Extending Object-Oriented Database Concepts to Support a Wider Range of Applications

Dr. Elizabeth Oxborrow
University of Kent at Canterbury
October 1993

ABSTRACT

This paper primarily addresses the issue of what we really mean by 'Object-Oriented Databases'. It is argued that the conventional view of an object-oriented database (OODB) − with the main object-oriented features consisting of object identity, classification, specialisation, composition and encapsulation − is rather basic and has limited applications. In order to more adequately support both non-traditional database applications and traditional ones, and provide continuing support into the future, a broader view is essential. Features, such as explicit relationships, version management, arbitrary collections, and multimedia support, should be available as standard. Examples are provided to illustrate applications which require these enhanced features.

1 INTRODUCTION

There has been much debate about what object-orientation means − about what can be considered to be object-oriented and what cannot − and as a result, there are many interpretations of the object-oriented paradigm (Blair 1989, Blair 1991). This debate has been carried out primarily in the context of object-oriented systems and programming languages in general. However, there is much more agreement on the main object-oriented (OO) features of an object-oriented database management system (OODBMS), partly as a result of the way in which database technology has evolved. An OODBMS must be able to provide the standard functionality of a traditional DBMS, but additionally it should provide enhanced semantics, behavioural support and an object-based data model. These additional features are considered to be the OO features of an OODBMS, but enhanced semantics have been supported by data modelling methodologies and techniques for many years.

Well before the era of OODBMSs, it was realised that a data model representing a network, hierarchical or relational database does not adequately reflect all the structural semantics of the real world application on which the model is based. Hence, various data modelling methodologies and techniques were developed. The ERM (Entity-Relationship Model) approach was probably the first of these to be widely used; this approach originated from work by Chen (1976) but has subsequently been developed to embrace a wide range of semantic information. In the 1980s, there was a great deal of research in the area of semantic data models (SDMs) (Abiteboul, 1987; Hammer, 1981; Hull, 1987; Peckham, 1988).

Enhanced semantics are common to both SDMs and object-oriented programming (OOP) and an object-based data model has some similarities with the entity-based data models of SDMs; hence, it can be argued that, in terms of concepts, only the behavioural support is really 'new' in a basic OODBMS. However, there are a number of other features which, though they may not be considered to be essential at present, will be essential requirements for future OODBMSs. These features are object-based, and support for them is facilitated by an object-oriented approach.

2 FUNDAMENTAL OBJECT-ORIENTED FEATURES

An OODBMS is obviously a DBMS and must therefore provide standard DBMS facilities which include persistence, a query facility, and underlying data management facilities such as secondary storage management, concurrency and recovery. In addition, of the eight mandatory OO features discussed in Atkinson (1989), five are fundamental and it is generally agreed that they must be supported by a DBMS before it can be considered to be an OODBMS. The consensus on these features is partly due to their consistency with traditional DBMS concepts; in fact, similar features are now accepted by the supporters of traditional approaches (Stonebraker 1990), and relational DBMSs are being enhanced to support them.

The fundamental OO features are:

- the object concept with *unique object identity*
- the distinction between *types* and *classes*
- *specialisation* (also known as inheritance, ‘is-a’ hierarchies, subtyping, subclassing)
- *composite objects* (also known as part-component or ‘part-of’ hierarchies)
- *encapsulation* of structure and behaviour (complete objects)

These features can be illustrated using the example in figure 1. The various aspects of the example are explained below and the terms used are consistent with those of the OMG (Object Management Group) object model (Soley, 1992).[1]

The diagram in figure 1 represents an instance of a particular *object* with *unique object identity* 12345. The object identifier is shown in a dotted ellipse as it is system defined; it is also independent of the application of which it forms a part in the sense that it does not depend on any of the object’s properties or characteristics.

The *type* of an object, which determines its *structure* and *behaviour*, is defined by its *interface* (the object interface is the collection of *signatures* of each of the public (externally visible) operations − Display(), Title(), etc.; the interface does not include the implementation of the methods). Object 12345 belongs to the *class* of objects of type Library Book which share the same implementation.

This object is *specialised* in that whereas all books in a library possess publication information, an accession number and the relevant operations, not all books have a shelf component and participate in an on-loan relationship with corresponding operations. The library may also have other types of specialised book, e.g. confined books which are subject to restricted access and reference books which cannot be lent out and are stored in particular locations in the library.

The library book can also be considered to be a *composite* object. A composite object is an object which is built from components which are concrete or abstract objects in their own right but are considered to be part of the containing object (hence the use of the term ‘part-of hierarchy’). A component is conceptually or spatially contained within some other object and is generally existence-dependent on that object. A car is a simple example of a composite object; its components − engine, body, wheels, etc. − possess a number of properties of their own and are concrete objects spatially contained within the car. Returning to the library book, the publication information is an example of an abstract component conceptually contained within another object. (However, it should be noted that this is not the only way to model library books with their publication information.)

Finally, the example shows the book as a complete object. Its structure (components, relationships, constraints) and behaviour (operations) are all *encapsulated* within the object.

All these features are provided to some extent by the main OODBMSs currently available or under development (e.g. O2 (O.Deux, 1991), Itasca/Orion (Kim, 1990), GemStone (Butterworth, 1991) and many others).

---

[1] Note: The OMG draws its membership from the computing industry; it aims to provide standard interfaces for inter-operable software components based on the object-oriented paradigm.

