve as object-oriented constructing fundamental to modeling different kinds of spatial information. This means that the prototype is expected to be customized for most situations, so the data model should be for general purpose in nature. Since it is impossible to include all kinds of components to correspond to all kinds of objects in the real world in this modeling prototype, the core of the modeling prototype should be of a high level of abstraction and contain very general class types. Support for specification from existing model components to generate new model components for particular situations is necessary. As this is expected to be an object-oriented modeling prototype, the following object-orientation features should also be supported: encapsulation, inheritance, aggregation, message communications, etc. All these mechanisms should be implemented in a software system, and used to specify what

form a system should take. An object-oriented programming language such Borland C++ provides direct support for several of the above mentioned object-oriented features at certain levels, which makes the implementation of further support easier. Thus Borland C++ has been used as the implementation language. The object-oriented analysis and design method was applied to implement these features more thoroughly.

Support of functions

Since the prototype is designed to be used to build up other specific applications, besides the behavior of features of each component, the prototype should also have modeling capabilities to do the work such as generating new model components by inheriting from superclass types. Thus, basically two kinds of functionality are necessarily considered: system functionality which supports the creation, use, maintenance of the model components in the prototype; and operation functionality which is built on the prototype components by the user for feature operation. These are implemented in the form of functions. As this is in an object-oriented development environment, the functions can be classified into two categories:

- the functions which mainly describe the objects' own behavior features such as displaying or changing features. Usually these are encapsulated within the class types of modeling components;

- functionalities to work with class types or operate on the class types. Usually these are encapsulated within the class types other than those of modeling components.

Generally speaking, the prototype should meet the following main objectives:

- modeling various type of spatial objects with one group of data modeling components;
- providing basic supporting manipulation functions, such as objects creation, object maintenance.

3.3 Fundamental Class Types for Spatial Information Modeling

3.3.1 Domain analysis

Domain analysis is the process of defining a precise, concise, and object-oriented model of the part of the real-world related to the application, or the problem domain. It is through this process that the detailed knowledge of the problem domain can be gained, which is needed to create a system capable of carrying out the required functions.

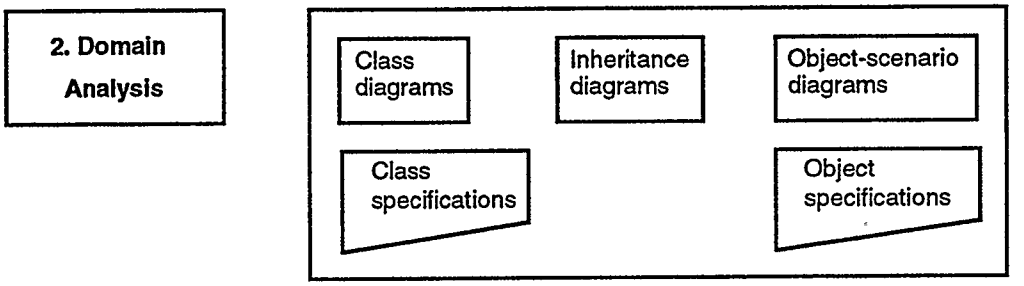

Figure 3.4 Domain analysis step and deliverables

More specifically, domain analysis identifies all major class types and objects in the domain, including all data and major operations that will be needed to carry out the system's functions. It produces a central model containing all the semantics of the system in a set of concise but detailed definitions of classes and objects, which will map directly to final implementation.

Good domain analysis not only simply refines and details the previous analysis results, but also adds appropriate levels of abstraction to a system. In fact, this is the phase when object-oriented analysis skills such as classification, instantiation, specialization, generalization can be applied further to lead to high efficient modeling. With classification, a limited number of class types can be obtained to represent the groups of countless entities in the real world, where each group includes entities sharing certain commonalities. Then with careful analysis and definition of the domain scopes, hierarchical structures can be built with more abstract class types on the root levels and more specified class types on the higher levels. In this way, reusability can be realized in the application system by support of inheritance mechanism to make the development

more efficient. It can also make it easier to further extend or modify the system by specializing the abstract types with the support of inheritance. Thus efficiency and flexibility can be obtained during both its development and its productive life.

3.3.2 Domain analysis steps

The following steps are performed during domain analysis: defining classes, defining relationships, defining operations, finding attributes, defining inheritance (Figure 3.5).

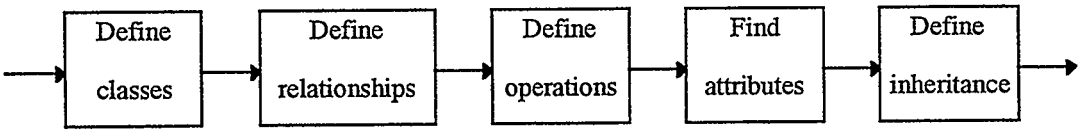

Figure 3.5 Steps of domain analysis.

3.3.3 Identify and define classes

To identify and define the key classes is the first important step towards finding out major abstractions in the problem domain. More specifically, it can help to obtain knowledge about what domain the application system works in and what data it contains.

This step can begin with identifying nouns in the system charter. It is important that the defining classes should stay at a logical level since the system charter usually contains some implementation characteristics. Domain related classes should be independent of any given implementation of a system, thus concentrating on the problem

domain, not on how domain entities map onto an implementation. By this, thorough understanding of problem and design at a certain level of abstraction can be guaranteed, which can lead to flexible implementation.

3.3.4 Classification

The identification of key classes and objects is the hardest part of object-oriented analysis and design. Through its ability to abstract a class type from and for a group of entities sharing a certain commonality in structure, classification is a very helpful tool at this stage. Classification recognizes the similarity among key abstractions, eventually leading to smaller and simpler architectures. It also affects understandibility and effective communication greatly. For example, assuming that an application requires one to model *"highways"* and *"streets"*. There are two types of objects, but they share similarities in terms of state and behavior. They can be represented by only one type of class *"transportation road"*, and *"highways"* and *"streets"* can be sets of instances, objects. The class can denote the common features such as *road name, road type, length, display method,* etc. The specialization can be obtained when the objects of a certain street or highway are constructed (Figure 3.6).

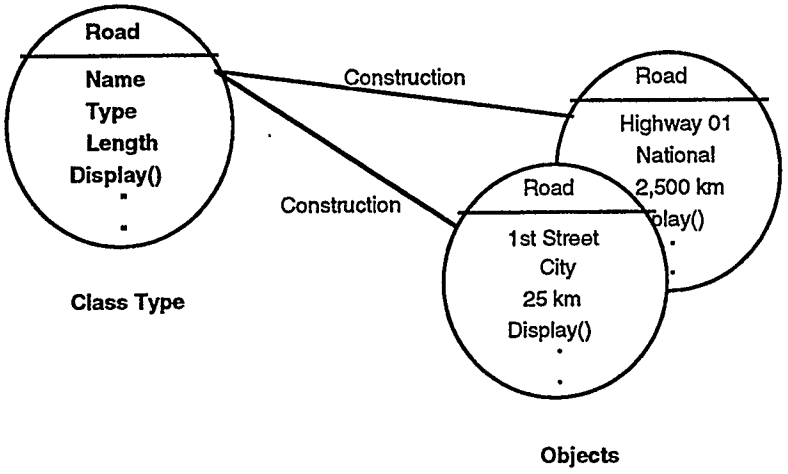

Figure 3.6 Classification and instantiation for the "Transportation Road" type.

3.3.5 Fundamentals of topology

A basic idea in algebraic topology is that most fundamental element for constructing geometric objects is called the simplex. A k-dimensional simplex or k-simplex is the convex hull s $(v_0,...,v_k)$ of k+1 points $v_0, \ldots, v_k$ with $(v_1-v_0, \ldots, v_k-v_0)$ linearly independent [Jänich 1984]. There are 0-simplex (point), 1-simplex (segment), 2-simplex (triangle), 3-simplex (tetrahedron), etc (Figure 3.7). Groups of simplices form complices, (Figure 3.8) and spatial space can be decomposed into complices [Jänich 1984, Gamelin and Greene. 1983].

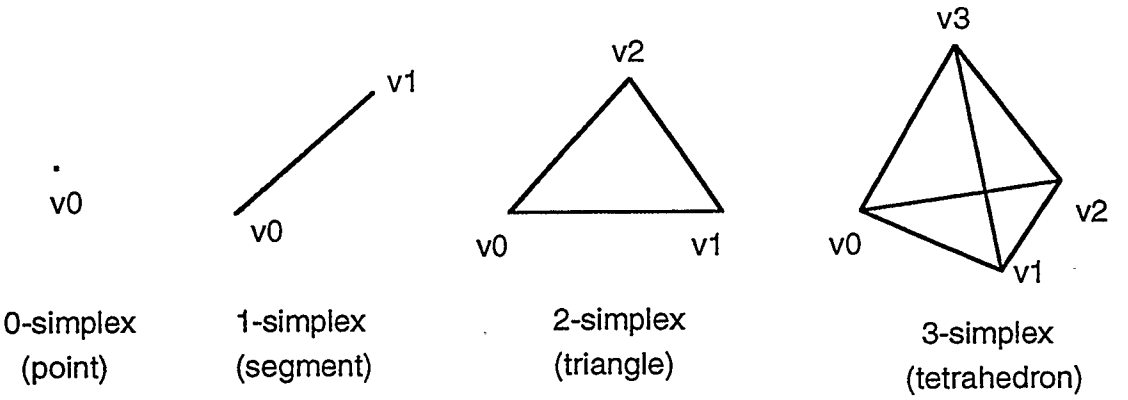

Figure 3.7 K-dimensional simplices.

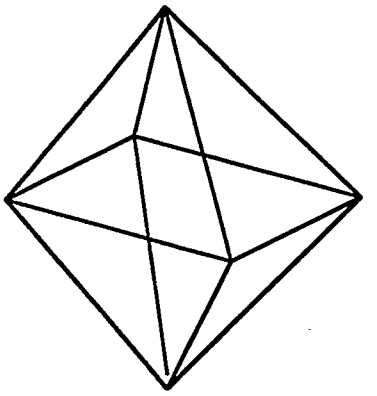

Figure 3.8 Octachedral surface composed of eight 2-simplices.

This idea was applied in the spatial information modeling. First, three geometric elements: node, arc and polygon were identified as fundamental composing elements for complex entities. Geometric information, in terms of both positional and topological information, is stored in terms of these elements. Complex entities contain the geometric information by holding logical pointers to the member elements. On the other hand, attributes of spatial entities are always directly related to the entities instead of the constructing elements. So three types of entity type: point, polyline and area, were also

identified to obtain direct access to the attribute information. More details about how these element types and basic entity types can be used to model general spatial entities will be discussed in the following paragraphs.

3.3.6 Classification of spatial information

Spatial information can be viewed as a combination of all kinds of spatial entities. Spatial entities can range from forestry coverage to land parcels. To have model components for every entity is not only impossible but also unnecessary. Abstraction can help to discover the commonality shared by spatial entities. Then classification techniques can be applied to draw several general classes which include entities based on the commonality found.

Geometric characteristics are the most important feature of a spatial entity for GIS application. Classification can proceed according to spatial entities' geometric characteristics. Studies have shown that all 2-D spatial entities can be classified into one of the following entity types: point, polyline, area and complex entity, these can serve as fundamental modeling units. More details about these four geometric abstract types will be given in the following paragraphs.

Point Entity

Point entity refers to a kind of spatial entity whose location is important, but whose size is too small to be represented as a line or an area. For example, in a

transportation network modeling system, the intersections can be represented as points (Figure 3.9). In reality, the ignorance of size really depends on different observation scales or different applications. If in another application of land usage administration, the size of intersection were to be important, and have to be modeled as polygons.

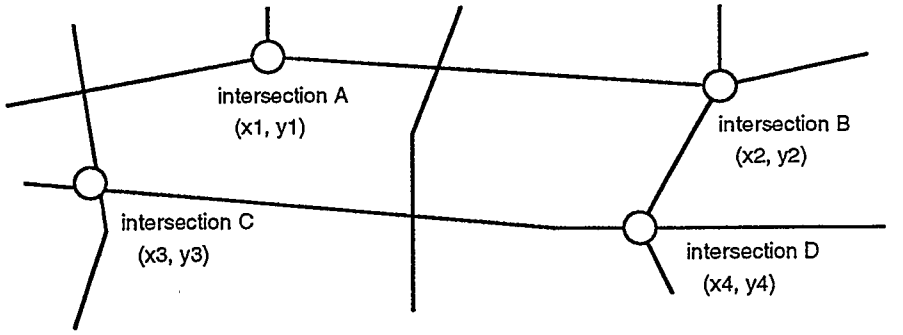

Figure 3.9 Intersections of roads can be represented as point type entities.

