AN OBJECT-ORIENTED MODELLING FRAMEWORK FOR GEOGRAPHIC INFORMATION

M. Isdale & Y.C. Lee
Department of Surveying Engineering
University of New Brunswick
Fredericton, N.B. Canada

ISPRS COMMISSION III

ABSTRACT

One of the drawbacks of many Geographic Information Systems is in the concepts provided for representing features in the real world. These concepts tend to be far removed from those employed by users when they think about geographic features. As a result, these concepts lack the power to represent the higher level, semantic aspects of geographic information. This paper describes the design and implementation of a framework for the storage of geographic models of the real world. It can operate at a level closer to that of GIS users, and supports concepts based on those of Entity/Relationship and Object-Oriented modelling. It provides a number of data types suitable for spatial processing, and they can be extended by the user. The implementation of the framework was carried out using an object-oriented programming language called Eiffel.

KEY WORDS: Data Base, Object-Orientation, GIS/LIS, System Design.

1. INTRODUCTION

The abstract view of the world provided by many of today's Geographic Information Systems (GIS) is based heavily on the data structures used to represent information. For example, the concepts provided by the graph abstraction of space — points, lines and polygons — are often given a great deal of emphasis. Such emphasis on particular structures reflects an over-dependance on them, and leads to a lack of flexibility in using others. In addition, the identities of the things being represented are often hidden away among a great deal of interconnected data. The combination of a polygon and identifier, a feature code and a set of relational tuples is often the extent of the powers of a GIS when it comes to representing a plot of land, for example.

This paper deals with the design and implementation of a framework for the storage of geographic information. In this framework, an attempt is made to provide better representational powers by supplying structures which explicitly represent the *existance* of the things of interest — nothing more. These structures can then be linked to others which represent information *about* the things of interest — names and positions, for example, along with other quantitative and qualitative information.

By elevating the things of interest to a position of greater importance, and by supplying the structures to represent them, two things are acheived. First, the representation in the GIS should correspond more closely to the user's perception of the real world. This should make it easier for the user to interact with the representation and to get work done. Second, with the de-emphasis of the structures used to represent information about the things of interest, it is anticipated that a more flexible approach to using such structures will be possible. It should be noted that although the importance of a structure such as a line graph may well be de-emphasised from an overall point of view, this in no way detracts from the fact that it is a powerful structure for modelling and analysing networks, and for providing a basis for polygonal structures.

The framework which was developed can be described as being object-oriented, both in terms of its implementation and the way in which it models the real world. This paper first sets out to explain what object-orientation is, and how it can be applied. It then goes on to describe the ideas behind the design of the framework, and the experiences encountered in trying to implement it.

2. OBJECT-ORIENTATION

The term 'object-orientation' is one of the biggest computer buzzwords of our time. It is one that is often used without much understanding of what it implies, and has, for some people, become a hi-tech synonym for 'good'. Object-orientation is based on some fundamental concepts which can be used at a number of different levels. They are used in software development, modelling, and as the basis for user interfaces. This can lead to some conflict in what people perceive as being object-oriented. In this section, the concepts behind object-orientation will be outlined, and their application in different situations described. These concepts need to be covered before a complete explanation of the modelling framework can be undertaken.

2.1 Concepts of Object-Orientation

The basic idea behind all things object-oriented is that in one way or another, a system can be viewed as a collection of inter-related **objects**. An object has a **unique identity** (implying its existance), a **state** (defining its structural characteristics at any point in time), and a **behaviour** (defining its operational characteristics, or what it can do). The concept of **classification** dictates that an object must belong to a *class* of objects, all with the same specifications of state and behaviour. A class defines the nature of the state and behaviour, while an object records the identity and state of one particular *instance* of a class.

