University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1994-54

Evaluation of OMG, ODMG, X.500, and X.700 Data Models*

Juha Taina

Helsinki, November 1994

Report C-1994-54

University of Helsinki
Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

* This report is based on work done in Darfin-project (Project No 30018) funded by Telecom Finland.

Evaluation of OMG, ODMG, X.500, and X.700
Data Models†

Juha Taina

Report C-1994-54
Helsinki, November 1994

Abstract. Database access in Intelligent Networks plays a significant role in the whole communications system. Still in the current ITU-T recommendations the database access models are left open. The documented areas of the ITU-T recommendations are all based on object oriented approach. Thus an object oriented approach to the database access is a logical decision.

We examined different object models that can be used in the IN database. The models examined are Object Management Architecture from the Object Management Group, Object Database Management Group Standard, Open systems Directory Recommendation (X.500), and Open Systems Management Recommendation (X.700.) This raport summarizes the models. It is intented to offer a compact reference to the models.

Key Words: Object Oriented Databases, Object modelling, Object model standards, Telecommunication, Intelligent Networks, Open Distributed Processing.

CR Classification: C.2.4, H.2.1, H.2.4, H.4.3

†This report is based on work done for Darfin-project (Project No 30018) funded by Telecom Finland.

Contents

1 Introduction

In the current ITU-T recommendations, the general IN model seems to be object oriented, while the database access model is usually defined as being based on the relational model. This forces an extra mapping between the object oriented standards and the relational standards, and thus enforces more complex and request-dependent solutions. However, an object oriented approach is a workable solution as long as the requirements for such a system are well defined.

The database and network communities have defined a few object oriented approaches and standards for different conditions. In this paper we will examine and evaluate four of the standards. We will also consider the requirements for the different object models within an IN database node, and how different object oriented standards can fulfill each of the requirements. Currently we have identified four different model levels in a distributed IN database:

- Internal core model, which defines the standard object behavior that all levels of an object-oriented database must follow.
- Language model, which defines the data definition language, the data manipulation language, and the high level language bindings.
- Directory model, which defines how data can be located and retrieved in a distributed manner.
- OSI management model, which defines how distributed nodes can be managed by a distributed management system.

While all the models have different views about object oriented approaches, they all have similarities in definitions. An important question is whether it is possible to create a simple mapping from one standard to the others, and thus allow the logical model of the IN distributed database to be defined with a single object oriented standard. If it is possible then it is also probable that within a few years working commercial solutions come to the market.

We will consider the following object model standards and recommendations:

- Object Management Architecture, which is a logical core object model.
- Object Database Management Group Standard, which is a more detailed object model for database management systems.
- Open Systems Directory Recommendation (X.500), which is the OSI directory information model, and
- Open Systems Management Recommendation (X.700), which is the OSI management information model.

The principal structures of the models are all alike. They all recognize the general principles of object modeling, such as object identity and inheritance. However, the differences are imminent in the description levels, the used models and the generality levels.

Our report is organized as follows. Section 2 is a brief introduction to object modelling. Section 3 is about the Object Management Architecture and its Core Object Model. The Section is strongly based on books Object Management Architecture Guice ([OMG92]) and The Common Object Request Broker: Architecture and Specification ([OMG91].) Section 4 is about the Object

Database Management Group standard, and its object model. The Section is strongly based on the book Object Database Standard: ODMG –93 ([Catt94].) Section 5 is about the X.500 Directory recommendation, and especially its object model. The Section is strongly based on the CCITT Blue Book Recommendations X.500 to X.521 – Data Communication Networks: Directory ([CCIT89].) Section 6 is about the X.700 Management recommendation, and especially its object model. The Section is strongly based on the CCITT recommendations in X.700-series ([CCIT92a], [CCIT92b], [CCIT91a], [CCIT91b], [CCIT92c], [CCIT93], [CCIT92d], [CCIT92e].) In Section 7 we will compare the four referenced object models. Section 8 is the summary.

2 Introduction to Object Modeling

The general idea behind object modelling is simple: in real world we usually deal with distinguishable objects that have a structure and standard behavior. For instance, a pair of socks on a chair consists of two objects – the socks. The objects have something in common (e.g. they can be recognized as socks no matter how dirty they are) but they also have differences (e.g. one sock is green, the other is yellow). The same idea can be moved into abstract worlds.

The principal item in an object model is an object. An object is an abstraction of a real life that can be exclusively identified, for instance a person, a service, or a language grammar. Objects are created and deleted, and during their lifetime they remain the same even if all their attributes are changed. This ability to be identifiable is called object identity. Every object has an unique object identity that distinguishes it from all other objects.

Object identity differs object oriented models from value oriented models where the values of an item define its identity. In value oriented models two items are the same if all their values are the same. In object oriented models two objects are the same if and only if their object identity is the same. Thus object identity can be considered as an automatically set static key that will distinguish all objects from each other.

An object is a frame for the abstraction of a real life entity. The actual abstraction is maintained by an object structure inside the object. Usually this is done with attributes and operations. An object has usually both an internal structure and an external structure. Only the external structure is visible to other objects.

An attribute is a data location inside an object that has a name, a value, and a type. The name is used for referencing the attribute, the value defines the current state of the attribute, and the type defines how the value can be interpreted. For instance in an object that abstracts a person there can be an attribute of birthdate. If the abstracted person was born January 15, 1928, then the corresponding attribute has name *birthdate*, value *January 15, 1928*, and type *string.*

Attributes can be either simple or structured. A simple attribute has an undivisible type such as integer or character. A structured attribute can be further divided into subattributes all of which can be structured themselves. For instance, in the previous example attribute birthdate can be considered a structured attribute or a simple attribute depending on the point of view. It is of an undivisible type string, but on the other hand a string can be interpreted as an array of characters.

A structured attribute may have named subattributes that themselves can be structured attributes. For instance, we could have formed the attribute birthday in the previous example the way that it has three subattributes: day of type integer, month of type string, and year of type integer.

An attribute can also be a reference to another object. Such an attribute is often called a relationship attribute because it forms a relationship between two objects. For instance, an object person might have a relationship attribute supervisor which is the person object of his supervisor. When the attribute supervisor is referenced the reference is actually done to the other object that is an abstraction of the supervisor. This relationship also states that the person has a supervisor, and thus defines a relationship.

In certain special cases it is possible that an attribute is unnamed. This usually occurs when a group of attributes are structured into a single structure.

There are usually at least the following attribute structures:

- Attribute groups. An attribute group is a static group of named attributes. An attribute group structure cannot change during object's lifetime. For instance, an object of a person might have an attribute group birthday that has attributes day of integer, month of string, and year of integer (see the example above.) Attribute groups are often called structures or structs.
- Sets. A set is an unordered group of named or unnamed attributes of a single type. A set does not allow duplicates. For instance, some possible sets are a set of integers (unnamed attributes), a set of birthdays (named attributes), and a set of supervisors (object attributes.)
- Bags. A bag is a set that allows duplicates.
- List. A list is an ordered bag. The specific order may or may not be defined.
- Arrays. An array is a one dimensional array of named or unnamed attributes of a single type. An array item can be referenced directly with an index number.

All structured attributes may have substructures of arbitary dimension. Thus it is possible to have an array of list of set of integers, or a collection that has a list of bags of strings and a set of objects of persons.

An operation of an object is a function that performs some kind of action into the object. Thus an operation is a function that of which main scope is the object whose operation it is. The action may change the external or internal state of the object such as changing its attribute values, it may do some side effect that can affect several objects, or it may return a value that describes the object state.

An operation consists of a type, a name, and a set of parameters. The type is the return type that the operation returns upon exiting. The name is the name that is used for calling the operation. The set of parameters is a set of values of certain types that are sent to the operation. Operation names do not have to be unique. An operation is identified through a triple {return type,name,parameters} which is also called operation's signature. This allows operations that have a unique name to have different parameter instances. For instance, an object of type imaginary number might have an operation *add* which adds a constant to the object. The first operation definition might be *imaginary add(imaginary us, imaginary a)* which adds another imaginary number to us. But it is also practical to have an add operation that adds a normal integer to an imaginary number object. Then the operation definition is *imaginary add(imaginary us, integer a)*. This is called operation overloading. The actual instance that is used is decided upon call time depending on the parameters that are given to the operation. This is called late binding.

The structure of an object is defined in a special static definition called object class or object type. All objects that belong to the same object class have the same attributes and operations, but not necessarily the same attribute values or operation implementations. Similarly, an object class is a set of similar objects all of which belong to the class itself and follow its definitions. Thus the object class is the information base of objects and their attributes.

Object classes maintain attribute and operation information about their objects. Objects themselves maintain the state information of themselves. Two objects that belong to the same class don't necessarily have the same state information. For instance, we can have two objects in class person. Both objects have then the same attributes and operations. But if the first object has an attribute birthday of value January 15, 1928, and the second object has an attribute birthday of value September 14, 1977, then the objects have different states. However, even if they had the same state they still would not be the same object as all objects have a unique object identity.

An object class may have its own attributes. Such attributes are called static and they can be referenced from all objects that belong to the class. When the attribute value is changed the change is immediately visible to all objects. An object class may also have its static operations but they are not as common as static attributes.

Usually the object class definition consists of a public interface and a private interface. The public interface is visible to other objects while the private interface is for internal use only. Sometimes the restrcition is too strong, for instance in situations where another object class must see a substructure of a class while the rest of the classes do not. This situation may occur in object class of linked lists. A linked list consists of linked objects all of which belong to another class. Thus every linked object should be able to reference the whole linked list object structure, not only the public part. The problem is solved with a *friend* definition. A friend of an object class is a class that is allowed to see both the public and the private interface.

Object classes usually form an object class hierarchy. A class hierarchy means that object classes are related to each other in a way where one class is a special case of another class. The whole set of classes forms then a complete class hierarchy. If a class A is a special case of class B then it is called the subclass of class B, and class B is called the superclass of class A.

The subclass/superclass hierarchy is related to inheritance. A subclass inherits the structure of its superclass. Thus all attributes and operations that are defined in the superclass are also available in the subclass. If a subclass has an operation or an attribute that is also defined in the superclass with the same type and signature then the subclass operation or attribute is used. This is called operation shadowing or attribute shadowing. There can be subclasses of subclasses to arbitary dimension.

The object class inheritance is an useful tool because it offers a good method to view the real world. For instance, class man is a subclass of class person, which is a superclass of class mammal. Class woman is also a subclass of class person. If class mammal has an attribute *integer blood_pressure* then the same attribute is also in class person, class man, and class woman. Class person may have its own attributes such as *string language*. All subclasses of person have the same attribute, but the superclasses do not necessarily have it. They may have it if the same attribute is defined there too. In that case, the attribute definition in class person shadows the attribute definition in class mammal.

In class inheritance a class may be an abstract class. An abstract class does not have any objects of its type but it lets its subclasses to inherit its attributes and operations. Usually such object classes are default classes that define default behavior. For instance, it is quite common that the whole class inheritance hierarchy has a single abstract class *root* which is a supertype of all other classes in the class hierarchy.

