An Object-Oriented GIS

Mary Garvey[+], Mike Jackson[+], Martin Roberts[*]
University of Wolverhampton
[+]School of Computing & IT, Wolverhampton, West Midlands, WV1 1EL, UK
[*] School of Applied Sciences, Wolverhampton, West Midlands, WV1 1SB, UK
{M.Garvey, M.S.Jackson, M.G.Roberts}@wlv.ac.uk

Abstract

Object oriented databases have been portrayed as being the solution for complex applications such as Geographical Information Systems. Traditional database systems, such as relational, are not adequate for the rich types typically required of such systems. This paper discusses an application using object-oriented database technology, detailing the schema used, problems encountered and benefits perceived.

Keywords: object-oriented databases, GIS, MVC, GemStone, PJama

1. Introduction

This paper describes the construction of a geographical information system (GIS) using an object-oriented database management system. The motivation for this approach is that geographical data does not conform to the traditional record structures found in a conventional relational DBMS. Geographical datasets are not distinct (i.e. made up of unconnected strings and numbers) but rather are multi-dimensional and continuous. Kent [1987] comments that *"records are an excellent tool for processing information that fits a certain pattern"*, the features appearing on maps are not well represented in traditional record structures. The artificial discretisation, which is necessary to achieve a representation in record structures, does not adequately reflect the natural and visually evident structure of the information itself.

Object-oriented data models possess rich modelling structures, enabling a more intuitive representation that more closely parallels the natural structure seen in maps. For example figure 1.1 shows a simple map and some data that represents two segments of a single carriage primary A-Road:



```
1LINE    226                          5
  350460.000000  312880.000000      .000000
  350406.375000  313047.125000      .000000
  350393.937500  313185.937500      .000000
  350406.375000  313325.562500      .000000
  350480.000000  313680.000000      .000000
2LINE    226                          2
  350060.000000  314320.000000      .000000
  350480.000000  313680.000000      .000000
```

Figure 1.1 Sample Map and Data

In a relational database each segment would have to be treated as a number of rows, rather than one object. A segment number would be needed to link the relate records:

LINENO	SEGMENTNO	TYPENO	X	Y	Z
1	1	226	350460	312880	0
2	1	226	350406.375	313047.125	0
3	1	226	350393.9375	313185.9375	0
4	1	226	350406.375	313325.5625	0
5	1	226	350480	313680	0
6	2	226	350060	314320	0
7	2	226	350480	313680	0

Whereas in an object-oriented database, a user-defined type could be created to hold the point co-ordinates and each segment could be represented as one object. In this approach, the segment number is unnecessary and it provides a more natural way of representing the information, which can be easily manipulated as a whole rather than in bits:

LINENO	TYPENO	POINT
1	226	{ (350460, 312880,0) (350406.375, 313047.125,0) (350393.9375, 313185.9375,0) (350406.375, 313325.5625,0) (350480, 313680, 0) }
2	226	{ (350060, 314320, 0) (350480, 313680, 0) }

An additional claim made for object–oriented software systems is that they make the task of constructing software considerably easier. This is particularly relevant when manipulating complex data structures and for creating visualisations of the data. A successful GIS system will invariably embody this type of data structure and will almost certainly require a visual representation of map data. A further motivation for the use of an object-oriented approach to the production of such a system is therefore the expectation that the approach will result in a system which has a clean interface and is easier to maintain than an equivalent system built using conventional programming techniques.

The paper discusses briefly the essential objects in a GIS. This data model is then used as the application model within the Model-View-Controller (MVC) paradigm in order to create a visualisation of the data. The resulting representation has been made persistent in the GemStone database but has led to performance problems and has led to a move towards PJama.

2. GIS and Databases

2.1 Geographical Features

Geographical features are usually defined with respect to two main data categories: spatial and aspatial. A spatial database describes a collection of entities, some of which have a permanent location in some global, dimensioned space. Normally a system will contain a mixture of spatial and aspatial entity types. Spatial data types have the basic topographical properties of location, dimension and shape.

