# Chapter 2

# Object Model

## 2.1 Introduction

This chapter defines the Object Model supported by ODMG-compliant object data management systems (ODMSs). The Object Model is important because it specifies the kinds of semantics that can be defined explicitly to an ODMS. 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 programming language–independent object specification languages. One such specification language, Object Definition Language (ODL), is used to specify application object models and 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, 6, and 7, respectively, define the C++, Smalltalk, and Java 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. Modifications are explained in Chapters 5, 6, and 7.

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

- The basic modeling primitives are the *object* and the *literal*. Each object has a unique identifier. A literal has no identifier.
- 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.
- 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. Operations may have a list of input and output parameters, each with a specified type. Each operation may also return a typed result.
- An *ODMS* stores objects, enabling them to be shared by multiple users and applications. An ODMS 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 ODMS's (logical) schema. The ODMG Object Model is the fundamental definition of an ODMS's functionality. It includes significantly richer semantics than does the relational model, by declaring relationships and operations explicitly.

## 2.2 Types: Specifications and Implementations

There are two aspects to the definition of a type. A type has an external *specification* and one or more *implementations*. The specification defines the external characteristics of the type. These are the aspects that are visible to users of the type: the *operations* that can be invoked on its instances, the *properties*, or state variables, whose values can be accessed, and any *exceptions* that can be raised by its operations. By contrast, a type's implementation defines the internal aspects of the objects of the type: the implementation of the type's operations and other internal details. The implementation of a type is determined by a language binding.

An external specification of a type consists of an implementation-independent, abstract description of the operations, exceptions, and properties that are visible to users of the type. An *interface* definition is a specification that defines only the abstract behavior of an object type. A *class* definition is a specification that defines the abstract behavior and abstract state of an object type. A *class* is an extended interface with information for ODMS schema definition. A *literal* definition defines only the abstract state of a literal type. Type specifications are illustrated in Figure 2-1.

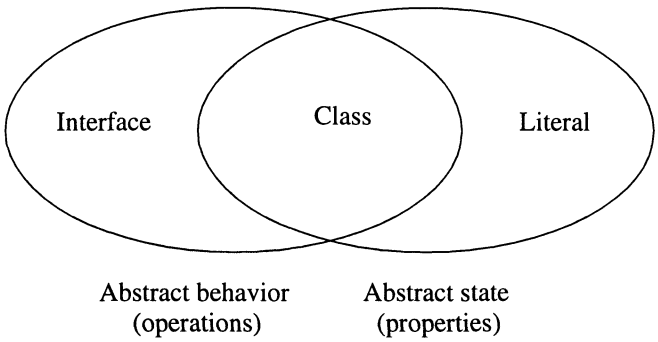


Figure 2-1. Type Specifications

For example, interface Employee defines only the abstract behavior of Employee objects. Class Person defines both the abstract behavior and the abstract state of Person objects. Finally, the struct Complex defines only the abstract state of Complex number literals. In addition to the struct definition and the primitive literal datatypes (boolean, char, short, long, float, double, octet, and string), ODL defines declarations for user-defined collection, union, and enumeration literal types.

```
interface Employee {...};
class Person {...};
struct Complex {float re; float im; };
```

An implementation of an object type consists of a *representation* and a set of *methods*. The representation is a data structure that is derived from the type's abstract state by a *language binding*: For each property contained in the abstract state there is an instance variable of an appropriate type defined. The methods are procedure bodies that are derived from the type's abstract behavior by the language binding: For each of the operations defined in the type's abstract behavior a method is defined. This method implements the externally visible behavior of an object type. 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 in an implementation that have no direct counterpart to the operations in the type's specification. The internals of an implementation are not visible to the users of the objects.

Each language binding also defines an implementation mapping for literal types. Some languages have constructs that can be used to represent literals directly. For example, C++ has a structure definition that can be used to represent the above Complex literal directly using language features. Other languages, notably Smalltalk and Java, have no direct language mechanisms to represent structured literals. These language bindings map each literal type into constructs that can be directly supported using object classes. Further, since both C++ and Java have language mechanisms for directly handling floating-point datatypes, these languages would bind the float elements of Complex literals accordingly. Finally, Smalltalk binds these fields to instances of the class Float. As there is no way to specify the abstract behavior of literal types, programmers in each language will use different operators to access these values.

