A Persistent View of Encapsulation

G.N.C. Kirby & R. Morrison

School of Mathematical and Computational Sciences,
University of St Andrews,
St Andrews, Fife KY16 9SS,
Scotland
E-mail: {graham, ron}@dcs.st-and.ac.uk

Abstract. Orthogonal persistence ensures that information will exist for as long as it is useful, for which it must have the ability to evolve with the growing needs of the application systems that use it. The need for evolution has been well recognised in the traditional (data processing) database community and the cost of failing to evolve can be gauged by the resources being invested in interfacing with legacy systems.

Zdonik has identified new classes of application, such as scientific, financial and hypermedia, that require new approaches to evolution. These applications are characterised by their need to store large amounts of data whose structure must evolve as it is discovered. Here, we discuss one particular problem of evolution in these new classes of application in relation to Object-Oriented Database Systems (OODBS): that of the tension between the encapsulation of data within objects and the need for the data be mapped dynamically to an evolving schema. We outline a solution taken from our persistent programming experience and show how it may be used in the O_2 OODBS.

1 Introduction

The growing requirements of database application systems challenge database architects to provide the appropriate mechanisms for system evolution. Database systems are designed under a number of *a priori* assumptions about how they will be used that fundamentally affect their ability to evolve. The particular example that will be addressed in this paper is the assumption that data is encapsulated within objects in an OODBS.

Zdonik [Zdonik, 1993] has identified new classes of application system that require new approaches to evolution. These include scientific applications, data mining, financial applications, multimedia applications, graphics and video applications, text applications and heterogeneous databases. The applications are characterised by their need to store large amounts of data whose structure must evolve as it is discovered. The changes may be additive, subtractive or descriptive [Connor et al., 1994], but all require that the data be mapped in complex ways, and dynamically, to an evolving schema. The nature of these applications may be illustrated by the example of a scientific application, again taken from [Zdonik, 1993].

Consider a satellite that is sending weather maps to a monitoring station. Image enhancement techniques may be applied to the data as it arrives, but it is stored initially as raw bitmaps without any higher level information concerning its contents. Further additional structure may be discovered by an application program that is able to perform feature extraction on the images and can identify and categorise various kinds of features, such as storms, weather fronts etc. While a weather map is represented as an object, each feature might also best be considered as an object, embedded inside the weather map. The fact that a particular weather map contains one or more storms would be stored as a part of that weather map object, which might cause it to become reclassified as an inclement weather map. Each storm might also contain some substructure. For example, if a storm is a hurricane, the storm object might have an eye, a size, and a location.

It should be noted that there is no requirement that the objects that are discovered in the weather map be disjoint. For example, weather fronts could intersect several of the storms. Furthermore, the embedded objects need not necessarily form a hierarchy.

The example illustrates many of the problems incurred by applications, which in turn pose the following new challenges for database architects:

1 **Run-time schema change** - As the new structure is discovered, for example in the discovery of hurricanes and weather fronts in a weather map, it must be possible to incorporate that new structure into the schema while the program is running.

2 **Complex mapping to the schema** - A mechanism is required that will keep track of the complex relationships amongst the data, meta-data and programs, and indeed the intended semantics defined by the users.

3 **Object type migration** - A mechanism is required for descriptive evolution, as in the classification of a weather map where it is also an inclement weather map, even where the new form of the data differs markedly from the old. Existing applications may operate with new data if the system supports a type system that allows new structure to be related to existing structure, such as subtyping [Cardelli, 1984] or mechanisms geared specifically to evolution [Zdonik, 1990, Connor et al., 1995]. A more powerful technique is required where such a mechanism is not supported or where the evolution does not follow the predicted path provided by the type system.

4 **Embedded and overlapping objects** - Encapsulation requires that objects hide their internal structure. Some objects, like a hurricane in a weather map, may, however, be part of others. To model this accurately requires a mechanism that will allow an aggregation superstructure to be placed on the object without encapsulation, while retaining the property of information hiding where necessary. Similarly two objects may overlap, such as two weather fronts in a weather map. In both cases, unlike in traditional object models, multiple objects share state and therefore pose problems of consistent update.

5 **Co-ordinate systems** - Applications may require more than one co-ordinate system to be active at one time. For example, a weather map may have its own co-ordinate system which is active at the same time as the co-ordinate system re-

quired by a hurricane object.