Aspatial data does not have a specific location, i.e., attributes have no intrinsic location. Their location is defined only by association with spatial data. For example, a road is a spatial entity, which has a street name, such as ‘High Lane’, this is the aspatial data.

For GIS, spatial data is a key feature and can be considered more important than aspatial data, since this differentiates GIS from vanilla information systems [Maguire and Dangermound 1991]. One aim of a good design, however, is to build an integrated database combining both the spatial and aspatial entities. In particular, the data model and data language should permit the use of spatial and aspatial relationships.

2.2 What Object-Orientation Offers a Database

An OODB must first of all be a DBMS and provide the features and functionality that we traditionally expect. OODBs behave like object-oriented languages, however the objects are persistent, that is, they remain after the program has terminated [Garvey and Jackson 1989]. Object-oriented languages offer richer data structures, which is important for applications with complex data such as GIS and an environment where systems can be developed quicker through reuse.

Most commercial OODBs have either a C++, Smalltalk or Java interface. A C++ interface has the drawback that once the data structures have been set-up, it is difficult to add new ones dynamically. This has important ramifications for aspatial data in a GIS. A user may wish to add new structures at a later point to hold aspatial data and this would involve recompiling the schema. Smalltalk on the other hand allows classes to be created dynamically. This has the disadvantage that the code is not compiled, so will run slower.

3. Problem outline

The problem area studied was the representation of typical information found on a map, for example, roads, rivers, urban areas, etc. The data source used was the Bartholomew (BART) datasets exported from Genamap in a portable format. The first platform for implementing the GIS was GemStone [Butterworth, Otis and Stein 1991], which is a commercial object-oriented database management system with a Smalltalk interface. The format of the data suggested a design for the initial object-oriented schema. Each dataset contains three types of data: Points, which are three dimensional points representing features such as spot heights, lighthouse, tourist information centres; Lines, which are a set of three dimensional points, each line representing a segment of the feature, such as a road or river and Areas, which are a set of three dimensional points, where the start and end points are the same, such as lakes or urban areas. An example of line data is seen in figure 1.1.

Simply adding the map data as it stands to the GIS is not sufficient. A point on the map may appear in a number of lines or areas. This occurs when more than one object can be found at a map reference. If the map consists of just collections of points, lines and areas then answering the query *"What objects are at point X"* will be a complex task, but highly relevant to contractors about to dig a road, who wants to know all the pipelines beneath them. To deal with this adequately each new point added to the GIS is added to a collection called *SetOfAllPoints,* which contains a point representing every place in the map where there is at least one feature. Curiously this is a case where the relational concept of a unique key is more help that the O-O concept of object identifiers [Garvey and Jackson 2000].

A user of a GIS will wish to investigate the relationships between different objects. In Smalltalk and GemStone, relationships are defined by creating an attribute (instance variable) which contains a reference to another object. A relationship of this type is, however, uni-directional, for example, a map item has an attribute tag which is constrained to be of type feature code. This allows a user to find the feature details of a given map item. A user looking at the feature code however could not find which map objects are related to the feature unless another relationship from the feature code to the map item is set-up. To

gain maximum flexibility, all classes in the schema have been defined with two-way relationships. This idea which appears as the inverse relationship in the ODMG standard [Cattell et. al. 1997] is not explicitly represented in GemStone.

4. GIS Schema

Given the data identified above, there are basically three strands to the schema: Point details, Map Features and Feature Code definitions. These are briefly described as follows:

4.1 Point

Point is fundamental to the whole schema. Smalltalk has a predefined class called Point, which has much of the required functionality already defined. Point however only applies to two-dimensional data, some datasets may require the capability to support three-dimensional data, so a new ThreeDPoint class was introduced as a subclass of Point, with an extra instance variable z.

The discussion in section 3.2 highlighted a need for every point on the map to be related to all the features that can be found at that point. For example, at a given point there could be a river, bridge, traffic lights, etc. An extra class GISPoint inherits from ThreeDPoint with an extra instance variable mapFeatures, which will hold the set of map features that are at that geographical location.