The objects dealt with are dictated by the kind of system being considered. When looking at a model of a real world system, objects such as roads or rivers may be dealt with. When concerned with the implementation of a software system, objects such as arrays and lists would be important — in addition to those representing things in the real world, were this the purpose of the software system. Object-oriented user interfaces use objects such as windows, icons & pointers to allow the user to control a computer environment. The common idea is that a system — be it the code for a computer programme, a model of reality or a computer environment — is viewed as a collection of inter-related, uniquely identifiable objects with a state and behaviour.

As well as the concepts of object and class, object-orientation also supplies the dual concepts of **generalisation** and **specialisation.** Together, these allow classes to be arranged in a hierarchy. A class which is a descendant of another in the hierarchy is said to be a more specialised *subclass*, while the more general class is the *superclass*. A subclass has the characteristics of the superclass automatically defined for it. It can also define more specialised characteristics of its own, however.

To make things happen in an object-oriented system, **messages** are passed to objects. This may be done by the user or by other objects. An object which has a message sent to it will then behave in some way, depending on what kind of system is being dealt with. Object-oriented systems are therefore dynamic in nature.

A GIS is a software system concerned with modelling the real world. There are three separate systems which can be identified. The *implementation system* is the programme itself — a collection of software components which define how the GIS works. The *environment system* is what the user sees when the GIS is running, and is defined by the user interface. The *model system* is the representation of the real world with which the user interacts — through the environment system. Any of these systems can be looked at through a pair of object-oriented glasses (o-o) — as a collection of objects.

2.2 Implementation System — O-O Programming

The implementation system (the 'programme') may be built using an **object-oriented programming language**, such as Smalltalk or Eiffel. Object-oriented programmers develop their software by defining classes. At run-time, the software system consists of a collection of interconnected objects, generated from the class definitions.

The state of a software object is the data it stores, and takes the form of a collection of **attributes**. These may consist of links to other objects, or direct internal representations (as with an integer object, for example). The behaviour of an object is the set of operations (often called **methods**) defined for it by its class. These methods are defined by routines written by the programmer, and are the only means of manipulating the data stored in the object. The operations which may be defined by a class for linked list objects, for example, may include append, insert and delete, for adding and removing items to and from a list. From this description, it can be seen why objects are said to **encapsulate** data storage capabilities along with the functionality to manipulate the data.

The execution of an object-oriented software system occurs through **message passing**. When a message is received by an object, it matches it with one of its methods or attributes. An attribute would then return a value, while a method would start executing. Methods can include 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, control is returned to the object that sent the message.

In addition to providing the concept of encapsulation, most object-oriented programming languages also support **information hiding**. This allows an **interface** to be specified for a class. An interface defines exactly what form messages sent to objects of that class should take. All access to the state and behaviour of an object must be made through its interface. As a result, the interface can control what attributes and methods are actually available to other objects. The interface is separate from the **implementation** of state and behaviour. It defines what messages an object will respond to, the form those messages must take, and what results will be yielded. It mentions nothing about how the results are obtained — that is left to the implementation. Information hiding, then, means that one object can in no way be dependant on how another acheives results.

Object-oriented programming languages incorporate a mechanism known as **inheritance** which facilitates specialisation and generalisation. One class can be made to *inherit* the state and behaviour of another, without having to define them itself. The subclass can then modify inherited attributes and methods, and add new ones of its own. Most inheritance mechanisms do not *support* specialisation and generalisation, because they do not enforce any constraints on the interfaces of classes. A subclass may inherit from a superclass, but it need not appear anything like it through its interface. The messages that it responds to may be entirely different, meaning that it can't be treated in the same way.

In some situations, it is important for the interface to be inherited rather than (or as well as) the implementation. Some programming languages address this problem, while others do not. The difference between inheritance and generalisation/specialisation leads to some conflict between the ideas behind object-oriented programming and object-oriented modelling.

2.3 Model System — O-O Modelling

A GIS allows a model of a real world system to be stored and analysed. The model represents the real world by storing information about it. The structure of the model system dictates how users interact with it to retrieve information or perform analyses.