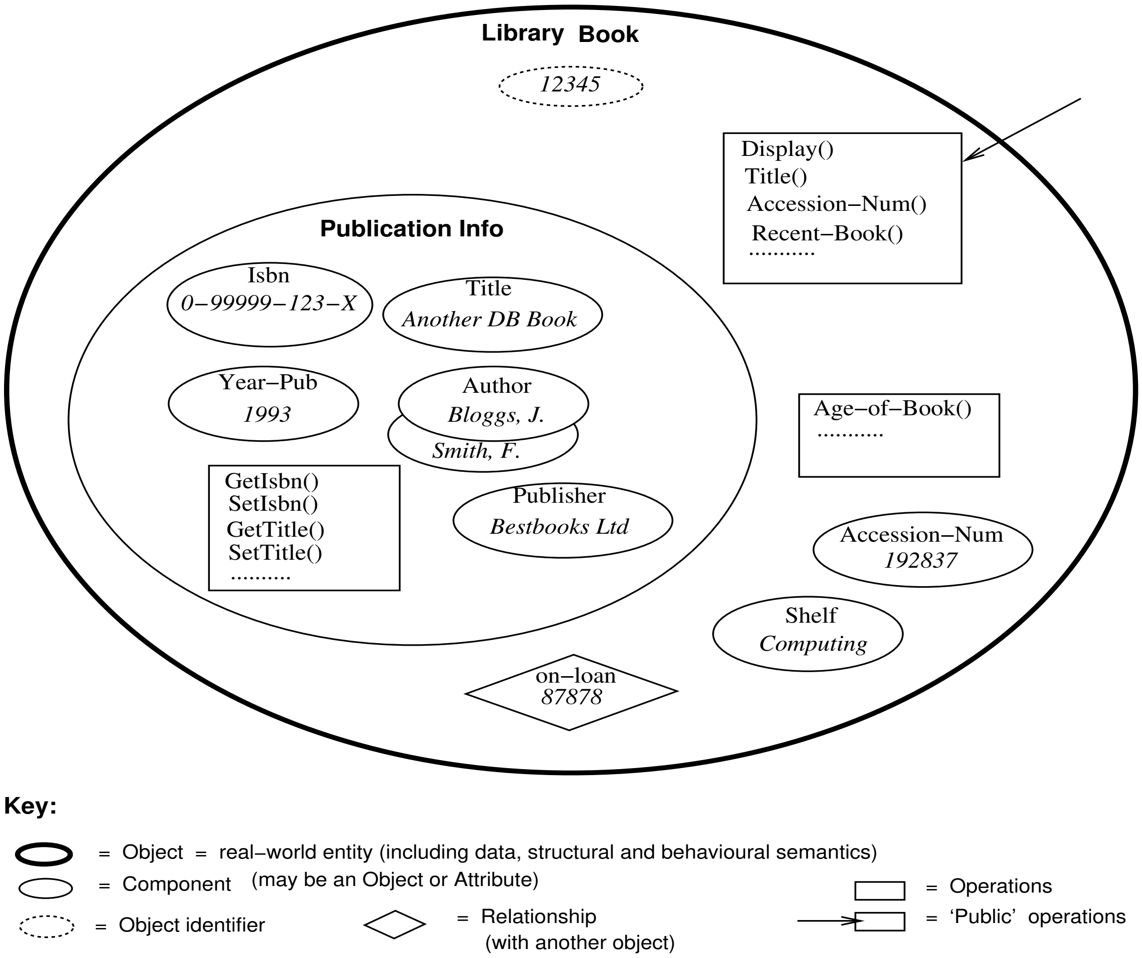
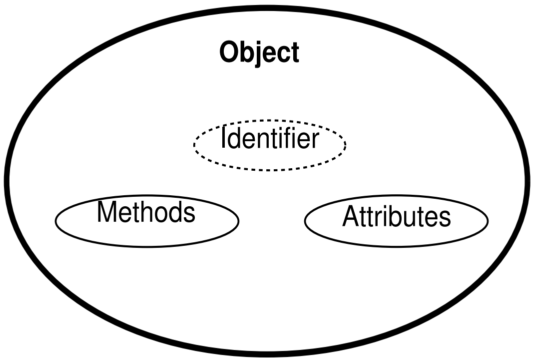


Figure 1 A Library Book Object



Notes:

Attributes may be of various types
(including 'object' for components and related objects)

An object of type 'class' is generally used
as a container for the specification and implementation,
as a factory for the creation of instances, and
as a container for the instances

Figure 2 The Basic Object Model

The basic object model which supports the features is illustrated in figure 2. As can be seen, it is quite a simple model. Objects have unique identifiers, at least one method and an optional collection of attributes. The methods provide access to the attributes, which may be of any standard type (string, integer, etc.) or of type ‘object’ (reference or pointer to object). All relationships with other objects, including component relationships (implicit in composite objects), are represented by attributes of type ‘object’, while specialisation is indicated by specifying the object’s ‘supertype’ (via an ‘is-a’, ‘inherits-from’ or similar clause in the definition). An object’s type definition is generally specified in a class definition, which means that class and type are synonymous. Class is therefore an overloaded concept, embracing the type definition, acting as a ‘factory’ for creating objects belonging to the class, and also representing the container for all objects belonging to the class.

3 WHY THE FIVE OBJECT-ORIENTED FEATURES ARE IMPORTANT

Unique object identity is essential if an object-based data model is to be supported and it facilitates enhanced semantics; specialisation and composition are concerned with semantics from a structural viewpoint; encapsulation is concerned with behavioural support and is fundamental to the OO paradigm; and the distinction between types and classes provides further semantic richness from a behavioural viewpoint. These aspects are considered in more detail below.