The distinction between specification and implementation views 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 external specifications of types in application object models. The language bindings of Chapters 5, 6, and 7, respectively, define the C++, Smalltalk, and Java constructs used to specify the implementations of these specifications.

A type can have more than one implementation, although only one implementation is usually 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 specifications from the implementations keeps the semantics of the type from becoming tangled with representation details. Separating the specifications from the implementations is a positive step toward multilingual access to objects of a single type and sharing of objects across heterogeneous computing environments.

Many object-oriented programming languages, including C++, Java, and Smalltalk, have language constructs called classes. These are implementation classes and are not to be confused with the *abstract classes* defined in the Object Model. Each language binding defines a mapping between abstract classes and its language's implementation classes.

### 2.2.1 Subtyping and Inheritance of Behavior

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, etc.). 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

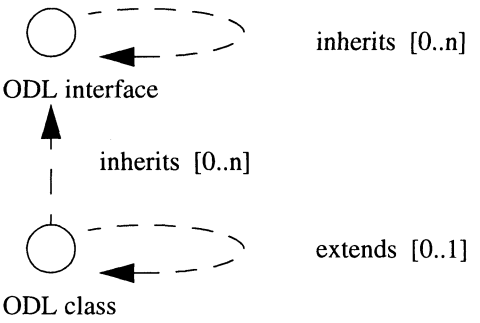


Figure 2-2. Class-Interface Relationships

specialize state and behavior. For example, the Employee type might have an operation for calculate_ paycheck. The Salaried_Employee and Hourly_Employee class implementations 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.

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

The ODMG Object Model supports multiple inheritance of object behavior. Therefore, it is possible that a type could inherit operations that have the same name, but different parameters, from two different interfaces. The model precludes this possibility by disallowing name overloading during inheritance.

ODL classes are mapped by a language binding to classes of a programming language that are directly instantiable. Interfaces are types that cannot be directly instantiated. For example, instances of the classes Salaried_Employee and Hourly_Employee may be created, but instances of their supertype interface Employee cannot. Subtyping pertains to the inheritance of behavior only; thus, interfaces may inherit from other interfaces and classes may also inherit from interfaces. Due to the inefficiencies and ambiguities of multiple inheritance of state, however, interfaces may not inherit from classes, nor may classes inherit from other classes. These relationships are illustrated in Figure 2-2.

### 2.2.2 Inheritance of State

In addition to the ISA relationship that defines the inheritance of behavior between object types, the ODMG Object Model defines an EXTENDS relationship for the inheritance of state and behavior. The EXTENDS relationship also applies only to *object* types; thus, only classes and not literals may inherit state. The EXTENDS relationship is a single inheritance relationship between two classes whereby the subordinate class inherits all of the properties and all of the behavior of the class that it extends.

```
class Person {
    attribute string name;
    attribute Date birthDate;
    };
// in the following, the colon denotes the ISA relationship
// the extends denotes the EXTENDS relationship
class EmployeePerson extends Person : Employee {
    attribute Date hireDate;
    attribute Currency payRate;
    relationship Manager boss inverse Manager::subordinates;
    };
class ManagerPerson extends EmployeePerson : Manager {
    relationship set<Employee> subordinates
        inverse Employee::boss;
    };
```

The EXTENDS relationship is transitive; thus, in the example, every ManagerPerson would have a name, a birthDate, a hireDate, a payRate, and a boss. Note also that, since class EmployeePerson inherits behavior from (ISA) Employee, instances of EmployeePerson and ManagerPerson would all support the behavior defined within this interface.

