Chapter 2

Object Model

This is not a standard. It is a work-in-progress draft of ODMG's Object Model which incorporates changes which have been approved by the ODMG Board for version 2.0. Additionally, we are working on a proposal to resolve the ambiguities of IDL and ODL interfaces with respect to representation of object state. This draft will expire in January 1997 at which time it will be replaced with an updated version.

2.1 Overview

This chapter defines the Object Model supported by ODMG-compliant object database management systems. The Object Model is important because it specifies the kinds of semantics that can be defined explicitly to an ODBMS. Among other things, the semantics of the Object Model determine the characteristics of objects, how objects can be related to each other, and how objects can be named and identified.

Chapter 3 defines a programming-language independent Object Definition Language (ODL), to be used to specify application object models. ODL syntax is presented for all of the constructs explained in this chapter for the Object Model. It is also used in this chapter to define the operations on the various objects of the object model. Chapters 5 and 6, respectively, define the C++ and Smalltalk programming language bindings for ODL and for manipulating objects. Programming languages have some inherent semantic differences; these are reflected in the ODL bindings. Thus some of the constructs that appear here as part of the Object Model may be modified slightly by the binding to a particular programming language. These differences are explained in the chapter about the particular binding.

The Object Model specifies the constructs that are supported by an ODBMS:

- The basic modeling primitives are the *object* and the *literal*. Each object has a unique identifier. A literal has no identifier.
- The state of an object is defined by the values it carries for a set of *properties*. These properties can be *attributes* of the object itself or *relationships* between the object and one or more other objects. Typically the values of an object's properties can change over time.
- The behavior of an object is defined by the set of *operations* that can be executed on or by the object. For example, a Document object includes a format operation.
- Objects and literals can be categorized by their *types*. All elements of a given type have a common range of states (i.e., the same set of properties) and

common behavior (i.e., the same set of defined operations). An object is sometimes referred to as an *instance* of its type.

- A *database* stores objects, enabling them to be shared by multiple users and applications. A database is based on a *schema* that is defined in ODL and contains instances of the types defined by its schema.

The ODMG Object Model specifies what is meant by objects, literals, types, operations, properties, attributes, relationships, and so forth. An application developer uses the constructs of the ODMG Object Model to construct the object model for the application. The application's object model specifies particular types, such as Document, Author, Publisher, and Chapter, and the operations and properties of each of these types. The application's object model is the database's (logical) schema.

Analogous to the ODMG Object Model for object databases is the relational model for relational databases, as embodied in SQL. The relational model is the fundamental definition of a relational database management system's functionality. The ODMG Object Model is the fundamental definition of an ODBMS's functionality. The ODMG Object Model includes significantly richer semantics than does the relational model, by declaring relationships and operations explicitly.

2.2 Types and Classes; Interfaces and Implementations

There are two aspects to the definition of a type. A type has an *interface* specification and one or more *implementation* specifications. The interface defines the external characteristics of the objects of the type. These external characteristics are the objects' aspects that are visible to users of the objects. These are the operations that can be invoked on the objects and the state variables whose values can be accessed. By contrast, a type's implementation defines the internal aspects of the objects of the type.

An implementation of a type consists of a *representation* and a set of *methods*. The representation is a data structure. The methods are procedure bodies. There is a method for each of the operations defined in the type's interface specification. A method implements the externally visible behavior of its associated operation. A method might read or modify the representation of an object's state or invoke operations defined on other objects. There can also be methods and data structures in an implementation that have no direct counterpart operations or state variables in the type interface. The internals of an implementation are not visible to the users of the objects.

The distinction between interface and implementation is important. The separation between these two is the way that the Object Model reflects encapsulation. The ODL of Chapter 3 is used to specify the interfaces of types in application object models. Chapters 5 and 6, respectively, define the C++ and Smalltalk constructs used to specify the implementations of these types.

A type can have more than one implementation specification, although only one implementation can be used in any particular program. For example, a type could have one C++ implementation and another Smalltalk implementation. Or a type could have one C++ implementation for one machine architecture, and another C++ implementation for a different machine architecture. Separating the interface from the implementations keeps the semantics of the type from becoming tangled with representation details. Separating the interface from the implementations is a positive step toward multi-lingual access to objects of a single type and sharing of objects across heterogeneous computing environments.

We sometimes loosely refer to an interface by itself as a type, and to an implementation of a type as a *class*. An object is an instance of a class. A class specification, then, is used to implement all instances of the type. For example, a C++ class specification is used by both a C++ compiler and an ODBMS to create instances (objects) of the type.

2.2.1 Subtyping and Inheritance

Like many object models, the ODMG Object Model includes inheritance-based type-subtype relationships. These relationships are commonly represented in graphs; each node is a type and each arc connects one type, called the *supertype*, and another type, called the *subtype*. The type/subtype relationship is sometimes called an *is-a* relationship, or simply an *ISA* relationship. It is also sometimes called a *generalization-specialization* relationship. The supertype is the more general type; the subtype is the more specialized.

```
interface Employee {...};
interface Professor : Employee {...};
interface Associate_Professor : Professor {...};
```

For example, Associate_Professor is a subtype of Professor; Professor is a subtype of Employee. An instance of the subtype is also logically an instance of the supertype. Thus an Associate_Professor instance is also logically a Professor instance. That is, Associate_Professor is a special case of Professor.

An object's *most specific type* is the type that describes all the behavior and properties of the instance. For example, the most specific type for an Associate_Professor object is the Associate_Professor interface; that object also carries type information from the Professor and Employee interfaces. An Associate_Professor instance conforms to all the behaviors defined in the Associate_Professor interface, the Professor interface, and any supertypes of the Professor interface (and their supertypes, ...). Where an object of type Professor can be used, an object of type Associate_Professor can be used instead, because Associate_Professor inherits from Professor.

A subtype's interface may define characteristics in addition to those defined on its supertypes. These new aspects of state or behavior apply only to instances of the subtype (and any of its subtypes). A subtype's interface also can be refined to specialize state and behavior. For example, the Employee type might have an operation for calculate_ paycheck. The Salaried_Employee and Hourly_Employee subtypes might each refine that behavior to reflect their specialized needs. The polymorphic nature of object programming would then enable the appropriate behavior to be invoked at runtime, dependent on the actual type of the instance.

```
interface Teaching_Assistant : Employee, Student {...};
```

The ODMG Object Model supports multiple inheritance. Therefore it is possible that a type inherits characteristics that have the same name, but different semantics, from two different supertypes. The model currently does not specify how name clashes are resolved; this is implementation-defined.

```
interface Salaried_Employee : Employee {...};
interface Hourly_Employee : Employee {...};
```

Some types are directly instantiable and are called *concrete types*. Others are called *abstract types* and cannot be directly instantiated. For example, if it is logically the case that all employees are either hourly or salaried, then Salaried_Employee and Hourly_Employee would be concrete types, and their supertype Employee would be an abstract type. There could be no direct instantiations of Employee. ODL does not explicitly denote whether a type is abstract or concrete.

2.2.2 Extents

The *extent* of a type is the set of all instances of the type within a particular database. If an object is an instance of type **A**, then it will of necessity be a member of the extent of **A**. If type **A** is a subtype of type **B**, then the extent of **A** is a subset of the extent of **B**.

A relational DBMS maintains an extent for every defined table. By contrast, the object database designer can decide whether the ODBMS should automatically maintain the extent of each type. Extent maintenance includes inserting newly created instances in the set and removing instances from the set as they are deleted. It may also mean creating and managing indexes to speed access to particular instances in the extent. Index maintenance can introduce significant overhead, so the object model definer specifies that the extent should be indexed separately from specifying that the extent should be maintained by the ODBMS.

2.2.3 Keys

In some cases the individual instances of a type can be uniquely identified by the values they carry for some property or set of properties. These identifying properties are called *keys*. In the relational model, these properties (actually, just attributes in relational databases) are called *candidate keys*. A *simple key* consists of a single property. A *compound key* consists of a set of properties. The scope of uniqueness is the extent of the type, thus a type must have an extent to have a key.