A wide variety of schemes exists for structuring information about the real world. Structures based on line graphs are commonly used for representing positional information, while for textual information, table or record structures are common. Knowing the organisation of the tables and/or graphs, a user can retrieve the information required, and use that to perform analyses. Interacting with a model structured in this way is not particularly intuitive for a human — a translation must be made between the human conception of the real world system and the table/graph structured conception of the model system. The problem here, for the users, is that too much emphasis is given to the storage of information *about* things, and not enough to representing the things themselves.

To be more useful to users, model systems and the real world systems they represent should correspond more directly. Acheiving this has been referred to as "closing the semantic gap", and **object-oriented modelling** of the real world is one way of making this possible. Taking such an approach allows users to conceive of the model system in terms of objects, each with a defined state and behaviour. Some of these objects may correspond directly to things of interest in the real world, while others may be dedicated to storing certain kinds of information.

Object-oriented models allow a kind of information hiding. Although object structures for storing information (such as line graphs and tables) may still be used, they can be hidden from users to a certain degree. If users can interact with the model through the objects directly representing things in the real world, and if such interaction can be performed using real world terminology, then the underlying representations and processes can be largely concealed — and users can obtain results without knowing too much about the underlying structures.

An object-oriented model of a real world system need not necessarily be implemented using an object-oriented programming language — as long as the required concepts are visible to users. This overloading of the term 'object-oriented' tends to cause some confusion. Programmers often think of it as being 'their' term, and that other people should therefore refrain from using it. Its use in such a manner can be justified, however, if we think of object-orientation as supplying some basic concepts. We can then apply those concepts to different systems — programmes, models and environments.

2.4 Environment System — O-O User Interfaces

Computers create and present us with an environment within which to work. In a GIS, we find a model of a real world system located within this environment. The way we interact with the model system is dictated by the nature of the environment and the way users can interact with it.

Modern user interfaces employ object-oriented concepts. The "point & click" paradigm of today's graphical user interfaces is basically object-oriented in nature — corresponding directly to the object & message concept. Metaphorical objects (such as windows, icons, menus, buttons, pointers and dialogue boxes) are given a physical presence on a screen. Users can send messages to the objects by moving a pointing device and clicking buttons, and this is how users communicate with the computer.

Generalisation/Specialisation can be found among interface objects. There are many different kinds of window, for example, but many have common features. Most have a 'close' button, for example, and many have scroll-bars. A window class hierarchy can be formed with the most general kind at the top, and application specific ones at the bottom (a text editor window being one example).

2.5 A Final Remark on Object-Orientation

The view of object-orientation presented here is perhaps more general than those presented elsewhere. Many views are restricted only to the application of object-orientation to software construction. There can be little argument about the impact of the approach on this field. In fact, the concepts behind object-orientation largely grew out of developments in programming languages. The concepts have, however, found application in other fields, and this is why object-orientation has been presented here simply as a way of looking at systems.

3. FRAMEWORK DESIGN

The framework which has been developed is for building and storing models of the real world. Ideally, the user's view of a model system should correspond as closely as possible to the view of the real world system it represents. This should make manipulation and analysis of the model as intuitive as possible. The things that users perceive as being important in the real world, therefore, should be easily perceived in the model. This means that users should be able to view the model as being formed of components which directly correspond the things we are interested in.

In GIS, the interest is in storing information about entities (such as roads or rivers) — including their locations and spatial distributions. The relationships between these entities also tend to be of some importance, and therefore need to be modelled. Modelling the spatial variation of phenomena such as temperature or population is another important capability. A framework for storing geographic information should therefore allow models to be built from components which explicitly represent these items of interest.

The framework which has been developed may be described as the core of a fledgling database management system. Using database terminology, the framework is based on the *Entity/Relationship (ER) Data Model.* It allows entity types, relationship types and attribute value types to be defined. These types are then used to create actual **entities**, **relationships** and **attribute values**. These can in turn be put together to build a model. Note that the concept of classification is involved in the relationship between entities, for example, and entity types.