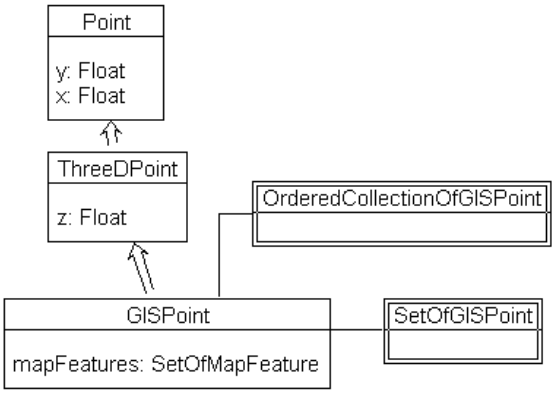


Figure 4.1 Point Classes

Figure 4.1 shows the structure, using ROME, a tool for object-oriented modelling [Barclay and Savage 1997]. The collection classes are explicitly modelled in this and the following diagrams because they exhibit differing behaviour from the classes they contain. The distinction between classes and collection classes is not often made in many modelling tools, but is important because each has its own separate properties and behaviour.

For line and area features there will be a number of points per feature, hence the collection class OrderedCollectionOfGISPoint is introduced. An ordered collection is used because the order of the points in any line or area segment is important.

4.2 Feature Codes

The class FeatureCode (figure 4.2) contains details of the features typically found on a map, for example, B-road single carriage, canal, urban area, etc. This class has three attributes: featureCode, description and mapFeature:

featureCode	Numeric code for the feature, for example, 12 represents a canal
description	Textual description of the code
mapFeature	Set containing all the points of that particular type of feature.

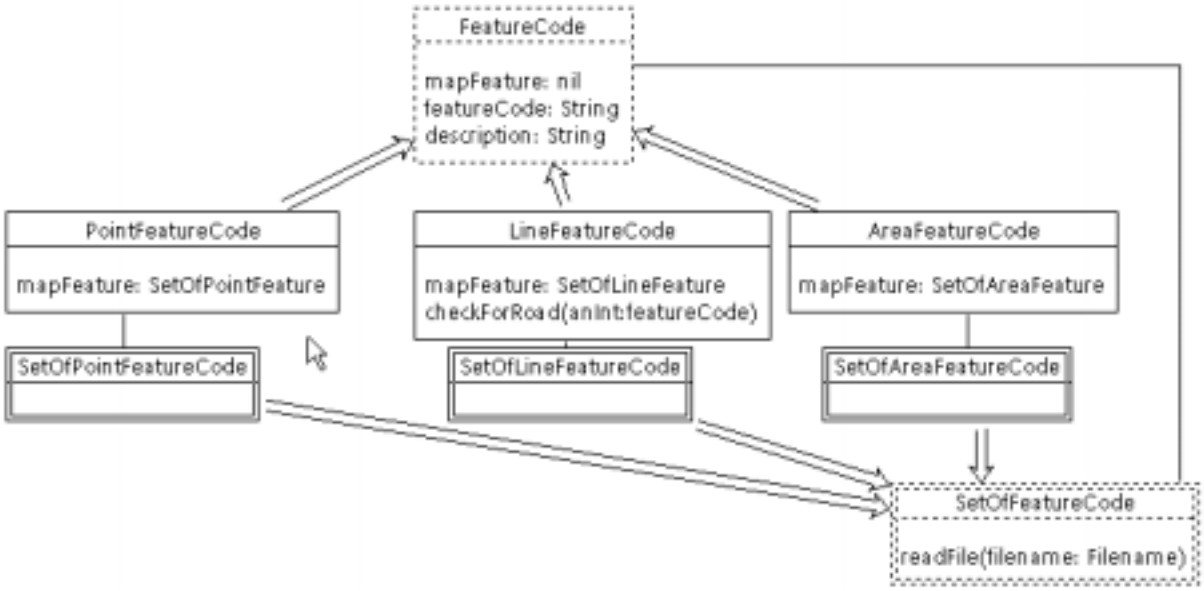


Figure 4.2 Feature Code Classes

4.3 Map Features

A class MapFeature has two attributes: tag and coOrdinates:

tag	Contains the relevant feature code, which will either be an object of type PointFeatureCode, LineFeatureCode or AreaFeatureCode.
coOrdinates	Holds either a GISPoint object in the case of points, or an OrderedCollectionOfGISPoint, for areas and lines.