2.3 Objects

This section considers each of the following aspects of objects:

- Identifiers, which are used by an ODBMS to distinguish one object from another and to find objects.
- Names, which are designated by programmers or end-users as convenient ways to refer to particular objects.
- Creation, which refer to the manner in which objects are created by the programmer.
- Lifetimes, which determine how the memory and storage allocated to objects are managed.
- Structure, which can be either atomic or not, in which case the object is comprised of other objects.

All objects have the following ODL interface, which is implicitly inherited by the definitions of all user-defined objects:

```
interface Object {
    boolean    same_as(in Object anObject);
    Object     copy();
    void       delete();
};
```

2.3.1 Object Identifiers

Because all objects have identifiers, an object can always be distinguished from all other objects within its *storage domain*. In this release of the ODMG Object Model, a storage domain is a database. All identifiers of objects in a database are unique, relative to each other.The representation of the identity of an object is referred to as its *object identifier* (or Object_Id). An object retains the same object identifier for its entire lifetime. Thus the value of an object's identifier will never change. The object remains the same object, even if its attribute values or relationships change. An object identifier is commonly used as a means for one object to reference another.

Note that the notion of object identifier is different from the notion of primary key in the relational model. A row in a relational table is uniquely identified by the value of the column(s) comprising the table's primary key. If the value in one of those columns changes, the row changes its identity and becomes a different row. Even traceability to the prior value of the primary key is lost.

Literals do not have their own identifiers and cannot stand alone as objects; they are embedded in objects and cannot be individually referenced. Literal values are sometimes described as being constant. An earlier release of the ODMG Object Model described literals as being immutable. The value of a literal cannot change. Examples of literal values are the numbers 7 and 3.141596, the characters A and B, and the strings Fred and April 1. By contrast, objects, which have identifiers, have been described as being *mutable*. Changing the values of the attributes of an object, or the relationships in which it participates, does not change the identity of the object.

Object identifiers are generated by the ODBMS, not by applications. There are many possible ways to implement object identifiers. The structure of the bit pattern representing an object identifier is not defined by the Object Model, as this is considered to be an implementation issue, inappropriate for incorporation in the Object Model. Instead, the operation same_as() is supported which allows the identity of any two objects to be compared.

2.3.2 Object Names

In addition to being assigned an object identifier by the ODBMS, an object may be given one or more names that are meaningful to the programmer or end-user. The ODBMS provides a function that it uses to map from an object name to an object identifier. The application can refer at its convenience to an object by name; the ODBMS applies the mapping function to determine the object identifier that locates the desired object. ODMG expects names to be commonly used by applications to refer to "root" objects, which provide entry points into databases.

Object names are like global variable names in programming languages. They are not the same as keys. A key is comprised of properties specified in an object type's interface. An object name, by contrast, is not defined in a type interface and does not correspond to an object's property values.

The scope of uniqueness of names is a database. The Object Model does not include a notion of hierarchical name spaces within a database, or of name spaces that span databases.

2.3.3 Object Creation

Objects are created by invoking creation operations on *factory interfaces* provided on factory objects supplied to the programmer by the language binding implementation. Each factory interface will create objects of one or more types, as specified by its operations. The new operation, defined below, causes the creation of a new instance of an object of the Object type.

```
interface ObjectFactory {
   Object      new();
};
```

2.3.4 Object Lifetimes

The *lifetime* of an object determines how the memory and storage allocated to the object are managed. The lifetime of an object is specified at the time the object is created.

Two lifetimes are supported in the Object Model:

- **transient**

- **persistent**

An object whose lifetime is transient is allocated memory that is managed by the programming language run-time system. Sometimes a transient object is declared in the heading of a procedure and is allocated memory from the stack frame created by the programming language run-time system when the procedure is invoked. That memory is released when the procedure returns. Other transient objects are scoped by a process rather than a procedure activation and are typically allocated to either static memory or the heap by the programming language system. When the process terminates, the memory is deallocated. An object whose lifetime is persistent is allocated memory and storage managed by the ODBMS run-time system. These objects continue to exist after the procedure or process that creates them terminates. Persistent objects are sometimes referred to as *database objects*. Particular programming languages may refine the notion of transient lifetimes in manners consistent with their lifetime concepts.

An important aspect of object lifetimes is that they are independent of types. A type may have some instances that are persistent and others that are transient. This independence of type and lifetime is quite different from the relational model. In the relational model, any type known to the DBMS by definition has only persistent instances, and any type not known to the DBMS (i.e., any type not defined using SQL) by definition has only transient instances. Because the ODMG Object Model supports independence of type and lifetime, both persistent and transient objects can be manipulated using the same operations. In the relational model, SQL must be used for defining and using persistent data, while the programming language is used for defining and using transient data.

2.3.5 Atomic Objects

An atomic object type is user-defined. There are no built-in atomic object types included in the ODMG Object Model. See Section 2.5 for information about the properties and behavior that can be defined for atomic objects.

2.3.6 Collection Objects

In the ODMG Object Model, object types that are not atomic are collections. Instances of these types comprise distinct elements, each of which can be an instance of an atomic type, another collection, or a literal type. Literal types will be discussed in section 2.4. An important distinguishing characteristic of a collection is that *all* the elements of the collection must be of the *same* type. They are either all the same atomic type, or all the same type of collection, or all the same type of literal.

The collection types supported by the ODMG Object Model include:

- **Set<t>**

- **Bag<t>**
- **List<t>**
- **Array<t>**
- **Dictionary<t,v>**

Each of these is a type generator, parameterized by the type shown within the angle brackets. All the elements of a Set object are of the same type **t.** All the elements of a List object are of the same type **t**. In the following interfaces, we have chosen to use the ODL type any to represent these typed parameters, recognizing that this can imply a heterogeneity which is not the intent of this object model.

Collections are created by invoking the operations on the CollectionFactory interface defined below. The new operation creates a Collection with a system-dependent default amount of storage for its elements. The new_of_size operation creates a Collection with the given amount of initial storage allocated, where the given size is the number of elements for which storage is to be reserved.

```
interface CollectionFactory : ObjectFactory {
    Collection      new_of_size(in long size);
};
```

Collections all have the following operations:

```
interface Collection : Object {
    exception               InvalidCollectionType{};
    unsigned long           cardinality();
    boolean                 is_empty();
    void                    insert_element(in any element);
    void                    remove_element(in any element);
    boolean                 contains_element(in any element);
    Iterator                create_iterator(in boolean stability);
    BidirectionalIterator   create_bidirectional_iterator(in boolean stability)
                                raises(InvalidCollectionType);
};

interface Iterator : Object {
    exception       NoMoreElements{};
    boolean         is_stable();
    boolean         at_end();
    void            reset();
    any             get_element() raises(NoMoreElements);
    void            next_position() raises(NoMoreElements);
};

interface BidirectionalIterator : Iterator {
    boolean         at_beginning();
    void            previous_position() raises(NoMoreElements);
};
```

An Iterator, which is a mechanism for accessing the elements of a Collection object, can be created to traverse a collection. The create_iterator and create_bidirectional_iterator operations create iterators which support forward only traversals on all collections and bidirectional traversals of ordered collections. The stability of an iterator determines whether an iteration is safe from changes made to the collection during iteration. A stable iterator ensures that modifications made to a collection, during iteration, will not affect traversal. If an iterator is not stable, the iteration supports only retrieving elements from a collection during traversal as changes made to the collection during iteration may result in missed elements or the double processing of an element. Creating an iterator automatically positions the iterator to the first element in the iteration. The get_element operation is used to retrieve the element currently pointed to by the iterator. The next_position operation increments the iterator to the next element in the iteration. The previous_element operation decrements the iterator to the previous element in the iteration. A copy of a collection (the copy operation is inherited from the Object interface) returns a new Collection object whose elements are the same as the elements of the original Collection object (i.e. this is a shallow copy operation).

2.3.6.1 Set Objects

A Set object is an unordered collection of elements, with no duplicates allowed. Set refines the following operations inherited from its Collection supertype:

```
interface Set : Collection {
    Set         union_with(in Set other);
    Set         intersection_with(in Set other);
    Set         difference_with(in Set other);
    boolean     is_subset_of(in Set other_set);
    boolean     is_proper_subset_of(in Set other_set);
    boolean     is_superset_of(in Set other_set);
    boolean     is_proper_superset_of(in Set other_set);
};
```

The inherited insert_element operation inserts the object passed as its argument into an existing Set object. If the object passed as an argument to the insert_element operation is already a member of the Set object, the argument is discarded and the Set remains unchanged. Equality is determined by the element’s same_as operator.