In the previous example the superclass person had two subclasses, class man and class woman. This is common, but it is also common that a subclass has several superclasses. This is called multiple inheritance. For instance, a class person can be a subclass of class mammal and a subclass of class customer. This superclass/subclass hierarchy states that a person is a mammal and a customer of a company. A company might have other customers, such as other companies.

Sometimes multiple inheritance is a useful tool but it can also cause serious problems if it is used improperly. Problems may occur if a class can inherit attributes via two different inheritance routes. For instance, in the previous example there might be an abstract superclass *root* that has default attributes and operations. In such a case class person inherits the class *root* twice: both via class mammal and via class company. Usually this problem is solved so that the operations and attributes are inherited only once, and everything that can be shadowed on lower levels are also shadowed in higher levels. For instance, if class mammal had an attribute *integer weight* then the same attribute type holds in class person even if the superclass *root* has the same attribute name with a different type, such as *double weight*. Another problem arises if class company has also an attribute *double weight*. Then only one of the weight-attributes can be inherited. Sometimes the attribute name has the class name in front of it so that it can be known where the attribute is defined. For instance, in the previous example we could have mammal.weight and root.weight. However, usually it is desired that a single attribute name is inherited only once.

3 Object Management Architecture

Object Management Architecture (OMA) was designed by the Object Management Group. The model is a basic abstract mode for object oriented applications that want to communicate with each other in a standardized manner. It is intented to be used in several different application platforms and in distributed environments.

The Object Management Architecture is designed to be portable. It allows design portability across several applications. This means that applications that follow the Object Management Architecture can be implemented in different platforms with different language bindings as long as the model standards are followed. The Object Management Architecture does not attempt to develop a syntax for an object definition language, or specific bindings for the object model to a programming language.

The Object Management Architecture is designed to be simple and abstract. It defines an object model that is based on objects, object types, and operations. Only the external interface of an object is defined in the standard. The external inteface consists of operations alone. Attributes are not supported in the model. However, attributes may be used in the internal object implementation that is not visible to other applications that follow the Object Management Architecture.

The Object Management Group has several objectives for their architecture. The objectives cover the total capabilities of the Object Management Architecture (OMA). The objectives are:

- Conformance to the OMG Object Model. OMA follows the OMG Object Model.
- Object distribution transparency. Objects do not need to know the distribution of remote applications.
- Performance for local and remote server object requests. It is possible to evaluate the performance of requests to objects in areas such as scalability, method invocation, time, storage overhead, resource consumption, parallelism, and throughput.
- Extensible and dynamic nature. The implementation of objects may be changed dynamically without affecting the interface to them.
- Provide a naming system architecture that supports multiple naming contexts. It is possible to name objects in unambiguous way and in several levels.
- Queries based on object names, attributes, and relationships with other objects.
- Support discretionary access control of objects. Access control can be set by an object's owner and cannot normally be circumvented.
- Provide concurrency control for objects. An object can have parallel simultaneous access to one or more other objects.
- Provide transactions for the purpose of committing the result of an operation or sequence of operations on distributed objects atomically. There is support for atomic short transactions, and atomic long transactions. The former is more important than the latter.
- Provide robust operation and a high level of availability. The environment can recover from errors and exceptions.
- Provide support for the versioning of objects.
- Provide support for the notification of events to interested objects. An object that triggers an event need not know the set of objects that are interested in receiving notification of the event.
- Provide support for inter-object semantic relationships by holding object references within other objects.
- Provide an application programming interface for all object management functions and capabilities with, as a minimum, a C binding.
- Require minimal administration of the environment and facilitates management of the object base.
- Support application internationalization. An application may carry out interaction with an user in different languages according to preferences of the user.
- Conformance to pertinent industry standards. The model follows standards that are broadly accepted and that cover the same area than the Object Management Standard. This includes X/Open endorsed standards, POSIX standard, and ANSI C.

3.1 The OMG Core Object Model

The OMA model defines a common object semantics for specifying the visible abstract interface to objects in a standardized way. The exact implementation of an object and its internal structure may vary across applications and environments. The common semantics characterize objects that exist in a system that follows OMG standards. Such systems perform operations and maintain state for objects. They must follow the OMG Core Object Model which defines the external characteristics of the objects within systems. The Core Object Model is on an abstract level.

3.1.1 Objects

The principal item in the Core Object Model is an object. An object is an abstraction of a real world entity, such as a person or a file. An object has an external interface which consists of operations. The operations together define the behaviour and state of the object.

Each object has a unique identity which distinguishes it from the other objects regardless of their other characteristics. In the Core Object Model, each object has an identifier that implements the unique identity of the object. The identity is stable over time and stays active as long as the object is active.

3.1.2 Types

Objects are created as instances of types. A type characterizes the behaviour of its instances by describing the operations that can be applied to the objects. Types are related to each other through supertype/subtype relationships. They are arranged into a type hierarchy that forms a directed acyclic graph. The root of this type hierarchy is the type *Object*. Applications introduce new types by subtyping from type *Object*. With a single root it is possible to define operations that accept parameters of any object type.

Objects are created as instances of their types. Types are static. An object cannot change its type after it has been created.

Subtyping is a relationship between types based on their interfaces. It defines the rules by which objects of one type are determined to be acceptable in contexts expecting another type.

The OMG model supports subtyping for object types. Intuitively one type is a subtype of another if the first is a specialization or a refinement of the second. Operationally, this means that any object of the first type can be used in any context which expects an object of the second type. If S is a subtype of T, an object of type S may be used wherever an object of type T may be used. Subtypes can have multiple parent types.

An object is a direct instance of a type if it is an instance of that type and not an instance of any subtype of that type. Objects are direct instances of only one type. That one type is the object's immediate type. An object cannot change its immediate type. For instance, in type hierarchy table → furniture → item an object that is an abstraction of a kitchen table is a direct instance of type table. The object of type table is also of type furniture and of type item.

If S is declared to be a subtype of T, then for each operation in type T there exists a corresponding operation in type S, such that the following conditions hold:

(1) The names of the operations match

(2) The number and types of the parameters are the same

(3) The number and types of the results are the same

The specifications of corresponding operations must match precisely. For instance, in the former example an object of type class has all the operations of both of type furniture and of type item. The exact definition prevents two object types that have an operation with the same name but different signatures from having a common subtype.

It is possible to have abstract types that are incomplete themselves, but define a common interface for their subtypes (that have extensions to the interface themselves.) It is also possible to have types that are complete themselves, but useful only when combined with other types in a new subtype. In both cases, the types do not have their own implementations. For instance, in the previous example type *item* may be an abstract type. Its external interface is combined with the subtypes but the type *item* itself does not have any actual objects.

The OMG model relates subtyping with inheritance. If S is declared to be a subtype of T, then S also inherits from T. It is possible to have multiple inheritance, which allows a subtype to have multiple supertypes. The OMG model does not provide a name conflict resolution mechanism nor does it allow subtypes to redefine inherited operation signatures. The definition prevents operation overloading.

3.1.3 Operations

An object is characterized by its operations. An operation to an object is a request of action. The action may change the state of the object, or several objects. For instance, operation marry that takes two person objects as parameters can change the state of operation spouse on both of them. Thus operations are also used to implement relationships between different objects.

Operations are defined in the type interface. An object that belongs to a certain type has all the operations of the type and its supertypes. The internal representation and implementation of objects and operations is not fixed in the standard. The OMG standard affects only the external representation. It is possible to have multiple implementations of a single object type and its operations.

Each operation has a signature, which consists of a name, a set of parameters, and a set of results. The set of operation signatures defined on a type is called the type's interface. A type's interface includes signatures that are inherited from supertypes.

An operation describes an action that can be applied to parameters. An operation invocation, called a request, is an event that indicates an operation and possibly lists some parameters on behalf of a requester, possibly causing results to be returned. The consequences of a request can include:

(1) An immediate set of results.

(2) Side effects that change object states.

(3) Exceptions (currently not part of the Core Object Model) that indicate that something unusual even happened. The information is passed to the exception handler.

Each operation has a signature. The signature includes the operation's name, list of parameters, and list of return values, if any. Parameters and results have both names and types.

Operations are always specified with a controlling parameter. An operation is part of the interface of its controlling parameter's type and the interface of all subtypes of that type. An operation is defined on the type of its controlling parameter. In other words, the controlling parameter defines the object type where the operation is defined. An operation is defined on a single type (the type of the controlling parameter), so the model does not recognize operations that are not bind to a specific type, or operations that are bind to several types.

When an operation request is issued, a specific operation implementation is selected for execution. The process of selection an implementation to invoke is based on the type of the object supplied as the controlling argument of the actual call. This means that the controlling parameter identifies the object which will get the operation request. The request may be sent from a different application from a remote source.

A type exports all of the operations that are defined on it. There is no notion of multiple named interfaces defined on a type that exports less than the full type specification. Visibility attributes such as private, public, subtype_visible, and friends are not supported either. For instance, object types do not have a specific private interface. Everything that is defined is also accessible.

3.1.4 Non-Objects

A Non-Object is a type or value that does not have an object identity. For instance, integers and strings are non-objects. A non-object cannot be the controlling parameter for an object request. Each non-object can be considered to belong to a type of value. They are not subtypes of Object. The rules for subtyping and inheritance do not apply to non-object types.

3.1.5 Unspecified areas

The Core Object Model does not require comparison operations to object identifiers. Thus it is not necessarily possible to tell if two references refer to the same object.

The Core Object Model does not require support for atomic operation execution or specific operation execution order. For instance, operation execution can be sequential or parallel.

The Core Object Model does not specify the syntax of operation specification. Nor does it require formal specification of the semantics of an operation.

The OMG model does not specify implementation aspects.

3.1.6 Core Object Model Summary

In summary, the OMG Core Object Model is as following:

The principal item is an object. Each object is an unique entity, that has object identity. An object is an instance of a specific type. Types form a type hierarchy that is an acyclic graph. The root of the graph is type Object. All types inherit from type Object. Types respect subtype/supertype notations. The Non-Object types do not have object identity, and they all belong to type of value.

Types are characterized through operations. An operation has an operation name, a number of parameters, and a number of results. Each operation must have at least one parameter, which is its controlling parameter. The controlling parameter is used to access the object in which the operation is performed. Operations can also be called as requests to a specific object.

Operations are the only external interface to an object. The implementations of objects and operations can vary between platforms.

A type's interface defines the externally visible state and behaviour of instance of the type. Behaviour is defined as a set of operation signatures. For instance, the following example follows the OMG rules (the syntax is not defined in the OMG standard):

```
type IMAGE
  supertypes OBJECT

  operations:
    INTEGER get_length (i:IMAGE)
    set_length (i:IMAGE, length: INTEGER)
    INTEGER get_width (i:IMAGE)
    set_width (i:IMAGE, width: INTEGER)
    scale (i:IMAGE, by:FACTOR)
    rotate (i:IMAGE, by:DEGREES)
    IMAGE crop (i:IMAGE, to:SIZE)

end_type
```

While the Core Object Model serves as the common ground, the OMG Object Model also allows for extensions to the Core to enable even greater commonality within different technology domains. The Object Model defines a concept of components, which are compatible extensions to the Core Object Model but are not required to be supported by all systems. The Object Model also defines a mechanism, called profiles, for technology domains to group pertinent components.