For each point entity, the most basic and important attribute is its position. Usually, it is represented by a pair of coordinates, which could be in the form of UTM, 3TM or other local map coordinates. Other concerned features, including logical identity in a GIS, can be attached as an attribute to describe the point. In the above example, the intersection point can have a set of attributes *"Traffic Control", "Control Type"* to hold information about whether there is traffic control in an intersection and what kind control it is. Other examples include in utility and transportation systems inventory and analysis, such point entities as the location of telephone poles or sewer manholes, or of individual houses with discrete addresses.

Polyline Entity

This is used to represent a linear entity which has location and length, but where occupation can be ignored. For example, on a map showing if there is a highway network connecting city A, city B and city C, the highways can be represented as polyline type (Figure 3.10). The entities may be directly observable or may be conceptual only, as is the case of air routes. They are not necessarily only in a horizontal plane, like a borehole or a water level in a mine. Again, the ignorance of the entities' occupation depends on different observation scales and different applications. The same highway may have to be modeled as polygons in a land usage administration application.

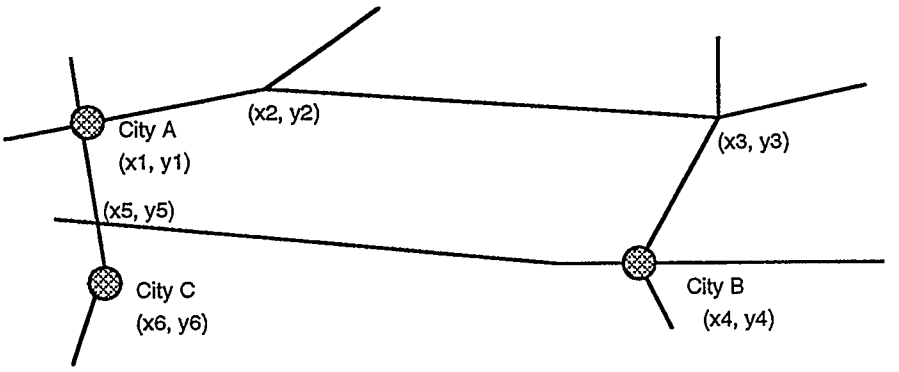

Figure 3.10 Transportation network represented as polyline type entities.

For a polyline entity, the information of the set of points on this linear entity is important to record their position. This information can be stored as a set of coordinate pairs directly, or a set of the points' IDs in the system and retrieved through the IDs when

required. The latter way is better for maintaining the correctness and consistency of the information. For most of the applications, attributes such as the length of the entity and the orientation are also very important. The specialties of each entity can be represented by the specification of its attribute set.

Linear features are important in transportation studies, hydrology, utilities management and geology, and are prominent features on many types of mapping.

Areal Entity

Sometimes referred to as regions or zones, they may be identified for natural or man-made phenomena whose occupation cannot be ignored in a GIS application. The areal units may be entities like lakes, islands, territory with a particular soil type, or land parcels (Figure 3.11). The units can also be artifacts used for statistical reporting like census zones or delivering mail like postal zones, or discretizations (the creation of pieces or segments) of continuous space like climate regions. Other sets of attributes can be designed to represent special cases of area type entities.

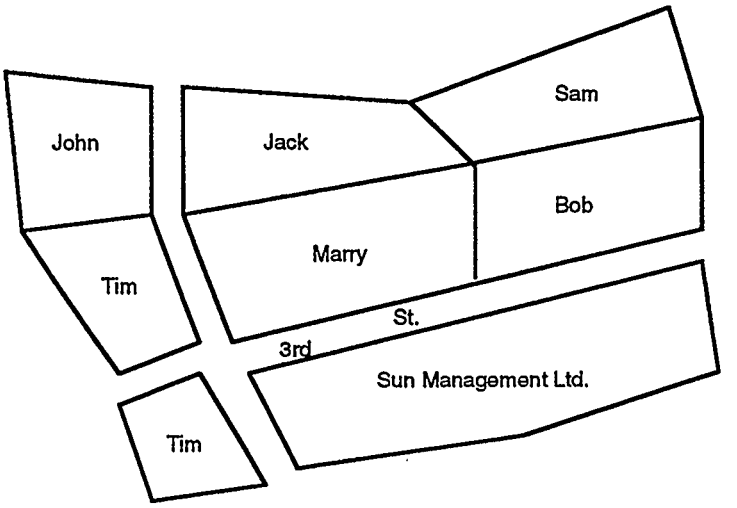

Figure 3.11 Land parcels can be modeled by area units.

In a simple areal entity, a polyline (modeled as polyline type) is included to serve as the boundary to locate the area. For some complicated cases where there are holes in an area, two or more polylines should be embedded to represent the location of the holes. In this project, concentration has been placed on simple areal modeling. The popular way to include polyline information is to store the system ID to build up the logical connection with physical data. Other particular spatial properties associated with area entities are: area extent (the size of forestry coverage), perimeter length (the extent of a shoreline), etc.

Areal units are important in socio-economic studies, analysis of terrain conditions, land use and natural resources inventory, and recordings of real estate.

Complex Entity

This kind of entity usually has a combination of more than one of point, polyline and area types. For example, a park might have paths as polyline type entities, picnic spots as point type entities, lake and meadow as area entities (Figure 3.12). Complex entities usually can be decomposed into and modeled by the three basic types. But in some cases when there are attributes and indexing mechanisms directly related to the complex entity as a whole instead of related to each component, a complex entity type can also be used in addition to its components of the three basic types. This can improve query speed at the cost of limited redundancy (extra complex entity type itself) by reducing the efforts to reconstruct the complex entity from components at running time. In fact, redundancies of storing the attributes of a complex entity repeatedly in all of its components can also be reduced this way.

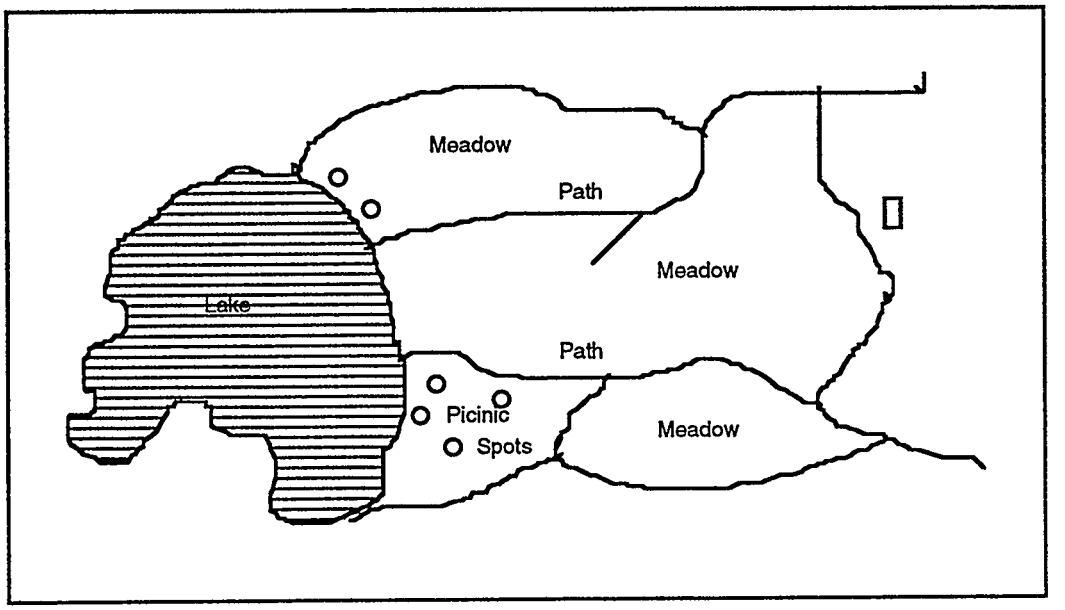

Figure 3.12 Park represented by a complex entity type consisting of point, polyline and area entity types.

Usually, a complex entity has a polyline to specify its boundary and the ID of the boundary polyline should be placed separately from its components' IDs. Complex entities include three sets of system IDs of the included point, polyline and area entity types to connect to the physical data. Which of the component entities should be listed in the ID sets depends on which of them are indexed or expected to be queried at this level. For example, both path intersections and picnic spots are point entities in park entity, but assuming that users are only concerned about where to find picnic spots, only IDs of picnic spots are necessary to be stored in the ID sets.

So these four fundamental spatial entity types can serve as four superclass types to model various kinds of spatial entities (Figure 3.13), and the complex entity type is based on the availability of the other three types.

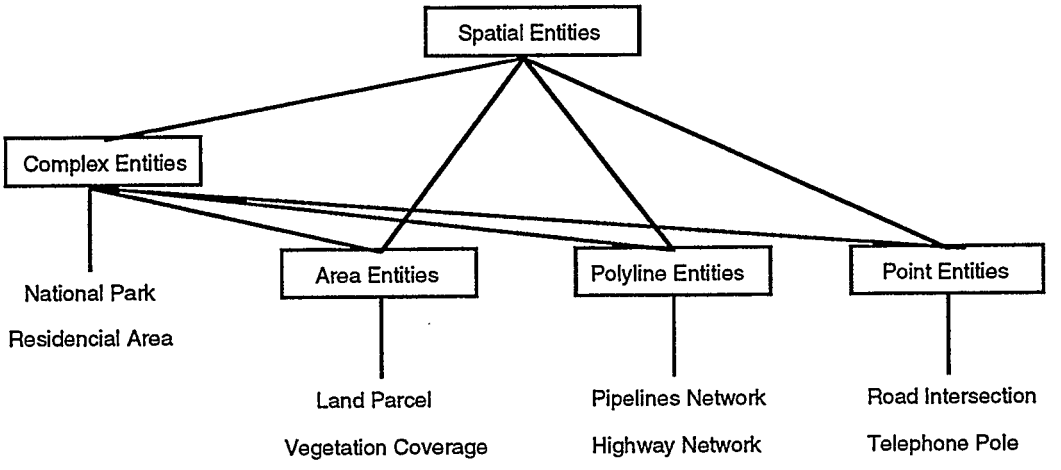

Figure 3.13 Spatial entities and modeling class types

3.3.7 Fundamental geometry elements

In spatial information, an important part is the topological relationships between spatial entities, which include enclosure, connectivity and adjacency [Blais 1987]. Good spatial information modeling must be able to reflect topological information. For this purpose, geometry knowledge and skills are naturally applied in a GIS study. In spatial information modeling, three geometric elements can be introduced to make all the spatial entities be built on them, then geometry can be applied for the topological information study.

Three basic geometric elements to construct complex objects are node, arc and polygon.

Node

Called a vertex in geometry, node refers to a point that terminates a line or a point at which lines cross. Therefore it has a property of connectivity, being related to the lines (Figure 3.14 (a)).

Node is different from point in this context: it is considered as the abstraction of point. The principal attributes of a node are its position and relative relationship with arcs and polygons, such as on what arcs or polygons the node is situated. A point can be defined based on a node to hold the positional information, but usually the focus is placed on non-positional features, such as what kind of object it stands for.

Arc

Referred to as edges in geometry, both ends of an arc terminate in two nodes. Every arc has, and only has, two nodes (Figure 3.14 (b)).

The difference between an arc and a polyline is very similar to the difference between a node and a point in terms of focus on the included information. One special case is a polyline may contain one or more arcs, so a polyline may contain one more piece of information: how it is composed.

Polygon

A polygon refers to a piece of surface bounded by a minimum of three arcs (Figure 3.14 (c)).

The relation between a polygon and an area entity type is similar to the difference between an arc and a polyline.