In addition to the operations inherited from its supertype, the Set type interface has the conventional mathematical set operations, as well as subsetting and supersetting boolean tests. The union_with, intersection_with, and difference_with operations each returns a new result Set object.

2.3.6.2 Bag Objects

A Bag object is an unordered collection of elements that may contain duplicates. Equality of elements is determined by the element’s same_as operator. Bag refines the following operations inherited from its Collection supertype: insert_element, remove_element.

The insert_element operation inserts into the Bag object the element passed as an argument. If the element is already a member of the bag, it is inserted another time, increasing its multiplicity in the bag.

In addition to the operations inherited from its Collection supertype, the Bag type has the following operations defined in its interface:

```
interface Bag : Collection {
    Bag         union_with(in Bag other);
    Bag         intersection_with(in Bag other);
    Bag         difference_with(in Bag other);
};
```

The union_with operation is equivalent to creating a new Bag object that is a copy of the receiver bag, then iterating through the argument's bag and performing an insert into the new bag for each element in that argument bag.

2.3.6.3 List Objects

A List object is an ordered collection of elements. In addition to the operations it inherits from its supertype Collection, the List type has these operations specified:

```
interface List : Collection {
    void    replace_element_at(in unsigned long index, in any element);
    any     remove_element_at(in unsigned long index);
    any     retrieve_element_at(in unsigned long index);
    void    insert_element_after(in any obj, in unsigned long index);
    void    insert_element_before(in any obj, in unsigned long index);
    void    insert_element_first (in any obj);
    void    insert_element_last (in any obj);
    any     remove_first_element();
    any     remove_last_element();
    any     retrieve_first_element();
    any     retrieve_last_element();
    List    concat(in List other);
    void    append(in List other);
};
```

These operations are positional in nature, either in reference to a given index or to the beginning or end of a List object. Indexing of a List object starts at 0 (zero).

2.3.6.4 Array Objects

An Array object is an ordered collection with a fixed number of elements that can be located by position. The Array type defines the following operations in addition to those inherited from its supertype Collection:

```
interface Array : Collection {
    void    replace_element_at(in unsigned long index, in any element);
    any     remove_element_at(in unsigned long index);
    any     retrieve_element_at(in unsigned long index);
```

```
    void     resize(in unsigned long new_size);
};
```

The remove_element_at operation replaces any current element contained in the cell of the array object identified by position with a null value. It does not remove the cell or change the size of the array. This is in contrast to the remove_element_at operation defined on type List, which does change the number of elements in a List object. The resize operation enables an Array object to change the maximum number of elements it can contain.

2.3.6.5 Dictionary Objects

A Dictionary object is an unordered sequence of key-value pairs with no duplicate keys. Each key-value pair is constructed as an instance of the following structure:

```
struct Association {any key; any value; };
```

In addition to the operations inherited from its Collection supertype, the Dictionary type has the following operations defined in its interface:

```
interface Dictionary : Collection {
    exception    KeyNotFound{any key; };
    void         bind(in any key, in any value);
    void         unbind(in any key) raises(KeyNotFound);
    any          lookup(in any key) raises(KeyNotFound);
    boolean      contains_key(in any key);
};
```

The insert_element, remove_element and contains_element operations, inherited from Collection, are valid for Dictionary types when an Association is specified as the argument. Iterating over a Dictionary object will result in the iteration over a sequence of Associations. Each get_element operation, executed on an Iterator object, returns a structure of type Association.

2.4 Literals

Literals do not have object identifiers. The Object Model supports three literal types:

- **atomic literal**
- **collection literal**
- **structured literal**

2.4.1 Atomic Literals

Numbers and characters are examples of atomic literal types. Instances of these types are not explicitly created by applications, but rather implicitly exist. The ODMG Object Model supports the following types of atomic literals:

- **long**
- **short**
- **unsigned long**
- **unsigned short**
- **float**
- **double**
- **boolean**
- **octet**
- **char (character)**
- **string**
- **enum (enumeration)**

These types are all also supported by the OMG Interface Definition Language (IDL). The intent of the Object Model is that a programming language binding should support the language-specific analog of these types, as well as any other atomic literal types defined by the programming language. If the programming language does not contain an analog for one of the Object Model types, then a class library defining the implementation of the type should be supplied as part of the programming language binding.

Enum is a type generator. An enum declaration defines a named literal type that can take on only the values listed in the declaration. For example, an attribute gender might be defined by

```
attribute enum gender {male, female};
```

An attribute state_code might be defined by

```
attribute enum state_code {AK,AL,AR,AZ,CA, ... WY};
```

2.4.2 Collection Literals

The ODMG Object Model supports collection literals of the following types:

- **set<t>**
- **bag<t>**
- **list<t>**
- **array<t>**
- **dictionary<t,v>**

These type generators are analogous to those of collection objects, but these collections do not have object identifiers. Their elements, however, can be of literal types or object types.

2.4.3 Structured Literals

A structured literal, or simply *structure*, has a fixed number of elements, each of which has a variable name and can contain either a literal value or an object. An element of a structure is typically referred to by a variable name, e.g., address.zip_code = 12345; address.city = "San Francisco". Structure types supported by the ODMG Object Model include

- **Date**
- **Interval**
- **Time**
- **Timestamp**

These types are defined as in the ANSI SQL specification by the following interfaces:

2.4.3.1 Date

The following interface defines the factory operations for creating Date objects:

```
interface DateFactory : ObjectFactory {
    exception InvalidDate{};
    Date        julian_date(in unsigned short year,
                                 in unsigned short julian_day)
                    raises(InvalidDate);
    Date        calendar_date(in unsigned short year,
                                    in unsigned short month,
                                    in unsigned short day)
                    raises(InvalidDate);
    boolean     is_leap_year(in unsigned short year);
    boolean     is_valid(in unsigned short year,
                            in unsigned short month,
                            in unsigned short day);
    unsigned short    days_in_year(in unsigned short year);
    unsigned short    days_in_month(in unsigned short year,
                                          in Date::Month month);
    Date        current();
};
```

The following interface defines the operations on Date objects:

```
interface Date : Object {
    typedef     unsigned short    ushort;
    enum        Weekday {Sunday, Monday, Tuesday, Wednesday,
                        Thursday, Friday, Saturday};
    enum        Month {January, February, March, April, May, June, July,
                August, September, October, November, December};
```

```
    // used to represent a Date object by a typed value
    struct asValue {ushort month, day, year;};
    ushort      year();
    ushort      month();
    ushort      day();
    ushort      day_of_year();
    Month       month_of_year();
    Weekday     day_of_week();
    boolean     is_leap_year();
    boolean     is_equal(in Date a_date);
    boolean     is_greater(in Date a_date);
    boolean     is_greater_or_equal(in Date a_date);
    boolean     is_less(in Date a_date);
    boolean     is_less_or_equal(in Date a_date);
    boolean     is_between(in Date a_date, in Date b_date);
    Date        next(in Weekday day);
    Date        previous(in Weekday day);
    Date        add_days(in ushort days);
    Date        subtract_days(in ushort days);
    long        subtract_date(in Date a_date);
};
```

2.4.3.2 Interval

Intervals represent a duration of time and are used to perform some operations on Time and TimeStamp objects. Intervals are created using the subtract_time operation defined in the Time interface below. The following interface defines the operations on Interval objects:

```
interface Interval : Object {
    typedef     unsigned short    ushort;
    ushort      day();
    ushort      hour();
    ushort      minute();
    ushort      second();
    ushort      millisecond();
    // used to represent an Interval object as a typed value
    struct      asValue {ushort day, hour, minute; float second;};
    boolean     is_zero();
    Interval    plus(in Interval an_interval);
    Interval    minus(in Interval an_interval);
    Interval    product(in short val);
    Interval    quotient(in short val);
    boolean     is_equal(in Interval an_interval);
    boolean     is_greater(in Interval an_interval);
    boolean     is_greater_or_equal(in Interval an_interval);
    boolean     is_less(in Interval an_interval);
    boolean     is_less_or_equal(in Interval an_interval);
};
```