The only legal exception to the name-overloading prohibition occurs when the same property declaration occurs in a class and in one of its inherited interfaces. Since the properties declared within an interface also have a procedural interface, such redundant declarations are useful in situations where it is desirable to allow relationships to cross distribution boundaries, yet they also constitute part of the abstract state of the object (see Section 2.6 on page 37 for information about the properties and behavior that can be defined for atomic objects). In the previous example, it would be permissible (and actually necessary) for the interfaces Employee and Manager to contain copies of the boss/subordinates relationship declarations, respectively. It would also be permissible for the interface Employee to contain the hireDate and/or payRate attributes if distributed access to these state variables was desired.

### 2.2.3 Extents

The *extent* of a type is the set of all instances of the type within a particular ODMS. 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 ODMS schema designer can decide whether the system 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 schema designer specifies that the extent should be indexed separately from specifying that the extent should be maintained by the ODMS.

### 2.2.4 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:

- Creation, which refers to the manner in which objects are created by the programmer.
- Identifiers, which are used by an ODMS 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.
- 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 composed of other objects.

All of the object definitions, defined in this chapter, are to be grouped into an enclosing module that defines a name scope for the types of the model.

```
module ODLTypes {
    exception    DatabaseClosed{};
    exception    TransactionInProgress{};
    exception    TransactionNotInProgress{};
    exception    IntegrityError{};
    exception    LockNotGranted{};

    // the following interfaces and classes are defined here
};
```

### 2.3.1 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. The new operation, defined below, causes the creation of a new instance of an object of the Object type.

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

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

```
interface Object {
    enum        Lock_Type{read, write, upgrade};
    void        lock(in Lock_Type mode) raises(LockNotGranted);
    boolean     try_lock(in Lock_Type mode);
    boolean     same_as(in Object anObject);
    Object      copy();
    void        delete();
};
```

Identity comparisons of objects are achieved using the same_as operation. The copy operation creates a new object that is equivalent to the receiver object. The new object created is not the "same as" the original object (the same_as operation is an identity test). Objects, once created, are explicitly deleted from the ODMS using the delete operation. This operation will remove the object from memory, in addition to the ODMS.

While the default locking policy of ODMG objects is implicit, all ODMG objects also support explicit locking operations. The lock operation explicitly obtains a specific lock on an object. If an attempt is made to acquire a lock on an object that conflicts with that object's existing locks, the lock operation will block until the specified lock can be acquired, some time-out threshold is exceeded, or a transaction deadlock is detected. If the time-out threshold is crossed, the LockNotGranted exception is raised. If a transaction deadlock is detected, the transaction deadlock exception is raised. The try_lock operation will attempt to acquire the specified lock and immediately return a boolean specifying whether the lock was obtained. The try_lock operation will return TRUE if the specified lock was obtained and FALSE if the lock to be obtained is in conflict with an existing lock on that object. See Section 2.9 for additional information on locking and concurrency.

The IntegrityError exception is raised by operations on relationships and signifies that referential integrity has been violated. See Section 2.6.2 for more information on this topic.

Any access, creation, modification, and deletion of persistent objects must be done within the scope of a transaction. If attempted outside the scope of a transaction, the TransactionNotInProgress exception is raised. For simplicity in notation, it is assumed that all operations defined on persistent objects in this chapter have the ability to raise the TransactionNotInProgress exception.

### 2.3.2 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 an ODMS. All identifiers of objects in an ODMS are unique, relative to each other. The representation of the identity of an object is referred to as its *object identifier*. 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 ODMS, 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.

### 2.3.3 Object Names

In addition to being assigned an object identifier by the ODMS, an object may be given one or more names that are meaningful to the programmer or end user. The ODMS provides a function that it uses to map from an object name to an object. The application can refer at its convenience to an object by name; the ODMS 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 the ODMS.

Object names are like global variable names in programming languages. They are not the same as keys. A key is composed 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 an ODMS. The Object Model does not include a notion of hierarchical name spaces within an ODMS or of name spaces that span ODMSs.

### 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 runtime 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 runtime 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 ODMS runtime system. These objects continue to exist after the procedure or process that creates them terminates. 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 Sections 2.6 and 2.7 for information about the properties and behavior that can be defined for atomic objects.