6 **Non-contiguous object specification** - Conceptually objects in a database system are regarded as if they occupy contiguous storage. Where the data contained in a new object is non-contiguous, a mechanism is required to group the parts into an apparently contiguous object with separate identity. For example a low-resolution view of a weather feature might be formed by composing alternate scan lines from the underlying bitmap.

7 **Breaking abstraction** - Hiding data in objects as they are formed runs the risk of improperly imposing structure on data. As we have seen, data may require several forms thus it must always be possible to access the original form, subject to appropriate authorisation.

Many researchers have investigated the support of evolution in object-oriented database systems [Andany et al., 1991, Coen-Porsini et al., 1991, Bratsberg, 1993, Ferrandina et al., 1995, Odberg, 1995, Lerner, 1996, Roddick et al., 1996]. They address such problems as adding and deleting class definitions in an existing populated database, and modifying definitions by adding, removing, altering and moving class attributes. Method definitions and any stored queries may have to be altered to accommodate the changes. Once a schema change has been made it is then necessary to transform the existing objects in the database to ensure consistency, either immediately or lazily at the time that they are next accessed. We have made our own contributions to points 1-3 above using hyper-programming and linguistic reflection in [Connor, et al., 1994, Kirby et al., 1996, Kirby et al., 1997].

The complex structure of the new applications identified above may be modelled using objects within an object database system. However, object modelling leaves some inherent problems unresolved. In this paper we address points 4-7 by identifying encapsulation as the offending concept. We develop an alternative to encapsulation which preserves information hiding, taken from our persistent programming experience [Connor et al., 1990], and show how the technique may be incorporated into systems using the O_2 OODBS.

Thus this paper addresses only a very restricted case of the general problem of evolution in object-based systems, that of monotonically increasing schema evolution. This is the case in which all changes to the schema simply add new definitions without invalidating the existing schema—the idea being to incrementally form new views over the same underlying data. It is thus not necessary to alter the existing data and programs. The problem of interest is how to support this kind of evolution efficiently.

2 Persistent Encapsulation Considered Harmful

The object model fulfils two functions: aggregation and information hiding (protection). Data within an object is aggregated into a unit, the object, which can be manipulated as a single entity. The data itself is hidden so that only the object, and not the data itself, can be manipulated directly. In most object-oriented languages and database systems, information hiding is synonymous with encapsulation, which means that data may be trapped in a single object.

The effect of encapsulation is that the finest grain of data sharing among objects in a pure object-oriented system is the object. There may exist multiple references to an object from outside the object, but not to individual data components within it. This means that if fine-grain sharing is required (for embedded or overlapping objects), either the objects must be made sufficiently small, or all the data to be shared and the data structures which share it must be encapsulated within the same large object. These alternatives are illustrated in Fig. 1.

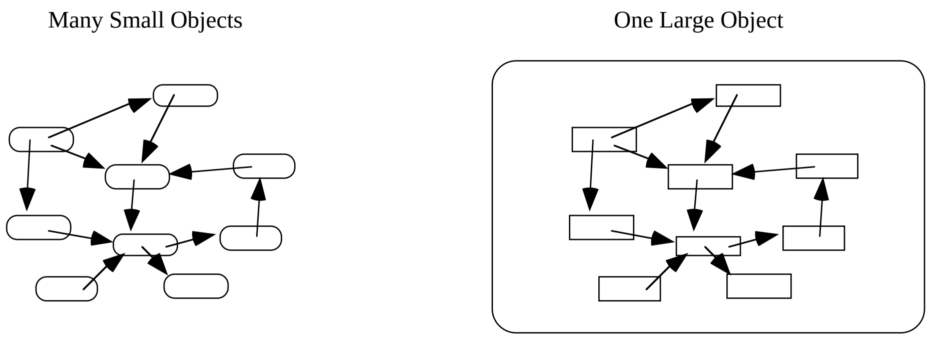


Fig. 1. Fine-grained Sharing with a Pure Object Model.

The problem with the first alternative is that the creation, storage and access overheads may be significant. Consider for example the extreme case in which sharing within a bitmap is required at the pixel level. This would involve the creation of a separate object for each pixel, the state of which would be accessed by method call. Such difficulties are avoided by encapsulating all the relevant data within the same object, but the advantages of using objects as a structuring mechanism are then lost. It is thus difficult to support fine-grained sharing satisfactorily with a pure object model, although compilation techniques such as method in-lining may assist in increasing efficiency of data access.