4.4 Relationships between the Classes

The fundamental class of the schema is GISPoint. A data collection called SetOfAllPoints, an instance of SetOfGISPoint which contains GISPoints, holds all the known points. For each co-ordinate, the mapFeatures instance variable contains a SetOfMapFeature, which will contain all the features at a given point. This set can contain a mixture of different types of objects, but all are subclasses of MapFeature. The objects may appear to be duplicated in several places, but they are only stored once and their object identifier is used to refer to them in several sets.

4.5 Network Analysis

Additional classes were needed when implementing some of the functionality required by a typical GIS: an IdentityDictionary called Graph that is used with the shortest path algorithm to speed up processing. The key of the dictionary is a GISPoint converted to a symbol for fast lookups and the value is the set of points connected to it; Network also of type IdentityDictionary and NetworkTable which contains the three arrays needed [Lambert and Naps 1996] for Dijkstra's shortest path algorithm [Dijkstra 1976]. The Graph data object is a permanent object that is stored with the relevant map when the dataset is loaded in.

Network and Network data objects are temporary constructs used whilst a map is active. When loading the data if the current point is a type of road, then it is also added to the map's graph. Each map has its own graph.

4.6 Map

A Map class is used to tie the points, lines and areas together from a particular map set. Map has five instance variables: pointSet, containing SetOfPointFeature, lineSet containing SetOfLineFeature, areaSet containing SetOfAreaFeature, xyArray, which holds the minimum and maximum points on a map and graph, which holds the map's Graph. A GIS user will typically have more than one map set, so an IdentityDictionary collection class is used to hold all the different map sets.

4.7 Collection Classes

Once an object is defined, it is then added to the appropriate collection class in the map. For example, a LineFeature will be added to lineSet, which is of type SetOfLineFeature. The object is also added to the mapFeature instance variable of LineFeatureCode, which is of type SetOfLineFeature, to cross-reference the object. This allows a user to find all objects of a feature type, e.g., find all A-Roads, dual carriageway, as well as looking at a specific object. Before adding new objects, a test is made for duplicate values, if a duplicate value is found it is ignored.

The methods associated with reading this data is are contained within the appropriate collection classes. Much of the behaviour was put in the abstract collection classes: SetOfMapFeature and SetOfFeatureCode. Overriding means that where necessary, some of the behaviour can be modified to cater for specific class needs. One main difference is in the data type of the coOrdinates instance variable found in MapFeature's concrete classes. In PointFeature, this is of type GISPoint, whereas in LineFeature and AreaFeature it is an OrderedCollection and must be treated differently. This was one area of the GIS where significant software reuse occurred via the inheritance mechanism.

For each new map added to the database, a Map object is created and the data loaded into this object. This is then stored in a data object called MapDictionary, which contains all the maps known to the database.

4.8 Attribute Data

The GemStone system allows a user to build their own classes and add methods to them. This involves using the meta-class classes, the important ones being Class, Behavior, ClassDescription found under the Kernel-Classes class category. The meta-class classes are the classes used by the system for creating classes, methods, etc.

5. GemStone Smalltalk Interface

Servio's GemStone and the ParcPlace VisualWork's Smalltalk Interface has been used to implement the GIS. Within the GemStone Smalltalk Interface (GSI), there are two different worlds: Smalltalk and GemStone. The GSI incorporates two different name spaces, two different execution engines (for Smalltalk-80 and Smalltalk DB) and two different sets of tools. The GIS application could be developed in either or both environments. Smalltalk provided a rich set of classes that could be utilised so the front-end was developed in Smalltalk. GemStone was used to store the data and provide the standard database functions required.