3.1 The Object Concept with *Unique Object Identity*

The object concept means that everything is viewed as an object. Each object may represent a concrete or abstract self-contained thing in which we are interested. In order that an object may be related to and interact with other objects, it is important that it should be able to be uniquely identified. The object must be able to exist independently of its properties (attributes, methods, etc.) as these may change over time, so the object identifier must purely act as a surrogate for the object.

Uniqueness and independence of the object identifier not only makes it easier to support interactions and relationships between objects, but it is also easier to support complex objects (incorporating both specialisation and composite object features), different views of, and interfaces to, objects, and evolution of objects.

3.2 The Distinction between *Types* and *Classes*

The specification of the properties belonging to an object (sometimes referred to as the interface to an object) defines the *type* of the object. The type of an object is not concerned with implementation, and hence an object specification should be implementation-independent. The *class* of an object not only defines the type of the object but also its implementation. All objects belonging to a particular class conform to the type associated with the class and have the implementation defined for the class. Two or more classes may have the same type associated with them but different implementations; hence, two objects which belong to different classes may conform to the same type but have different implementations.

As mentioned above, the class concept is an overloaded concept. In some systems/languages (e.g. C++), it is used for type/interface specification (e.g. C++ headers) as well as implementation definition. Apart from these two functions associated with class (the first of which should really be associated with type), a class may perform other functions. It commonly acts as a ‘factory’ for creating objects (c.f. C++ constructors), but this function need not be associated with the class concept − it could be treated as a separate function. In an OODB environment, a class also performs the function of representing the collection (set) of objects belonging to the class (i.e. having the same type and implementation); it is the OO equivalent of a relation in relational DBMSs.

If type and class are intimately connected (as in C++ and many existing OODBMSs), object base design flexibility is limited, a class being the only construct generally provided as a building block for complex objects and collections of objects. Due to the fact that the new applications requiring data management facilities often consist of more complex objects than those of the traditional data management applications, the extra flexibility offered by distinguishing between type and class is particularly useful.

3.3 *Specialisation*

Specialisation enables variations between otherwise fairly similar objects to be catered for, and hence provides more sophisticated classification of objects. Objects of a particular generalised type may possess specialised features which are significant in the context of an application. For example, some employees may be salespersons possessing a commission property in addition to the properties possessed by all employees; other employees may be managers, secretaries, accountants, etc.

In traditional systems, employees, salespersons, managers, etc. would be treated as though they were different entity sets (being represented by different relation or record type descriptions). Hence, a particular salesperson, for example, would appear as two separate 'entities' in the database; to create that person as a single entity, the separate records would need to be combined by the user. In an OODBMS, on the other hand, a single entity would be represented logically by a single object.

3.4 *Composite Objects*

Composite objects occur naturally in the real world. Many objects are built from other objects or, conversely, are part of other objects. In traditional systems, the conceptual or spatial containment implied by composite objects cannot be represented explicitly. The component objects must be modelled as separate objects, with the composite object being related to the component objects.

3.5 *Encapsulation* of Structure and Behaviour

The most important aspect of encapsulation from an OODBMS viewpoint is that it enables objects to be completely defined. Traditional DBMSs only support the data aspects of entities, although recent enhancements to some systems provide some support for functional aspects (the ability to define procedures associated with the entities in a database). The advantages of OO encapsulation are many.

Firstly, the integration of the data (structural) and functional (behavioural) properties of an object is an aid to application design. When traditional methodologies are used, it is not uncommon for separate teams to work on the data and functional aspects, and it is also not uncommon for there to be problems when trying to integrate the results. The entities and their data properties which are identified in the data modelling and database design stages are defined in the database; the functional aspects which are identified in the functional analysis and design stages are defined in the application program, but it is unlikely that the functions/procedures can be associated one-to-one with the entities in the database. This mis-match can give rise to problems in maintaining the database and applications, as well as in their integration during implementation.

A second advantage of encapsulation is that the resulting 'complete' objects more closely reflect our own view of the real-world equivalents. Hence, an object-oriented approach should provide a better basis for end-user communication and involvement in the analysis and design processes.

A third advantage is that it offers great potential for the development of diagrammatic techniques and notations for use in graphical user interfaces (GUIs). This is of benefit not only to application designers but also to end-users. In traditional analysis and design methodologies, different techniques and diagrammatic notations apply to the data and functional aspects, and these are database- and program-oriented rather than being real-world oriented. Although the object-oriented analysis and design techniques which have so far been developed tend not to be particularly well-integrated and user-friendly, the potential exists with the object-oriented approach.

4 WHAT'S NEW?

Most of the object-oriented features mentioned above are recognisable as semantic data model (SDM) features – entities with unique identifiers, classification (though without the distinction between class and type), specialisation and composition. SDMs were an important area of research in the 1980's; they provided enhanced modelling facilities but did not 'take off' as implemented systems. What OODBMSs can offer is the ability to implement the SDM features directly, providing a close mapping between data model design and database implementation.

The object-oriented feature which is really 'new' (as far as DBMSs are concerned) is encapsulation. From the viewpoint of a single object, encapsulation means that the object can be designed to be complete in terms of structure and behaviour. However, although this is new, it is not really revolutionary; it can be considered as a further enhancement to database (DB) concepts which are already familiar to DB specialists. (As an aside, object-oriented analysis and design can be considered to be more revolutionary, but that is a topic for discussion in its own right and beyond the scope of this paper.)