A second related problem is a lack of flexibility: the degree to which sharing is possible is determined at the time the encapsulating object is formed. If it is later desired to share a component within the object, at best this involves extracting the data and creating a new object which can be shared—and this is only possible if the necessary access methods are provided in the original object.

The application of persistence to the object-oriented model highlights the fact that once some data is encapsulated within an object, it remains so encapsulated forever despite the unknown future modelling requirements of the data. This problem is not well identified in non-persistent object-oriented languages since all objects are transient and disappear at the end of a program execution.

The strongest argument for encapsulation, that of security, is also exposed as a naive static description of access protection. Protection by access path, espoused by capability systems [Dennis and Van Horn, 1966], is not available by this technique, since all the protection is associated with the object. Thus the shortcomings of encapsulation can be summarised as:

- binding data to objects too early which causes modelling decisions to be made prematurely especially where sharing may be involved, and
- a static view of security.

Some object-oriented systems relax the strict object-oriented model so that data can be aggregated into an object without encapsulating it. This is achieved by allowing some object fields or slots to be *public*, meaning that those fields can be accessed directly from outside the object, while other fields remain encapsulated. This gives the same modelling capability as would be obtained by adding a *record* construct to the model. Sharing must, however, still be identified at object creation time.

The technique proposed in this paper avoids using encapsulation as an information hiding mechanism, and thus is applicable to such systems, although it will first be described in the context of the persistent language Napier88 [Morrison et al., 1994].

2.1 Information Hiding Without Encapsulation

The technique used to implement information hiding without encapsulation is to regard all data constructors as views over the data. Initially data is placed in the persistent store in the manner in which it is collected or generated, which will then be regarded as its most primitive form. Applications are built in terms of multiple views of the primitive data and may involve many layers of views.

Views may be open, and defined implicitly by the data constructor, or abstract and defined explicitly by a set of functional interfaces. Thus a view forms a functional dependency over an existing set of values. Every view in the system is defined by a signature, consisting of a set of operations available over the constituent values and an implementation consisting of the operations. In the abstract case the definition of the signatures and the implementations are divorced allowing different implementations to be used at different times. Signatures are, however, fixed for the lifetime of a view. Co-ordinates, vectors and records, for example, may be used to provide open views and existentially quantified types to provide abstract views.

Using encapsulation in object-oriented database systems, the raw data may only be viewed through one interface and the information is essentially trapped in the object once instantiated. In this technique, the data is placed in an object (view) dynamically when the data modelling requires it. The viewed value and the viewing value are both available to other views of the data and there is no sense in which the viewed value becomes unavailable, or encapsulated, in the viewing object. Of course, it is not always desirable to expose all views of data, and techniques are required to limit the visibility of certain data. However this is achieved by access protection not encapsulation.

2.2 An Illustrated Example

Consider again the meteorological database. Initially the data is held in an unstructured form, just a collection of photographic images captured by various satellites. This data is then used by a weather forecasting application to build higher level models of weather patterns, and ultimately to produce forecasts. This may involve running a feature extractor application over the photographic data. The purpose of this application is to use image analysis to determine the locations of static features, such as the outlines of countries, and dynamic features such as fronts, storms and hurricanes. Thus as the program is running it discovers new structure in the original data and forms new views over it; thereafter the data can be accessed both in its original form and via the new views. For example, a feature extraction application may locate the countries and weather fronts shown in Fig. 2:

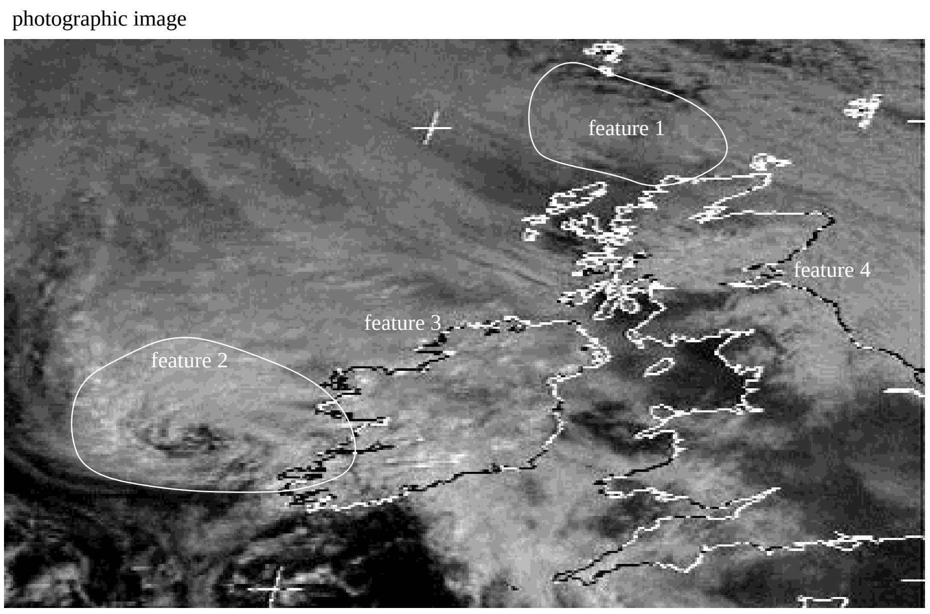


Fig. 2. Features Located in a Weather Map.

Fig. 3 shows a simple database schema in Napier88, which contains a set of type definitions, three persistent variables, and a procedure which operates over them. The variables refer to sets of objects of the corresponding types.

```
! Storage type
type RawData is structure (chunks : Set [image])

! View types
type PixMap is … ! arbitrarily shaped region of pixels;
                 ! storage structure not specified here
type Region is … ! representation of region in real-world coordinates
type Feature is structure (featureType : string;
                           featurePixels : PixMap;
                           featureRegion : Region)
```

```
type Country is structure (countryName : string; countryPix : PixMap)

! Persistent data and procedures
RAWDATA :   RawData
FEATURES :  Set [Feature]
COUNTRIES : Set [Country]
extractFeatures : proc (RawData)
```

Fig. 3. Weather Map Database Schema in Napier88.

As each raw photographic image arrives it is stored initially as a chunk in the set associated with the persistent variable *RAWDATA*. Later, when the feature extraction application is run against the *RAWDATA* images it creates additional views comprising *Feature* and *Country* structure objects which contain references to the raw data. These objects are entered in the *FEATURES* and *COUNTRIES* persistent sets; subsequently users can access the data via the views provided by any of the three persistent variables. A simplified diagram of this structure is shown in Fig. 4:

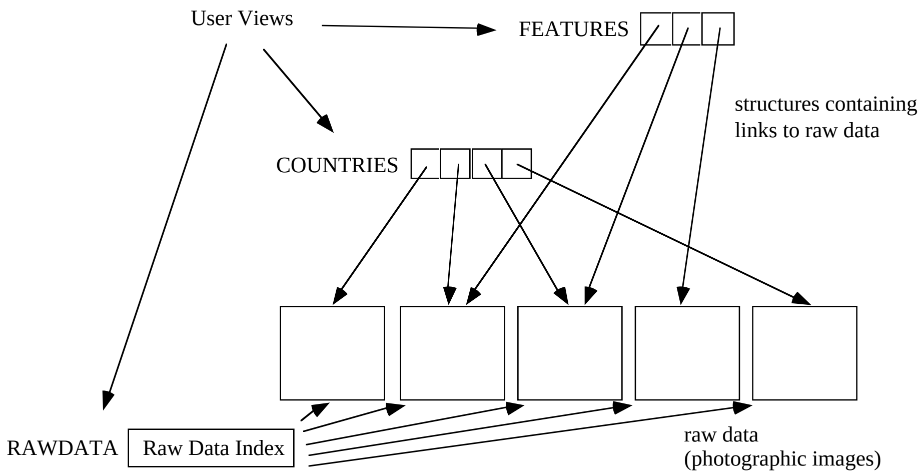


Fig. 4. Structure Views with General Access.

Here the raw data is partitioned into chunks and accessed by users through views provided by structures which contain links to the raw data. The perceived ordering of the raw data may vary depending on the view. There is nothing to stop users accessing the raw data directly if they wish, via the variable *RAWDATA* which contains links to all the raw data chunks.

Notice that these viewing mechanisms can support embedded and overlapping objects. For example, the pixel regions associated with various *Feature* and *Country* instances may overlap or be contained within one another. Other embedded and overlapping objects may be formed by the same object being in more than one view.

Although not demonstrated here, views may be formed over other views to any depth and any mixing of levels.