5.1 Model-View-Controller Architecture

Interactive applications Smalltalk are built using an architecture that identifies a *model*, a *view* and a *controller*. This known as the Model-View-Controller (MVC) paradigm [Krasner and Pope 1988] and is used extensively throughout VisualWorks and is the key to the rapid development of graphical applications.

MVC separates a graphical application into two parts: abstract application (or model) and the interface. This means the application and interface can be developed separately. An interface can be used in several applications and an application can have different interfaces. For example, a map could have several interfaces, one that just shows lines and another that shows all features.

The interface is subdivided into two parts: the view, which handles all output (display) operations and the controller, which handles all user input from the mouse or keyboard.

The dependency mechanism is used throughout VisualWorks, this is a standard Smalltalk mechanism for transmitting changes between classes. Objects typically communicate with each other in object-oriented programming languages by sending messages. The dependency mechanism adds another form of communication between two objects and does not rely on one class sending a message to another class. It is a relationship between two objects, where one object is dependent on another. The relationship is one-way, though an object can both be dependent and have dependants. Each object can take part in as many dependency relationships as necessary.

6. Persistent Data

6.1 Data Storage using GemStone

Smalltalk on its own is sufficient for a single-user system, as the data could be stored in an image file. To realise the benefits of a DBMS the Smalltalk environment must be linked to GemStone. The option of using forwarders was chosen. Here the classes required for storing the data are kept solely in the GemStone database, whilst a dummy class is created in the Smalltalk environment, acting as a link to the real data. If any messages are sent to the dummy class, they are forwarded on to GemStone for processing. This saves the overhead of transferring the datasets from one environment to another. The non-dataset classes used in the view and controller were not affected.

Spatial data by itself is not sufficient for most GIS users, they also want the ability to add attribute, or aspatial data. For example, a Local Authority could want to store details of roadworks and link these to the actual roads on a map. Such data could already be stored in a relational database, such as Oracle. It is not envisaged that this system will provide a direct link to an external database, instead existing data could be exported in a standard format such as comma separated values (CSV) or XML and loaded into the GIS.

6.2 Performance

Initially the data was loaded into Smalltalk and then committed to GemStone. With a small set of data (325K), this took less than a minute to load and commit. Problems occurred when larger datasets were used (5.6Meg) [Garvey and Jackson 2000]. GemStone proved an ideal environment to develop the system quickly, but has not proved to be scalable with the given schema. Once major advantage of this system is the ability to dynamically create attribute classes and methods, which will be difficult to emulate in other compiled languages.

Because of the performance problems however it was decided to port the schema to Java, using PJama as the persistent store [Jordan 1999].

6.3 PJama

PJama is a platform that provides orthogonal persistence for Java [Arnold and Gosling 1996]. Classes and methods can be developed in Java and with a minimum of additional code, the persistence normally associated with a DBMS is provided. Porting the schema developed in GemStone has proved easy and Java™ also supports the MVC architecture [Horstmann and Cornell 1999] leading to a rapid development of the Windows interface. The algorithm utilised in GemStone have also ported readily.

Below is a cut-down version of the class that contains the dictionary of all maps (the equivalent of the IdentityDictionary). The two lines in bold text is the only additional code needed to make standard Java code persistent:

```
import java.util.*;
import org.opj.*;

public class MapDictionary {
  static { OPRuntime.roots.add(MapDictionary.class); }
  private static Hashtable mapList = new Hashtable();
  private static boolean firstRun = true;

public Object getPRoot(String name) {
      return mapList.get(name);
 }
public void newPRoot(String name, Object root) {
      mapList.put(name, root);
  }
/* other methods required…. */
}
```

The classes must be compiled and run using the Orthogonal Persistence for Java™ (OPJ) platform rather than the normal JDK environment. When storing data in the persistent object, the following method must be called to commit the database, otherwise no additional code is necessary:

```
OPRuntime.checkpoint();
```

The code below then shows the class in action, if data is just being retrieved and not updated, then no extra code is necessary:

```
public class ShowShrew
{public static void main(String[] args) {
      MapDictionary md = new MapDictionary();
      MapData map;

      map = (MapData) md.getPRoot("Shrewsbury");
     if (map != null) {
           JFrame frame = new MapFrame(map);
           frame.show();
         }
}
```