5 WHAT MORE DO WE NEED?

Object technology can offer much more than just the commonly-accepted features discussed above. The object concept is the key to much greater flexibility. The notion that an object represents a concrete or abstract thing (i.e. a real-world entity or concept) suggests that object technology ought to enable some more aspects of reality to be reflected in the OODB. Oxborrow (1992) identified some important additional facilities which should be provided as standard in future OODBMSs. Support for the full semantics of relationships, for the evolution of objects in terms of different versions, for arbitrary collections of objects, and for multimedia objects was considered to be essential; these additional features which need to be supported are all specifically object-oriented in nature. Enhancements to traditional DBMS mechanisms for access control, concurrency control and secondary storage mechanisms were also identified as being necessary, and the importance of extensible and reusable systems was also stressed. The remainder of this paper takes a look at the additional OO features in the context of two example applications, and then considers the solution to the problem of how to support them which was explored in the recently-completed Zenith research project.

5.1 A Building Plot Example

The first example application represents part of an application which includes a fairly complex object. It is by no means an unusual application and such complex objects could (and do) occur in many other applications. It should be stressed that this is not a complete example − no operations are shown, and only a few relationships are given; to avoid cluttering up the diagram, the example includes only those aspects which are needed for illustrative purposes. Figure 3 represents the application, which involves plots of land owned by a builder and illustrates the information requirements needed for drawing up plans, getting planning permission, and everything else through to the actual building. On each plot of land, a number of houses can be built, so there is a plot plan and the builder has to obtain planning permission for the plot. Each house is architect designed and the house plan includes plans for each room. Related to each plot of land is the planning authority which is responsible for approving/rejecting the planning application, and associated with each house design is the architect who provides the sketch and plans for the house.

Not all of the components which are objects in their own right are expanded in detail. For example the Planning Information component of Plot is only shown in outline; if expanded, it would be complex like the House component. This example highlights the need for versioning and multimedia support.

5.1.1 Versioning

Many of the components of the Plot object may be modified over time with a requirement to retain old versions. For example, the Plot Plan may be changed if a planning application is rejected, but the old version of the Plan must be kept for reference purposes, together with the old version of the planning application. Furthermore, the architect may modify the design of one or more of the houses. In fact, if he reuses designs he may have a number of variants of a particular design and use different variants for different houses and possibly on different plots as well.

This example illustrates the need for the management of complex version trees, with the potential for a number of default versions depending on the context (e.g. the specific design associated with an actual house which is being built) and on the user (e.g. the architect may be working on a particular variant of a design and so his default version will be different from the builder's default for a particular house on the plot).

Version management is required in all design environments (software design, engineering design, etc.). Efficient support for version management, though, offers the possibility of making versioning available in areas where it has been required in the past but could not feasibly be used. Hospital DBs are an example in which chronological versions are required for patients' records. Special-purpose functionality has to be built into such applications at present, due to lack of support by traditional DBMSs. Local authority DBs provide another example; from the planning viewpoint, plots of land exist in different versions over time as houses are built/extended, shops change use, etc.; from the council housing viewpoint, the history of different versions of houses are needed to reflect their maintenance/modernisation; from the social services viewpoint, the status and other aspects of those claiming benefit change over time. Some of these changes require simple chronological version trees, while others (e.g. planning) may require more complex version trees.