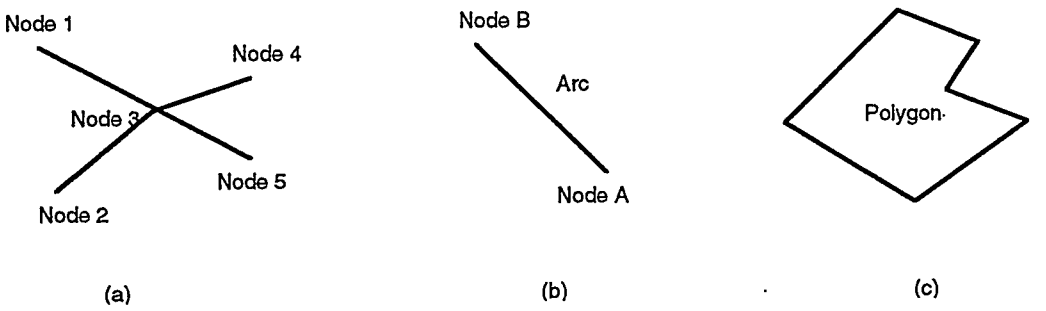

Figure 3.14 Geometric elements. (a) Nodes. (b) Arc. (c) Polygon.

These three elements can be grouped together with the four entity types as the seven fundamental modeling class types. Every complicated spatial entity can be first modeled as one of the four entity types, then the entity types can be decomposed into the very three basic types of geometric elements, and at this level, topological information is easier to store because the relationship between the elements is relatively clearer and more systematic from the support of geometry theory.

3.3.8 Defining relationships

Classes do not exist in isolation. Rather they are related in a variety of ways to form the class structure for the domain. Finding relationships help further define the classes by exposing their contents and dependency to the contents of others. There is an important relationship in spatial information modeling: aggregation, which denotes a "part of" relationship. It occurs when an entity is physically constructed from other entities, or an entity logically contains another entity (Figure 3.15). It reflects the way spatial entities are structured. It is also essential for the modeling of complex entities. Thus aggregation should be clarified and defined in this modeling prototype.

Cardinality refers to the number occurred in a relationship between two objects. It is usually expressed by four numbers defining the minimum and maximum number of objects occurring in the relationship. For example, an arc can and does only have two nodes, a node can be connected with one or many arcs (Figure 3.15). So, from node to arc, the cardinality is 1-N, where N represents *always greater than one*, from arc to node, the cardinality is 2-2. Cardinalities are important aspects necessary to include in implementation. Some of them are part of integrity constraints, too. As cardinality reflects the state of a relationship at any given time, and some relationships may have varying cardinality depending on the states of the objects involved, so the constraints which may be restricting at certain times should be also included in the relationship specifications.

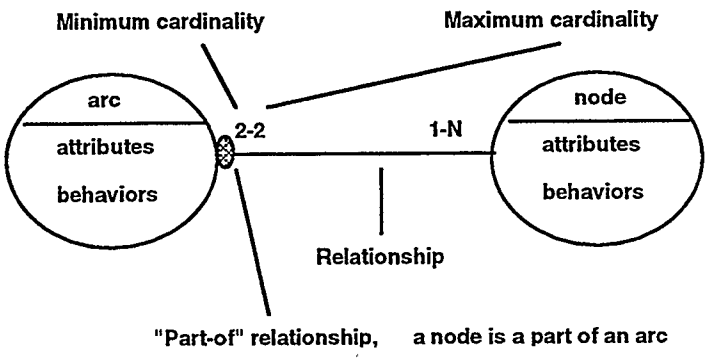

Figure 3.15 Cardinality of the relationship between the classes arc and node.

3.3.9 Relationships between the seven classes

The main class types have been identified, but the relationships between the classes and the detailed definitions still need to be specified.

The relationships between them can be clarified as follows. First, complex entity types consist of point, polyline and polygon types.

A polygon is part of an area entity. An area entity can include one or more polygon.

A polygon is bounded by three or more arcs. When no attribute connected with arcs are of interest in an application, the polygon can be directly modeled by three or more nodes.

A polyline contains one or more arcs, thus it also contains two or more nodes. When no attribute connected with arcs are of interest in a polyline, the polyline can be directly modeled by nodes.

An arc has two nodes, a node can be on several arcs. When a node is extended to model an independent point, a node can be on no arcs.

Figure 3.16 shows aggregation existing between these seven class types with corresponding cardinality constraints.

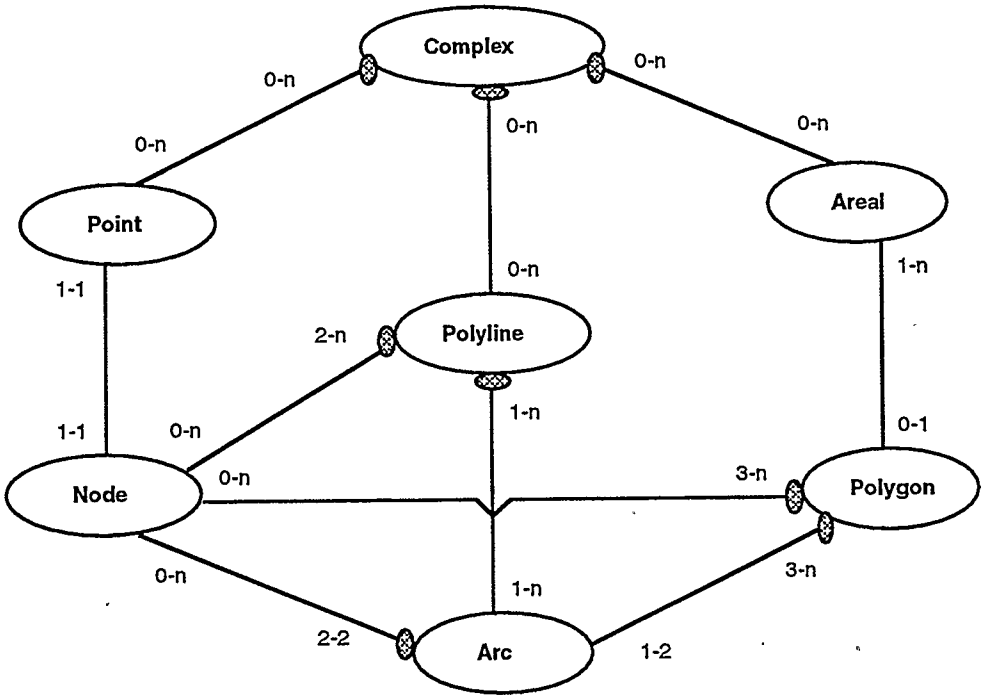

Figure 3.16 Class diagram.

3.3.10 Define operations

With the above analysis, a general abstraction about what classes need to be included in an application domain and how they relate to one another has been obtained. But this is just a static model. How the instances of the classes, objects, functions in the

application are not known yet. An additional step to identify the major operations required to support class structure and system functions is needed.

As mentioned before, the complete functionality of the system is defined by the use cases listed in the system function statement. Expanding a use case into a detailed scenario shows the operations needed to accomplish the use case. Modeling scenarios shows which objects collaborate in the use case and identify the operations needed with each object.

The use case analysis method was used in the operation definition. As the focus of the prototype is model construction, the main use cases are creation, alternation and deletion of the entity or element classes. The design of a function for creating a polyline entity can be used as an example here. To create a polyline entity, the set of arcs should be read in and checked if they already exist. If not, a message is passed to the arc class to activate the construction function to generate them. To create an arc, the pair of nodes have to be checked, too, and if necessary, a message is passed to node class to generate a node object. Then the uniqueness of the arc should be checked before the arc object created. So for this simple function, several cooperating membership functions have to be defined in classes of nodes and arcs. Similar analysis on other use cases can lead to the full list of functions for each class. Figure 3.17 is the object-scenario diagram of the example.

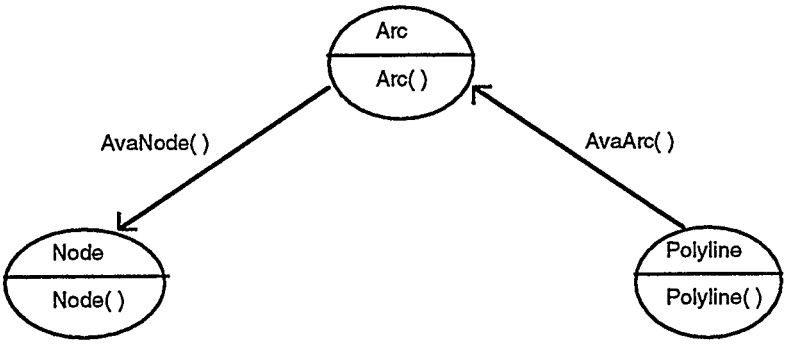

Figure 3.17 Scenario for adding a new polyline.

An object-scenario diagram is used to help in the analysis. It provides details about how objects collaborate to realize a use case, tracing the execution of a scenario. The diagram is usually incorporated with detailed script. The steps in the script align with the message invocations, and express conditional statements and iterations, which lead to a design close to the computational model. Developing object-scenario diagrams gives a more complete picture of the operations needed for each class.

3.3.11 Attribution

Attribution is the process of determining the application related properties that describe the classes. This section will introduce the some of the main attributes defined in the classes to model position information.

First of all, every class has a system ID attribute to contain a unique identifier of objects. The IDs can be numbers generated by the system, and a copy of ID is always

stored in a LookUpTable (LUT). The LUT is updated by the system whenever an object is created or deleted so that the system can keep track of objects and ensure the uniqueness of the IDs. Ideas are also proposed to contain certain semantics in IDs. One example is to add extra numbers in the ID to hold class type information to ease some retrievals.

For each class, the most important attributes are their positional ones. For the node class, the pair of coordinate values should be included, as well as the IDs of the arcs connected with the node. An arc contains the IDs of a pair of the nodes and IDs of right and left polygons so as to locate the arc and record an adjacency of two polygons. A polygon keeps IDs of the boundary arcs (or nodes if directly modeled by them), together with the IDs of surrounding polygons. For arcs and polygons, the location retrieval needs to go through two more levels, but this organization method can help to maintain data consistency and correctness.

To set up attributes for node, arc and polygon classes in this way, the position information is contained in a safe way in terms of consistency, and the relationships among these three types of entities can be traced easily. For example, if one wants to find out which polygons a node A is situated on (Figure 3.18), first the arcs connected to the nodes can be obtained (AB, AC, AD), then all the polygons contain any of the selected arcs can be extracted (P1, P2, P3).

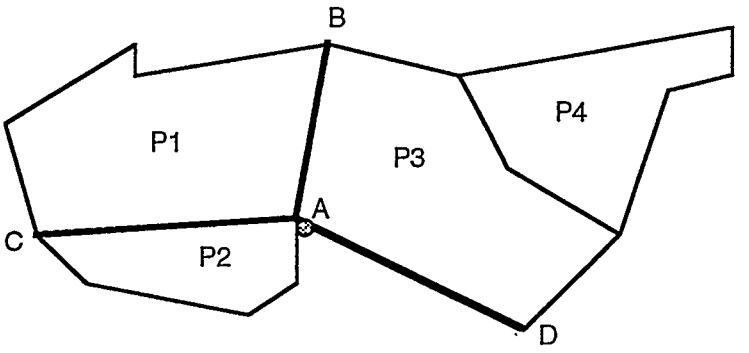

Figure 3.18 Polygons on which node A situates.

For the entity class types of points, polylines, areas, complex entities, all of them carry the IDs of the composing elements to make the logical connections to the element classes, where the data are physically stored.

More attributes can be added in the entity and element classes to fit different application requirements.

3.3.12 Defining inheritance

This step is to discover generalizations, which are usually called superclasses, and specialization, which are usually called subclasses, within similar domain types.

Finding semantically correct inheritance structures provides good reuse, because the states and behaviors of the superclass do not have to be rewritten for each of the subclasses. It also allows simplification, since developers can work with the specific or general object as appropriate. Generalization and specialization are the main tools used to do this job.

As these seven class types are designed as fundamental modeling units to compose other classes with more specialties, they are superclasses themselves without much inheritance occurring among them. Only the definition of point type inherit from the node type. But whenever building an application based on this, inheritance can improve reuse of code greatly.