3.2 OMG Reference Model

The Reference Model identifies and characterizes the components, interfaces, and protocols that compose Object Management Group's Object Management Architecture . An application that is OMA-compliant consists of a set of interworking classes and instances that interact via an Object Request Broker (ORB). The common object management architecture has the following parts (figure 1):

Object Request Broker provides the mechanisms by which objects transparently make and receive requests and responses. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments. It interconnects multiple object systems. An ORB has a well defined interface to the objects and clients that send requests to the objects. It must address the following areas:

- Name Service, which covers mapping object names between different domains.

- Request Dispatch, which determines which method to invoke. The OMG object model does not require a request to be delivered to any particular object.
- Parameter Encoding, which covers parameter mapping between local representations.
- Requests and results that must be delivered to the proper location, which can be on a different node.
- Synchronization, which primarily deals with handling the parallelism of the objects.
- Activation, which is the housekeeping processing necessary before a method can be invoked.
- Exception Handling, which covers processing in failure conditions.
- Security Mechanisms, which ensure the secure conveyance of requests among objects.

The OMG standard defines a Common Object Request Broker Architecture (CORBA) that fills the above requirements. CORBA defines a primary interface for both clients and object services, covering areas such as objects, requests, object creation and destruction, types, interfaces, operations, and attributes. It also defines an Interface Definition Language (IDL) that is based on C++, a mapping from IDL to C, dynamic invocation interface that allows dynamic creating and invocation of requests to objects, and persistent storage of interface definitions.

Object services provides basic operations for the logical modeling and physical storage of objects. It defines a set of operations that all classes should implement or inherit. Objects do not have to use the implementation of basic operations provided by Object Services nor do objects have to provide all basic operations.

The operations that Object Services can provide include:

- Class Management
- Instance Management
- Storage
- Integrity
- Security
- Query
- Versions

The operations provided by Object Services are made available through the ORB, or through other interfaces.

Common Facilities comprises facilities that are useful in many application domains and which are made available through OMA-compliant object interfaces. Common Facilities are optional.

A service becomes a Common Facility when it:

- Communicates using the ORB
- Implements a facility that OMG chooses to adopt
- Has an OMA-compliant object interface

Examples of possible candidates for Common Facilities cover cataloging and browsing of classes and objects, link management, reusable user interfaces, printing and spooling, error reporting, help facility, electronic mail facility, tutorials and computer-based training, common access to remote information repositories, agent facilities, interfaces to external systems, object querying facilities, and user preferences and profiles.

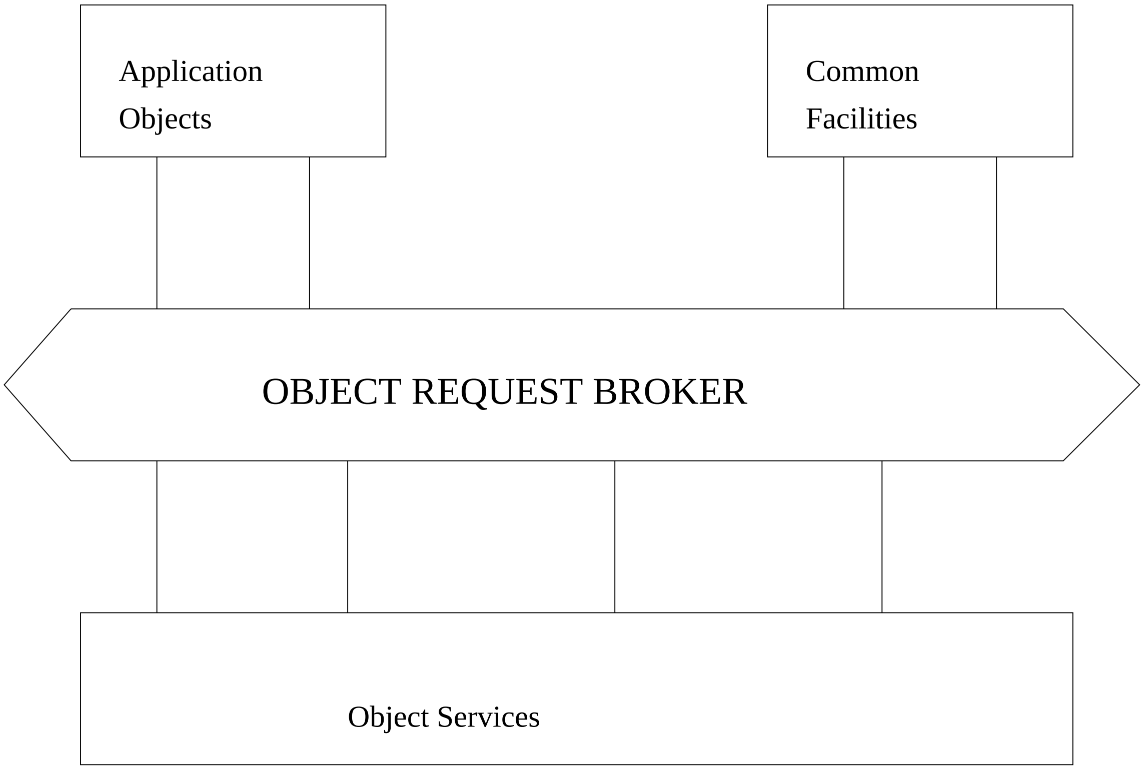

Figure 1: Object management architecture overview

Application objects is an object-oriented equivalent term to the traditional notion of an application. An OMA-compliant application consists of a collection of interworking objects. Classes and objects may be used in multiple applications. Applications communicate through the ORB.

Thus the OMG Reference Model defines a platform frame where OMG-compilant application objects may interact. The Object Request Broker is an information highway for object requests to application objects, object services, and common facilities.

4 Object Database Management Group Standard

Object Database Management Group Standard (ODMG) is a specification for object oriented databases. Its primary goal is to put forward a set of standards allowing an Object Database Management System (ODBMS) customer to write portable applications. The data schema, programming language binding, and data manipulation and query languages must be portable.

The major components of the ODMG-standard are:

- Object Model. The common data model to be supported by ODBMSs is an Object Model. The basis of the ODMG Object Model is the OMG Object Model that was summarized in Section 3.

- Object Query Language. The ODMG standard defines a declarative language for querying and updating database objects. The object query language is based on the relational standard SQL-language.
- C++ language binding. The most important programming language for ODBMSs has proven to be C++. The C++ Object Manipulation Language allows writing portable C++ code that manipulates persistent objects.
- Smalltalk language binding. The ODMG standard also defines an ODBMS binding for Smalltalk. It is currently a draft version.

4.1 Object model

The ODMG Object Model is summarized as following:

- The basic modeling primitive is the object.
- Objects can be categorized into types. All objects of a given type exhibit common behaviour and a common range of states.
- The behaviour of objects is defined by a set of operations that can be executed on an object of the type.
- The state of objects is defined by the values they carry for a set of properties. These properties may be either attributes of the object itself or relationships between the object and one or more other objects.

A type has one interface and one or more implementations. The interface defines the external interface supported by instances of the type; that is, their properties and the operations that can be invoked on them. An implementation defines data structures in terms of which instances of the type are physically represented and the methods that operate on those data structures to support the externally visible state and behaviour defined in the interface.

Types are themselves objects, and may therefore have properties. The interface definition of a type specifies values for these type properties:

- Supertypes. Object types are related in a subtype/supertype graph. All of the attributes, relationships, and operations defined on a supertype are inherited by the subtype. The subtype may add additional properties and operations to introduce behaviour or state unique to instances of the subtype. it may also refine the properties and operations it inherits to specialize them to the behaviour and range of state values appropriate for instances of the subtype.
- Extents. The set of all instances of a give type is termed its extent. The type programmer can request that the system automatically maintain a current index to the members of this set by including an extent declaration in the type definition. The maintenance of the extent is optional and need not be done for all types.
- Keys. In some cases the individual instances of a type can be uniquely identified by the value they carry for some property or set of properties. These properties or sets of properties are termed keys. A key can be either simple, when it consists of a single property, or complex, when it consists of several properties. Both relationships and attributes may serve as keys or part of keys.

The interface definition for an object type also contains instance property and instance operation declarations. Instance properties are the properties for which objects of the type carry values. Instance operations are the operations which objects of the type support.

The full built-in type hierarchy is as following:

```
Denotable_Object
        Object
                Atomic_Object
                        Type
                        Exception
                        Iterator
                Structured_Object
                        Collection <T>
                                Set <T>
                                Bag <T>
                                List <T>
                                        String
                                        Bit_String
                                Array <T>
                        Structure <e1:T1...en:Tn>
        Literal
                Atomic_Literal
                        Integer
                        Float
                        Character
                        boolean
                Structured_Literal
                        Immutable_Collection <T>
                                Immutable_Set <T>
                                Immutable_Bag <T>
                                Immutable_List <T>
                                        Immutable_String
                                        Immutable_Bit_String
                                Immutable_Array
                                Enumeration
                        Immutable_Structure <e1:T1...en:Tn>
                                Date
                                Time
                                DateTime
                                Interval
Characteristic
        Property
                Attribute
                Relationship
        Operation
```

The type hierarchy consists of denotable objects and characteristics.

- Denotable Objects. The hierarchy of object types is rooted at the type Denotable_Object. Its subtypes are Objects and Literals.
- Objects. Objects have object identity. They are either atomic or structured.
- Atomic Objects. Object types, Exception objects that allow exception handling, and Iterator objects that allow iteration over a set of objects, are atomic objects. They do not have internal structure.
- Structured Objects. All user defined objects are structured objects. They can be either collections or structures.
- Collections. Possible collections are a set of type, a bag of type, a list of type, or an array of type. A list of type has two special subclasses, string (a list of characters), and a bit_string (a list of bits.)
- Literals. A literal is not an object and it does not have identity. Literals can either be atomic or structured.
- Atomic Literals. An atomic literal cannot be divided into subliterals. Integers, floats, characters and boolean values are atomic literals.
- Structured Literals. A structured literal is similar to an structured object except that its state cannot be changed. There is a special structured literal called Enumeration. The ODMG-standard does not define its behaviour, but most probably an enumeration literal is a set of atomic literals that have an order.
- Characteristics. Characteristic is a supertype for object characteristics. Its subclasses are Property and Operation.
- Properties. The state of an object is defined by its properties. Possible properties are attributes that define the state of a single object, and relationship, that define relationship states between several objects.
- Operations. The state of an object or several objects may be modified or read with operations. Operations define the behaviour of the object.

4.1.1 Types and Instances

A type defines the state and behaviour of its instances. A state is defined as a set of properties; behaviour is defined as a set of operations.

Types may be organized into a graph of subtypes and supertypes. A subtype inherits all of the characteristics of its supertypes. It may also define additional characteristics that apply only to its instances.

Some types are directly instantiable. Others are termed as abstract types. Abstract types only define characteristics inherited by their subtypes. They do not define an implementation and therefore cannot be directly instantiated.