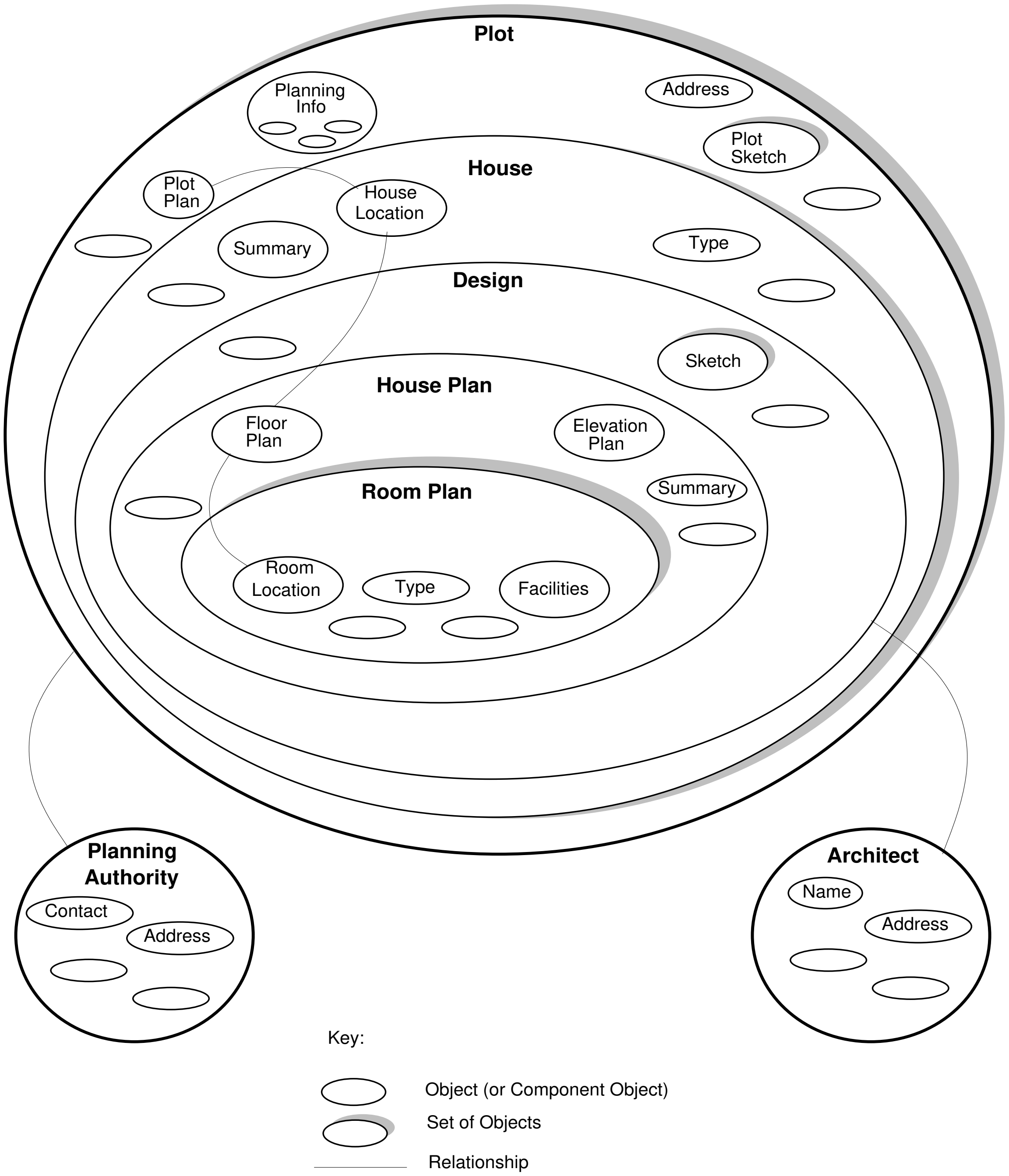


Figure 3 A Building Plot Object

5.1.2 Multimedia

Some of the components in the Plot object will be in non-standard media. The sketches may be originally hand-drawn and inserted into the database by means of a scanner. The plans could be computer-produced using a CAD tool. There may also be photographs of the plot taken at various stages during the building process.

5.2 An Art Gallery Example

The other example application illustrated in figure 4 relates to an art gallery which maintains a catalogue of artists and paintings; some of the paintings are displayed in the gallery in rooms supervised by members of staff, some of whom have expert knowledge of artists in the catalogue and are responsible for maintaining the relevant parts of the catalogue. Although not fully developed, this example illustrates how a complete application can be modelled in terms of objects; it includes objects which provide a service or perform management or control functions − the Enquiry Server, Acquisition Server and Supervision Scheduler. It clearly shows how applications consist of *interacting* and *interrelated* objects and highlights the need for full support for relationships and arbitrary collections of objects.

5.2.1 Relationships

A number of relationships are shown with their full semantics. For example, the relationship between Artist and Painting shows that an artist must have painted at least one painting and may have painted many, while a painting must be painted by exactly one artist. On the other hand, the relationship between Artist and Staff Member is completely optional indicating that there may be no staff who are experts on a particular artist and staff do not need to be experts on any artist.

Note that interacting objects do not need to be related. For example, on the assumption that the Catalogue object has an EnterArtist() function, a Staff Member object can pass a message to the catalogue to enter a new artist, without that staff member being logically related to it in any way.

The explicit representation of relationships is important for ensuring that the integrity of the database is not violated. It is not sufficient merely to know that two objects are related, as is the general case in OODBs in which relationships are represented by attributes of type ‘object’. It has already been indicated that this mechanism does not enable one to distinguish between component relationships and general conceptual relationships, but it also does not enable the cardinality and mandatory/optional existence of relationships to be represented and hence such constraints cannot automatically be checked.

Without full support for relationships, it will be difficult to integrate application design and implementation. Design diagrams explicitly represent different types of relationship and relationship semantics. Graphical user interfaces to applications must also enable these to be represented and hence the object model underlying the database must include the appropriate concepts.

5.2.2 Object collections

How should the Catalogue be represented? In figure 4, it is represented as a *collection*. This is a single object which contains (is a collection of) all the artists and paintings in the gallery’s catalogue, together with their interrelationships. The ‘class’ concept is well-suited to collections of objects all of the same type and with the same implementation, but it is not so well suited to arbitrary collections of objects (e.g. the Catalogue in figure 4, in-trays and committees in an Office Information System (OIS), configurations in a software database, etc.). Although many systems do provide limited (generalised) facilities for arbitrary collections (e.g. sets, bags), different types of collection in an application may have specific semantics associated with them − semantics which identify the nature of the collection and determine the constraints on membership (e.g. which objects can or must be members of a collection, whether they can be a member of more than one, etc.). It is important for such application semantics to be able to be explicitly specified.

Collections differ from composite objects (‘part-of’ hierarchies) in which the components are logically part of the containing object and can, in general, only be ‘part-of’ one containing object at a time; in collections, the components (or members of the collection) are logically independent. In figure 4, for example, the objects representing artists and paintings are complete in themselves without the catalogue, whereas in figure 3, the room plans are not meaningful on their own − they are only relevant in the context of the house plan. Also, in the case of a company database, an employee can obviously exist as an independent object, as well as being able to be a member of a committee and other types of collection (grouping) in the company.