2.4.3.3 Time

Times denote specific world times. The following interface defines the factory operations for creating Time objects:

```
interface TimeFactory : ObjectFactory {
    Time_Zone     default_time_zone();
    Time          from_hms(in unsigned short hour,
                           in unsigned short minute,
                           in float second);
    Time          from_hmstz(in unsigned short hour,
                             in unsigned short minute,
                             in float second,
                             in short tzhour,
                             in short tzminute);
    Time          current();
};
```

The following interface defines the operations on Time objects:

```
interface Time : Object {
    typedef short         Time_Zone;
    const   Time_Zone  GMT = 0;
    const   Time_Zone  GMT1 = 1;
    const   Time_Zone  GMT2 = 2;
    const   Time_Zone  GMT3 = 3;
    const   Time_Zone  GMT4 = 4;
    const   Time_Zone  GMT5 = 5;
    const   Time_Zone  GMT6 = 6;
    const   Time_Zone  GMT7 = 7;
    const   Time_Zone  GMT8 = 8;
    const   Time_Zone  GMT9 = 9;
    const   Time_Zone  GMT10 = 10;
    const   Time_Zone  GMT11 = 11;
    const   Time_Zone  GMT12 = 12;
    const   Time_Zone  GMT_1 = -1;
    const   Time_Zone  GMT_2 = -2;
    const   Time_Zone  GMT_3 = -3;
    const   Time_Zone  GMT_4 = -4;
    const   Time_Zone  GMT_5 = -5;
    const   Time_Zone  GMT_6 = -6;
    const   Time_Zone  GMT_7 = -7;
    const   Time_Zone  GMT_8 = -8;
    const   Time_Zone  GMT_9 = -9;
    const   Time_Zone  GMT_10 = -10;
    const   Time_Zone  GMT_11 = -11;
    const   Time_Zone  GMT_12 = -12;
    const   Time_Zone  USeastern = -5;
    const   Time_Zone  UScentral = -6;
    const   Time_Zone  USmountain = -7;
    const   Time_Zone  USpacific = -8;
```

```
    ushort      hour();
    ushort      minute();
    ushort      second();
    ushort      millisecond();
    short       tz_hour();
    short       tz_minute();

    boolean     is_equal(in Time a_Time);
    boolean     is_greater(in Time a_Time);
    boolean     is_greater_or_equal(in Time a_Time);
    boolean     is_less(in Time a_Time);
    boolean     is_less_or_equal(in Time a_Time);
    boolean     is_between(in Time a_Time,
                        in Time b_Time);

    Time        add_interval(in Interval an_interval);
    Time        subtract_interval(in Interval an_interval);
    Interval    subtract_time(in Time a_time);
};
```

2.4.3.4 TimeStamp

TimeStamps consist of an encapsulated Date and Time. The following interface defines the factory operations for creating TimeStamp objects:

```
interface TimeStampFactory : ObjectFactory {
    TimeStamp    current();
    TimeStamp    create(in Date a_date, in Time a_time);
};
```

The following interface defines the operations on TimeStamp objects.

```
interface TimeStamp : Object {
    typedef     unsigned short    ushort;
    Date                 the_date();
    Time                 the_time();
    ushort               year();
    ushort               month();
    ushort               day();
    ushort               hour();
    ushort               minute();
    ushort               second();
    ushort               millisecond();
    short                tz_hour();
    short                tz_minute();

    TimeStamp            plus(in Interval an_interval);
    TimeStamp            minus(in Interval an_interval);

    boolean              is_equal(in TimeStamp a_Stamp);
    boolean              is_greater(in TimeStamp a_Stamp);
    boolean              is_greater_or_equal(in TimeStamp a_Stamp);
    boolean              is_less(in TimeStamp a_Stamp);
    boolean              is_less_or_equal(in TimeStamp a_Stamp);
    boolean              is_between(in TimeStamp a_Stamp,
```

```
                                            in TimeStamp b_Stamp);
};
```

2.4.3.5 User-defined Structures

Because the Object Model is extensible, developers can define other structure types as needed. The Object Model includes a built-in type generator struct, to be used to define application structures. For example:

```
attribute struct Address {string dorm_name, string room_no} dorm_address;
```

The operations defined by the generator for the structure types include the following:

```
interface Struct {
   unsigned long  size();
   void           set_element (in any field, in any value);
   any            get_element(in any field);
   void           clear_element(in any field);
   Struct         copy();
   void           delete();
};
```

Structures may be freely composed. The Object Model supports sets of structures, structures of sets, arrays of structures, and so forth. This composability allows the definition of types like Degrees, as a list whose elements are structures containing three fields:

```
struct Degree {
   string           school_name;
   string           degree_type;
   unsigned short   degree_year;
};
typedef list<Degree>Degrees;
```

Each Degrees instance could have its elements sorted by value of degree_year.

An implementation may bind the Object Model structures and collections to classes that are provided by the programming language. For example, Smalltalk includes its own Collection, Date, Time, and Timestamp classes.

2.5 Modeling State — Properties

A type defines a set of properties through which users can access, and in some cases directly manipulate, the state of instances of the type. Two kinds of properties are defined in the ODMG Object Model: *attribute* and *relationship*. An attribute is of one type. A relationship is defined between two types, each of which must have instances that are referenceable by object identifiers. Thus literal types, because they do not have object identifiers, cannot participate in relationships.

2.5.1 Attributes

The attribute declarations in an interface define the abstract state of a type. For example, the type Person might contain the following attribute declarations:

```
interface Person {
    attribute short age;
    attribute string name;
    attribute enum gender {male, female};
    attribute Address home_address;
    attribute set<Phone_no> phones;
    attribute Department dept;
};
```

A particular instance of Person would have a specific value for each of the defined attributes. The value for the dept attribute above is the object identifier of an instance of Department. An attribute's value is always either a literal or an object identifier.

It is important to note that an attribute is not the same as a data structure. An attribute is abstract, while a data structure is a physical representation. While it is common for attributes to be implemented as data structures, it is sometimes appropriate for an attribute to be implemented as a method. For example, the age attribute might very well be implemented as a method that calculates age from a stored value of the person's date_of_birth and the current date.

In this release of the ODMG Object Model, attributes are not "first class." This means that an attribute itself is not an object and therefore does not have an object identifier. It is not possible to define attributes of attributes or relationships between attributes or subtype-specific operations for attributes.

2.5.2 Relationships

Relationships are defined between types. The ODMG Object Model supports only binary relationships, i.e., relationships between two types. The model does not support n-ary relationships, which involve more than two types. A binary relationship may be one-to-one, one-to-many, or many-to-many, depending on how many instances of each type participate in the relationship. For example, *marriage* is a one-to-one relationship between two instances of type Person. A woman can have a one-to-many *mother of* relationship with many children. Teachers and students typically participate in many-to-many relationships. Relationships in the Object Model are similar to relationships in entity-relationship data modeling.

Relationships in this release of the Object Model are not named and are not "first class." A relationship is not itself an object and does not have an object identifier. A relationship is defined implicitly by declaration of *traversal paths* that enable applications to use the logical connections between the objects participating in the relationship. Traversal paths are declared in pairs, one for each direction of traversal of the binary relationship. For example, a professor *teaches* courses and a course *is taught by* a professor. The teaches traversal path would be defined in the interface declaration for the Professor type. The is_taught_by traversal path would be defined in the interface declaration for the Course type. The fact that these traversal paths both apply to the same relationship is indicated by an inverse clause in both of the traversal path declarations. For example:

```
interface Professor {
    ...
    relationship set<Course> teaches
        inverse Course::is_taught_by;
     ...
}
```

and

```
interface Course  {
    ...
    relationship Professor is_taught_by
        inverse Professor::teaches;
     ...
}
```

The relationship defined by the teaches and is_taught_by traversal paths is a one-to-many relationship between Professor and Course objects. This cardinality is shown in the traversal path declarations. A Professor instance is associated with a set of Course instances via the teaches traversal path. A Course instance is associated with a single Professor instance via the is_taught_by traversal path.