### 2.3.6 Collection Objects

In the ODMG Object Model, instances of *collection objects* are composed of 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 collections 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 Object to represent these typed parameters, recognizing that this can imply a heterogeneity that is not the intent of this object model.

Collections are created by invoking the operations on the factory interfaces defined for each particular collection. The new operation, inherited from the ObjectFactory interface, 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.

Collections all have the following operations:

```
interface Collection : Object {
    exception                InvalidCollectionType{};
    exception                ElementNotFound{Object element; };
    unsigned long            cardinality();
    boolean                  is_empty();
    boolean                  is_ordered();
    boolean                  allows_duplicates();
    boolean                  contains_element(in Object element);
    void                     insert_element(in Object element);
    void                     remove_element(in Object element)
                                 raises(ElementNotFound);
    Iterator                 create_iterator(in boolean stable);
    BidirectionalIterator    create_bidirectional_iterator(in boolean stable)
                                 raises(InvalidCollectionType);
    Object                   select_element(in string OQL_predicate);
    Iterator                 select(in string OQL_predicate);
    boolean                  query(in string OQL_predicate,
                                     inout Collection result);
    boolean                  exists_element(in string OQL_predicate);
};
```

The number of elements contained in a collection is obtained using the cardinality operation. The operations is_empty, is_ordered, and allows_duplicates provide a means for dynamically querying a collection to obtain its characteristics. Element management within a collection is supported via the insert_element, remove_element, and contains_element operations. The create_iterator and create_bidirectional_iterator operations support the traversal of elements within a collection (see Iterator interface below). The select_element, select, query, and exists_element operations are used to evaluate OQL predicates upon the contents of a collection. The boolean results of the query and exists_element operations indicate whether any elements were found as a result of performing the OQL query.

In addition to the operations defined in the Collection interface, Collection objects also inherit operations defined in the Object interface. Identity comparisons are determined using the same_as operation. A copy of a collection 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). The delete operation removes the collection from the ODMS and, if the collection contains literals, also deletes the contents of the collection. However, if the collection contains objects, the collection remains unchanged.

An Iterator, which is a mechanism for accessing the elements of a Collection object, can be created to traverse a collection. The following operations are defined in the Iterator interface:

```
interface Iterator {
    exception       NoMoreElements{};
    exception       InvalidCollectionType{};
    boolean         is_stable();
    boolean         at_end();
    void            reset();
    Object          get_element() raises(NoMoreElements);
    void            next_position() raises(NoMoreElements);
    void            replace_element (in Object element)
                        raises(InvalidCollectionType);
};
interface BidirectionalIterator : Iterator {
    boolean         at_beginning();
    void            previous_position() raises(NoMoreElements);
};
```

The create_iterator and create_bidirectional_iterator operations create iterators that 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 retrieves the element currently pointed to by the iterator. The next_position operation increments the iterator to the next element in the iteration. The previous_position operation decrements the iterator to the previous element in the iteration. The replace_element operation, valid when iterating over List and Array objects, replaces the element currently pointed to by the iterator with the argument passed to the operation. The reset operation repositions the iterator to the first element in the iteration.

#### 2.3.6.1 Set Objects

A Set object is an unordered collection of elements, with no duplicates allowed. The following operations are defined in the Set interface:

```
interface SetFactory : ObjectFactory {
    Set             new_of_size(in long size);
};
class Set : Collection {
    attribute       set<t> value;
    Set             create_union(in Set other_set);
    Set             create_intersection(in Set other_set);
    Set             create_difference(in Set other_set);
    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 Set type interface has the conventional mathematical set operations, as well as subsetting and supersetting boolean tests. The create_union, create_intersection, and create_difference operations each return a new result Set object.

Set refines the semantics of the insert_element operation inherited from its Collection supertype. If the object passed as the argument to the insert_element operation is not already a member of the set, the object is added to the set. Otherwise, the set remains unchanged.

#### 2.3.6.2 Bag Objects

A Bag object is an unordered collection of elements that may contain duplicates. The following interfaces are defined in the Bag interface:

```
interface BagFactory : ObjectFactory {
    Bag             new_of_size(in long size);
};
class Bag : Collection {
    attribute       bag<t>value;
    unsigned long   occurrences_of(in Object element);
    Bag             create_union(in Bag other_bag);
    Bag             create_intersection(in Bag other_bag);
    Bag             create_difference(in Bag other_bag);
};
```

The occurrences_of operation calculates the number of times a specific element occurs in the Bag. The create_union, create_intersection, and create_difference operations each return a new result Bag object.

Bag refines the semantics of the insert_element and remove_element operations inherited from its Collection supertype. 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 the multiplicity in the bag. The remove_element operation removes one occurrence of the specified element from the bag.

#### 2.3.6.3 List Objects

A List object is an ordered collection of elements. The operations defined in the List interface are positional in nature, in reference either to a given index or to the beginning or end of a List object. Indexing of a List object starts at zero. The following operations are defined in the List interface:

```
interface ListFactory : ObjectFactory {
    List            new_of_size(in long size);
};
class List : Collection {
    exception   InvalidIndex{unsigned long index; };
    attribute   list<t>value;
    void        remove_element_at(in unsigned long index)
                    raises(InvalidIndex);
    Object      retrieve_element_at(in unsigned long index)
                    raises(InvalidIndex);
    void        replace_element_at(in Object element, in unsigned long index)
                    raises(InvalidIndex);
    void        insert_element_after(in Object element, in unsigned long index)
                    raises(InvalidIndex);
    void        insert_element_before(in Object element, in unsigned long index)
                    raises(InvalidIndex);
    void        insert_element_first (in Object element);
    void        insert_element_last (in Object element);
    void        remove_first_element()
                    raises(ElementNotFound);
    void        remove_last_element()
                    raises(ElementNotFound);
    Object      retrieve_first_element()
                    raises(ElementNotFound);
    Object      retrieve_last_element()
                    raises(ElementNotFound);
    List        concat(in List other_list);
    void        append(in List other_list);
};
```

The List interface defines operations for selecting, updating, and deleting elements from a list. In addition, operations that manipulate multiple lists are defined. The concat

operation returns a new List object that contains the list passed as an argument appended to the receiver list. Both the receiver list and argument list remain unchanged. The append operation modifies the receiver list by appending the argument list.

List refines the semantics of the insert_element and remove_element operations inherited from its Collection supertype. The insert_element operation inserts the specified object at the end of the list. The semantics of this operation are equivalent to the list operation insert_element_last. The remove_element operation removes the first occurrence of the specified object from the list.

#### 2.3.6.4 Array Objects

An Array object is a dynamically sized, ordered collection of elements that can be located by position. The following operations are defined in the Array interface:

```
interface ArrayFactory : ObjectFactory {
    Array           new_of_size(in long size);
};
class Array : Collection {
    exception   InvalidIndex{unsigned long index; };
    exception   InvalidSize{unsigned long size; };
    attribute   array<t> value;
    void        replace_element_at(in unsigned long index, in Object element)
                    raises(InvalidIndex);
    void        remove_element_at(in unsigned long index)
                    raises(InvalidIndex);
    Object      retrieve_element_at(in unsigned long index)
                    raises(InvalidIndex);
    void        resize(in unsigned long new_size)
                    raises(InvalidSize);
};
```

The remove_element_at operation replaces any current element contained in the cell of the Array object identified by index with an undefined 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. The exception InvalidSize is raised, by the resize operation, if the value of the new_size parameter is smaller than the actual number of elements currently contained in the array.

Array refines the semantics of the insert_element and remove_element operations inherited from its Collection supertype. The insert_element operation increases the size

of the array by one and inserts the specified object in the new position. The remove_element operation replaces the first occurrence of the specified object in the array with an undefined value.

#### 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 {Object key; Object value; };
```

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.