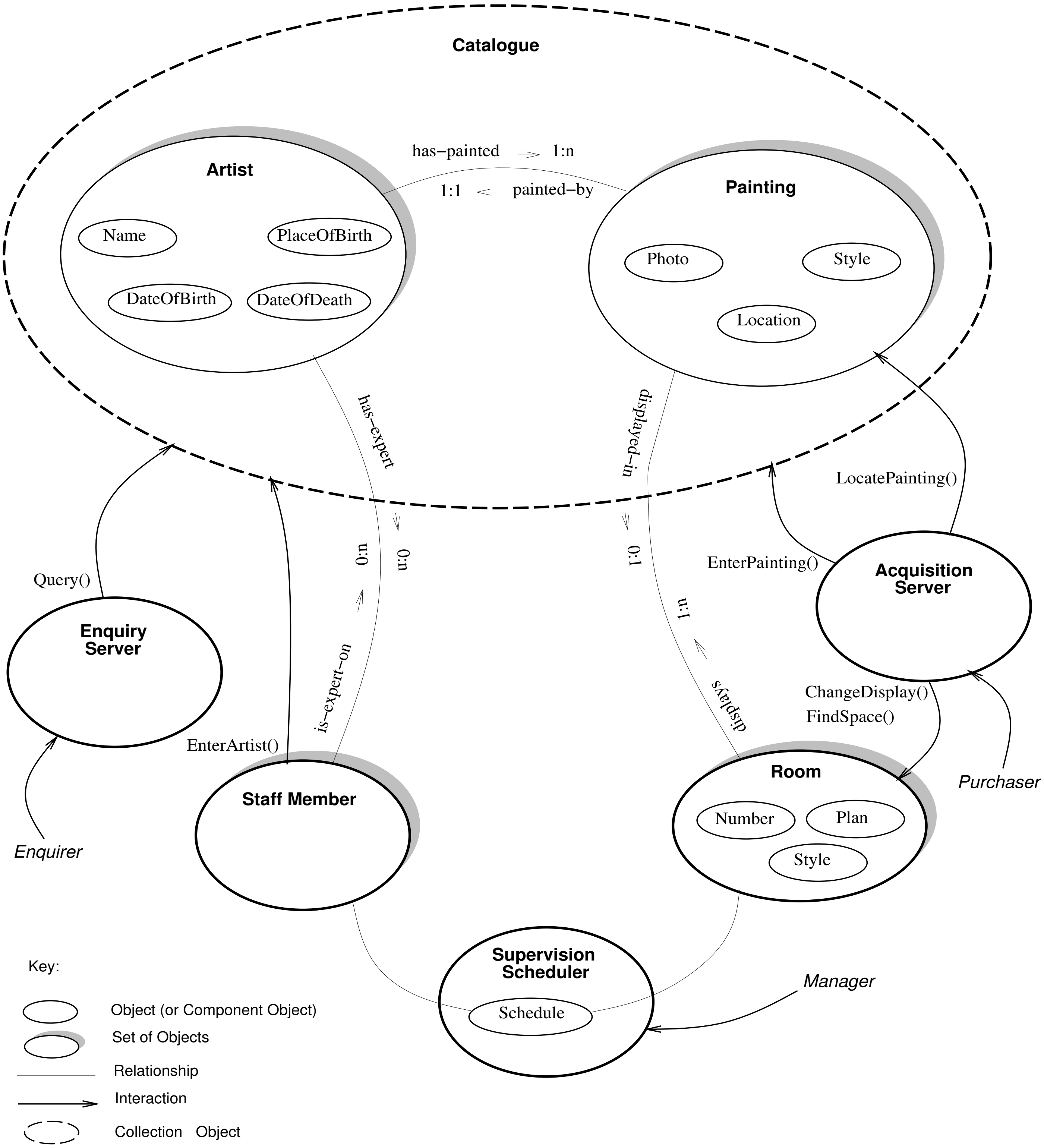


Figure 4 An Art Gallery Example

An object may be a member of many different collections or groupings, and may be just a transient member (as in the case of objects in an in-tray) or a (semi-)permanent member (as in the case of artists and paintings in the art gallery catalogue). Figure 5 shows part of the art gallery example with the catalogue collection and an alternative representation for the display of paintings. In this example, paintings are not only members of the catalogue collection but may also be members of one of the display collections. Each display collection is an object which represents the collection of paintings displayed in a room. A further example to illustrate the relevance of different types of collection is a company in which committees, project teams, working parties, etc. (all of which are collections of employees) may exist or be created from time-to-time.

Obviously, the choice of construct to use in modelling a particular application depends on the modeller's view of that application, but there is a need for a generalised 'collection' concept to enable real-world complexities to be more appropriately modelled. As mentioned before, class is an overloaded (multivalued) concept; one aspect of it − the container for objects conforming to the type and implementation of the class − is just a special case of a collection.