2.3 Data Protection

Access to the raw data may be restricted to the administrator by imposing password protection on the access to the raw data, as shown in Fig. 5. Users may now access only those parts of the raw data allowed by their views; since user address arithmetic is forbidden there is no way to access one chunk directly from another. The administrator may gain access to the index and thus the raw data by presenting the correct password to the checking procedure which contains a link to the raw data index hidden within its closure.

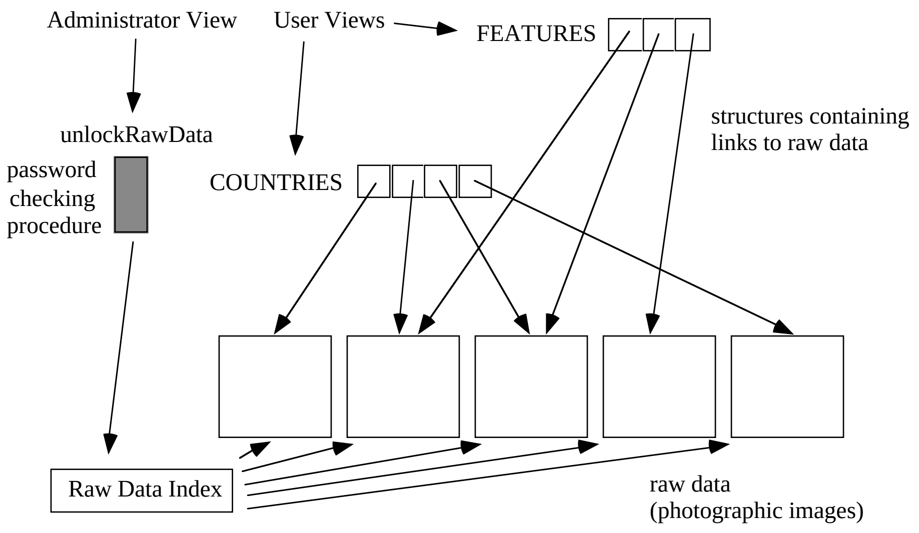

Administrator View
User Views
FEATURES
structures containing links to raw data
unlockRawData
password checking procedure
COUNTRIES
Raw Data Index
raw data (photographic images)

Fig. 5. Structure Views with Restricted Access.

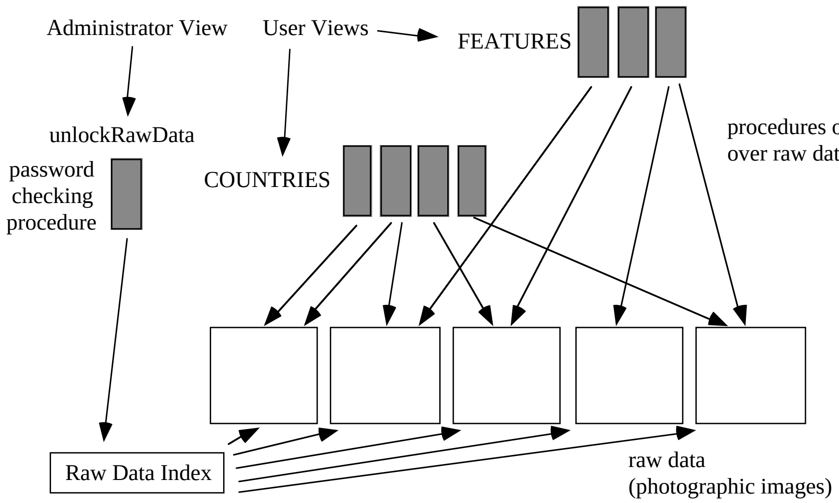
Fig. 6 shows how such protection can be coded in Napier88. The procedure *passwordCheckGen* takes as parameters the password string and the data to be protected as an instance of the infinite union type *any*. It returns a procedure which itself returns either the protected value or a null value depending on whether the password presented is the correct one. The procedure *unlockRawData* is created in this way and made persistent. Direct access to the raw data is then removed by dropping the persistent variable *RAWDATA*. This simply removes the access path to the raw data rather than deleting the data itself.

```
let passwordCheckGen = proc (key : string ; data : any ->
                             proc (string -> any))
   proc (try : string -> any); if try = key then data else any (nil)
```

```
in root let unlockRawData = passwordCheckGen ("secret", any(RAWDATA))
drop RAWDATA from root
```

Fig. 6. Password Protected Access to Raw Data in Napier88.