The built-in types at the root of the object model are organized as a tree. The user-defined types in a database schema may in general be a directed acyclic graph. A type may inherit characteristics that have the same name but different semantics from several different supertypes. This form of name clash is handled by requiring the inheriting type to redefine the name of one of the inherited characteristics.

A type has one or more implementations. In the base model it is only object types that have a user-definable implementation. An implementation of an object type consists of a representation and a set of methods. The representation is a set of data structures. The methods are procedure bodies. There is one method for each of the operations defined in the specification. These methods implement the behaviour specified for their associated operation by modifying the representation of the object and/or invoking operations defined on related objects. There may be additional methods and data structures in the implementation that have no counterpart operations in the type interface.

Implementations are named. Names of implementations are unique within the scope defined by a type. The implementations associated with a type are lexically separate in the Object Definition Language (ODL.)

A combination of the type interface specification and one of the implementations defined for the type is termed a class. This use of the term class makes a subset of the ODMG model roughly consonant with the C++ model. A C++ class has a single public part and a single private part. The private part corresponds to the implementation in ODL. The ODMG model allows multiple implementations for a given interface. Which implementation an object uses is specified at object creation time. The base model provides no facilities for dynamically changing the implementation of an object later.

4.1.2 Objects

Objects are the things of which characteristics are predicated. Thus objects have state and behaviour. Objects also have identity. Individual objects have an intrinsic identity in and of themselves.

The hierarchy of object types is rooted at the type Denotable_Object. There are two orthogonal lines along which the set of denotable objects can be decomposed:

(1) mutable versus immutable, and

(2) atomic versus structured.

The ODMG definition takes the mutable/immutable as the first distinction, and then repeating the atomic/structured distinction under both the mutable and immutable branches of the tree. The mutable/immutable distinction is captured within the type hierarchy using the more traditional terms Object and Literal. Objects are mutable; literals are immutable.

All denotable objects have identity, but the representation used to maintain that identity is typically different for objects and literals. The representation of the identity of an object is what we refer to as an object identifier.

Instances of type Object are mutable. The values of their attributes may change. The relationships in which they participate may change. The identity of the object remains invariant across these changes. An object's identity uniquely distinguishes the object from all other objects within the domain in which the object was created. In the base model the domain of an object identifier is the database in which the object exists.

Individual objects may be given names meaningful to the programmer or end-user in addition to object identifiers. Any object will have a single object identifier, but it may have more than one name. A name must refer uniquely to a single object within the scope of the definition of the name. In the base level model there is a single flat name space across the entire database.

Objects may also be identified by predicates defined on their characteristics. A predicate is applied to a collection to select the members of the collection that satisfy the predicate. A predicate is a boolean conjunction or disjunction of operations supported by the object types that appear within the predicate. A common simple case of this is selecting an object that has a particular value for one of its properties from the extent of a type. To accelerate the processing of such selection requests a type definer may declare that the values of certain properties are sufficient to uniquely identify objects within the extent of their associated type.

An object type may be defined as a subtype of one or more other object types. If object type B is declared to be a subtype of object type A, this means that any operations defined on A are also available on instances of B, any attributes defined on A are also defined on B, and any relationships defined on A are also available on instances of B.

An Object has a set of built-in properties and operations that are always accessible regardless of the Object type. For instance, a built-in property *has_name?:Boolean* has the value *TRUE* if the object has a name. A built-in operation *delete()* deletes the object.

A lifetime of a mutable object is orthogonal to its type. Three lifetimes are supported in the base model:

- in procedure,
- in process, and
- in database

The first two are common to C++, and the last one defines a persistent object form. In the base model the lifetime of an object is specified at the time the object is created and cannot be changed thereafter. Objects of longer lifetimes can validly refer to objects of shorter lifetimes only when the objects of shorter lifetimes are known to exist.

Literals are objects whose instances are immutable. Two subtypes of literal are defined by the model: atomic literals and structured literals. Numbers and characters are examples of atomic literals.

The type Structured_Literal has two subtypes: Immutable_Collection and Immutable_Structure. These types are the direct analogues of their counterpart Structured_Object types, Structure and Collection, except that they are immutable.

It is possible to define additional subtypes of type Literal. It is not possible to redefine operations defined on any of the built-in literal types.

4.1.3 Properties

An object type defines a set of properties through which users of instances of the type can interrogate and in some cases directly manipulate the state of these instances. Two kinds of property are defined in the model – attribute and relationship. Attributes are defined on one object type and take literals as their values. Relationships are defined between two object types, both of which are mutable.

Attributes are defined on a single object type. The declaration of an object type includes declarations of each of the attribute types for which an instance of the object type carries a specific value. It is possible to read and change all attributes. For instance, if object type *person* has been defined to have attributes *string name*, and *string address* and nothing else, then objects of the type person have exactly these attributes and inherited attributes from supertypes.

Attributes define abstract state. They therefore appear within the interface definition of an object type rather than in the implementation. In the base model it is not possible to define attributes of attributes, to define relationships between attributes, or to add subtype specific operations to attributes. It is possible to override the built-in operations of particular attribute types.

Relationship types are defined between mutable object types. The Base Object Model supports only binary relationships. One-to-one, one-to-many, and many-to-many relationships are supported. For instance, a student object may have a relationship *takes: Set¡course¿ inverse Course::is_taken_by* which states that a student may take several courses and each course has a reference back to the student. A course object has a relationship *is_taken_by: Set¡Student¿ inverse Student::takes* which states that a course may be taken by several students and each student has a reference back to the course. This is a many-to-many relationship.

Relationships themselves have no names. Instead, named traversal paths are defined for each direction of traversal. For instance, a student *takes* a set of courses; conversely, a course *is_taken_by* a set of students (see the previous example.) These names are declared within the interface definitions of the object types that participate in the relationship. If the type programmer does not specify an inverse path for a relationship, only one direction of traversal will be possible.

Relationships maintain referential integrity. If an object that participates in a relationship is deleted, a subsequent attempt to traverse the relationship in either direction will raise an exception.

4.1.4 Operations

The potential behaviour of instances of an object type is specified as a set of operations. Operations are always defined on a single object type. Operation names need be unique only within a single type definition. This leaves open the possibility that operations defined on different types will have the same name. These are called overloaded operations. The operation defined on the most specific type of the argument object is chosen for invocation.

An argument for an operation may be any denotable object - mutable or literal, structured or atomic.

An operation may return a denotable object or a set of such objects. There is a distinguished denotable object returned which is treated as the value of the operation itself to allow functional composition of operations. For instance, in an object screen a composed operation *home().print(”Home base”)* might move an object cursor to home position and print the text ”Home Base”.

The model assumes sequential execution of operations. It does not require support for remote, concurrent or parallel operations. However, it does not preclude an ODBMS from providing such support.

4.1.5 Exceptions

The object model supports dynamically nested exception handlers. Exceptions are themselves objects and may be organized into a subtype/supertype hierarchy. A root type Exception is provided by the ODBMS. This type includes an operation to print out a message noting that an unhandled exception of type ¡exception_type¿ has occurred and to terminate the process. For instance, referencing an nonexisting object through a relationship will cause an exception. The type programmer may have defined a special exception handler procedure for such situations, or he may have relied on the default handler.

4.1.6 Structured Objects

The type Structured_Object has two subtypes: Structure and Collection.

Structures have a fixed number of named slots each of which contains an object or a literal. The types of the elements that fill the slots may be different. Insertion of objects into slots of the structure and removal of objects from slots in the structure are both done by referring to these named slots, e.g. address.zip_code = 01907.

Collections contain an arbitrary number of elements, do not have named slots, and contain elements that are all instances of the same type. it is possible to have elements that are instances of different subtypes of the type over which the collection was defined. Insertion of elements is based on either absolute position within the collection or at a point established by a cursor. Retrieval is based on either absolute position, current cursor-relative position, or a predicate that uniquely selects an element from the collection based on the values that object carries for one or more of its properties.

Within collections, the model supports both ordered and unordered collections, where the order is defined either by the sequence in which objects are inserted or by the value of one of the properties of the objects that are the members of the collection. Collections may also allow or disallow the same object to be present in the collection more than once.

Structured objects may be freely composed. The model supports sets of structures, structures of sets, arrays of structures, etc. This allows the definition of types like Dictionary as an ordered set whose elements are structures containing two fields. a key and an object identifier.

The Object Model defines both mutable and immutable collections. Mutable collections are defined as subtypes of Structured_Object. Immutable collections are defined as subtypes of Structured_Literal.

The object model defines a standard set of built-in collection type generators that are subtypes of type `Collection<T>`:

- `Set<T>`. Sets are unordered collections of Type T that do not allow duplicates.
- `Bag<T>`. Bags are unordered collections of Type T that allow duplicates.

- `List<T>`. Lists are ordered collections of Type T that allow duplicates. The elements of a list are ordered by the order of their insertion.
- `Array<T>`. Arrays are one dimensional arrays of Type T of varying length. An initial size for the dimension is specified at array creation time. The length of the array can be either explicitly or implicitly changed after its creation. The array will be implicitly extended by assignments to array elements beyond the current end of the array.

The type `Collection<T>` cannot be directly instantiated. It is an abstract type only.

The model defines a built-in type generator *structure*. A structure is an unnamed group of elements. Each element is a (name,value) pair, where the value may be any subtype of type Denotable_Object. Since structures and collections are subtypes of Denotable_Object, a structure may contain other structures or collections as the value of one or more of its elements. Structures are literals, so they may occur as the value of attributes in an object interface declaration, for instance the following declaration is valid:

```
interface Student {
...
    address: Structure<
        street_address: Structure<
            street_number: Integer
            street_name: String>
        city: String
        state: String
        zip: String>
}
```

4.1.7 Structured Literals

The model defines both atomic literals and structured literals. There are two subtypes of structured literal: Immutable_Collection and Immutable_Structure. The built-in subtypes of Immutable_Collection mirror the built-in subtypes of Collection.

Immutable collections behave just like their mutable counterparts, except that they may not be modified.

Immutable structures may be used to capture update constraints on the value of a property. They are also often returned as the result of queries.

4.1.8 Transactions

Programs that use persistent data are organized into transactions. Transactions are units of atomicity, consistency, and integrity. Transactions happen as a whole or not at all. If a transaction commits, then all of the changes made by that transaction are permanently installed in the database and made visible to other users of the database. If the transaction aborts, then none of the changes made by it are installed in the database.

A transaction is assumed to take the database from one internally consistent state to another internally consistent state. Concurrent users of a shared database always see internally consistent versions of the database. Once a transaction has committed, the DBMS guarantees that changes made by that transaction are never lost.

The object model supports a nested transaction model, with dynamic scoping of transactions. The commit of a nested transaction is only relative to the commit of its containing parent transaction. If a nested transaction aborts, this does not cause abort of the containing transaction.

The ODMG model uses a standard lock-based approach. Locks can be acquired on particular objects.

The ODMG model supports a traditional pessimistic concurrency control model as the default, though it does not preclude systems from supporting a wider range of concurrency control policies. This default model requires acquisition of a read lock on an object before it can be read and acquisition of a write lock on an object before that object can be modified. Readers of an object do not conflict with other readers, but writers conflict with both readers and writers. If there is a conflict, the holder of the lock proceeds, and the transaction that has requested the lock waits until the holder completes. All access, creation, modification, and deletion of persistent objects must be done within a transaction.