3.4 System Design and Implementation

The previous analysis steps focus on understanding the domain and abstracting the computational model. System design focuses on how the computational model can be implemented. It is the process of expanding what was learned from the domain analysis and then determining effective, efficient, and cost-effective implementation to carry out the functions and store the data defined in domain analysis (Figure 3.19).

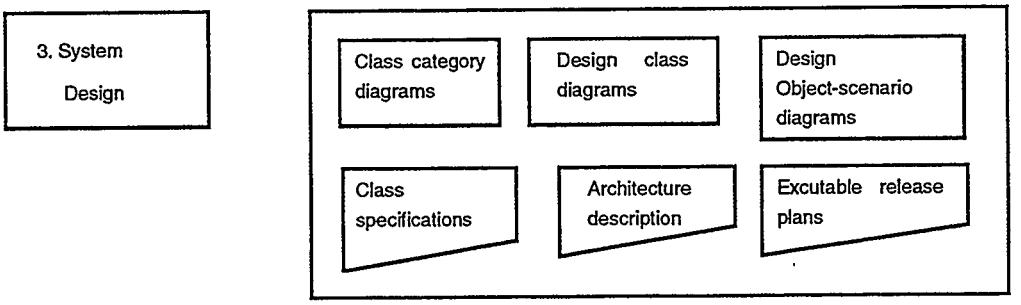

Figure 3.19 System design step and the deliverables

As more details are required for a working implementation than for the domain analysis, an iterative approach rather than a full scale leap is even more imperative than it was during domain analysis. The Booch method encourages series of smaller steps and graduate integration leading to a working system.

3.4.1 Structure of the prototype

How the functional components are organized to construct a software system is usually referred to as system architecture. It is one of the major standards to judge the quality of a system, since a clean and efficiently organized internal structure makes a system easy to understand, maintain and extend. As stated in Chapter 2, integrated architectures show more advantages.

During system development, the complicated integrated system can be decomposed into loosely coupled partitions which carry out relative independent and complete functionalities. In traditional design, partitions are groups of functions or procedures. In the Booch method, these partitions are called class categories, groups of classes cooperating in certain use cases.

Interface should be clearly defined and provided between class categories to accomplish the interactions. The interfaces tell the other categories of the system what kind of functionalities are provided without specifying how they are implemented internally. This guarantees the independence of each class category's design and

implementation. During the system maintenance, the modification can be limited within certain class categories.

Two main class categories are identifed in this prototype: the spatial entity and geometry element groups. According to the design principle, they are designed with different emphasis. The element group concentrates on the positional information of each element and the relationships among them. Whereas the entity classes group concentrates on the non-spatial information, such as what kind of objects they represent, how they are displayed on a map, etc. The entity classes have logical pointers to their composing elements to contain the access to positional information. To make it easier for end users to use the classes, some simple MS Windows style interfaces are implemented based on the Object Window Library (OWL) provided by Borland C++ which developers can use to customize for their own window style interface. This brings the third class category in the prototype. Figure 3.20 shows the Class Categories in this prototype.

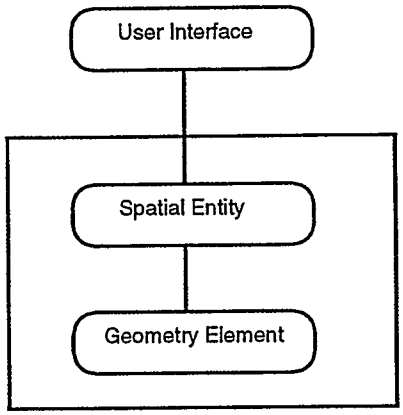

Figure 3.20 Class categories in the spatial data modeling prototype.

3.4.2 Form of implementation

As this research focuses on object-oriented spatial data modeling, the prototype is designed to be a fundamental one which other GIS applications can make use of and build on. The prototype is not implemented as an executable system including all the popular GIS functions, instead, it is implemented as a group of classes providing basic modeling components or units, as well as modeling construction functionalities. They can work as the kernel of data modeling module to handle the data conversion and storage. To make use of these capabilities, an application systems can embed the classes just in the same way as including other standard classes in C++. For particular applications, other tools can be implemented by adding more classes for more convenient interface, more complicated retrieval, more sophisticated processing functions, etc. This provides flexibility for users to utilize the modeling capabilities of this prototype for different purposes.

3.4.3 Implementation of modeling functionality

The prototype is designed as an object-oriented modeling fundamental group for spatial information, so one of the focus of the implementation is the supporting modeling functionalities and object-oriented features discussed in Section 3.2.3. In this section, how these functions and mechanisms are implemented is going to be reviewed.

Class and encapsulation

All types of modeling components are implemented in the form of classes. As analyzed before, to accomplish the required functionalities for the whole system, each class should carry part of responsibilities which considered to be its own behavior. For example, a node class includes the following main functions: *Node(), ~Node()* for constructing and destructing node object; *Read( ), Write( )* for reading and writing data about the node; *GetConnectedArc(), AddConnectedArc(), RmConnectedArc()* for getting, adding and deleting the IDs of connected arcs, etc. Figure 3.21 shows the main definition of the Node class.

```
class Node
    {
    protected:
        long int  ID;
        EleArray  ConnectedArc;
        void Read( );
        void Write( );
    public:
        Node( );
        ~Node( );
        long int  GetID( );
        EleArray GetConnectedArc( );
        EleArray AddConnectedArc( );
        EleArray RmConnectedArc( );
                :
    }
```

Figure 3.21 Main attributes and functions in class Node.

In fact, all the classes have included functions for creating and maintaining the corresponding objects, changing the components of an entity to reflect the change of real situation. Thus the classes are relatively complete modeling units. Directly including these classes in an application can access these functionalities of object creation and operation to produce proper computation models for various applications.

As an object-oriented programming language, C++ provides basic support for some of the mechanisms expected in the prototype, thus using C++ surely facilitates part of the work. Implementation of encapsulation is a good example. Since every type of modeling components is implemented as classes, the encapsulation of state and behavior within model components is achieved through the use of the class in C++. As analyzed before, to accomplish the required functionalities for the whole system, each class carries part of responsibilities such as creating and maintaining the corresponding objects, changing the components of an entity to reflect the change of real situation. These duties are implemented in the form of functions as part of a class. While using C++ to instantiate objects from the defined classes or inherit and specify new classes with more features added to build up an application system, encapsulation is still obtained.

Message passing and inheritance

Similarly, communication with messages is also directly supported by C++, as well as with the inheritance mechanism. Besides the inheritance already existing among the classes defined in the prototype, all of the classes can be superclasses that others can

be derived from. For example, a national park can be defined as a subclass of the class ComplexEntity, inheriting all the state and behavior from the class ComplexEntity instead of defining them again. Of course, more features can be added to meet the need of a particular application. With this support, the hierarchical structure of the modeling can be obtained based on the classes defined (Figure 3.22). The very roots are the geometry element classes, providing fundamental functions such as positional information modeling, then the entity classes providing information of geometry construction. New classes can be derived by specialization of these basic types with more special functions and states introduced for a particular purpose.

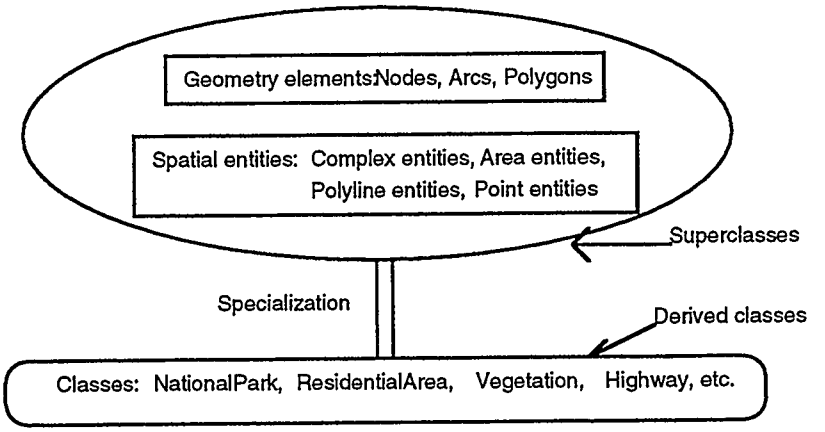

Figure 3.22 Hierarchical structure of the spatial information modeling

It was found that the *template* feature of C++ could also promote code reuse in GIS applications. Template provides a mechanism for indicating those types that need to change with each class instance [Lippman 1995]. This is done by parametrizing the types within a template class definition. Not like inheritance enabling developers to share code

between classes on different levels of the inheritance hierarchy, template enables developers to share code between objects on the same level of hierarchy. This feature can be very useful for modeling entities sharing the same attributes and functions but of different types, which is very common in GIS. But this feature was not used in the project because the data used were relatively simple.

Aggregation

As mentioned before, object-oriented programming languages support complex objects. These allow the construction of complex classes based on already defined classes by directly declaring them as parts of the complex class. This implies one of the key modeling capability expected in this prototype: aggregation. In practice, there are two ways to implement this capability. One way is to directly declare a defined class as a member of a new class. Another way is to include an ID of the defined class as a logical pointer in the new class to make a connection to the defined class. The first method is very practical for the situation when immediate access to an entity's components is necessary, whereas the second one is clearer and more concise in structure and efficient in storage. In fact, definition of class *ComplexEntity* is an example of using the complex class mechanism to generate complicated class types based on pre-defined simple class types. Figure 3.23 illustrates how *ComplexEntity* includes other class types as its members. The *EntArray* is a pre-defined array type containing the IDs of the component entities such as points, polylines in a complex entity. The IDs logically point to the

objects of the component entities where the detailed information is stored. Adding or removing a component from a complex entity is achieved by maintaining the array of IDs, as shown in the Figure 3.23.

```
class ComplexEntity
    {
    protected:
        long int  ID;
        EntArray  EntityIDs;
    public:
        ComplexEntity ( );
        ~ ComplexEntity;
        float  GetID( );
        EntArray GetEntity( );
        EntArray AddEntity( );
        EntArray RmEntity( );
        float GetArea();
        float GetPerimeter();
        EleArray GetBound;
        EleArray GetBoundPoint;
        void Remove ();
                :
    }
```

Figure 3.23 Main attributes and functions in class ComplexEntity.

With the support of aggregation, more complex class types can be generated by utilizing the basic classes. Extending the basic element to model more complicated spatial entities can be much more straightforward.

3.4.4 Object indexing

Indexing of data is the key factor in retrieval efficiency. According to various GIS applications, the index could be built on certain non-spatial features or on spatial positions. Due to the specialty of GIS, spatial indexing is usually more important and more challenging. For years, lots of researchers have worked on it, and lots of methods, such as space-filling curves, quadtrees, R- and R+-trees, etc, have been proposed and applied successfully [Laurini et al. 1992]. No matter what kind of index it is and what kind of algorithm is used, the indexing result is usually a collection of pointers (physically or logically) pointing to the stored physical data in certain orders to guide retrievals. So in this research, instead of concentrating on discussion on the mature indexing algorithms, some efforts have been made to design a new class to contain the indexing result from an indexing program. Thus this class actually serves as an interface between indexing programs and the data the index applied on (in this prototype, objects). The following information is included in the class to make the connection: object ID and its relative position in the array, size of an index array, etc. Users can write indexing programs using certain algorithms and put the results into the array of index objects. Figure 3.24 illustrates the main member attributes and functions of the index class.

```
class Index
    {
    protected:
        long int  ID;
        int  Offset;
        char * IndType;
      IndArray ObjectIDs;
```

```
  int num;
public:
      Index( );
      ~Index( );
       long int  GetObjectID( );
      int  GetOffset( );
      IndArray AddIndex();
                :
```

Figure 3.24 Main attributes and functions in class Index.

Chapter 4

AN IMPLEMENTATION FOR CREOSOTE PROJECT USING THE PROTOTYPE

In this project, the main purpose is to investigate the applicabilities of object-orientation in spatial information modeling. As described in Chapter 3, a general object-oriented spatial data model is first implemented as the core of a prototype. In this chapter, discussion will be made on how the data model can be adapted to construct an application module to model and manipulate DEM data obtained in the creosote project.