Each entity in a model represents the existence of something of interest, such as a `river` or a `forest`. Entities can be associated with one another by relationships. A `flows_into` relationship might associate a `river` entity with a `lake` entity. Both entities and relationships may have attributes which describe them in some way by linking to an attribute value. A `river` entity might have three attributes — `name`, `position` and `length`. An attribute must have a declared type which states what kind of value can be attached to it. The attribute `name`, for example, may have the type `string` declared for it.

The generalisation/specialisation of entities is added to the basic concepts supplied by the ER Data Model. This is acheived by allowing a hierarchy of entity types to be defined, rooted at the most general type, `entity`. A more specialised type may first be defined, called `water-body`. This might specify the attributes `name`, `position` and `permanance`. A `river` type may then be defined as a descendant of `water-body`, so it would have all the attributes of that type. In addition, however, it may have the attribute `length`. A `lake` type may also be defined as a descendant of `water-body`, and have the attribute `area` added.

A hierarchy of entities is useful in several respects. It can be useful in the selection of features for further investigation. Rather than having to say "select all rivers and lakes", for example, the more concise "select all water-bodies" would suffice. Relationships can be used more effectively too. A `river`, for example, may flow into a `lake` or another `river`. By making the `flows_into` relationship associate a `river` entity with any `water-body` entity (rather than being more specific and allowing it only to relate a `river` to a `lake`), the real world situation can be modelled more flexibly.

A similar hierarchy of relationships can be defined, rooted at the most general type, `relationship`.

In many database management systems, the capabilities for handling different types of attribute value are rather limited. A few basic types are allowed, such as integer, character, string and real. The support for handling more complex types is often restricted to the ability to group together two or more attribute values, each of basic types. For example, a real and a string taken together may represent a distance (a magnitude and a unit). This framework is more flexible, in that new types can be designed and 'plugged in', allowing different kinds of attribute value to be stored. A `distance` type may be designed, a value of which would include a magnitude and a unit. Functionality would also be incorporated, however. Some operations may be for setting the initial distance, converting the distance from one unit to another, or for adding two distances, regardless of whether they are have the same units or not. An addition operation would obviously have to include a conversion process if the units were different.

The ability to put storage capabilities and functionality together in a type like this is obviously the same idea as the concept of encapsulation which was mentioned earlier. In fact, attribute value types are not the only ones which can have operations specified for them. Entity types and relationship types can too. Some basic functionality is included in the definitions for the general types `entity` and `relationship` — the functionality to allow an entity to have an attribute value attached to it, for example, or for an entity to enter into a relationship with another. Further operations can be added for more specialised entities and relationships, but more will be said about this later.

The concept of generalisation/specialisation can also be applied to attribute values and this would appear to hold some potential for the integration of data from different sources.

Sticking with the hydrological example, a river may have an attribute called position. We know that the position of a river is linear in nature, but if we are dealing with data from a variety of sources, the actual representation of the position may take a variety of forms. Small rivers may have their positions defined by single lines of co-ordinates, while larger ones may have theirs defined by very long, thin polygons. Another possibility is that a river's position may have been determined by remote sensing techniques, and be defined by a long, thin block of pixels. This raises a problem when it comes to declaring the type of position in the river entity type definition

Each of the representations described above is fundamentally different, and yet because of the linearity of what they are representing, they have some common conceptual qualities. The concept of length, for example, is often important for linear features, as is the ability to identify segments along the length of a feature. A general attribute value type called linear could be defined as a descendant of the general type, attribute_value. It would specify a common state and set of behaviours for all linear positions. Any attribute value with linear as an ancestor would then have the state and behaviour enabling it to be treated as we would expect to be able to treat a linear position. Three useful descendants of linear could be defined — e.g. vector_line, polygonal_line and raster_line.

The implementation of functions to compute the lengths of these different representations of position would, of course, be very different. The functionality would be there, however, and would work for any descendant of linear. More will be said about this during the discussion of the implementation of the framework. For now, it is sufficient to note that if the type for position is declared as linear, then it is quite allowable for an attribute value of any of the more specialised types to be associated with that attribute.