Transactions must be explicitly created and started.

4.2 Object definition language

The Object Definition Language (ODL) is a specification language used to define the interfaces to object types that conform to the ODMG Object Model. The primary objective of the ODL is to facilitate portability of database schemas across conforming ODBMSs. ODL is not intended to be a full programming language. It is a specification language for interface signatures. ODL defines the characteristics of types, including their properties and operations. ODL does not address definition of the methods that implement the operations.

The ODL is intented to define object types that can be implemented in a variety of programming languages. It defines the definition properties such as type definitions, type characteristics, and instance properties (attributes, relationships, and operations). It is language-independent but it can be mapped to various languages, such as C+ and Smalltalk.

4.3 Object query language

The Object query language in the ODMG model is called OQL. It supports the ODMG data model. The design of the OQL is based on the following principles and assumptions:

- OQL is not computationally complete. It is a query language which provides easy access to an object database.
- OQL provides declarative access to objects.
- OQL relies on the ODMG object model.
- OQL has an abstract syntax.

- The formal semantics of OQL can easily be defined.
- OQL has one concrete syntax which is SQL-like, but it is easy to change the concrete syntax. Other concrete syntaxes are defined of merging the query language into programming languages.
- OQL provides high-level primitives to deal with sets of objects but does not restrict its attention to this collection construct. Thus, it also provides primitives to deal with structures and lists, and treats all such constructs with the same efficiency.
- OQL does not provide explicit update operators but relies on operations defined on objects for that purpose.
- OQL can be easily optimized by virtue of its declarative nature.

As a stand-alone language, OQL allows to query denotable objects starting from their names, which act as entry points into a database. A name may denote any kind of object.

As an embedded language, OQL allows to query denotable objects which are supported by the native language through expressions yielding atoms, structures, collections, and literals.

The query language supports both types of objects: mutable and literal, depending on the way the objects are constructed or selected.

A query expression may return

- a collection of objects with identity,
- an object with identity,
- a collection of literals, or
- a literal

Therefore the result of a query is an object with or without object identity: some objects are generated by the query language interpreter, and others produced from the current database.

4.4 C++ binding

The most important programming language for ODBMSs has proven to be C++. ODMG allows database programmers to use object manipulation commands that are embedded to C++ programs. This is called the C++ object manipulation language (OML.) The C++ binding also includes a version of the ODL that uses C++ syntax, a mechanism to invoke OQL, and procedures for operations on databases and transactions.

The C++ binding of ODL is expressed as a class library and an extension to the standard C++ class definition grammar. The class library provides classes and functions to implement the concepts defined in the ODMG object model. The extension consists of a single additional keyword and associated syntax that add declarative support for relationships to the C++ class declaration. The C++ OML syntax and semantics are those of standard C++ in the context of the standard class library.

ODL/OML specifies only the logical characteristics of objects and the operations used to manipulate them. It does not discuss the physical storage of objects. An additional set of constructs called physical pragmas is defined to give the programmer some direct control over physical storage issues, or at least to enable a programmer to provide hints to the storage management subsystem.

The programming language-specific bindings for ODL/OML are based on one basic principle: the programmer feels that there is one language, not two separate languages with arbitrary boundaries between them. This means that

(1) There is a single unified type system across the programming language and the database; individual instances of these common types can be persistent or transient.
(2) The programming language-specific binding for ODL/OML respects the syntax and semantics of the base programming language into which it is being inserted.
(3) The binding is structured as a small set of additions to the base programming language. it does not introduce sublanguage-specific constructions that duplicate functionality already present within the base language.
(4) Expressions in the OML compose freely with expressions from the base programming language and vice versa.

The C++ binding maps the Object Model into C++ by introducing a set of classes that can have both persistent and transient instances. These classes are distinct from the normal classes defined by the C++ language, all of whose instances are transient; that is, they do not outlive the execution of the process in which they were created.

Instances of database classes may contain occurrences of C++ built-in types or user-defined classes. However, they may refer to such classes only during execution of a transaction. At the time a transaction is committed, there must either be no such references within objects committed to persistent storage, or such references must be set to NULL by the commit operation.

5 OSI Systems Directory Recommendation

The OSI Systems Directory Recommendation (X.500) defines a directory architecture for receiving information that can be distributed across a number of distributed nodes. It is called the Directory.

The Directory is a collection of open systems which cooperate to hold a logical data base of information about a set of objects in the real world. The users of the Directory can read or modify the information. Each user is represented in accessing the Directory by a Directory User Agent (DUA) which is considered to be an application-process.

The information held in the Directory is collectively known as the Directory Information Base (DIB.) The DIB is made up of information about objects. It is composed of (directory) entries, which consist of a collection of information on one object. Each entry is made up of attributes, each with a type and one or more values. The types of attribute which are present in a particular entry are dependent on the class of object of which the entry describes.

The entries of the DIB are arranged in the form of a tree, the Directory Information Tree (DIT.) Every entry has a distinguished name, which uniquely and unambiguously identifies the entry. These properties of the distinguished name are derived from the tree structure of the information.

Some of the entries at the leaves of the tree are alias entries, while all other entries are object entries. Alias entries point to object entries, and provide the basis for alternative names for the corresponding objects.

5.1 General architecture

A Directory System Agent (DSA) is an OSI application process which is part of the Directory and whose role is to provide access to the DIB, to DUAs, and/or other DSAs. A DSA may use information stored in its local database or interact with other DSAs to carry out requests. Alternatively, the DSA may direct a requestor to another DSA which can help to carry out the request. Local databases are entirely implementation dependent.

A set of one or more DSAs and zero or more DUAs managed by a single organization may form a Directory Management Domain (DMD.) A DMD may be an Administration DMD (ADDMD), or a Private DMD (PRDMD), depending on whether or not it is being operated by a public telecommunications organization.

The DUA interacts with the Directory by communicating with one or more DSAs. A DUA need not be bound to any particular DSA. It may interact directly with various DSAs to make requests. For some administrative reasons, it may not always be possible to interact directly with the DSA which needs to carry out the request. It is also possible that the DUA can access the Directory through a single DSA. For this purpose, DSAs need to interact with each other.

The DSA is concerned with carrying out the requests of DUAs and with obtaining the information where it does not have the necessary information. It may take the responsibility to obtain the information by interacting with other DSAs on behalf of the DUA.

5.2 Directory information model

The purpose of the Directory is to hold, and provide access to, information about objects of interest which exist in some "world". An object can be anything in that world which is identifiable.

The complete set of information to which the Directory provides access is known as the Directory Information Base (DIB.) All of the pieces of information which can be read or manipulated by the operations of the Directory are considered to be included in the DIB.

An object class is an identified family of objects which share certain characteristics. Every object belongs to at least one class. An object class may be a subclass of another object class, in which case the members of the former class (the subclass) are also considered to be members of the latter (the superclass.) There may be subclasses of subclasses to arbitrary dimension.

The DIB is composed of Directory entries each containing information about a single object. For any particular object there is precisely one object entry, this being the primary collection of information in the DIB about that object. The object entry is said to represent the object. For any particular object there may, in addition to the object entry, be one or more alias entries for that object which are used to provide alternative names. Aliases permit object entries to achieve the effect of having multiple immediate superiors. Only object entries can have aliases. Thus aliases of aliases are not permitted.

Each directory entry contains an indication of the object class and the superclasses of that object class with which the entry is associated. In the case of an object entry, this indicates the classes to which the object belongs. In the case of an alias entry, this indicates that it is in fact an alias entry, and may also indicate to which subclasses of the alias object class the entry belongs.

In order to satisfy the requirements for the distribution and management of a potentially very large DIB, and to ensure that objects can be unambiguously named and their entries found, a flat structure of entries is not likely to be feasible. The hierarchical relationship can be exploited, by the arrangement of the entries into a tree, known as the Directory Information Tree (DIT.)

The component parts of the DIT have the following interpretations:

- The vertices are the entries. Object entries may be either leaf or non-leaf vertices, whereas alias entries are always leaf vertices. The root is not an entry as such, but it can be viewed as a null object entry.
- The arcs define the relationship between entries. An arc from vertex A to vertex B means that the entry A is the immediately superior entry of the entry at B, and conversely that the entry at B is an immediately subordinate entry of the entry at A. The definition of superior/subordinate entries is not connected to the definition of super/subclasses. A superior entry is not necessarily a superclass of the object in the subordinate entry. The superior entries of a particular entry are its immediate superior together with its superiors (recursively.) The subordinate entries of a particular entry are its immediate subordinates together with their subordinates (recursively.)
- The object represented by an entry is associated with the naming authority for its subordinates.
- The root represents the highest level of naming authority for the DIB

A definition of an object class consists of

- the position of the managed object class in the inheritance hierarchy
- a collection of attributes

An entry consists of a set of attributes. Each attribute provides a piece of information about, or describes a particular characteristic of, the object to which the entry corresponds.

An attribute consists of an attribute type, which identifies the class of information given by an attribute, and the corresponding attribute values, which are the particular instances of that class appearing in the entry.

Some attribute types will be internationally standardized. other attribute types will be defined by a national administrative authorities and private organizations. All attributes in an entry must be of distinct attribute types.

There are a number of attribute types which the Directory knows about and uses for its own purposes. They include:

- ObjectClass, which appears in every entry and indicates the object class and superclasses to which the object belongs
- AliasedObjectName, which appears in every alias entry and holds the distinguished name of the object which this alias entry describes.

The Directory Schema is a set of definitions and constraints concerning the structure of the DIT and the possible ways entries are named, the information that can be held in and entry, and the attributes used to represent that information. Formally, the Directory Schema comprises a set of:

- DIT Structure definitions that define the distinguished names that entries may have and the ways in which they may be related to one another through the DIT
- Object Class definitions that define the set of mandatory and optional attributes that must be present, and may be present, respectively in an entry of a given class
- Attribute Type definitions that identify the object identifier by which an attribute is known, its syntax, and whether it is permitted to have multiple values
- Attribute Syntax definitions that define for each attribute the underlying data type and matching rules.

The Directory Schema is distributed, like the DIB itself.

A DIT Structure Rule defines the permitted hierarchical relationships between entries and their permitted relative distinguished names (RDNs.) The definition of a DIT Structure Rule involves:

- identifying the subordinate and superior object classes
- identifying the attribute types which may be involved in subordinate entries' RDNs
- additional information

The definition of an object-class involves:

- optionally, assigning an object-identifier for the object-class
- indicating which classes this is to be a subclass of
- listing the mandatory attribute types that an entry of the object class must contain in addition to the mandatory attribute types of all its superclasses
- listing the optional attribute types that an entry of the object class may contain in addition to the optional attributes of all its superclasses.

There is one special object class, of which every other class is a subclass. This object class is called "Top".

Every entry contains an attribute of type ObjectClass to identify the object class and superclasses to which the entry belongs. The ClassObject attribute is managed by the Directory. It may not be modified by the user.