4.1 Creosote Project

4.1.1 Creosote Problem

Creosote is a kind of compound obtained from distillation of coal tar. It is a colorless or yellowish oily liquid containing a mixture of phenolic compounds. Being denser than water, it can penetrate the vadose and ground water zones. Between the early 1920s and 1964, a large amount of creosote was introduced into the soil and ground water by the former Canada Creosote Ltd. plant located on the Bow River in downtown Calgary. Release of creosote into the subsurface have resulted in the accumulation of significant quantities of creosote liquids in the sand and gravel aquifer underlying the site. Recently, blobs have been observed in the Bow River due to this creosote extensive discharge.

4.1.2 Objectives of creosote project

Several groups of scientists have been working on the creosote problem. The research work include evaluating the creosote migration and dissolution at the Creosote Site in Calgary and studying the mechanism of creosote movement into the Bow River at the same site. Our group has carried out research on the construction appropriate evolutionary spatial model of the creosote site to help scientists to visualize the surface changes over the past sixty eight years and decide on a proper course of action. The main idea is, first, to use photogrammetry technologies to reconstruct the Digital Elevation Models (DEMs) from available historical data, in this case, twelve pairs of historical aerial photographs. Then the corresponding images has been processed and draped over the reconstructed DEMs for more realistic rendering and visualization. Morphing software has also been used on the DEMs to generate intermediate views and demonstrate evolutionary changes of the creosote site over the years [Blais et al. 1995].

4.1.3 Data used in creosote project

In order to reconstruct evolutionary digital terrain models with good quality, it is required to find all spatial information available about the creosote site on the Bow River, dating back to as early as 1924. Historical aerial photographs and control information, as well as some related environmental data for the past sixty-eight years (from 1924 to 1991) were obtained from the City of Calgary, Alberta Environment and MacKimmie Library in the University of Calgary. On consideration of photo quality and reasonable

time intervals, twelve pairs of aerial photographs, corresponding to epochs of: 1924, 1949, 1951, 1953, 1956, 1958, 1966, 1975, 1979, 1980, 1982 and 1991 respectively, were selected and digitized at 450 dots per inch (dpi) resolution.

4.2 Reconstruction of the DEMs

4.2.1 DEM Measurements

The PC based system Digital Video Plotter (DVP) [DVP, 1991] was used to construct the stereomodels. After sufficient fiducial marks were located on the whole twenty four photos, the DVP software was used to carry out the interior orientations and camera calibrations with the expected accuracy. Then for each epoch, from four to six ground control points were used to carry out the absolute orientation. Due to the fact that quality and completeness of the available data varied from one epoch to another, selection of the control points for absolute orientation was complicated. Details can be found in the final report of this project [Blais et al. 1995].

Following the generation of the stereomodels, regular grids of 131 x 46 points (1 point every 10 pixels) for each epoch were measured with DVP, where each point has X, Y and Z coordinates. So about 6000 points were measured for each of the eleven epochs, except for the 1924 model. The photos for 1924 do not cover the area of interest completely, as the north and west parts are missing. So a smaller grid of 112 x 30 points was measured. Surface reconstruction of the creosote site can then be carried out based on the measured grids of points.

4.2.2 Triangulation and Gridding

Triangulation and gridding were also used in this project to fill in the blank areas in the DEM coverage provided by the City of Calgary and to form a regular grid over the terrain for the whole creosote site area. They were also used to densify the grid data in order to provide better visualization of the reconstructed ground surface. The mathematical library IMSL, which is available on IBM RS/6000 computer on the University of Calgary campus, was used in the gridding program GRID to process the data sets and produce gridded data. A Delaunay triangulation was carried out internally by the program GRID and a smoothing algorithm was applied to the interpolated data to form a smooth grid surface.

4.2.3 Results of measurement

After the above described operations, DEMs of twelve epochs were reconstructed with a 10m by 10m resolution. The whole creosote site is contained in a 1300m by 450m area.

4.3 Geometric Transformation of Digital Images

A geometric transformation was applied to the digital images before they were draped over the corresponding reconstructed DEMs for visualization in order to make sure that the extracted image does geometrically correspond to the ground DEM grid. Figure 4.1 shows the relationship between an aerial photo and ground DEM in two

dimensions disregarding parallax effects between the photo and the ground spatial coordinate system.

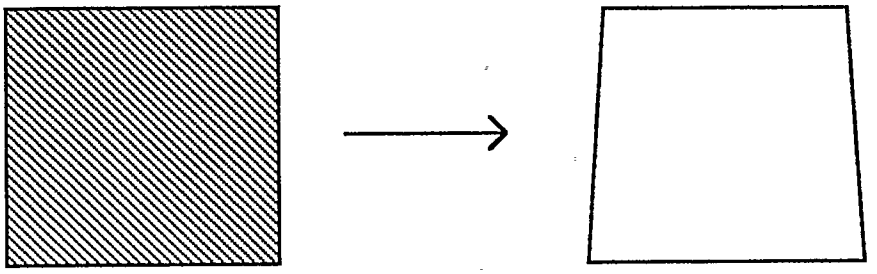

Figure 4.1 Relationship between ground area and corresponding photo area.

To transform the ground area into the corresponding photo area, a 2D projective transformation was applied. The formulas of a 2D projective transformation are:

$$X = \frac{ax + by + g}{ex + fy + 1} \qquad (4.1)$$

$$Y = \frac{cx + dy + h}{ex + fy + 1}$$

where X, Y are the photo coordinates, x and y are the original ground coordinates, and a, b, c, d, e, f, g and h are the coefficients which can be estimated with a least-squares method.

In the ground system, the size of the creosote site area is 1300 m x 450 m. By maintaining the relative scale factor between the two axes and considering the size of all the images that cover the same area at different epochs, the optimal size for the transformed images was chosen to be 2200 pixels x 760 pixels. Six to nine points with

recognized ground features were selected within the creosote area to be used as control points for each epoch and the corresponding photo coordinates of the control points of the ground area have been obtained with DVP and then used to solve Eq. (4.1) for the coefficients. Then, these coefficients are used to compute the corresponding coordinates of the original image for each pixel of the output image so that the images have a uniform size of 2200 pixels x 760 pixels. The gray value of the point located at these coordinates is then interpolated by using a nearest neighbor resampling approach and assigned to the corresponding point in the output image. The processing has been done on either the left or right photos of the stereomodels covering the creosote site.

4.4 Generation of Intermediate Views Using Morphing

In order to visualize the evolutionary changes of the creosote site, the generation of intermediate views was proceeded to fill up the gaps between the available epochs. One 3D model for each year from 1924 to 1991 has been generated. This operation has been done in two steps. The first step is to use a simple linear interpolation approach to generate the DEM grids between two available epochs. For example, eight interpolation values have been estimated for each elevation data between 1982 and 1991.

The second step consists in metamorphosing one image into another. In image processing terms, metamorphosing is replaced by the new term "morphing". Another linear interpolation of grey level has been implemented to generate intermediate views between existing epochs (Figure 4.2).

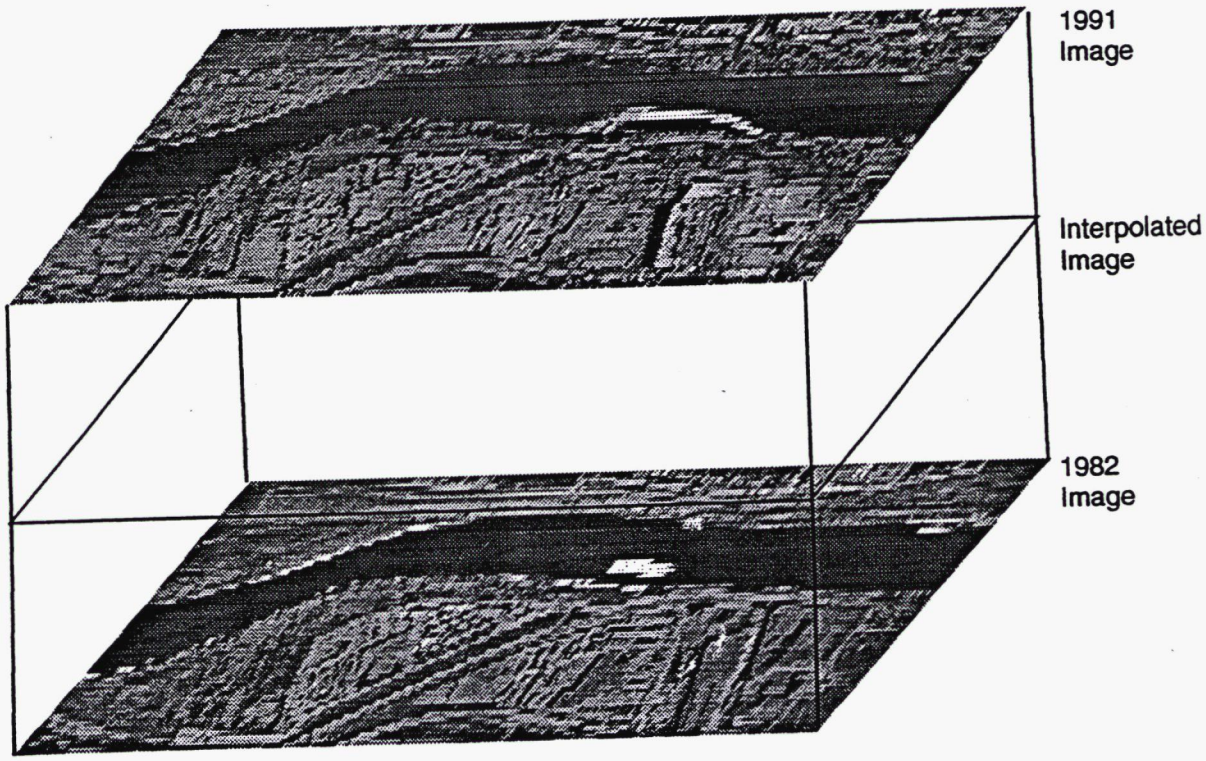

Figure 4.2 Interpolation between 1991 and 1982.

4.5 Rendering the 3D Surface

Visualization of the DEMs with images draped over is one of the most important parts of this project. With all the DEMs of the twelve epochs completely reconstructed and other fifty six epochs interpolated, as well as all the corresponding digitized photographs preprocessed in the way described in the previous section, the surface of the creosote site can be visualized with the corresponding images draped over. The Advanced Visualization System (AVS) on IBM RS6000 was used to reconstruct the 3D geometry view of the DEM for the creosote site based on the measured elevation data

sets for sixty eight epochs. Then the images were draped over the re-generated 3D model to rebuild the historical scenes of the creosote site in three dimensions over the sixty-eight years. Figure 4.3 illustrates this procedure.

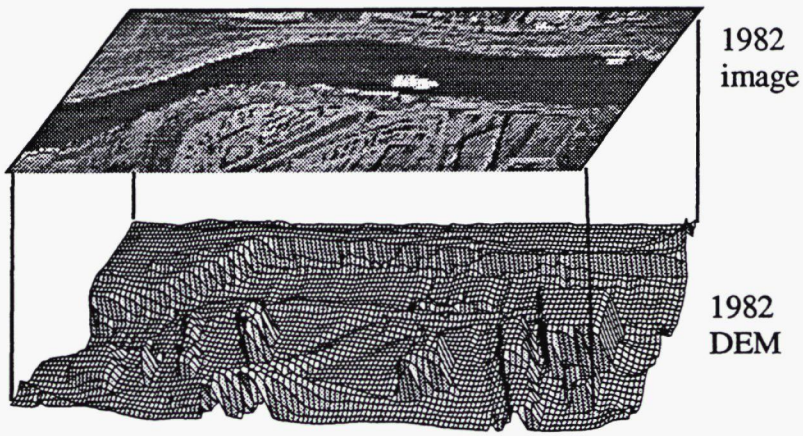

Figure 4.3 Draping of image over the resconstructed DEM of the creosote site.

For visualization purposes, the following two groups of images are also generated in TIFF format by using AVS: the images of the perspective view of DEMs reconstructed; images of the reconstructed DEM with corresponding images draped over to provide more information.

4.6 A DEM Data Management Module

The continuous change for the creosote contamination site can be viewed by the series of generated 3D surfaces. It could be also very helpful to have some incorporating tools to manage and retrieve the elevation information when users require detailed elevation for certain positions at a certain time. So an application module was designed and implemented using the developed prototype of spatial data model to provide the following functions: DEM data conversion and storage, elevation data query, image display.