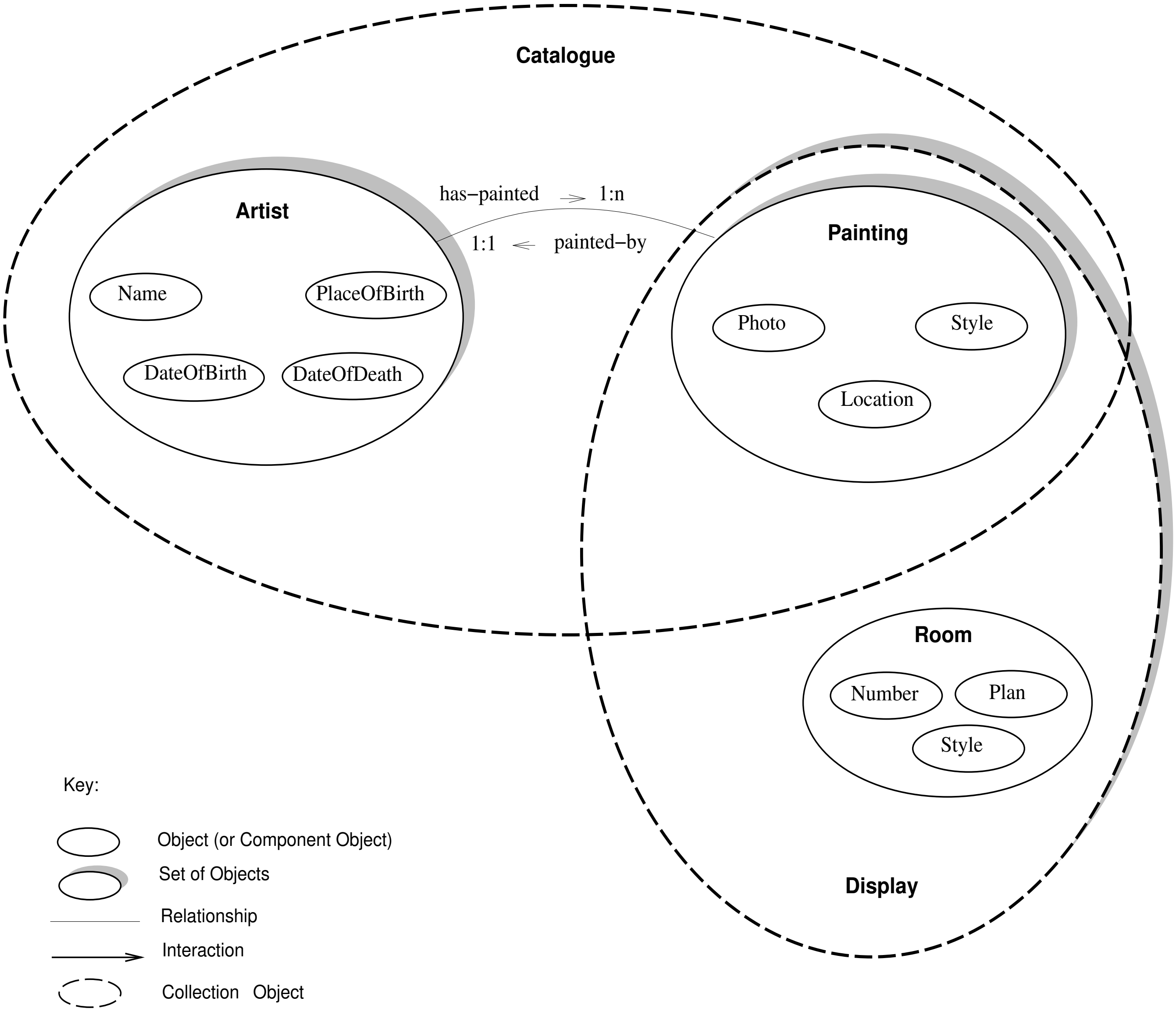


Figure 5 Part of the an Art Gallery Example
Alternative representation for the Display of Paintings

6 FLEXIBLE OBJECT MANAGEMENT SUPPORT FOR A VARIETY OF APPLICATIONS

The problem of how to support the object management requirements of a range of applications from different environments was addressed by the Zenith project (Kemp, 1992), a collaborative project carried out jointly by the Universities of Kent and Lancaster between October 1989 and September 1992 (funded by the SERC under grant nos. GR/F37610 and GR/F37627, and 'uncled' by BT Labs.). The most important feature of Zenith in the context of this paper is the object model. The basic object model of figure 2 was found to be inadequate, and an enhanced model was designed to support a variety of different requirements.

Figure 6 illustrates the main characteristics of the Zenith object model from the viewpoint of the specifier or 'user' of the object (as distinct from the implementer of the object). The model includes a unique object identifier as in the basic model. It also includes methods, but only the interface information which determines the type of the object (the implementation is not of relevance to specifiers and general users). The main differences between this and the basic model concern the treatment of attributes and the class concept. The Zenith object model can provide all the facilities needed to support the traditional class concept but does not explicitly possess a class construct. Instead it is sufficiently flexible to enable different types of collection (including class) to be defined by a user of the model.