The Directory enforces the defined object class for every entry in the DIB. Any attempt to modify an entry that would violate the entry's object class definition fails. In particular, the Directory will prevent:

- attribute types absent from the object class definition being added to an entry of that object class
- an entry being created with one or more absent mandatory attribute types for the object class of the entry
- a mandatory attribute type for the object class of the entry being deleted

The special object class Alias is defined. Every alias entry shall have an object class which is a subclass of this class.

An attribute set is a set of attributes identified by an object identifier. The definition of an attribute set involves assigning an object identifier to the set and listing the object identifiers of the attributes and other attribute sets whose members together form the set.

The definition of an attribute type involves assigning an object identifier to the attribute type, indicating or defining the attribute syntax for the attribute type, and indicating whether an attribute of this type may have only one, or may have more than one value.

The Directory ensures that the indicated attribute syntax is used for every attribute of this type. The Directory also ensures that attributes of this type will have one and only one value in entries if attributes of this type are defined to have only one value.

The definition of an attribute syntax involves

- optionally, assigning an object identifier to the attribute syntax
- indicating the data type
- defining appropriate rules for matching a presented value with a target attribute value held in the DIB.

The Directory offers the following services:

- Service qualification
  - Service controls.
  - Security parameters
  - Filters
- Directory interrogation
  - Read. A Read operation is used to extract information from an explicitly identified entry. It may also be used to verify a distinguished name.
  - Compare. A Compare operation is used to compare a value with the values of a particular attribute type in a particular object entry.
  - List. A List operation is used to obtain a list of the immediate subordinates of an explicitly identified entry.
  - Search. A Search operation is used to search a portion of the DIT for entries of interest and to return selected information from those entries.
  - Abandon. An Abandon operation is used to abandon an operation specified in the Abandon request.
- Directory modification
  - Add entry. An AddEntry operation is used to add a leaf entry to the DIT.
  - Remove entry. A RemoveEntry operation is used to remove a leaf entry from the DIT.
  - Modify entry. A ModifyEntry operation is used to perform a series of one or more of the following modifications to a single entry:

    (1) add a new attribute

    (2) remove an attribute

    (3) add attribute values
    (4) remove attribute values
    (5) replace attribute values
    (6) modify an alias
  - Modify relative distinguished name. A ModifyRDN operation is used to change the Relative Distinguished Name of a leaf entry in the DIT.
- Other outcomes
  - Errors. The Directory does not continue to perform an operation beyond the point at which it determines that an error is to be reported.
  - Referrals. A Referral redirects the service-usr to one or more access points better equipped to carry out the requested operation.

Operations may have filter parameters. A filter parameter applies a test that is either satisfied or not by a particular entry. The filter is expressed in terms of assertions about the presence or value of certain attributes of the entry, and is satisfied if it evaluates to TRUE.

A Filter is either a Filter item, or an expression involving simpler Filters composed together using the logical operators and, or, and not.

A Filter Item asserts:

- equality, if there is a value of the attribute which is equal to that asserted
- substrings, if there is a value of the attribute in which the specified substrings appear in the given order. The substrings are not overlapping, and may be separated from the ends of the attribute value and from one another by zero or more string elements.
- greater or equal, if the relative ordering (as defined by the appropriate ordering algorithm) places the supplied value before or equal to any value of the attribute.
- less or equal, if the relative ordering places the supplied value after or equal to any value of the attribute.
- present, if such an attribute is present in the entry.
- approximate match, if there is a value of the attribute which matches that which is asserted by some locally-defined approximate matching algorithm.

The OSI Directory standard defines several object classes that may be found useful across a range of applications of the Directory. The following classes are defined:

- Top. The Top object class is the class of which every other object class is a subclass of.
- Alias. The Alias object class is the class of which all alias entries may be derived. It is not a subclass of Top so it doesn't belong to the regular object inheritance tree.
- Country. The Country object class is used to define country entries in the DIT. It is a subclass of Top.
- Locality. The Locality object class is used to define locality in the DIT. It is a subclass of Top.
- Organization. The Organization object class is used to define organization entries in the DIT. It is a subclass of Top.

- Organizational Unit. The Organizational Unit object class is used to define entries representing subdivisions or organizations. It is a subclass of Top.
- Person. The Person object class is used to define entries representing people generically. It is a subclass of Top.
- Organizational Person. The Organizational Person object class is used to define entries representing people employed by, or in some other way associated with, an organization. It is a subclass of Person.
- Organizational Role. The Organizational Role object class is used to define entries representing an organizational role. It is a subclass of Top.
- Group of Names. The Group of Names object class is used to define entries representing an unordered set of names which represent individual objects or other groups of names. It is a subclass of Top.
- Residential Person. The Residential Person object class is used to define entries representing a person in the residential environment. It is a subclass of Person.
- Application Process. The Application Process object class is used to define entries representing application processes. It is a subclass of Top.
- Application Entity. The Application Entity object class is used to define entries representing application entities. It is a subclass of Top.
- DSA. The DSA object class is used to define entries representing Directory System Agents. It is a subclass of Application Entity.
- Device. The Device object class is used to define entries representing devices. It is a subclass of Top.
- Strong Authentication User. The Strong Authentication User object class is used in defining entries for objects which participate in strong authentication. It is a subclass of Top.
- Certification Authority. The Certification Authority object class is used in defining entries for objects which act as certification authorities. It is a subclass of Top.

The following attribute sets are defined in the recommendation:

- Telecommunication Attribute Set
- Postal Attribute Set
- Locale Attribute Set
- Organizational Attribute Set

The following attribute types are defined in the recommendation

- Object Class
- Aliased Object Name
- Knowledge Information
- Labeling Attribute Types
- Common Name

- Surname
- Serial Number
- Geographical Attribute Types
- Country Name
- Locality Name
- State or Province Name
- Street Address
- Organization Name
- Organizational Unit Name
- Title
- Description
- Search Guide
- Business Category
- Postal Address
- Postal Code
- Post Office Box
- Physical Delivery Office Name
- Telecommunications Addressing Attribute Types
- Telephone Number
- Telex Number
- Teletex Terminal Identifier
- Facsimile Telephone Number
- X.121 Address
- International ISDN Number
- Registered Address
- Destination Indicator
- Preference Attribute Types
- Preferred Delivery Method
- Presentation Address
- Supported Application Context
- Member
- Owner
- Role Occupant
- See Also
- User Password

- User Certificate
- CA Certificate
- Authority Revocation List
- Certificate Revocation List
- Cross Certificate Pair

6 OSI Systems Management Recommendation

The basic reference model for Open Systems defines how distributed nodes can interact with each other. The OSI Systems Management Recommendation (X.700) defines how the Open Systems network can be managed.

Management is related to activities which control or monitor the use of resources. Within open systems the resources can be those which provide data storage or processing capabilities, or they can be those which provide interconnection capabilities. The latter is dealt with OSI systems management.

OSI management supports the users' needs for

- Activities which enable managers to plan, organize, supervise, control and account for the use of interconnection services
- the ability to respond to changing requirements
- facilities to ensure predictable communications behavior
- facilities which provide for information protection and for the authentication of sources of, and destinations for, transmitted data

6.1 General architecture

The OSI management environment is the area of OSI environment which works with the tools and services needed to monitor, control and coordinate interconnection activities.

A managed object is the OSI management view of a resource that is subject to management, such as a layer entity, a connection, or an item of physical communications equipment. A managed object is an abstracted view of a resource that represents its properties as seen by management.

A managed object is defined in terms of attributes it possesses, operations that may be performed on it, notifications that it may issue and its relationships with other managed objects. The set of managed objects within a system constitutes that system's management information base (MIB).

OSI management can be divided into the following functional areas:

- fault management, which concentrates on fault detection, isolation and the correction of abnormal operation of the environment
- accounting management, which enables charges to be established for the use of resources in the OSI environment, and for costs to be identified for the use of the resources

- configuration management, which identifies, exercises control over, collects data from and provides data to open systems for the purpose of preparing for, initializing, starting, providing for the continuous operation of, and terminating interconnection services.
- performance management, which enables the behavior of resources in the OSI environment and the effectiveness of communication activities to be evaluated.
- security management, which supports the application of security policies.

Management is effected through a set of management processes. The processes may be distributed over a number of systems. OSI management is accomplished through

- systems management, which provides mechanisms for the monitoring, control and coordination of managed objects through the use of application-layer systems management protocols.
- (N)-layer management, which is like systems management, but works on layer N.
- (N)-layer operation, which provides mechanisms for the monitoring and control of a single instance of communication.

6.2 Management information model

The purpose of the information model is to give structure to the management information and to model management aspects of the related resources. The information model deals with managed objects that are abstractions of data processing and data communications resources for the purposes of management. The precise properties abstracted and the specific effects of management operations on a resource must be specified as part of the managed object class specification.

The managed object as visible to management is an encapsulated abstraction of the actual resource. The attributes, operations and notifications that are needed for management are visible at the managed object boundary, while the internal structure of the resource remains hidden to the management system.

A managed object class is defined as a collection of packages. Each package is a collection of attributes, operations, notifications and related behavior. Packages are either mandatory or conditional upon some explicitly stated condition. A managed object is an instance of a managed object class.

The definition of a managed object class consists of

- the position of the managed object class in the inheritance hierarchy
- a collection of mandatory packages of attributes, operations, notifications and behavior.
- a collection of conditional packages of attributes, operations, notifications and behavior, together with the condition under which each package will be present.

A package structure consists of

- the attributes visible at the managed object boundary.
- the operations which can be applied to the managed object
- the behavior exhibited by the managed object

- the notifications which can be emitted by the managed object.

New managed object classes and functions will be added as needs are identified. The information model makes use of object-oriented design principles. In the information model object-oriented design is applied to the specification of management information. It need not be applied to system implementation.

Encapsulation ensures that the integrity of an object is preserved. This requires that all operations to be performed are accomplished by sending a message to the object. The internal operation of a managed object is not visible at the object boundary unless attributes, operations, or notifications are defined to expose this information. The definition of the managed object class specifies what operations can be performed and what consistency constraints are required to maintain the integrity of the managed object.

A package is a collection of characteristics which is an integral module of a managed object class definition. Packages are specified as either mandatory or conditional when referenced in a managed object definition. Packages have the following properties

- only one instance of a given package can exist in a managed object
- since only one instance of a package can exist in any managed object, no name bindings are assigned to packages
- once encapsulated in a managed object, the attributes, operations, notifications and behavior become an integral part of the managed object and are accessible only as as a part of that managed object.
- a package can never be instantiated without the managed object that encapsulates it
- a package must be instantiated at the same time as the managed object
- packages must be deleted at the same time as the managed object
- operations are always performed on managed objects

Managed objects have attributes. An attribute has an associated value, which can consist of a set or sequence of elements. A set-valued attribute is an attribute whose value is an unordered set of members of a given type. A sequence-valued attribute has an order. A value of a set-valued attribute can be an empty set.

The value of an attribute may or may not be observable. It can determine or reflect the behavior of the managed object. It is observed or modified by sending a request to a managed object to read (get) or write (replace) the value. Additional operations are defined for set-valued attributes.

Within a managed object class specification the properties of an attribute are further defined in terms of the permitted value set and required value set. The permitted value set and required value set specify value restrictions on the attribute.