Traversal paths that lead to many objects can be unordered or ordered, as indicated by the type of collection specified in the traversal path declaration. If set is used, as in set<Course>, the objects at the end of the traversal path are unordered.

The ODBMS is responsible for maintaining the referential integrity of relationships. This means that if an object that participates in a relationship is deleted, then any traversal path to that object must also be deleted. For example, if a particular Course instance is deleted, then not only is that object's reference to a Professor instance via the is_taught_by traversal path deleted, but also any references in Professor objects to the Course instance via the teaches traversal path must also be deleted. Maintaining referential integrity ensures that applications cannot dereference traversal paths that lead to non-existent objects.

```
attribute Student top_of_class;
```

An attribute may be Object-valued. This kind of attribute enables one object to reference another, without expectation of an inverse traversal path or referential integrity.

It is important to note that a relationship traversal path is not equivalent to a pointer. A pointer in C++ or Smalltalk has no connotation of a corresponding inverse traversal path, which would form a relationship. The operations defined on relationship parties and their traversal paths vary according to the traversal path's cardinality.

The implementation of relationships is encapsulated by public operations which *form* and *drop* members from the relationship, plus public operations on the relationship target classes to provide access and to manage the required referential integrity constraints. When the traversal path has cardinality "one," operations are defined to form a relationship, to drop a relationship, and to traverse the relationship. When the traversal path has cardinality "many," the object will support methods to add and remove elements from its traversal path collection. Traversal paths support all of the behaviors defined above on the Collection class used to define the behavior of the relationship. Implementations of form and drop operations will guarantee referential integrity in all cases. In order to facilitate the use of ODL object models in situations where such models may cross distribution boundaries, we define the relationship interface in purely procedural terms by introducing a mapping rule from ODL relationships to equivalent IDL constructions. Then, each language binding will determine the exact manner in which these constructions are to be accessed.

2.5.2.1 Cardinality "one" Relationships

For relationships with cardinality "one" such as

```
relationship   X   inverse Z;
```

we expand the relationship to an equivalent IDL attribute and operations:

```
attribute      X   Y;
void           form_Y(in X target);
void           drop_Y(in X target);
```

For example, the relationship in the above example interface *Course* would result in the following definitions (on the class *Course*):

```
attribute      Professor      is_taught_by;
void                          form_is_taught_by(in Professor aProfessor);
void                          drop_is_taught_by(in Professor aProfessor);
```

2.5.2.2 Cardinality "many" Relationships

For ODL relationships with cardinality "many" such as

```
relationship       set<x>         Y inverse Z;
```

we expand the relationship to an equivalent IDL attribute and operations. To convert these definitions into pure IDL, the ODL collection need only be replaced by the keyword *sequence*. Note that the add_Y operation may raise an IntegrityError exception in the event that the traversal is a set which already contains a reference to the given target X. This exception, if it occurs, will also be raised by the form_Y operation which invoked the add_Y

```
readonly attribute      set<X> Y;
void                    form_Y(in X target) raises(IntegrityError);
void                    drop_Y(in X target);
void                    add_Y(in X target) raises(IntegrityError);
void                    remove_Y(in X target);
```

For example, the relationship in the above example interface *Professor* would result in the following definitions (on the class *Professor*):

```
readonly attribute      set<course> teaches;
void                    form_teaches(in Course aCourse)
                          raises(IntegrityError);
void                    drop_teaches(in Course aCourse);
void                    add_teaches(in Course aCourse)
                          raises(IntegrityError);
void                    remove_teaches(in Course aCourse);
```

2.6 Modeling Behavior — Operations

Besides the attribute and relationship properties, the other characteristic of a type is its behavior, which is specified as a set of *operation signatures*. Each signature defines the name of an operation, the name and type of each of its arguments, the types of value(s) returned, and the names of any *exceptions* (error conditions) the operation can raise. Our Object Model specification for operations is identical to the OMG CORBA specification for operations.

An operation is defined on only a single type. There is no notion in the Object Model of an operation that exists independent of a type, or of an operation defined on two or more types. An operation name need be unique only within a single type definition. Thus different types could have operations defined with the same name. The names of these operations are said to be *overloaded*. When an operation is invoked using an overloaded name, a specific operation must be selected for execution. This selection, sometimes called *operation name resolution* or *operation dispatching*, is based on the most specific type of the object supplied as the first argument of the actual call.

The ODMG had several reasons for choosing to adopt this single-dispatch model rather than a multiple-dispatch model. The major reason was for consistency with the C++ and Smalltalk programming languages. This consistency enables seamless integration of ODBMSs into the object programming environment. Another reason to adopt the classical object model was to avoid incompatibilities with the OMG CORBA object model, which is classical rather than general.

An operation may have side effects. Some operations may return no value. The ODMG Object Model does not include formal specification of the semantics of operations. It is good practice, however, to include comments in interface specifications, for example remarking on the purpose of an operation, any side effects it might have, pre- and post-conditions, and any invariants it is intended to preserve.

The Object Model assumes sequential execution of operations. It does not require support for concurrent or parallel operations, but does not preclude an ODBMS from taking advantage of multiprocessor support.

2.6.1 Exception Model

The ODMG Object Model supports dynamically nested exception handlers, using a termination model of exception handling. Operations can raise exceptions, and exceptions can communicate exception results. Exceptions in the Object Model are themselves objects and have an interface which allows them to be related to other exceptions in a generalization-specialization hierarchy.

A root type Exception is provided by the ODBMS. This type includes an operation to issue a message noting that an unhandled exception of type Exception_type has occurred to terminate the process. Information on the cause of an exception or the context in which it occurred is passed back to the exception handler as properties of the Exception object.

Control is as follows:

1. The programmer declares an exception handler within scope **s** capable of handling exceptions of type **t**.
2. An operation within a contained scope **sn** may "raise" an exception of type **t**.
3. The exception is "caught" by the most immediately containing scope that has an exception handler. The call stack is automatically unwound by the run-time system out to the level of the handler. Destructors are called for all objects allocated in intervening stack frames. Any transactions begun within a nested scope, that is unwound by the run-time system in the process of searching up the stack for an exception handler, are aborted.
4. When control reaches the handler, the handler may either decide that it can handle the exception or pass it on (reraise it) to a containing handler.

An exception handler that declares itself capable of handling exceptions of type **t**, will also handle exceptions of any subtype of **t**. A programmer who requires more specific control over exceptions of a specific subtype of **t** may declare a handler for this more specific subtype within a contained scope.

The signature of an operation includes declaration of the exceptions that the operation can raise.

2.7 Metadata

Metadata is descriptive information about database objects which defines the *schema* of a database. It is used by the ODBMS to define the structure of the database and at runtime to guide its access to the database. Metadata is stored in an *ODL Schema*

Repository which is also accessible to tools and applications using the same operations that apply to user-defined types. In OMG CORBA environments, similar metadata is stored in an IDL Interface Repository.

The following interfaces define the internal structure of an ODL Schema Repository. These interfaces are defined in ODL using *relationships* which define the graph of interconnections between *meta objects* which are produced, for example, during ODL source compilation. While these relationships guarantee the referential integrity of the meta object graph they do not guarantee its semantic integrity or completeness. In order to provide operations which programmers can use to correctly construct valid schemas, several creation, addition and removal operations are defined which provide automatic linking and unlinking of the required relationships and appropriate error recovery in the event of semantic errors.