4.6.1 DEM data modeling

In GIS applications, there are several ways to model the DEM information, such as a grid, an irregular triangulated network (TIN), contour lines surface, gradients, etc. In grid format, only points at the intersection of two imagined orthogonal lines in x and y are stored. To fit the real condition of the irregular terrain, tighter grid lines are used in the area with much roughness, while sparse grid lines are used for relatively plain area. However, regular gridding lattice (Figure 4.4) with the finest gridding interval is much more often used due to the ease of generation, maintenance and processing. In this project, a regular grid is the final generated data format to represent terrain.

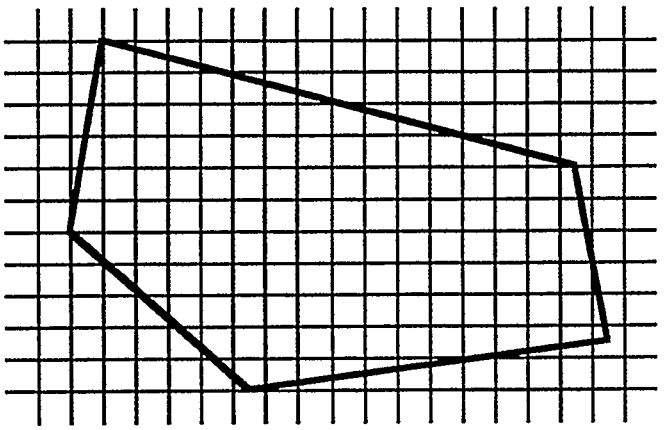

Figure 4.4 A regular grid lattice for the terrain.

Figure 4.5 is the class diagram for using the implemented object-oriented prototype to model the regular grid DEM data. Only a new class of *EleNode* needs to be defined in C++ based on the node class type. The *EleNode* can directly inherit from the basic model component *Node* class with *TerrainID, Elevation,* etc. additional attributes to hold position and elevation information and some more functions defined and re-defined. So the terrain model can be built directly on the *EleNode* class type.

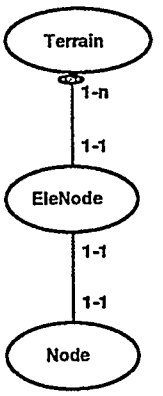

Figure 4.5 Class diagram for grid model of DEM.

4.7 Tool Implementation

When the C++ class types have been added, the model for DEM information has been constructed. Information can be stored and some tools can be implemented to manipulate and view the information.

4.7.1 Data construction

The original data sets are stored in flat files as a series of (*X, Y, ELEVATION*). The data construction function is designed to convert the data set for the new model. It reads the regular gridded DEM data set, sends appropriate messages to the model handling function to generate and connect model components, and put the actual values into them. A member function *PutElevation( )* is implemented to carry out this task. Once the objects for the model have been constructed, they can be saved and accessed by other C++ tools later.

For information retrieval efficiency, the data need to be indexed. Member functions *IndexOnElevation()* and *IndexOnLocation()* were coded to generate index on elevation and position, corresponding to the most possible situations of retrieval occurrence. The indexing result containing a list of objects' IDs in ascending order of elevation then have been stored in the objects of Index class type described in Chapter 3. The lists can be used to speed up retrievals when a query requires them. Figure 4.6 list the main state and functions defined in the *EleNode* class.

```
class EleNode: public Node
{
        protected:
            long int  ID;
            long int  TerrainID;
            float   Elevation;
        public:
            EleNode( );
            float  GetElevation( );
            float  PutElevation( );
            void  IndexOnElevation( );
            void  IndexOnLocation( );
                    .
                    .
```

Figure 4.6 Main attributes and functions in class EleNode.

4.7.2 Information query

The information query tool is implemented to retrieve and display the stored DEM model consisting of time, elevation and position information. The retrievals are mainly in terms of one of the above three items versus the other two. This tool is implemented using MS Windows style interface to enable the user to interactly select query fields, query criteria and view the query results.

For example, a user may want to know in 1982, which areas in the creosote contamination site have elevation between 1052.00 m and 1053.50 m. This is a query on positions versus time and elevation. Using a common dialogue box as an interactive selection mechanism, the user can first choose the type of selection criteria. In this example, the query is applied on position so the button besides 'Position' is clicked on. Then the values of selection criteria are entered (Figure 4.7). This information is then

passed to the search function and the function filters the points meeting the criteria and then a new window pops up containing the list of the points (Figure 4.8).

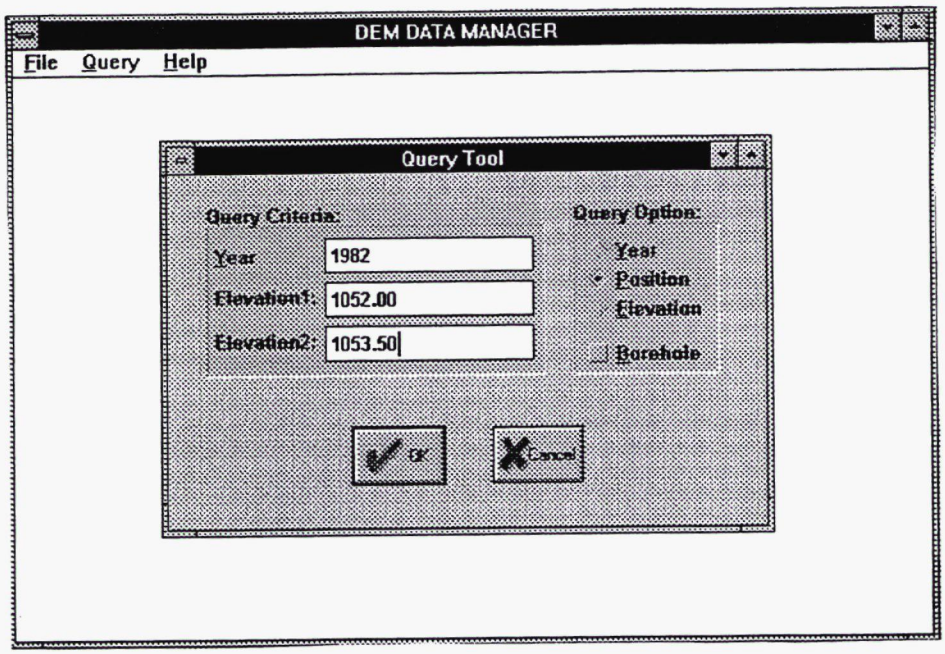

Figure 4.7 Dialogue box for query field selection and query criteria input.

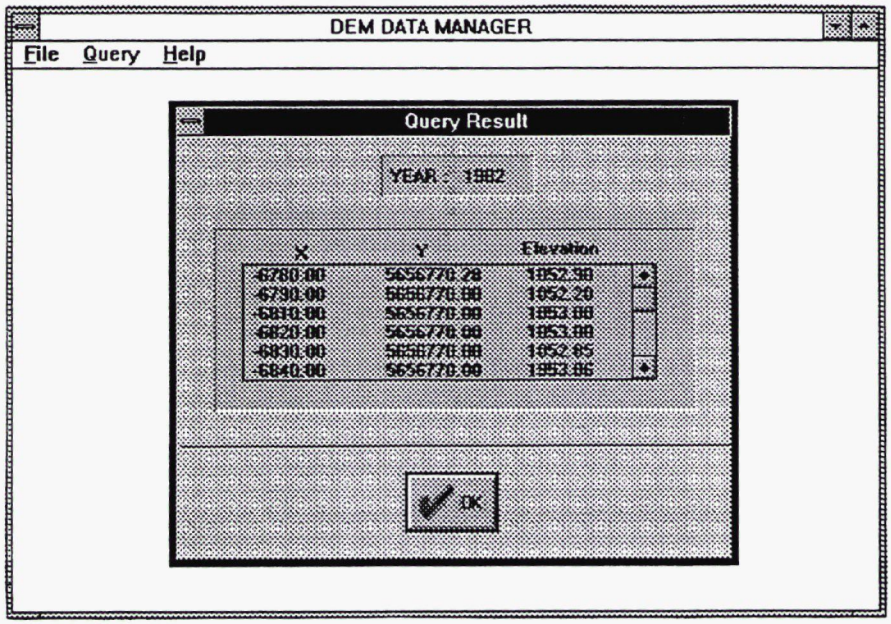

Figure 4.8 Query list result.

Another type of necessary query function in this project is to find out the nearest boreholes to a certain point. This module allows users to select the borehole query option, enter the coordinates of the point, then calculate the distances between this points and boreholes, pop out the coordinate list of the first ten nearest boreholes. Figure 4.9, 4.10 show an example of this type of query.

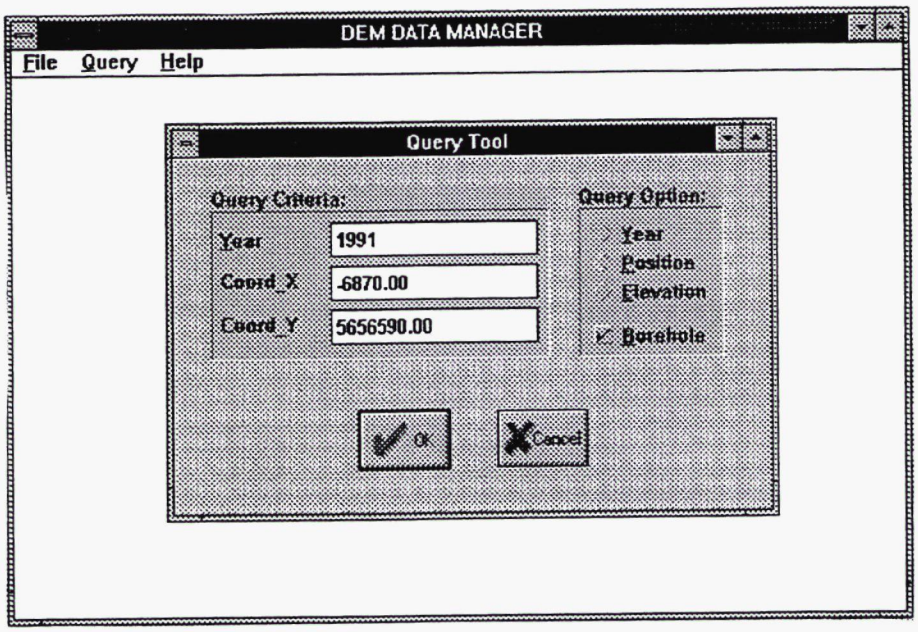

Figure 4.9 Dialogue box for query fields selection and query criteria input.

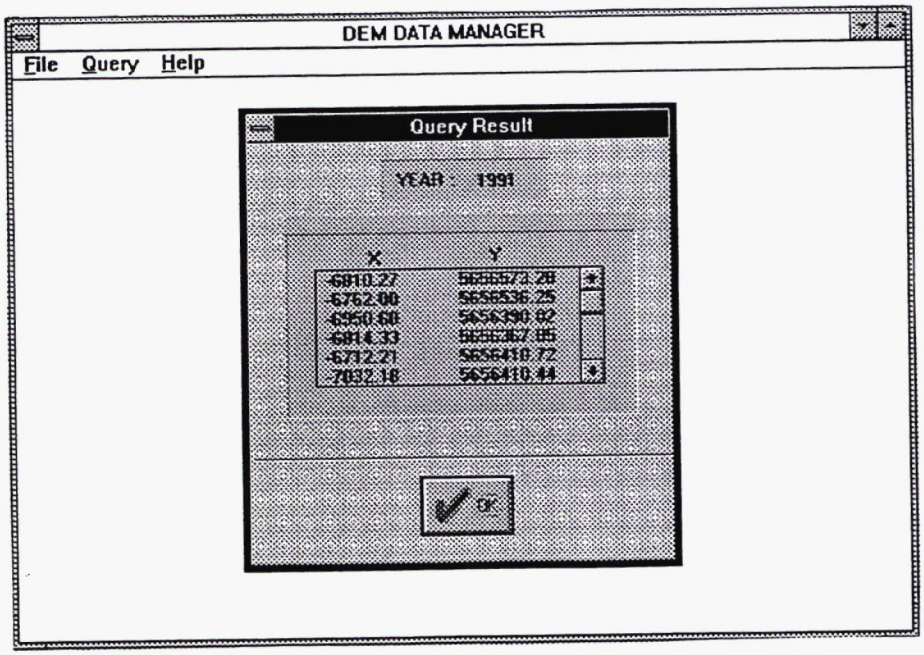

Figure 4.10 Coordinate list of the queried nearest boreholes.