An attribute group provides a means of referring to a collection of attributes within a managed object that contains the collection of attributes. Two types of attribute groups may be defined: fixed and extensible. A fixed attribute group is an attribute group whose set of attributes is defined as part of the initial attribute group definition and whose set of attributes may not be changed in any way. An extensible attribute group is an attribute group to which attributes can be added as a result of specialization. For extensible attribute groups the attributes specified for each extension shall either be defined in the same conditional package as the attribute group or in a mandatory package. An attribute group does not have a value. The only operations that are allowed on attribute groups are those which do not require a value to be specified.

A managed object class can have several attribute groups, and a single attribute can be included to several attribute groups.

Managed objects may be defined to emit notifications when some internal or external event occurs. Notifications are specific to the managed objects that emit them. The notifications and the information they contain are part of the definition of the managed object class of which the managed object is an instance.

A managed object class members have a certain behavior. The behavior can define

- the semantics of the attributes, operations, and notifications
- the response to management operations being invoked on the managed object
- the circumstances under which notifications will be emitted
- dependencies between values of particular attributes, which must be expressed in a way that takes account of the possible presence or absence of conditional packages
- the effects of relationships on the participating managed objects
- consistency constraints on attributes
- preconditions that identify the conditions when operations and notifications can be assumed to have valid meaning
- postconditions that identify the results of the processing of a management operation or the emission of a notification
- invariants that are in effect for the entire lifetime of the managed object and that describe conditions that are true for operation of the managed object
- synchronization properties of the managed object

One managed object class is specialized from another managed object class by defining it as an extension of the other managed object class. Such an extension is made by defining further packages that include one or more of the following:

- new management operations
- new attributes
- new notifications
- new behavior
- extensions to the characteristics of the original managed object class

A managed object class that is specialized from another managed object class is a subclass of that class (its superclass). An object class called Top is the ultimate superclass in the class hierarchy. All other classes are subclasses of Top. Top is an uninstantiable managed object class.

Multiple inheritance is allowed for subclasses. Thus a subclass can have several superclasses. The subclass inherits the operations, attributes, notifications, packages and behavior from all superclasses. When a class has multiply inherited the same characteristic from multiple superclasses, then that class is defined as though that characteristic were inherited from only a single superclass.

Two kinds of management operations are defined: those which can be sent to a managed object to be applied to its attributes, and those which apply to the managed object as a whole.

The following management operations can be sent to a managed object to be applied to its attributes:

- Get attribute value. Read the list of attribute values requested, or if a list is not supplied, read all attribute values, and return the values of the attributes that could be read and indicate an error for the attribute values that could not be read.
- Replace attribute value. Replace the values of specified attributes with the supplied values. The replacement is exact, except where the attribute definition explicitly states otherwise. For instance, such a situation may occur in rounding of a floating point number.
- Replace-with-default value. Replace the values of specified attributes with default values.
- Add member. For each set-valued attribute specified, replace the attribute's value with an union of the old set members and the new set members.

The following management operations apply to managed objects as a whole:

- Create. The operation requests the creation and initialization of a new managed object. The operation might have side effects to the resource that the new object represents.
- Delete. The operation requests the managed object to delete itself. The operation might have side effects to the resource that the deleted object represented.
- Action. The Action operation requests the managed object to perform the specified action and to indicate the result of that action.

Operations may have filter parameters. A filter parameter is used to determine whether or not an operation should be performed on a managed object. A filter parameter applies a test that is either satisfied or not by a particular managed object. A filter condition returns TRUE if the condition holds, and FALSE otherwise.

Filters can be combined together with logical operators and, or, and not. An operation is performed only if the total result of all filter conditions combined with logical operators is TRUE.

Eight different matching rules can be used in attribute value condition tests:

- equality
- greater or equal
- less or equal
- present
- subset of
- superset of

- non-null set intersection

It's possible to register items either globally or locally. A registered item can then be used freely without the risk of having another item with the same name but different characteristics. At least the following items can be registered for later use:

- Classes
- Packages
- Attribute types

The OSI management standard defines several object classes that are used in management functions, or can be used as superclasses for other objects. The following classes are defined:

- Top. The top managed object class is the class of which every other managed object class is a subclass of.
- Discriminator. The discriminator managed object class is used to define the criteria for controlling management services. It is derived from top.
- Event Forwarding Discriminator. The eventForwardingDescriminator managed object class is used to define the conditions that shall be satisfied by potential event reports before the event report is forwarded to a particular destination. It is derived from discriminator.
- Log. The log management object class is used to define the criteria for controlling the logging of the information in an open system. It is derived from top.
- Log record. The logRecord managed object class is used to define the records contained in a log managed object. It is derived from top.
- Event log record. The eventLogRecord managed object class is used to define the information stored in the log as a result of receiving notifications or event reports. It is derived from logRecord.
- Alarm record. The alarmRecord managed object class is used to define the information stored in the log as a result of receiving alarm notifications or alarm reports. It is derived from eventLogRecord.
- Attribute value change record. The attributeValueChangeRecord managed object class is used to define the information stored in the log as a result of receiving attribute value change notifications or attribute value change event reports. It is derived from eventLogRecord.
- Object creation record. The objectCreationRecord managed object class is used to define the information stored in the log as a result of receiving object creation notifications or Object Creation event reports. It is derived from eventLogRecord.
- Object deletion record. The objectDeletionRecord managed object class is used to define the information stored in the log as a result of receiving object deletion notifications or Object deletion event reports. It is derived from eventLogRecord.
- Relationship change record. The relationshipChangeRecord managed object class is used to define the information stored in the log as a result of receiving relationship change notifications or relationship change event reports. It is derived from eventLogRecord.

- Security alarm report record. The securityAlarmreportRecord managed object class is used to define the information stored in the log as a result of receiving security alarm notifications or security alarm event reports. It is derived from eventLogRecord.
- State change record. The stateChangeRecord managed object class is used to define the information stored in the log as a result of receiving state change notifications or state change event reports. It is derived from the eventLogRecord.

The following packages are defined in the recommendation:

- Additional information
- Additional text
- Attribute identifier list
- Attribute list
- Availability status
- Correlated notifications
- Notification identifier
- Daily scheduling
- Duration
- External scheduler
- Source indicator
- Weekly scheduling

The following attribute types are defined in the recommendation

- Threshold
- Discriminator Id
- Log Id
- Log record Id
- System Id
- System title
- Counter
- Counter-threshold
- Gauge
- Gauge-threshold
- Tide-mark
- Additional information
- Additional text
- Attribute identifier list
- Attribute list

- Attribute value change definition
- Backed-up status
- Correlated notifications
- Event time
- Event type
- Monitored attributes
- Notification identifier
- Preceived severity
- Probable cause
- Proposed repair actions
- Relationship change definition
- Security alarm cause
- Security alarm detector
- Security alarm severity
- Service provider
- Service user
- Source indicator
- Specific problems
- State change definition
- Threshold info
- Trend indication
- Administrative state
- Alarm status
- Availability status
- Control status
- Operational state
- Procedural status
- Standby status
- Unknown status
- Usage state
- State
- Back-up object
- Backed-up object
- Member
- Owner

- Peer
- Primary
- Provider object
- Relationships
- Secondary
- User object
- Active destination
- Allomorphs
- Back-up destination list
- Capacity alarm threshold
- Confirmed mode
- Current log size
- Destination
- Discriminator construct
- Intervals of day
- Log full action
- Logging time
- Managed object class
- Managed object instance
- Max log size
- Name binding
- Number of records
- Object class
- Packages
- Scheduler name
- Start time
- Stop time
- Supported features
- Week mask

The following notifications are defined in the recommendation:

- Attribute value change
- Communications alarm
- Environment alarm
- Equipment alarm
- Integrity violation

- Object creation
- Object deletion
- Operational violation
- Physical violation
- Processing error alarm
- Qualify of service alarm
- Relationship change
- Security service or mechanism violation
- State change
- Time domain violation

The following functions are defined in the recommendation:

- Object management function. The object management function fills the following requirements:
  - The reporting of creation and deletion of managed object
  - The reporting of changes to attribute values of managed object

  Additionally it defines the following pass-through services:
  - The creation and deletion of managed objects
  - Performing actions upon managed objects
  - Attribute changing
  - Attribute reading
  - Event reporting
- State management function. The state management function fills the following requirements:
  - ability to examine and be notified of changes in state
  - ability to monitor overall operability and usage of resources in a consistent manner
  - ability to control the general availability of specific resources
  - ability get reports of changes in state attributes
  - ability to read state attributes
  - ability to change state attributes
- Relationship management function. The relationship management function fills the following requirements:
  - ability to examine the relationships among various parts of the system(s) to see how the operation of one part of the system depends upon or is depended upon by other parts
  - ability to locate other parts that are related to a known part
  - ability to change relationships between parts and to be notified of such changes when they occur due to other causes
- Alarm reporting function. The alarm reporting function fills the following requirements:
  - ability to get reports of alarms, errors and related information, in a standard fashion
- Event report management function. The event report management function fills the following requirements:

  - The definition of a flexible event report control service which will allow systems to select which event reports are to be sent to particular managing systems
  - The specification of the destinations to which event reports are to be sent
  - The specification of a mechanism to control the forwarding of event reports
  - The ability for an external managing system to modify the conditions used in reporting of events
  - The ability to designate a backup location to which event reports can be sent if the primary location is not available.

- Log control function. The log control function fills the following requirements:
  - The definition of a flexible log control service which will allow selection of records that are to be logged by a management system in a particular log
  - The ability for an external system to modify the criteria used in logging records
  - The ability for an external system to determine whether the logging characteristics were modified or whether log record have been lost
  - Specification of a mechanism to control the time during which logging occurs
  - The ability for an external system to retrieve and delete log records
  - the ability for an external system to create and delete logs
- Security alarm reporting function. The security alarm reporting function fills the following requirements:
  - alerting users whenever an event indicating an attack or potential attack on system security has been detected
- Security audit trail function. The security audit trail function fills the following requirements:
  - ability to record in a security audit trail log of security-related events that occur in the management domain.
  - ability to control the operation of the security audit trail function

7 Object Model Comparison

In the previous four sections we introduced four different object models: OMG, ODMG, X.500, and X.700. The models are meant for different tasks, starting from a very broad OMG model and ending to very specific X.500 and X.700 models. Thus it is imminent that the models have a lot of differences, but also a lot of similarities.

The following table summarizes the similarities and differences between the referenced object models in a property level. A code `X` means that the model supports the certain property. A code `(X)` means that the property is possible to easily implement with the model or the model supports something that is closely related to the property.