All of the meta object definitions, defined below, are to be grouped into an enclosing module which defines a name scope for the elements of the model.

```
module ODLMetaObjects {
    // the following interfaces are defined here
};
```

2.7.1 Scopes

Scopes define a naming hierarchy for the meta objects in the repository. They support a bind operation for adding meta objects, a resolve operation for resolving path names within the repository, and an un_bind operation for removing bindings.

```
interface Scope {
    exception      DuplicateName{};
    void           bind(in string name, in MetaObject value)
                        raises(DuplicateName);
    MetaObject     resolve(in string name);
    MetaObject     un_bind(in string name);
};
```

2.7.2 Meta Objects

All objects in the repository are subclasses of three main interfaces: MetaObject, Specifier, and Operand. All MetaObjects, defined below, have name and comment attributes. They participate in a single definedIn relationship with other meta objects which are their defining scopes. DefiningScopes are Scopes which contain other meta object definitions using their defines relationship and which have operations for creating, adding and removing meta objects within themselves.

```
interface MetaObject {
    attribute      stringname;
    attribute      stringcomment;
    relationship   DefiningScopedefinedIn
```

```
                              inverse DefiningScope::defines;
};

enum PrimitiveKind {pk_boolean, pk_char, pk_short, pk_ushort, pk_long,
                              pk_ulong, pk_float, pk_double, pk_octet, pk_string,
                              pk_void, pk_any};

enum CollectionKind {ck_list, ck_array, ck_bag, ck_set, ck_dictionary};

interface DefiningScope : Scope {
    relationship     list<MetaObject>defines
                              inverse MetaObject::definedIn;
    exception        InvalidType{string reason; };
    exception        InvalidExpression{string reason; };
    exception        CannotRemove{string reason; };
    PrimitiveType  create_primitive_type(in PrimitiveKind kind);
    Collection       create_collection_type(in CollectionKind kind,
                              in Operand maxSize, in Type subType);
    Operand          create_operand(in string expression)
                              raises(InvalidExpression);
    Member           create_member(in string memberName,
                              in Type memberType);
    UnionCase        create_case(in string caseName, in Type caseType,
                              in list<Operand> caseLabels)
                              raises(DuplicateName, InvalidType);
    Constant         add_constant(in string name, in Operand value)
                              raises(DuplicateName);
    TypeDefinition add_typedef(in string name, in Type alias)
                              raises(DuplicateName);
    Enumeration    add_enumeration(in string name,
                              in list<string> elementNames)
                              raises(DuplicateName, InvalidType);
    Structure        add_structure(in string name, in list<Member> fields)
                              raises(DuplicateName, InvalidType);
    Union            add_union(in string name, In Type switchType,
                              in list<UnionCase> cases)
                              raises(DuplicateName, InvalidType);
    Exception        add_exception(in string name, in Structure result)
                              raises(DuplicateName);
    void             remove_constant(in Constant object)
                              raises(CannotRemove);
    void             remove_typedef(in TypeDefinition object)
                              raises(CannotRemove);
    void             remove_enumeration(in Enumeration object)
                              raises(CannotRemove);
    void             remove_structure(in Structure object)
                              raises(CannotRemove);
    void             remove_union(in Union object) raises(CannotRemove);
    void             remove_exception(in Exception object)
                              raises(CannotRemove);
};
```

2.7.2.1 Modules

Modules and the Schema Repository itself, which is a specialized module, are DefiningScopes which define operations for creating modules and interfaces within themselves.

```
interface Module : MetaObject, DefiningScope {
    Module      add_module(in string name) raises(DuplicateName);
    Interface   add_interface(in string name, in list<Interface> inherits)
                    raises(DuplicateName);
    void        remove_module(in Module object) raises(CannotRemove);
    void        remove_interface(in Interface object) raises(CannotRemove);
};

interface Repository : Module {};
```

2.7.2.2 Operations

Operations model the behavior which application objects support. They are maintain a signature list of Parameters, and refer to a result type. Operations may raise Exceptions. The ScopedMetaObject interface consolidates Scope operations for its sub-classes Operation and Exception.

```
interface ScopedMetaObject : MetaObject, Scope {};

interface Operation : ScopedMetaObject {
    relationship   list<Parameter>   signature
                       inverse Parameter::operation;
    relationship   Type              result
                       inverse Type::operations;
    relationship   list<Exception>   exceptions
                       inverse Exception::operations;
};
```

2.7.2.3 Exceptions

Operations may raise Exceptions, and thereby return a different set of results. Exceptions refer to a Structure which defines their results and keep track of the Operations which may raise them.

```
interface Exception : MetaObject {
    relationship   Structure      result
                       inverse Structure::exceptionResult;
    relationship   set<Operation>operations
                       inverse Operation::exceptions;
};
```

2.7.2.4 Constants

Constants provide a mechanism for statically associating values with names in the repository. The value is defined by an Operand sub-class which is either a literal value (Literal), a reference to another Constant (ConstOperand), or the value of a constant expression (Expression). Each constant has an associated type, and keeps track of the other ConstOperands which refer to it in the repository. The value operation allows the constant's actual value to be computed at any time.

```
interface Constant : MetaObject {
    relationship    Operand             hasValue
                         inverse Operand::valueOf;
    relationship    Type                type
                         inverse Type::constants;
    relationship    set<ConstOperand>   referencedBy
                         inverse ConstOperand::references;
    relationship    Enumeration         enumeration
                         inverse Enumeration::elements;
    any             value();
};
```

2.7.2.5 Properties

Properties form an abstract class over the Attribute and Relationship meta objects which define the abstract state of an application object. They have an associated type.

```
interface Property : MetaObject {
    relationship    Type        type
                    inverse Type::properties;
};
```

2.7.2.5.1 Attributes

Attributes are properties which maintain simple abstract state. They may be read-only, in which case there is no associated accessor for changing their values. Attributes are used as sorting criteria by sorted Relationships and maintain a relationship with these clients.

```
interface Attribute : Property {
    attribute       boolean             isReadOnly;
    relationship    set<Relationship>   sorterFor
                         inverse Relationship::sortedBy;
};
```

2.7.2.5.2 Relationships

Relationships model bilateral object references between participating objects. In use, two relationship meta objects are required to represent each traversal direction of the relationship. Operations are defined implicitly to form and drop the relationship, as well as accessor operations for manipulating its traversals.

```
enum Cardinality {c1_1, c1_N, cN_1, cN_M};

interface Relationship : Property {
    exception       integrityError{};
    relationship    Relationship   traversal
```

```
                    inverse Relationship::traversal;
    relationship    list<Attribute> sortedBy
                    inverse Attribute::sorterFor;
    Cardinality     getCardinality();
};
```

2.7.2.6 Types

TypeDefinitions are meta objects which define new names, or aliases, for the types to which they refer. Much of the information in the repository consists of type definitions which define the datatypes used by the application.

```
interface TypeDefinition : Type {
    relationship    Type        alias
                    inverse Type::typeDefs;
};
```

Type meta objects are used to represent information about datatypes. They participate in a number of relationships with the other meta objects which use them. These relationships allow Types to be easily administered within the repository and help to ensure the referential integrity of the repository as a whole.

```
interface Type : MetaObject {
    relationship    set<Collection>        collections
                    inverse Collection::subtype;
    relationship    set<Specifier>         specifiers
                    inverse Specifier::type;
    relationship    set<Union>             unions
                    inverse Union::switchType;
    relationship    set<Operation>         operations
                    inverse Operation::result;
    relationship    set<Property>          properties
                    inverse Property::type;
    relationship    set<Constant>          constants
                    inverse Constant::type;
    relationship    set<TypeDefinition>    typeDefs
        inverse            TypeDefinition::alias;
};

interface PrimitiveType : Type {
    attribute    PrimitiveKind  kind;
};
```

2.7.2.6.1 Interfaces

Interfaces are the most important types in the repository. Interfaces define the abstract state of application objects as well as their public behavior and contain operations for creating and removing Attributes, Relationships, and Operations within themselves in addition to the operations inherited from DefiningScope. Interfaces also may define keys and extents over their instances. Interfaces are linked in a multiple-inheritance graph with other Inheritance objects by two relationships, inherits and derives. They may contain most kinds of MetaObjects, excepting Modules and Interfaces.

```
interface Interface : Type, DefiningScope {
    struct ParameterSpec {
        string       paramName;
        Direction    paramMode;
        Type         paramType; };
    attribute        list<string>        extents;
    attribute        list<string>        keys;
    relationship     set<Inheritance>    inherits
                         inverse Inheritance::derivesFrom;
    relationship     set<Inheritance>    derives
                         inverse Inheritance::inheritsTo;
    exception        BadParameter{string reason; };
    exception        BadRelationship{string reason; };
    Attribute        add_attribute(in string attrName, in Type attrType)
                         raises(DuplicateName);
    Relationship     add_relationship(in string relName,
                         in Type relType,
                         in Relationship relTraversal)
                         raises(DuplicateName, BadRelationship);
    Operation        add_operation(in string opName,
                         in Type opResult,
                         in list<ParameterSpec> opParams,
                         in list<Exception> opRaises)
                         raises(DuplicateName, BadParameter);
    void             remove_attribute(in Attribute object)
                         raises(CannotRemove);
    void             remove_relationship(in Relationship object)
                         raises(CannotRemove);
    void             remove_operation(in Operation object)
                         raises(CannotRemove);
};

interface Inheritance {
    relationship     Interface    derivesFrom
                         inverse Interface::inherits;
    relationship     Interface    inheritsTo
                         inverse Interface::derives;
};
```