Procedural information hiding can be extended further to provide user views that have no access to the raw data itself. Instead, a user view contains procedures which contain hidden links to the raw data; the user is restricted to the functionality provided by the procedures, as shown in Fig. 7:



Administrator View
User Views
FEATURES
unlockRawData
password checking procedure
COUNTRIES
procedures operating over raw data
Raw Data Index
raw data (photographic images)

Fig. 7. Raw Data Completely Hidden.

Fig. 8 shows how this mechanism, called first order information hiding, can be coded. The elements of the set *FEATURES* now have the type *Feature2*. This structure contains only the type of the feature and a procedure *iterate* which can be used to apply any given procedure to each pixel of the underlying raw data in turn. A similar adaptation can be applied to the set *COUNTRIES*.

```
type Feature2 is structure (featureType : string;
                            iterate : proc (proc (pixel))
```

Fig. 8. First Order Information Hiding in Napier88.

The previous mechanism prevents users from accessing the raw data at all. Another possibility is to use abstract data types to provide user views, allowing users to access parts of the raw data directly but with limited type information. This restricts the operations a user program may perform on the raw data, while retaining the ability to pass references to the raw data to interface procedures [Connor, et al., 1990]. For example, the database may implement a feature object as a structure and provide a procedure in a user view which creates a new feature object. That procedure returns to the user a reference to the structure implementing the object, with a restricted type. The user cannot discover the contents of the structure, or even that it is a structure. All they can do with the reference is to pass it to other interface procedures which operate on weather features.

The technique uses the language type system to give users access to ‘handles’ which refer to the underlying data but have strictly limited functionality. Although achieved in a different way, the modelling ability is as powerful as, and similar to, that of capability systems [Morrison et al., 1990]. Both apply protection to the access paths to the data, rather than to the data itself, in order to retain flexibility, and both ultimately rely on some form of user authentication.

2.4 Co-ordinate Systems

Independent co-ordinate systems are a special case of overlapping objects that require separate calculation on each view. The Napier88 language gives an example of how this may be accommodated. It supports images as a basic co-ordinate system of pixels in an infinite two dimensional integer space. If one of the photographic images is modelled by a Napier88 image type, then views over part of the image may be formed by the *limit* operation. For example the following defines two view images which are embedded in the main image and which overlap with each other:

```
let firstView = limit photoImage1 to 150 by 100 at 50, 50
let secondView = limit photoImage1 to 200, 160 at 150, 80
```

All three images have their own co-ordinate system. The image *firstView*, for example, is a view over *photoImage1* that starts at the pixel (50, 50) and has the size 150 x 100. The situation is illustrated in Fig. 9.

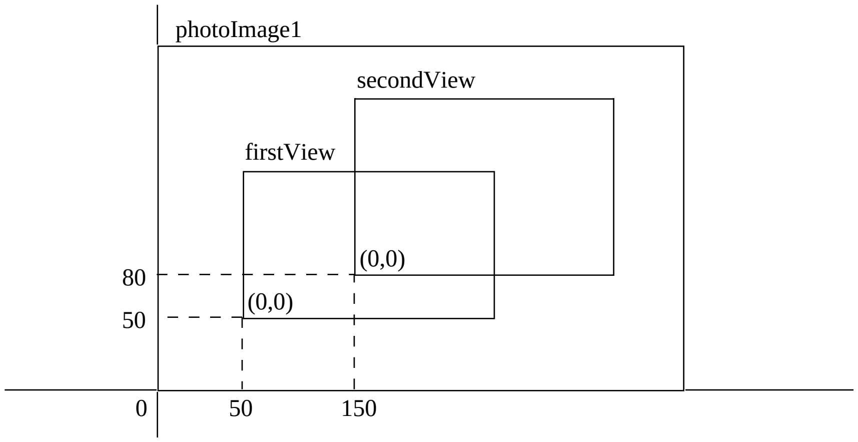


Fig. 9. Co-ordinate Systems in Overlapping Napier88 Images.

Operations on a view are expressed in terms of the local co-ordinate system and translation to the correct origin is performed automatically in that context. For example a raster update on the pixel (110, 100) of *firstView* is reflected in changes to pixel (160, 150) of *photoImage1* and pixel (10, 70) of *secondView*, since all of these are actually the same pixel.