	OMG	ODMG	X.500	X.700
Objects	X	X	X	X
Object Identity	X	X	X	X
Inheritance	X	X	X	X
Multiple Inheritance	(X)	X		
Attributes		X	X	X
Attribute Sets		(X)	X	X
Attribute Groups		(X)	X	X
Operations	X	X		X
Private Part		X	(X)	(X)
Public Part	X	X	X	X
Relationships	(X)	X		
Type hierarchy	X	X	X	X
Exceptions		X	X	X
Non-objects	X	X		
Atomicity		X		
Transactions		X		
Language Binding	(X)	X		
Extents		X		
Keys		X		
Literals		X		
Transient Objects	(X)	X		
Predefined attribute structures		X	X	X
Alias-entries		(X)	X	
Hierarchical Object Classes		(X)	X	
Naming Standard			X	
Multivalued Attributes			X	X
Optional Attributes			X	X
Predefined Object Classes			X	X
Predefined Attribute Types			X	X
Predefined Attribute Sets			X	X
Predefined Exceptions				X
General Object Management Operations			X	X

1) Objects All models share the same view of an Object. An Object is a distinguishable entity that has its own identity which is not dependable on the object attributes.

2) Object Identity All models have object identities to their objects. OMG, ODMG, and X.700 only tell that object identity is unique to a single object. X.500 defines an explicit object identity attribute that must be present on every object.

3) Inheritance All models allow objects to have supertypes and subtypes. X.500 and X.700 define an abstract supertype Top, of which all other types are subtypes. OMG calls the abstract supertype Object. ODMG doesn't mention an abstract supertype, but due to its principles to follow OMG, it must also have an abstract supertype Object.

4) Multiple Inheritance Only ODMG allows multiple inheritance on all cases. OMG allows multiple inheritance in all cases, except when two supertypes have an equal operation. X.500 and X.700 do not mention multiple inheritance, but X.500 does not allow multiple similar attribute sets, and X.700 does not allow multiple similar packages. These rules restrict multiple inheritance from two subtypes that have common packages or attribute sets.

5) Attributes All models except OMG define attributes. OMG model defines only operations, but attributes can be present in the internal structure of an object. Also an operation pair get/set can be considered an attribute.

6) Named Attribute Sets X.500 defines named attribute sets. X.700 calls them named attribute groups. The similar concept can be defined on ODMG with structures. OMG does not define the exact non-object type hierarchy in the model.

7) Named Attribute Groups Named attribute groups are a similar concept than Named Attribute Sets, but they are defined on X.700.

8) Operations OMG, ODMG, and X.700 define Operations that can be applied to objects. X.500 does not define operations due to its directory nature. All models allow operations performed on objects. X.500 has a stable operation set. OMG and ODMG define operations through object interfaces. X.700 has both a stable operation set and operations through object interfaces.

9) Private Part ODMG defines C++ style private attributes and operations. OMG states that all visible operations are public. However, the private part can be considered implementation-dependent, and OMG doesn't set any rules on that area. X.500 and X.700 allow implicit private parts, because the models see only the attributes and operations of an object that are of interest to them.

10) Public Part All models define public interface. OMG, X.500, and X.700 define that everything visible is public. ODMG allows also private part that is visible on definition stage.

11) Relationships ODMG defines relationships between objects. OMG defines a similar concept in terms of object state, which can be modified via operation side effects. X.500 and X.700 don't define relationships.

12) Type hierarchy All models define type hierarchy. Objects are instances of types.

13) Exceptions ODMG, X.500, and X.700 define exceptions, though all models call them with different names. Exceptions are currently not part of the OMG Core Object Model.

14) Non-objects OMG and ODMG define non-objects (e.g. regular types) that are not part of the object type hierarchy. X.500 and X.700 use only objects.

15) Atomicity ODMG defines atomicity of operations. OMG does not guarantee that operations are atomic. X.500 and X.700 do not mention atomicy.

16) Transactions ODMG defines transactions that use persistent objects. OMG, X.500, and X.700 do not use transactions.

17) Language Binding ODMG has language binding definitions for C++ and Smalltalk. OMG defines language binding through Object Request Brokers that allow different object applications to communicate with each others. X.500 and X.700 have specified communication protocols.

18) Extents ODMG defines extents that are additional information to object types. OMG, X.500, and X.700 do not use them.

19) Keys ODMG defines keys that are additional identification information to object types. OMG, X.500, and X.700 do not use keys.

20) Literals OMG and ODMG define literals. X.500, and X.700 use only objects.

21) Transient Objects ODMG defines transient (non-persistent) objects. OMG does not define whether objects are transient or persistent. X.500, and X.700 use only persistent objects.

22) Predefined attribute structures OMG does not define any attribute structures. X.500, X.700, and ODMG define special attribute structures. ODMG defines only a small subtype of them. X.500 and X.700 use a very rich type system that is related to telecommunications.

23) Alias-entries X.500 uses alias-entries that are actually references to real objects elsewhere. ODMG allows alias-entries through relationships, but then the alias-entry is an object itself.

24) Hierarchical Object Classes X.500 and X.700 consider objects in a hierarchical structure. The structure is related to naming, where an object can have a local name, or a global name that is based on the total hierarchical structure. OMG and ODMG use local naming only. ODMG can be used to define hierarchical object classes, but the model does not require them.

25) Naming Standard X.500 and X.700 have a common naming standard that is related to other ITU-T recommendation standards (such as X.400). OMG and ODMG use only local naming standards.

26) Multivalued Attributes X.500 and X.700 allow multivalued attributes. OMG does not define attributes at all. ODMG allows multivalued attributes via structures and sets, but the mapping is not exactly the same than in X.500 and X.700.

27) Optional Attributes X.500 and X.700 allow optional attributes in objects. OMG does not define attributes. ODMG does not allow objects to change type definitions, so objects cannot have optional attributes.

28) Predefined Object Classes X.500 and X.700 define a group of predefined classes that are useful in telecommunications. OMG and ODMG do not define any type instances, only the actual type information.

29) Predefined Attribute Types X.500 and X.700 define a group of predefined attribute types that are useful in telecommunications. OMG does not define attributes. ODMG defines a built-in attribute type hierarchy, but does not define predefined attribute types.

30) Predefined Attribute Sets X.500 and X.700 define a group of predefined attribute sets (X.500) and predefined packages (X.700) that are useful in telecommunications. OMG does not define attributes. ODMG does not define predefined structures.

31) Predefined Exceptions X.700 defines a group of predefined exceptions. X.500 defines a group of errors. OMG and ODMG do not define any exceptions.

32) General Object Management Operations X.500 and X.700 define a group of operations that can be used on all objects. OMG does not define operations, only operation semantics. ODMG defines object management operations via default object operations that are common to all objects, but are activated via the object interface.

8 Summary

In this report we have summarized four different object standards or recommendations: OMG standard, ODMG standard, X.500 Directory recommendation, and X.700 Management recommendation. All of the standards and recommendations define an object model that suits their needs.

The defined object models are meant for quite different tasks. OMG object model is defined for general interoperability between different applications, different platforms, and different implementations. Thus it is in a very general level. ODMG object model is defined for object database management aspects, and thus it is in a more detailed level than OMG. Yet ODMG is a true expansion to the OMG standard which it follows completely. X.500 Directory model is meant for accessing objects that may be distributed across a wide area and a number of independent nodes. It's a very specific model with a lot of predefined object classes and attribute types. X.700 Management model is similar to X.500, except that it concentrates on managing objects across a distributed environment. Like X.500, X.700 is a very specific object model. It's imminent that a distributed object oriented database needs the functionality of all the models.

OMG defines the frames that summarize the de facto standards in object oriented design. It also allows different object oriented applications to communicate with each other via well defined object request brokers. However, it does not specify relationships between objects, transactions, attributes, or non-object type hierarchy. It doesn't restrict access to the areas either, but we think it is probable that there won't be a pure OMG based database implementation in the near future that would fulfill all questions. After all, OMG is not an object database standard.

The ODMG object model, on the other hand, is designed to fulfill object database model requirements. As the model follows the OMG standard it also allows the same functionality than OMG.

It also defines extra functionality that is necessary for an object-oriented database. In ODMG, well defined data description and data manipulation languages, and a well formed C++ bindings, allow great expandability. Altogether the ODMG standard seems to offer an working solution to object oriented database design. However, it also means that building a prototype database management system that is based on ODMG standard is not a simple task. A big advantage in ODMG is that it is probable that there will be commercial ODMG-based object-oriented database management systems available in the near future.

X.500 Directory recommendation defines the object model from the point of view of a distributed directory service. In principle the X.500 model has the necessary operations to maintain an object oriented database. However, the lack of data definition and data manipulation languages and the pure protocol oriented nature don't make X.500 alone a good choice for human interactions or for expandability. Also the object model in X.500 is not as general and complete than in OMG and ODMG. X.500 has already been implemented several times, but there are no database architectures that are based purely on the X.500 recommendation.

X.700 Management recommendation defines the object model from the point of view of a management system. A distributed management system is actually quite close to a database management system, but the same drawbacks than in X.500 hold on X.700 as well. X.700 doesn't define any other interface than a pure protocol oriented one. Unlike X.500, it does allow objects to have their own operations, but it doesn't define what they are. Also, X.700 states that the local databases where the management system data resides are independent of the standard itself. Thus, although X.700 model is quite finished, it still doesn't answer the demands of an object oriented database system.

As a conclusion, it looks like that the best alternative for an implementation model is ODMG. The object model is quite finished and it alone can be used to define and implement most of X.500 and X.700, and all of OMG. The mapping from ODMG to the other models should therefore not be too complex. A C++ binding can be used to finish the X.500 and X.700 implementations on the database.

References

[Catt94] Cattel R. G. G., Object Database Standard: ODMG – 93. Morgan Kaufman Publishers, 1994

[CCIT89] CCITT, Recommendations X.500 to X.521 – Data Communication Networks: Directory. pp. 1 – 225 in CCITT Blue Book Volume VIII – Fascicle VIII.8. CCITT, 1989

[CCIT91a] Data Communication Networks: Open Systems Interconnection (OSI); Management – Common Management Information Service Definition for CCITT Applications. Recommendation X.710, CCITT, 1991

[CCIT91b] Data Communication Networks: Open Systems Interconnection (OSI); Management – Common Management Information Protocol Specification for CCITT Applications. Recommendation X.711, CCITT, 1991

[CCIT92a] Data Communication Networks – Management Framework for Open Systems Interconnection (OSI) for CCITT Applications. Recommendation X.700, CCITT, 1992

[CCIT92b] Data Communication Networks – Information Technology - Open Systems Interconnection - Systems Management Overview. Recommendation X.701, CCITT, 1992

[CCIT92c] Data Communication Networks – Information Technology - Open Systems Interconnection - Common Management Information Protocol: Protocol Implementation Conformance Statement Proforma. Recommendation X.712, CCITT, 1992

[CCIT92d] Data Communication Networks – Information Technology - Open Systems Interconnection - Structure of Management Information: Definition of Management Information. Recommendation X.721, CCITT, 1992

[CCIT92e] Data Communication Networks – Information Technology - Open Systems Interconnection - Structure of Management Information: Guidelines for the Definition of Managed Objects. Recommendation X.722, CCITT, 1992

[CCIT93] Data Communication Networks – Information Technology - Open Systems Interconnection - Management Information Model. Recommendation X.720, CCITT, 1993

[OMG91] OMG, The Common Object Request Broker: Architecture and Specification. Object Management Group, 1991

[OMG92] OMG, Object Management Architecture Guide – Revision 2.0 - Second Edition. Object Management Group, 1992