2.7.2.6.2 Collections

Collections are types which aggregate variable numbers of elements of a single subtype and provide different ordering, accessing, and comparison behaviors. The maximum size of the collection may be specified by a constant or constant expression. If unspecified, this relationship will be bound to the literal 0.

```
interface Collection : Type {
    attribute        CollectionKind      kind;
```

```
    relationship    Operand         maxSize
                        inverse Operand::sizeOf;
    relationship    Type            subtype
                        inverse Type::collections;
    boolean         isOrdered();
    unsigned long   bound();
};
```

2.7.2.6.3 Constructed Types

Some types contained named elements which themselves refer to other types and are said to be constructed from those types. The ScopedType interface is an abstract class which consolidates these mechanisms for its sub-classes Enumeration, Structure, and Union. Enumerations contain Constants, Structures contain Members, and Unions contain UnionCases. Unions, in addition, have a relationship with a switchType which defines the discriminator of the union.

```
interface ScopedType : Scope, Type {};

interface Enumeration : ScopedType {
    relationship    list<Constant>  elements
                        inverse Constant::enumeration;
};

interface Structure : ScopedType {
    relationship    list<Member> fields
                        inverse Member::structure_type;
    relationship    Exception       exceptionResult
                        inverse Exception::result;
};

interface Union : ScopedType {
    relationship    Type            switchType
                        inverse Type::unions;
    relationship    list<UnionCase> cases
                        inverse UnionCase::union_type;
};
```

2.7.3 Specifiers

Specifiers are used to assign a name to a type in certain contexts. They consolidate these elements for their sub-classes. Members, UnionCases and Parameters are referenced by Structures, Unions and Operations respectively.

```
interface Specifier {
    attribute       string      name;
    relationship    Type        type
                        inverse Type::specifiers;
};

interface Member : Specifier {
    relationship    Structure   structure_type
```

```
                            inverse Structure::fields;
};

interface UnionCase : Specifier {
    relationship    Union           union_type
                            inverse Union::cases;
    relationship    list<Operand> caseLabels
                            inverse Operand::caseIn;
};

enum Direction {mode_in, mode_out, mode_inout } ;

interface Parameter : Specifier {
    attribute       Direction   parameterMode;
    relationship    Operation  operation
                            inverse Operation::signature;
};
```

2.7.4 Operands

Operands form the base type for all constant values in the repository. They have a value operation and maintain relationships with the other Constants, Collections, UnionCases and Expressions which refer to them. Literals contain a single literalValue attribute and produce their value directly. ConstOperands produce their value by delegating to their associated constant. Expressions compute their value by evaluating their operator on the values of their operands.

```
interface Operand {
    relationship    Expression      operandIn
                            inverse Expression::hasOperands;
    relationship    Constant        valueOf
                            inverse Constant::hasValue;
    relationship    Collection      sizeOf
                            inverse Collection::maxSize;
    relationship    UnionCase       caseIn
                            inverse UnionCase::caseLabels;
    any             value();
};

interface Literal : Operand {
    attribute    any    literalValue;
};

interface ConstOperand : Operand {
    relationship    Constant   references
                            inverse Constant::referencedBy;
};
```

Expressions are composed of one or more Operands and an associated operator. While unary and binary operators are the only operations allowed by ODL, this structure allows generalized n-ary operations to be defined in the future.

```
interface Expression : Operand {
    attribute       string          operator;
    relationship    list<Operand>   hasOperands
                                        inverse Operand::operandIn;
};
```

2.7.5 The Full Built-in Type Hierarchy

Figure 2-1 shows the full set of built-in types of the Object Model type hierarchy. Concrete types are shown in non-italic font and are directly instantiable. Abstract types are shown in italics. In the interests of simplifying matters, both types and type generators are included in the same hierarchy. Type generators are signified by angle brackets (e.g., Set<>).

2.7.6 Type Compatibility Rules

The ODMG Object Model is strongly typed. Every object or literal has a type, and every operation requires typed operands. The rules for type identity and type compatibility are defined in this section.

Two objects or literals have the same type if and only if they have been declared to be instances of the same named type. Objects or literals that have been declared to be instances of two different types are not of the same type, even if the types in question define the same set of properties and operations. Type compatibility follows the subtyping relationships defined by the type hierarchy. If **TS** is a subtype of **T**, then an object of type **TS** can be assigned to a variable of type **T**, but the reverse is not possible. No implicit conversions between types are provided by the Object Model.

Two atomic literals have the same type if they belong to the same set of literals. Depending on programming language bindings, implicit conversions may be provided between the scalar literal types, i.e., long, short, unsigned long, unsigned short, float, double, boolean, octet, char. No implicit conversions are provided for structured literals.

Literal_type

  Atomic_literal

    long
    short
    unsigned long
    unsigned short
    float
    double
    boolean
    octet
    char

string
enum<>
Collection_literal
set<>
bag<>
list<>
array<>
Structured_literal
Date
Time
Timestamp
Interval
Structure<>
Object_type
Atomic_object
Collection
Set<>
Bag<>
List<>
Array<>

Figure 2-1. Full Set of Built-in Types

2.7.7 Null Value

For every literal type (e.g., float or Set<>) there exists another literal type supporting a null value (e.g., nullable_float or nullable_Set<>). This nullable type is the same as the literal type augmented by the null value “nil”. The semantics of null is the same as that defined by SQL-92.

2.7.8 Table Type

In order to make clear that the ODMG data model encompasses the relational data model, the type generator Table<> is defined in the ODMG data model as a synonym of the type generator bag<struct<>> such that

```
Table(a1:t1, a2:t2, ... , an:tn)
```

is equivalent to the definition

```
bag<struct< a1:t1, a2:t2, ... , an:tn>>
```

2.8 Transaction Model

Programs that use persistent objects are organized into transactions. Transaction management is an important ODBMS functionality, fundamental to database integrity, shareability, and recovery. Any access, creation, modification, and deletion of persistent objects must be done within a transaction.

A transaction is a unit of logic for which an ODBMS guarantees *atomicity*, *consistency*, *isolation*, and *durability*. *Atomicity* means that the transaction either finishes or has no effect at all. *Consistency* means that a transaction takes the database from one internally consistent state to another internally consistent state. There may be times during the transaction when the database is inconsistent. However, *isolation* guarantees that no other user of a database sees changes made by a transaction until that transaction commits. Concurrent users always see an internally consistent database. *Durability* means that the effects of committed transactions are preserved, even if there should be failures of storage media, loss of memory, or system crashes. Once a transaction has committed, the ODBMS guarantees that changes made by that transaction are never lost. When a transaction commits, all of the changes made by that transaction are permanently installed in the database and made visible to other users of the database. When a transaction aborts, none of the changes made by it are installed in the database, including any changes made prior to the time of abort. The execution of concurrent transactions must yield results which are indistinguishable from results that would have been obtained if the transactions had been executed serially. This property is sometimes called *serializability*.

2.8.1 Distributed Transactions

Distributed transactions are transactions which span multiple processes and/or which span more than one database, as describes in ISO XA and the OMG Object Transaction Service. The ODMG does not define an interface for distributed transactions because this is already defined in the ISO XA standard and since it is not visible to the programmers, but only used by transaction monitors.Vendors are not required to support distributed transactions, but if they do, their implementations must be XA-compliant.

2.8.2 Transactions and Processes

The ODMG Object Model assumes a linear sequence of transactions executing within a thread of control; that is, there is exactly one current transaction for a thread, and that transaction is implicit in that thread's database operations. If an ODMG language binding supports multiple threads in one address space, then transaction *isolation* must be provided between the threads. Of course, transaction *isolation* is also provided between threads in different address spaces or threads running on different machines.