In the implementation of Napier88 the pixels of the base image are stored contiguously. The result is that a view image is a non-contiguous object since it comprises only part of each scan-line of the base image. Napier88 thus supports a restricted case of the general need for embedded, overlapping and non-contiguous objects, in that all views are rectangular and contain every scan-line within the view region. The language model and implementation could be extended, however, with new more general view forming constructs.

2.5 View Update

With the layered views technique, views may be formed over other views to any depth and any mixing of levels. Thus, there is an obligation to maintain consistency in the way that updates are reflected through the viewing interfaces. Just as experienced in relational views [Dayal and Bernstein, 1978], all updates may not be efficient or even legal and some care has to be taken in allowing them.

Abstract views are the easiest to deal with since the abstract interface can be manipulated to reflect the correct update semantics. In the weather map example, storm information may be stored initially as a co-ordinate system within the weather map. The storm view may be written in terms of functions which operate over the weather map, consisting of the translation rules for mapping from the weather map to the storm and in reverse. The functions guarantee the consistency of the use of the views under some concurrency control for update.

Open views, created by non-abstract constructors, are not so easily dealt with. *Ad hoc* solutions can be devised for particular cases. The problem, however, is no greater than that found in relational databases.

3 Information Hiding in O_2

This style of constructing layered views is not restricted to Napier88. For example, Fig. 10 shows a schema for the initial non-protected views in O_2C.

```
/* Storage type */
class RawData public
     type tuple (chunks : set (Bitmap))
end;

/* View type */
type PixMap : …  /* arbitrarily shaped region of pixels */
type Region : …  /* representation of region in real-world coords */
```

```
class Feature public
     type tuple (featureType : string, featurePixels : PixMap,
                 featureRegion : Region )
end;

class Country public
     type tuple (countryName : string, countryPix : PixMap)
end;

/* Persistent names and functions */
name RAWDATA :   set (RawData);
name FEATURES :  set (Feature);
name COUNTRIES : set (Country);
function extractFeatures (rawData : RawData);
```

Fig. 10. Weather Map Database Schema in O_2C.

This schema is equivalent to the one given in Fig. 3 for Napier88. Notice that all of the object fields are public thus avoiding encapsulation of data. Data protection may also be provided in a similar manner. For example password protection of access to the raw data can be achieved as shown in Fig. 11 [1]. The class *PasswordCheck* defines private fields to contain the password and a reference to the protected data, and a method *unlock* to return the data if the correct password is presented. The persistent name *unlockRawData* is initialised with a new instance of the class and the direct path to the raw data is then deleted.

```
class PasswordCheck
     type tuple (password : string, data : Object)
     method public unlock (try : string) : Object
end

method public unlock (try : string) : Object in class PasswordCheck {
     o2 Object res = nil;
     if (try == self->password) { res = self->data; }
     return res;
}

name unlockRawData = new PasswordCheck ("secret", RAWDATA);
delete name RAWDATA;
```

Fig. 11. Password Protected Access to Raw Data in O_2C.

The raw data can be completely hidden by replacing the set *FEATURES* with instances of the class *Feature2* as shown in Fig. 12. The field *featureType* is now the only public field. The method *iterate* takes as parameter an instance of the class

[1] Note however that the query language OQL does not restrict access to private fields, and so could be used to circumvent this style of protection.

PixelIterator and invokes its method *process* for every pixel in the feature. The user can thus define a subtype of *PixelIterator* which overrides *process* in a suitable way, and pass an instance of that subtype to the *iterate* method.

```
class PixelIterator
     type tuple ()
     method public process (pix : Pixel)
end;

class Feature2
     type tuple (public featureType : string, feature : Feature)
     method public iterate (iterator : PixelIterator)
end;
```

Fig. 12. First Order Information Hiding in O_2C.

This illustration suggests that the strategy of providing users with layered views over the underlying data can be employed in O_2C applications. Rather than specifying classes which both encapsulate the raw data and define a single set of operations on it, the programmer can separate the storage of the data and multiple views on it into distinct classes, tailored to different users if necessary. This allows further views to be layered on as the need arises, without the need for explicit schema evolution mechanisms such as [Ferrandina, et al., 1995].

4 Conclusions

Our conclusion is that encapsulation of data is a poor method of structuring systems that are expected to evolve. Persistent systems, where structured data is long-lived, bring this problem sharply into focus. We have developed a mechanism (views) that retains information hiding but does so in a manner that does not require the encapsulation of data.