4.7.3 Image viewer

Two groups of images for the geometry reconstruction of the DEM were obtained as an important part of the information. A tool named IMG_VIEW is also implemented to handle these images to help to visualize the information. Again, featured with Microsoft Windows style interface, this tool can be used with ease.

The name of the image file is retrieved from the user using the common dialogue box shown in Figure 4.11, which is provided with Microsoft Windows. This allows the users to make their selection with ease and flexibility.

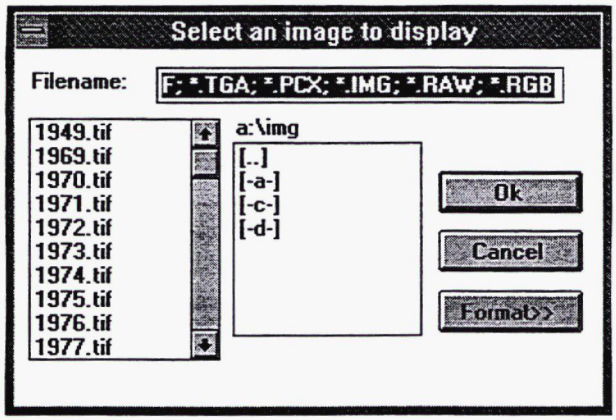

Figure 4.11 The dialog box for image selection.

The image_viewer display the image at the top-left of the window; however, if the window is too small, the edges of the image may extend beyond the boundaries of the window. To deal with this kind of situation, a scroller control feature is added in the

display window (Figure 4.12). A user has the flexibility of utilizing the scroller bar to move the image up and down, left and right in the window.

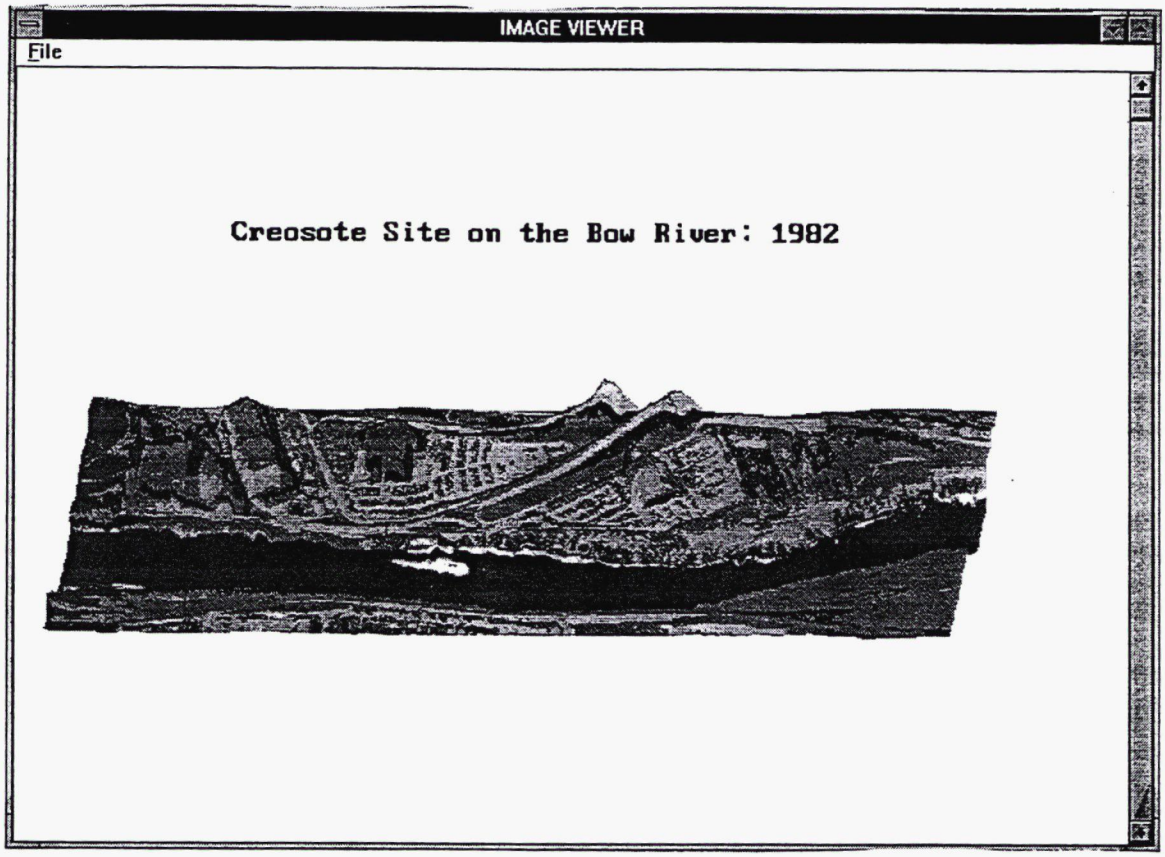

Figure 4.12 Image viewing window.

This implementation is a test of the applicability of the object-oriented spatial data modeling prototype. Due to the nature of data set used in this project, only NODE and INDEX classes were applied for DEM modeling. But this does not imply that only these two classes are useful. As a matter of fact, for applications which involve more types of spatial features, the other classes are very useful to model various complex features. One

ongoing research project is to use this method in geomorphological and hydrological applications. All the seven classes are being used to model the complicated geomorphological features, such as channel and ridge networks. The prototype is designed to be adaptable to various type of applications, simple or complex.

Chapter 5

CONCLUSIONS AND RECOMMENTATIONS

The main purpose of this project was to investigate the applicabilies of object-orientation in spatial information modeling. The experiments have included investigations of using the Booch Method to analyze and design a general object-oriented data model for representing geospatial information, implementation of the prototype data model in C++ and adaption of the model for an application handling the data set obtained in the creosote project. This chapter summarizes the main conclusions and recommendations obtained in this research and development.

5.1 Conclusions

1. Traditional logical models like hierarchical and network models have limitations in handling the complexity of spatial information. The relational model is the dominant logical model applied in current GISs. Simplicity and data independence are the major features of a relational DBMS, but these features can also cause deficiencies in the following sense for spatial information modeling: manipulation efficiency, modeling capability, data semantics, modeling extension, programming interface, etc. With its ability to address these deficiencies, the object-oriented model shows great potential in GIS applications.

2. The object-oriented analysis and design (OOA & OOD) method is the integration of the development of information system analysis and design methodologies and object-oriented programming. Using OOA & OOD and an object-oriented programming language for software development can help implement object-oriented mechanisms in the system more thoroughly.

3. As the basic modeling unit in object-oriented analysis and design, a class encapsulates both state and behavior features. The encapsulation makes the modeling closer to a one-to-one mapping procedure. During the modeling, a developer can concentrate on **what role an entity takes** (what it can do) in the world, instead of **what kind of role it takes** and **how it carries out its task**. Thus not only does the development of the prototype benefited from it, but also the utilization of the prototype to construct the application module had a smoother implementation using this idea. Of course, it is still far away from a human being's way of thinking. But compared with conventional strategies, such as in relational modeling, object orientation is one step closer to it, thus more natural, more direct and more efficient.

4. The complex object and class mechanisms provide the aggregation support in object-oriented modeling. This enables the system to model complex entities by simply composing the fundamental class types instead of going into very primitive elements. The definition details are shared by different components with each one handling certain information and functionalities to facilitate the application needs. This is very

beneficial for modeling geometric entities and general information. The modeling efficiency can be improved greatly, and system extension can have more flexibilities.

5. The inheritance mechanism further enhances an object-oriented systems' modeling capabilities. An inheritance hierarchy can clearly show the similarities shared by the interested entities, and the sharing implies reusing of code or specification of model components, which results in a dramatic increase in modeling and programming productivity. In GIS modeling, developers only need to consider the special features of certain interesting entities while building or extending an application based on well defined fundamental class types.

6. Smoother program interfacing can be achieved through the similarities shared by object-oriented programming languages (C++ in this project), the Booch method and the way to organize the data in the prototype. During the development of the prototype, the analysis and design using classes as the basic working units and the standard deliverables of each step describe the model in a method very close to object-oriented programming language, which can ease the implementation. During the utilization of the prototype, because the class is the common building block of the programming language and the data model, the programming gaps between the two systems are thereby reduced. Model extension and Microsoft Windows style interface development benefited greatly from this.

7. Compared with the conventional method, data semantics in object-oriented spatial data models has been enriched greatly because the object-oriented system can distinguish between classification, generalization and aggregation due to the support of class, complex class types and inheritance. This eases the understanding and usage of the spatial data model.

8. Direct support of class and complex class types imply logical and physical pointers connecting various information components, either physically or logically. Information access is usually achieved by navigation through pointers and no physical operations are needed to reconstruct the information from pieces like in the relational model. So the information access becomes more direct and straightforward both conceptually and physically, and manipulation efficiency is improved.

5.2 Recommendations

This project aimed at exploring how the object-oriented mechanisms can be applied and implemented in GIS applications instead of building a working system, the data set used and the experiments done are relatively limited. Recommendations for further research are as follows:

1. The development of the prototype has concentrated on design of general spatial data model which can be adapted for some specific GIS applications, such as the application for handling the DEM data set. To extend it to a complete and general

purpose GIS, more design in terms of data creation, storage and spatial analysis functions, etc., should be considered from the initial analysis phases. These are very important components in GIS and need further investigation.

2. In the implementation, data operations and system extensions still need C++ coding, which is not suitable for general users. More research should be done to investigate some high level language, such as CO2 in O2 system, to provide easier data manipulation.

3. In the application module for the creosote project, most of the data queries are relatively straightforward, and limited in query types and complexities. More investigation should be done on more general data retrievals for different kinds of situations.

4. The data set used was relatively small and simple. Larger size and more general types of data sets should be tested to investigate various kinds of situations. Moreover, analysis of storage and manipulation efficiency should be performed to refine the system.

5. This study can provide some general ideas for data modeling and system architecture for the ongoing Crown of the Continent Environment Information System (EIS) development [Blais, 1996]. The Crown of the Continent project involves multidisplinary users, multitype and multiresolution data collections, multilocation computer systems in southwestern Alberta, southeastern B.C. and Western Montana.

An integrated object-oriented architecture GIS obviously is the most optimal design approach to facilitate the variety of requirements. Of course, as mentioned above, more comprehensive studies on design and implementation need to be done to build a system of such scale. A recent doctoral study on handling large objects using object-oriented approach in GIS [Zhou, 1995] also provides very valuable experience on data storage facilities, query tools, spatial analysis functions, data conversion utilities, etc, in GIS development.

6. Most of the object-oriented analysis and design methods, including the Booch method, do not have sound theoretical support. System design and implementation still largely depend on developers' experience. More study in this fields can improve the design quality and consistency for GIS and other applications.

References

Abel, D.J. [1989]: SIRO-DBMS: A Database Tool Kit for Geographical Information Systems. International Journal of Geographical Information Systems 3, pp.103-115.

Bancilhon, F., C. Delobel and P. Kanellakis (editors) [1992]: Building an Object-oriented Database System: The Story of O2 (San Mateo, California: Morgen-Kaufmann).

Bancilhon, F. [1988]: Object-oriented Database System. In Proc. of ACM Symposium on Principles of Database Systems. Austin TX. May 1988.

Blais, J.A.R., W. Zhou and A.W. Colijn [1996]: On the Optimal Design of an Environmental Information System for the Crown of the Continent, Proceeding of 96's Canadian GIS Conference, Ottawa, on CD.

Blais, J.A.R., K. He, C. Larouche and C. Zheng [1995]: Final Report on the Project of Evolutionary Spatial Modeling of Creosote Site on the Bow River, Calgary, Dept., of Geomatics Engineering, The university of Calgary.

Blais, J.A.R. [1994]: On Database Considerations for Geoscience Information Systems, Proceeding of 94's Canadian GIS Conference, Ottawa, pp.912-923.