A transaction runs against a single logical database. Note that a single logical database may be implemented as one or more physical databases, possibly distributed on a network. The transaction model neither requires nor precludes support for transactions that span multiple threads, multiple address spaces, or more than one logical database.

In the current Object Model, transient objects in an address space are not subject to transaction semantics. This means that aborting a transaction does not restore the state of modified transient objects.

2.8.3 Locking and Concurrency Control

The ODMG Object Model uses a conventional lock-based approach to concurrency control. Locks can be acquired on particular objects. Some compliant implementations may either force or allow locks to be escalated to some other level of granularity.

The ODMG Object Model supports traditional pessimistic concurrency control as its default policy, but does not preclude a DBMS from supporting a wider range of concurrency control policies. The 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 DBMS allows the holder of the lock to proceed and the transaction that requested the conflicting lock waits until the holder completes. The holder may complete by either committing or aborting, at which point all its locks are released. Transactions subject to this protocol serialize in commit order.

2.8.4 Transaction Operations

There are two types that are defined to support transaction activity within an ODBMS, a Transaction, and a TransactionalThread. Once a Transaction object is created, to apply database operations to it within a thread, it must be associated with a TransactionalThread object. Exactly one TransactionalThread object is automatically created for each thread of control; it may be obtained from thread-specific data. A TransactionalThread object may have zero or one Transaction objects associated with it at any time.

An ODBMS provides a type Transaction with the following operations:

```
interface Transaction
    void        begin();
    void        commit();
    void        abort();
    void        checkpoint();
    boolean     active();
};
```

New Transaction objects are created using the new operation defined in ObjectFactory.

After a Transaction object is created it is initially inactive; an explicit begin operation is required to make it active. If a transaction is already active, additional begin operations will generate an error.

The commit operation causes all persistent objects created or modified during this transaction to be written to the database, and become accessible to other Transaction objects running against the database. All locks held by the Transaction object are released. Finally, it also causes the Transaction object to complete and become inactive. An error is generated if the Transaction object is inactive.

The abort operation causes the Transaction object to complete, and become inactive. The database will be returned to the state it was in prior to the beginning of the transaction. All locks held by the Transaction object are released. An error is generated if the Transaction object is inactive.

A checkpoint operation is equivalent to a commit operation followed by a begin operation, except that locks held by the Transaction object are NOT released. Therefore, it causes all modified objects to be committed to the database and it retains all locks held by the Transaction object. The Transaction object remains active.

The operations of the TransactionalThread object are described as follows:

```
interface TransactionalThread
    void            join(in Transaction tran);
    void            leave();
    Transaction     currentTrans();
};
```

Database operations are applied to the database during a transaction. Therefore, to execute any database operations, an active Transaction object must be associated with the current thread. The join operation associates the TransactionalThread object with a Transaction object. If the Transaction object is active, database operations may be executed, otherwise an error is generated. An error is generated when attempting to join a Transaction object which is already the current Transaction object.

If an implementation allows multiple active Transaction objects to exist, the join and leave operations allow a thread to alternate between them. To move to another Transaction object, a leave operation is first required to disassociate the TransactionalThread from the current Transaction object. This is followed by a join operation to again associate the thread with another Transaction object. Moving from one Transaction object to another does not commit or abort a Transaction object. When there is no current Transaction object, the leave operation is ignored.

After a Transaction object is completed, to continue executing database operations, either another active Transaction object must be associated with the TransactionalThread object, or a begin operation must be applied to the current Transaction object to make it active again.

Multiple threads of control in one address space can share the same transaction through multiple join operations on the same Transaction object. In this case, no locking is provided between these threads; concurrency control must be provided by the user. When any one of the threads executes a commit or abort operation against this Transaction object, it will complete.

Finally, if a child process is spawned from the current thread, by default it's TransactionalThread object will be associated with the Transaction object of the parent thread.

2.9 Database Operations

An ODBMS may manage one or more logical databases, each of which may be stored in one or more physical databases. Each logical database is an instance of the type Database, which is supplied by the ODBMS. The type Database supports the following operations:

```
interface Database {
    void        open(in string database_name);
    void        close();
    void        bind(in any an_object, in string name);
    any         lookup(in string object_name);
};
```

The open operation must be invoked, with a database name as its argument, before any access can be made to the persistent objects in the database. The Object Model requires only a single database to be open at a time. Implementations may extend this capability, including transactions that span multiple databases. The close operation must be invoked when a program has completed all access to the database. When the ODBMS closes a database, it performs necessary cleanup operations.

The lookup operation is used to find the identifier of the object with the name supplied as the argument to the operation. This operation is defined on the Database type, because the scope of object names is the database. The names of objects in the database, the names of types in the database schema, and the extents of types instantiated in the database are global to the database. They become accessible to a program once it has opened the database. Named objects are convenient entry points to the database. A name is bound to an object using the bind operation.

The Database type may also support operations designed for database administration, e.g., move, copy, reorganize, verify, backup, restore. These kinds of operations are not specified here, as they are considered an implementation consideration outside the scope of the Object Model.

2.10 Possible Future Revisions

The current ODMG Object Model is intended to be extensible to support additional functionality through successive releases of the ODMG standard. In the interest of achieving a standard we have intentionally restricted the functionality of the Object

Model. However, there are many areas of additional functionality we could support. This section outlines the areas we expect to consider in future revisions. This section is included for two reasons: first, to let reviewers of the base model know what functionality has been considered and deferred, and second, to allow implementors to begin developing pilot implementations in some of these areas. These pilots will give us a solid body of implementation experience on which to base our discussions of incorporation of new functionality.

2.10.1 Types, Extents

1. Multiple implementations are visible to the programmer. An implementation may be chosen at object creation time.
2. Indices may be dynamically created and destroyed either by explicit programmer request or by the query processor if indices would be useful.
3. Multiple types may share one implementation.
4. Instance properties may be defined on types, e.g., the *color* attribute inherited from type *Bird* is defined to be “red” for birds of subtype *Cardinal*.
5. Named sub-extents of a type may be defined. If a predicate determining instance membership in a particular sub-extent is specified in the type declaration, then the ODBMS automatically maintains the sub-extent as well as, or in lieu of, the full extent.

2.10.2 Objects

1. An object may be an instance of more than one type.
2. An object may dynamically acquire/lose a type.
3. An object may dynamically change representations (i.e., implementations).
4. An object’s lifetime may be changed. The typical use for this is for a transient object to become persistent.
5. Object versions are supported. Configurations containing specific versions of component objects are supported.
6. Versions of types are supported.
7. The type programmer may extend or replace the built-in collection types.

2.10.3 Attributes

1. Attributes may have properties.
2. Attribute types may participate in supertype/subtype relationships with other attribute types.
3. Integrity constraints can be declared on attribute types.

2.10.4 Relationships

1. Relationships become first-class objects.
2. Relationships may have properties, e.g., transitive, reflexive, or other attributes.
3. Relationship types may participate in supertype/subtype relationships with other relationship types.
4. The *consists_of* relationship is supported, with predefined *delete, move, copy ...* semantics.
5. An object may acquire attribute values by virtue of its participation in a relationship, e.g., *Course.enrollment* defines the number of students enrolled in a course, *Student.course_load* defines the number of courses a student is currently taking.
6. The type definer may supply an implementation for a relationship type rather than using one of the built-in implementations.

2.10.5 Operations

1. Remote operations are explicitly supported.
2. Parallel operations are explicitly supported.
3. Free-standing operations, not defined on a single type, are supported.

2.10.6 Transactions

1. Long-lived transactions are supported.
2. A single transaction may access objects in more than one database (distributed transactions and XA protocol).
3. More than one process may participate in a single transaction.
4. Transaction consistency applies to transient as well as persistent objects.
5. Nested transactions are supported.

2.10.7 Databases

1. Versions of database schemas are supported.
2. Subschemas are supported.
3. Versions of subschemas are supported.
4. Access control and security considerations are supported.
5. Type definitions are treated as objects.
6. Metadata is exposed as a predefined schema.
7. Hierarchical name spaces are supported.