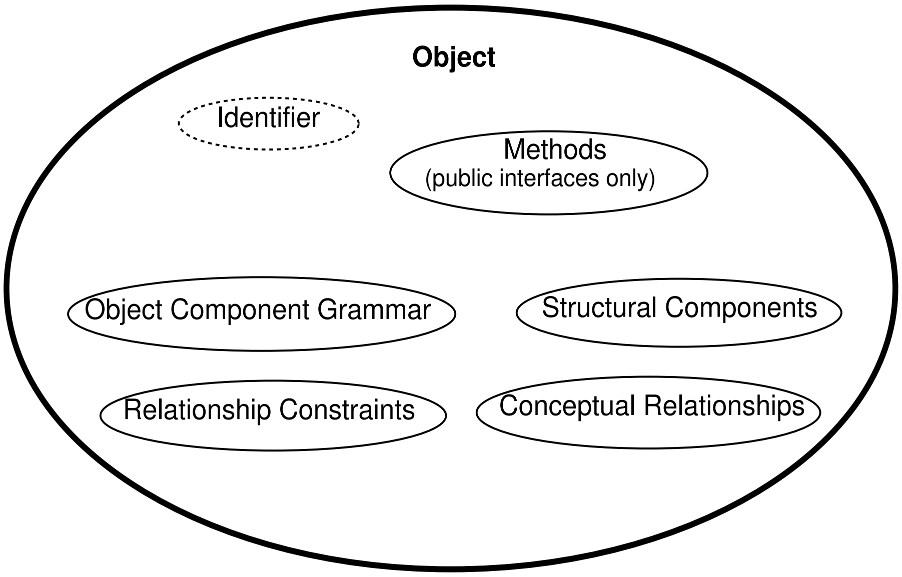


Figure 6 The Zenith Object Model

One feature which aids flexibility is the ability to distinguish between components and relationships at the conceptual level. Components and relationships replace attributes in the basic model. Components may be implemented as simple attributes, may be calculated from the stored internal state of the object, or may be objects in their own right. To the user, the implementation is not important; what is of importance is that methods exist which provide access to them. Components are related to the containing object but in a special way. General conceptual relationships with independent objects are separately identified in the model (and accessible via appropriate methods).

Associated with the actual components of an object is an object component grammar which defines the permitted component types and their semantics. Associated with the actual relationships between an object and other objects are the relationship constraints which define the permitted relationships in which this object may participate and their semantics. The object component grammar and relationship constraints are part of the type (specification) of the object. They are logically part of the object but may be physically separated and shared by other objects.

Using the Zenith object model, the additional features mentioned earlier in the paper can easily be supported. Relationships form part of the model. Composite objects and different kinds of collection are supported by specifying particular component semantics in the object component grammar. For example, components of a composite object which cannot exist independently are associated with delete-propagation semantics; the components of a configuration can exist independently and must not automatically be deleted when a configuration is deleted; the components of an office in-tray may have a priority ordering in the in-tray according to the type of component (letter, telephone message, etc.); and, a ‘class’ object must contain all the objects which conform to its particular type and implementation.

The versioning feature can be supported with the aid of relationships. In the Zenith project, a prototype object management system was developed. It was designed in terms of Zenith objects and version management was easily added to the basic system by defining version manager objects and relationships with special version management semantics to enable version trees to be built and the different versions and variants to be linked together.

The Zenith object model not only enabled an extensible prototype to be developed but was also sufficiently flexible to enable distributed and multimedia facilities to be incorporated, and fine granularity access mechanisms to be provided (down to the single-method level).

7 CONCLUSION

Key object-oriented concepts include the object concept with unique identity, classification, specialisation, composition and encapsulation. These are ‘enabling’ concepts and are necessary but not sufficient in the context of the wide-ranging applications which require object management support today. It has been shown that the following are also of importance: distinction between the components and relationships of an object (in order to provide a higher level of semantic support for applications than is possible with conventional attributes), versioning of objects (to enable the inherent immutability of objects to be represented), and object collections (to enable more general groupings of objects to be represented than is possible by means of the class concept); furthermore, multimedia support is essential.

In conclusion, a flexible model and extensible facilities are the keys to the success of future DBMSs. The object-oriented approach appears to be sufficiently flexible and extensible to enable the current and future requirements of a wide range of applications to be supported. More research and development is needed, but it is suggested that OODBMSs will have an important role to play in the future. However, their successful use for major application development will depend to a certain extent on the object-oriented analysis and design expertise of the OODB designers. Only when a truly object-oriented approach to design is adopted, will it be possible to effectively and efficiently integrate analysis, design and implementation, and hence fully exploit the potential of object-oriented database technology.

REFERENCES

Abiteboul, S. and Hull, R. (1987) IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12 (4), pp.525-565.

Atkinson, M. et al. (1989) The object-oriented database system manifesto, in *Proceedings of the Conference on Deductive and Object-Oriented Systems (DOODS-89).*

Blair, G. et al. (eds) (1991) *Object-Oriented Languages, Systems and Applications*, Pitman.

Blair, G., Gallagher, J.J. and Malik, J. (1989) Genericity vs Inheritance vs Delegation vs Conformance vs ... *Journal of Object-Oriented Programming (JOOP)*, 2 (3).

Butterworth, P., Otis, A. and Stein, S. (1991) The GemStone Object Database Management. *Communications of the ACM*, 34 (10), pp.64-77.

Chen, P.P-S. (1976) The Entity-Relationship Model – toward a unified view of data. *ACM Transactions on Database Systems*, 1 (1), pp.9-36.

Hammer, M. and McLeod D. (1981) Database description with SDM: a Semantic Data Model. *ACM Transactions on Database Systems*, 6 (3), pp.351-386.

Hull, R. and King, R. (1987) Semantic database modeling: survey, applications and research issues. *ACM Computing Surveys*, 19 (3), pp.201-260.

Kemp, Z.P. et al. (1992) Zenith system for object management in distributed multimedia design environments. *Information and Software Technology*, 34 (7), pp.427-436.

Kim, W. (1990) *Introduction to Object-Oriented Databases*, MIT Press.

O.Deux et al. (1991) The O2 System. *Communications of the ACM*, 34 (10), pp.34-49.

Oxborrow, E. (1992) What should go into an OODBMS project (and why there is no simple answer!), in *Object Management*, (eds R. Tagg and J. Mabon), pp.37-46.

Peckham, J. and Maryanski, F. (1988) Semantic data models. *ACM Computing Surveys*, 20 (3), pp.153-189.

Soley, R.M. (ed) (1992) *Object Management Architecture Guide*, 2nd edn, Object Management Group Inc.

Stonebraker, M. et al. (1990) Third-generation database system manifesto, in *Object-Oriented Databases: Analysis, Design and Construction*, (eds Meersman, Kent and Khosla), North-Holland.