Another important modelling concept which the framework supports is that of **aggregation** — the ability to handle **complex entities**. Some entities can be seen as collections of other entities. A road_system, for example can be seen as a collection of roads and junctions. Complex entities could be handled by using is_part_of relationships. The framework, however, supports the representation of complex entities directly, without requiring the use of relationships. In the same way that entities can have attributes, complex entities can have **components** as well. A component has a type declared for it which defines what kind of entity (or, indeed, complex entity) can be taken as a value. A complex entity may be composed of single components or groups of components. A land_parcel, for example, may consist of one plot, and zero or more buildings.

It should be clear from what has been said, that, at least in terms of the kind of model that can be built, the framework which has been developed can be described as object-oriented. The general types — entity, relationship and attribute_value — can be seen as classes which specify state and behaviour and which can be specialised. More specialised descendant types allow uniquely identifiable components to be generated (which can be seen as objects — instances of classes), and these allow useful models of the real world to be built. Some of these components correspond directly to items of interest in the real world (entities and relationships), while others are dedicated to storing information about them (attribute values). At the moment, no attempt has been made to incorporate components in the model which represent phenomena (such as elevation) in any way. It is hoped that future developments will address this deficiency.

It was mentioned that all the types — entity, relationship and attribute value — specify state and behaviour. Attribute values were described as having specialised operations defined for them (recall the distance type). It was also stated that entities could have specialised operations defined, as well as the most basic ones used for setting attributes and so on. As a demonstration of this, consider the following example :

> A road_system is a complex entity with road and junction entities for components. All three entity types have an attribute, position, specified for them. For the road_system type, the type for position is declared as network. For the road type, it is declared as linear, and for the junction type, it is nodal.
>
> In a particular model, a road_system entity is associated with a bunch of road and junction entities. The position attributess of the roads are of type vector_line, while those of the junctions are of type vector_point. These attribute value types are descendants of linear and nodal respectively.
>
> The positions of the roads and junctions are also tied together into an attribute value of a more complex type — a vector_network, which is a descendant of network. This attribute value is taken by the road_system as the value for its position attribute.
>
> Attribute values of type network (and its descendants) have the functionality to compute the shortest path between two nodals which are part of it. If an appropriate querying mechanism is in place on top of the framework, it should therefore be possible to establish what the shortest route between two junctions in the road_system is.
>
> By selecting two junctions, the vector_points which represent their positions can be found. These can then be used as input to the shortest_path function of the vector_network associated with the road_system. This function should return a list of vector_lines, and by comparing these to those defining the positions of the roads, it should be possible to determine the shortest route between the chosen junctions in terms of the roads that must be followed. This route may then be saved as an entity of type route.
>
> This method of finding the shortest route depends a lot on the knowledge of users, and it should be possible to specify a shortest_route operation in the road_system type. Such an operation would accept two junctions as its input, and return a list of road entities, or even a route entity, as its result. It would automatically perform the selections and comparisons outlined above, hiding the processes and the representations from the users.

Specifying the kind of information hiding behaviour that has been outlined for the road_system entity type is possible within this modelling framework. This allows models to be built which can be manipulated and analysed in an intuitive manner, using terms with which users are familiar.

This concludes the description of the design of the framework. The remainder of the paper will describe how it has been implemented.

4. FRAMEWORK IMPLEMENTATION

The implementation of the framework was carried out using the object-oriented programming language, Eiffel. To write programmes in Eiffel, classes are defined and compiled to produce executable systems. At run-time, objects are generated from these classes, and execution occurs as messages are passed among the objects.

The framework itself is not executable. Instead it consists of a library of classes which provide the capabilities to build, query and analyse models of the real world. Tools (or applications) which need to make use of these capabilities can be built on top of the framework. Tool classes can be written and compiled, along with the framework classes, to produce executable systems. When such a system is executed, tool objects provide a user interface. This allows users to access the framework objects and make use of its model handling capabilities.