The result does not invalidate the object model, merely refines it. Object hierarchies, sub-typing and object modelling may be performed on the views with the same benefits as before. Protection of data may also be preserved and even enhanced through the use of password systems.

Acknowledgements

This work was supported by EPSRC Grants GR/J67611 “Delivering the Benefits of Persistence” and GR/L32699 “Compliant Systems Architecture”, and the ESPRIT Working Group EP22552 “PASTEL”.

References

Andany, J., Leonard, M. and Palisser, C. (1991) *Management of Schema Evolution in Databases*. In Proc. 17th International Conference on Very Large Data Bases (VLDB), Barcelona, Spain, pp. 161-170.

Bratsberg, S.E. (1993) *Evolution and Integration of Classes in Object-Oriented Databases*. PhD thesis, Norwegian Institute of Technology.

Cardelli, L. (1984) *A Semantics of Multiple Inheritance*. In *G. Kahn, D.B. MacQueen and G. Plotkin (ed) Lecture Notes in Computer Science 173*. Springer-Verlag, pp. 51-67.

Coen-Porsini, A., Lavazza, L. and Zicari, R. (1991) *Updating the Schema of an Object-Oriented Database*. IEEE Data Engineering Bulletin, **14**: 2. pp. 33-37.

Connor, R.C.H., Balasubramaniam, D. and Morrison, R. (1995) *Investigating Extension Polymorphism*. In Proc. 5th International Workshop on Database Programming Languages, Gubbio, Italy, pp. 13-22.

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. and Morrison, R. (1994) *Using Persistence Technology to Control Schema Evolution*. In Proc. 9th ACM Symposium on Applied Computing, Phoenix, Arizona, pp. 441-446.

Connor, R.C.H., Dearle, A., Morrison, R. and Brown, A.L. (1990) *Existentially Quantified Types as a Database Viewing Mechanism*. In *F. Bancilhon, C. Thanos and D. Tsichritzis (ed) Lecture Notes in Computer Science 416*. Springer-Verlag, pp. 301-315.

Dayal, U. and Bernstein, P.A. (1978) *On the Updatability of Relational Views*. In Proc. 4th International Conference on Very Large Data Bases, West Berlin, Germany, pp. 368-377.

Dennis, J.B. and Van Horn, E.C. (1966) *Programming Semantics for Multiprogrammed Computations*. Communications of the ACM, **9**: 3. pp. 143-145.

Ferrandina, F., Meyer, T., Zicari, R., Ferran, G. and Madec, J. (1995) *Schema and Database Evolution in the O2 Object Database System*. In Proc. 21st International Conference on Very Large Data Bases, Zürich, Switzerland, pp. 170-181.

Kirby, G.N.C., Connor, R.C.H., Morrison, R. and Stemple, D. (1996) *Using Reflection to Support Type-Safe Evolution in Persistent Systems*. University of St Andrews Report CS/96/10.

Kirby, G.N.C., Morrison, R., Connor, R.C.H. and Zdonik, S.B. (1997) *Evolving Database Systems: A Persistent View*. University of St Andrews Report CS/97/5.

Lerner, B.S. (1996) *A Model for Compound Type Changes Encountered in Schema Evolution*. University of Massachusetts at Amherst Report 96-044.

Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. and Munro, D.S. (1994) *The Napier88 Reference Manual (Release 2.0)*. University of St Andrews Report CS/94/8.

Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Dearle, A., Rosenberg, J. and Stemple, D. (1990) *Protection in Persistent Object Systems*. In *J. Rosenberg and J.L. Keedy (ed) Security and Persistence*. Springer-Verlag, pp. 48-66.

Odberg, E. (1995) *MultiPerspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases*. PhD thesis, Norwegian Institute of Technology.

Roddick, J.F., Craske, N.G. and Richards, T.J. (1996) *Handling Discovered Structure in Database Systems*. IEEE Transactions on Knowledge and Data Engineering, **8**: 2. pp. 227-240.

Zdonik, S.B. (1990) *Object-Oriented Type Evolution*. In *F. Bancilhon and O.P. Buneman (ed) Advances in Database Programming Languages*. Addison-Wesley, pp. 277-288.

Zdonik, S.B. (1993) *Incremental Database Systems: Databases from the Ground Up*. In Proc. ACM SIGMOD, Washington D.C., USA, pp. 408-412.