Blais, J.A.R. [1987]: Theoretical Considerations for Land Information Systems, The Canadian Surveyor, vol 41. No. 1, pp. 51-64.

Bertino, E. and L. Martino [1993]: Object-oriented Database System, Concepts and Architectures, Addison-Wesley.

Booch, G. [1994]: Object-oriented Analysis and Design with Application (Redwood City, California: Benjamin/Cummis).

Chen, P. [1976]: The Entity-relationship Model Toward a Unified View of Data. ACM Trans. on Database System, 1(1), pp.9-36.

Coad, P. and E. Yourdon [1991]: Object-oriented Analysis, Prentice-Hall, Inc.

David. R., I. Raynal, G. Schorter and V. Mansart [1993]: GeO2: Why Objects in a Geographical DBMS. In Advances in Spatial Database (Proceedings of the Third Symposium Database), volume 692 of Lectures Notes in Computer Science, edited by D. Abel and B. C. Ooi (Heidelberg: Sprinder-Verlag), pp. 264-276.

DVP Geomatic System Inc. [1991]: User Guide.

Egenhofer, M.J. and A. Frank [1992]: Object-oriented Modeling for GIS. Journal of the Urban and Regional Information Systems Association, 4, pp. 3-9.

Faison, T. [1995]: Borland C++ 4.5 Object-oriented Programming, Fourth edition, SAMS publishing.

Gamelin, T.W. and R.E. Greene [1983]: Introduction to Topology, Saunders college publishing.

Garvey, M.A. and M.S. Jackson [1989]: Introduction to Object-oriented Database, Information and Software Technology, vol. 31, pp. 521-528.

Goldberg [1983]: Smalltalk-80: The Language and Its Implementation, Addison-Wesley.

Herring, J. [1992]: TIGRIS: A Data Model for an Object-oriented Geographic Information System. Computers and Geosciences, 18, pp. 443-452.

Herring, J. [1987] TIGRIS: Topological Integrated Geographic Information System, In Auto-Carto 8, pp. 282-291.

Hull, R. and R King [1987]: Semantics Database Modeling: Survey, Application and Research Issues. ACM Computing Surveys, 19(3), pp.201-260.

Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard [1992]: Object-oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley Publishing Company.

Jänich, K. [1984]: Topology, Springer-Verlag.

Koshafian, S. [1993]: Object-oriented Database (New York: Wiley).

Koshafian, S. and R. Abnous [1990]: Object-orientation: Concept, Languages and Database (New York: Wiley).

Laurini, R. and D. Thompson [1992]: Fundamentals of Spatial Information Systems, Academic Press.

Lee, Y.C. [1990]: A Comparison of Relational and Object-oriented Models for Spatial Data. Technical Report, The University of New Brunswick.

Lippman, S.B. [1995]: C++ Primer, 2nd Edition, Addison Wesley.

Lukatela, H., G. Murdeshwar and M. Shandro [1989]: An Object-oriented Database for GIS, Unpublished Manuscript.

Maier, D. [1982]: The Theory of Relational Database (Rockville, Marland: Computer Science Press).

Milne, P., S. Milton and J.L. Smith [1993]: Geographic Object-oriented Database: A Case Study, International Journal of Geographical Information Systems, 7, pp. 39-56.

Muller, J. [1993]: Latest Developments in GIS/LIS, International Journal of Geographical Information Systems, vol. 7, No. 4, pp. 293-303.

Newell, R. [1992]: Practice Experience of Using Object-oriented Database to Implement a GIS, In Proceedings, GIS/LIS Annual Conference (Bethesda: ASPRS and ACSM), pp.624-629.

Ross, D.T. [1986]: Applications and Extensions of SADT, IEEE Computer, April, pp. 98-106.

Rowe, L.A. and M.R. Stonebraker [1987]: The Postgres Data Model. In Proceedings of the 13th VLDB Conference, edited by P. Stocker and W. Kent (San Mateo, California: Morgan Kaufmann Publishers), pp. 83-96.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen [1991]: Object-oriented Modeling and Design (Englewood Cliffs, New Jersey: Prentice-Hall).

Smith, P.D. and G.M. Barnes [1987]: Files & Database, An Introduction, Addison-Wesley.

Snodgrass, R.T. [1992]: Temporal Databases. In Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, volume 639 of Lecture Notes in Computer Science, edited by A. U. Frank, I. Campari and U. Formentini (Heidelberg: Springer-Verlag). pp. 22-64.

Tansel, A.U., J. Clifford, S. Gadia, J. Sushil, A. Segev and R.T. Snodgrass (editors) [1993]: Temporal Databases: Theory, Design and Implementation (California Benjamin/Cummings).

Unisys [1994]: System 9 Manuals.

Van Oosterom, P. and J. Van den bos [1989]: An Object-oriented Approach to the Design of Geographic Information Systems. Computer and Graphics, vol.13, pp.409-418.

Veldon, H.E., Ten. and M.Van. Lingen [1990]: Geographical Information Systems and Visualization. In Geographical Information System for Urban and Regional Planning, edited by H. J. Scholten and J. C. H. Stillwell (Amsterdam: Kluwer Academic Publishers), pp. 229-237.

Vijlbrief, T. and P. van Oosterom [1992]: The GEO++ System: An extensible GIS, Proceeding of 5th International Sysmposium on Spatial Data Handling, IGU Commision on GIS. August3-7, Charleston, South Carolina, pp.40-50.

Vossen, G. [1991]: Data Models, Database Languages and Database Management Systems, Addison-Wesley.

Worboys, M.F. [1994a]: Unified Model for Spatial and Temporal Information. The Computer Journal, vol. 37, pp. 26-34.

Worboys, M.F. [1994b]: Object-oriented Approaches to Geo-referenced Information. International Journal of Geographical Information Systems, vol. 8, pp. 385-399.

Worboys, M.F., H.M. Hernshaw and D.J. Maguire [1990]: Object-oriented Data Modeling for Spatial Database. International Journal of Geographical Information Systems, vol. 4, pp. 369-383.

Yourdon, E. and L.L. Constantine [1979]: Structured Design: Fundamentals of a Displine of Computer Program and Systems Design. Englewood Cliffs, NJ: Prentice-Hall.

Zhou, W. [1995]: Large Object Support Using an Object-oriented Approach in Spatial Information Systems, Doctoral Dissertation, UCGE Report No.20084, The University of Calgary.

APPENDIX:

Main Member Attributes and Functions Defined in the Class Library

```
class Node
{
 attributes:
        long int ID;                          // object ID
        EleArray ConnectedArc;                // IDs of the connected arcs
        EntArray PointID;                     // containing point
        float Coord_x, Coord_y;               // coordinates
 functions:
        Node();                               // constructor
        ~Node();                              // destructor
        BOOL AveNode();                       // check existance of a node
        long int NewNode();                   // generate new object
        long int GetID;                       // get object ID
        EleArray GetConnectedArc;             // read all connected arcs' ID
        EleArray AddConnectedArc();           // add a connected arc
        EleArray DeleteConnectedArc();        // remove the disconnected arc ID
        void ReadCoord();                     // read coordinates
        void Remove;                          // Erase a node
        void write();
        void read();
};


class  Arc
{
attributes:
        long int ID;                          // object ID
        long int Node1ID, Node2ID;            // IDs of the two end nodes
        long int PolygonRID, PolygonLID;      //IDs of polygons left right to the arc
        EntArray PolylineID;                  // containing polyline
        float length;                         // length
functions:
        Arc();                                // constructor
        ~Arc();                               // destructor
        BOOL AveArc();                        // check availability of an arc
        long int NewArc();                    // generate new arc
        long int GetID;                       // get object ID
        float GetLength();                    // get length
```

```
        EntArray GetPolyLine;          // get containing Polyline
        void GetPolygon;               // get containing polygon
        void ChangNode();              // change ending nodes
        void ChangePolygon();          // change containing polygon
        void ChangePolyLine();         // change containing polyline
        void Remove;                   // delete arc
        void write();
        void read();
};


class  Polygon
{
 attributes:
        long int ID;                   // object ID
        EleArray BoundArc;             // IDs of closing arcs
        EntArray ArealID;              // ID of areal containing polygon
        float perimeter;               // perimeter
        float area;                    // area
functions:
        Polygon();                     // constructor
        ~Polygon();                    // destructor
        BOOL AvePolygon();             // check availability of an polygon
        long int NewPolygon();         // generate new polygon
        long int GetID;                // get object ID
        float GetPerimeter();          // get Perimeter
        float GetArea();               // get area
        EntArray GetAreal();           // get containing areal
        EleArray GetArcs();            // get closing arcs
        ElementArray GetNodes();       // get bound nodes
        void ChangArcs();              // change boundary
        void ChangeAreal();            // change containing real
        void Remove;                   // remove polygon
        void write();
        void read();
};


class Point: Node
{
 attributes:
```

```
        long int ID;
        long int NodeID;
        Entarry CompEnt;                    // containing complex entity
functions:
        Point();
        ~Point ();
        BOOL AvePoint();                    // check availability of the entity
        long int NewPoint ();               // generate new entity
        long int GetID;                     // get the point ID
        long int GetNodeID;                 // get the node ID
        long int ChangeNode;                // change the node
        EntArray GetCompEnt();              // get containing complex entity
        EntArray ChangeCompEnt();           // change containing complex entity
        void Remove;                        // remove the point
        void GetPosition();                 // get location
        void write();
        void read();
};


class Polyline
{
 attributes:
        long int ID;                        // Object ID
        EleArray ArcIDs;                    // IDs of arcs contained
        int NumArcs;                        // number of composing arcs
        Entarry CompEnt;                    // containing complex entity
        float length;
 functions:
        Polyline ();
        ~Polyline ();
        long int NewPoyline();              // Generate new entity
        long int GetID;
        BOOL AvePolyline();                 // check availability of the entity
        EleArray RmArc();                   // remove an arc
        EleArray AddArc ();                 // add an arc
        EntArray GetCompEnt();              // get containing complex entity
        EntArray ChangeCompEnt();           // change containing complex entity
        void Remove ;                       // remove the entity
        float GetLength();                  // get length
        EleArray GetArcs ();                // get composing arcs
```

```
        EleArray GetPoint;                  // get location
        void write();
        void read();
};


class  Areal
{
 attributes:
        long int ID;
        EntityArray  Polygons;
        EntityArray  Bound;
        float Perimeter;
        float Area;
functions:
        Areal ();
        ~Areal ();
        BOOL AveAreal();                    // check Availability of the entity
        long int NewAreal ();               // generate new entity
        long int GetID;
        EntityArray ChangeBound ();         // change boundary
        EntArray GetCompEnt();              // get containing complex entity
        EntArray ChangeCompEnt();           // change containing complex entity
        ElementArray GetPolygon ();         // get composing elements
        void Remove;                        // remove the complex entity
        float GetPerimeter();               // get perimeter
        float GetArea();                    // get occupation
        ElementArray GetBoundPoint();       // get boundary location
        ElementArray GetBound();            // get boundary arcs
        void write();
        void read();
};


class  ComplexEntity
{
 attributes:
        long int ID;                        // object ID
        EntArray EntityIDs;                 // IDs of entities contained
        float Perimeter;
        float Area;
```

```
functions:
        ComplexEntity ();
        ~ComplexEntity ();
        BOOL AveComEnt();                   // check Availability of the entity
        long int NewCompEnt ();             // generate new entity
        long int GetID;
        EntArray GetEntity();               // get included entities
        EntArray AddEntity ();              // add an entity in the complex type
        EntArray RmEntity ();                       // remove a member entity

        void Remove;                        // remove the complex entity
        float GetPerimeter();               // get perimeter
        float GetArea();                    // get occupation
        ElementArray GetBoundPoint;         // get boundary location
        ElementArray GetBound;              // get boundary arcs
        void write();
        void read();
};


class Index
{
 attributes:
        long int ID;
        int Offset;
        IndArray ObjectIDs;                         // Ids of indexed object
        char* IndType;                              // index fields
 functions:
        Index();
        ~Index();
        IndArray NewIndex();                        // new index
        long int GetObjectID ();                    // get object ID in array
        int GetOffset();                            // offset of ID in array
        void Remove;                                // remove an index
        IndArray AddIndex();                        // enter new member
        IndArray RmIndex();                         // remove a member
};
```