In the example above, MapDictionary represents the Model and MapFrame is the View with respect to the MVC paradigm. PJama achieves persistence by reachability, that is, when an object is stored in a persistent store, every item that is necessary to interpret it correctly is also stored in that store [Sun 1998]. This had the unexpected effect that the view was also made persistent, even though the model did not reference the view. This initially meant that the window displaying the map was saved after each program run, so that after three runs there were three map windows. Care has to be taken to dispose of the window religiously when it is no longer needed, plus the *–nowindowresume* parameter must be used when running the program.

8. Conclusions

It has been claimed that object oriented database products are suitable for the development of systems similar to the GIS developed at Wolverhampton. This research supports that claim to an extent. The effort required implement this system has been considerably less the effort that would be anticipated if a relational or object-relational database system had been used. The main drawback has been the speed when running the system with large datasets, which led to the move to PJama. Initial performance figures with the equivalent datasets have proved promising.

Standard algorithms have been used for typical GIS functionality, which have fitted in with the schema, no amendments have been necessary to the data structures. The only exception so far was adding the network analysis. The methods that read the data in had to be amended to test for road segments, which were added to the map's network graph. This network graph was added for convenience, rather than necessity and acts like an index to the road objects. The algorithms as implemented do not depend on the existence of these indexes, but work faster with them.

One peculiar feature of the implementation method chosen which would not be available either in relational systems or in most other object-oriented systems is the ability to dynamically define new data types. This has been shown to be central to the integration of aspatial data with the representation of the spatial as described in this paper. Achieving a similar feature in PJama could prove harder than the GemStone equivalent.

9. References

ARNOLD K. and GOSLING J. (1996), *The Java Programming Languag*e. Addison-Wesley.

BARCLAY K. and SAVAGE J. (1997), *Object-Oriented Design with C++*, Prentice Hall Europe.

CATTELL R.G.G., BARRY D., BARTELS D., BERLER M., EASTMAN J., GAMERMAN S., JORDAN D., SPRINGER A., STRICKLAND H., WADE D. (1997) *The Object Database Standard: OMDG 2.0.* CA:Morgan Kaufmann.

DIJKSTRA E.W. (1976), *A Discipline of Programming*, Englewood Cliffs, NJ: Prentice-Hall.

GARVEY M.A. and JACKSON M.S. (1989), Introduction to Object-Oriented Databases. *Information and Software Technology*, 31(10), Dec. 1989, pp. 521-528.

GARVEY M.A. and JACKSON M.S. (2000), A GemStone GIS, in *Proceedings of 2nd International Conference on Enterprise Information Systems*, 4-7 July 2000, Stafford, UK.

GEMSTONE (1996) *GemBuilder for VisualWorks*, GemStone Version 5.0, GemStone Systems Inc.

HORSTMANN C.S. and CORNELL G. (1999), C*ore Java™2, Volume 1 – Fundamentals*, Sun Microsystems Press, Prentice-Hall.

KENT W. (1979), Limitations of Record-Based Information Systems, *ACM TODS* 4(1), pp. 85-97.

KRASNER G. and Pope S.T. (1988), A Cookbook for using the model-view-controller user interface in Smalltalk-80, in *Journal of Object-Oriented Programming*, 1(3), Aug-Sept. 1988, pp. 26-49.

LAMBERT K.A. and NAPS T.L. (1996), *Understanding Program Design and Data Structures with C++*, MN: West Publishing Company.

MAGUIRE D.J. and DANGERMOND J. (1991) The Functionality of GIS, *in* (Eds.) MAGUIRE D.J., GOODCHILD M.F. and RHIND D.W., *Geographical Information Systems. Volume 1: Principles*. Longman, Scientific and Technical, pp. 319-335.

SUN MicroSystems (1998) PJama Tutorial: Persistence by Reachability, Glasgow: <http://www.dcs.gla.ac.uk/pjava/tutorial/opj.tutorial.reachability.html> (updated 17 April 1998, accessed 15 May 2000)