The framework classes can be broken down into three groups. The first consists of those classes which define entity and relationship types. Two root classes — `entity` and `relationship` — define the state and behaviours which are common to all such types. The class `complex_entity` is defined as a specialised subclass of `entity`. It adds the state and behaviour for handling components.

Eiffel's inheritance mechanism is used to implement the generalisation/specialisation modelling concept. When a model is to be built, the user must first of all decide what entities and relationships need to be represented. New classes must then be defined which inherit from the appropriate root classes. These new classes add definitions of the attributes (and components) of the specialised types. They also define any specialised behaviours, such as the one for finding the shortest route through a `road_system`. Once these classes have been defined, an executable system can be compiled which will allow a model to be built.

With hindsight, it seems that this use of the inheritance mechanism is not the best way to implement the object-oriented *modelling* concept of generalisation/specialisation — at least for entities and relationships. Future developments should allow the definition of specialised entity and relationship types to be done at run-time — before, during and after the building of a model. Such definitions should not require recompilation, as is the case at present. A simple to understand, application language could be developed to allow users to write 'scripts' defining functions and attributes for entities and relationships.

The second group of framework classes implement attribute value types. Instances of them are attribute values, and store different kinds of information. A class is defined for each available attribute value type in the modelling framework. Some simple, vector based spatial attribute types have been designed and implemented, along with some simple textual attribute types. The ability to plug new attribute value types into a hierarchy as they are developed, and without affecting existing models is an attractive benefit of the framework. It comes about because of the de-emphasis of structures for storing information, and the increased emphasis on the things being represented.

The third group of classes is concerned with the management of a model. It allows the creation of the model, and its analysis through a querying mechanism which has access to the analytical functions contained within it. The querying mechanism is, at the moment, rather rudimentary, the emphasis to date having been on the building of models. A simple selection capability is in place, however, and allows access to the states and behaviours of model components.

Some very simple tools have been developed which make use of the capabilities supplied by the framework. The BUILD tool interprets a simple text files written in a simple language which enables models to be built. It allows entities to be created and associated with one another through relationships. Attribute values can also be set and attached to the entities and relationships. The INFO tool allows a model to be examined — textual attributes can be displayed as text, while positions can be displayed graphically. The SH_PATH tool is a special tool for displaying a road system, and for allowing the user to pick junctions and determine shortest paths between then. Both INFO and SH_PATH are rather specific in their applicability, and work needs to be done in developing more general purpose tools.

5. SUMMARY

A framework for storing geographic information has been designed and implemented at the University of New Brunswick. It employs object-oriented concepts both in its modelling capabilities and in its implementation. It allows users to build models which bear more resemblance to the real world, and is very flexible in its handling of structures for storing information.

The development of this framework has provided a great deal of insight into the potential of object-orientation in modelling and programming. Continued work should result in an improved and extended framework, as well as more useful tools making use of its capabilities.

ACKNOWLEDGEMENTS

This reserch was partially funded by an NSERC operating grant and two EMR research agreements.

SELECTED BIBLIOGRAPHY

Feuchtwanger, M., 1989. An Object-oriented Semantic Association Model for GIS. In: Proceedings of the 1989 National Conference on GIS, Ottawa, Canada, pp. 689-703.

Kemp, Z., 1990. An Object-Oriented Model for Spatial Data. In: Proceedings of the 4th International Symposium on Spatial Data Handling, Zurich, Switzerland, pp.659-668.

Khoshafian, S. and R. Abnous, 1990. Object Orientation : Concepts, Languages, Databases, User Interfaces. John Wiley & Sons, Inc., New York.

Meyer, B., 1988. Object-oriented Software Construction. Prentice Hall International (UK) Ltd., U.K.

Morehouse, S., 1990. The Role of Semantics in Geographic Data Modelling. In: Proceedings of the 4th International Symposium on Spatial Data Handling, Zurich, Switzerland, pp.689-698.