The following operations are defined in the Dictionary interface:

```
interface DictionaryFactory : ObjectFactory {
    Dictionary          new_of_size(in long size);
};
class Dictionary : Collection {
    exception           DuplicateName{string key; };
    exception           KeyNotFound{Object key; };
    attribute           dictionary<t,v>value;
    void                bind(in Object key, in Object value)
                            raises(DuplicateName);
    void                unbind(in Object key) raises(KeyNotFound);
    Object              lookup(in Object key) raises(KeyNotFound);
    boolean             contains_key(in Object key);
};
```

Inserting, deleting, and selecting entries in a Dictionary object are achieved using the bind, unbind, and lookup operations, respectively. The contains_key operation tests for the existence of a specific key in the Dictionary object.

Dictionary refines the semantics of the insert_element, remove_element, and contains_element operations inherited from its Collection supertype. All of these operations are valid for Dictionary types when an Association is specified as the argument. The insert_element operation inserts an entry into the Dictionary that reflects the key-value pair contained in the Association parameter. If the key already resides in the Dictionary, the existing entry is replaced. The remove_element operation removes the entry from the Dictionary that matches the key-value pair contained in the Association passed as an argument. If a matching key-value pair entry is not found in the Dictionary, the ElementNotFound exception is raised. Similarly, the contains_element operation also uses both the key and value contained in the Association argument to locate

a particular entry in the Dictionary object. A boolean is returned specifying whether the key-value pair exists in the Dictionary.

### 2.3.7 Structured Objects

All *structured objects* support the Object ODL interface. The ODMG Object Model defines the following structured objects:

- Date
- Interval
- Time
- Timestamp

These types are defined as in the INCITS SQL specification by the following interfaces.

#### 2.3.7.1 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_date(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:

```
class Date : Object {
    enum            Weekday {Sunday, Monday, Tuesday, Wednesday,
                             Thursday, Friday, Saturday};
    enum            Month {January, February, March, April, May, June, July,
                           August, September, October, November,
                           December};
    attribute       date     value;
    unsigned short  year();
    unsigned short  month();
    unsigned short  day();
    unsigned short  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 long days);
    Date            subtract_days(in long days);
    long            subtract_date(in Date a_date);
};
```

#### 2.3.7.2 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:

```
class Interval : Object {
    attribute          interval  value;
    unsigned short     day();
    unsigned short     hour();
    unsigned short     minute();
    unsigned short     second();
    unsigned short     millisecond();
    boolean            is_zero();
    Interval           plus(in Interval an_interval);
    Interval           minus(in Interval an_interval);
    Interval           product(in long val);
    Interval           quotient(in long 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);
};
```

#### 2.3.7.3 Time

Times denote specific world times, which are internally stored in Greenwich Mean Time (GMT). Time zones are specified according to the number of hours that must be added or subtracted from local time in order to get the time in Greenwich, England.

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

```
interface TimeFactory : ObjectFactory {
    void               set_default_time_zone(in TimeZone a_time_zone);
    TimeZone           default_time_zone();
    TimeZone           time_zone();
    Time               from_hmsm(in unsigned short hour,
                                 in unsigned short minute,
                                 in unsigned short second,
                                 in unsigned short millisecond);
    Time               from_hmsmtz(in unsigned short hour,
                                 in unsigned short minute,
                                 in unsigned short second,
                                 in unsigned short millisecond,
                                 in short tzhour,
                                 in short tzminute);
    Time               current();
};
```

The following interface defines the operations on Time objects:

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

```
        unsigned short    hour();
        unsigned short    minute();
        unsigned short    second();
        unsigned short    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);
};
```

#### 2.3.7.4 Timestamp

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

```
interface TimestampFactory : ObjectFactory {
    exception         InvalidTimestamp{Date a_date, Time a_time; };
    Timestamp         current();
    Timestamp         create(in Date a_date, in Time a_time)
                          raises(InvalidTimestamp);
};
```

The following interface defines the operations on Timestamp objects:

```
class Timestamp : Object {
    attribute       timestamp   value;
    Date            get_date();
    Time            get_time();
    unsigned short  year();
    unsigned short  month();
    unsigned short  day();
    unsigned short  hour();
    unsigned short  minute();
    unsigned short  second();
    unsigned short  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 Literals

This section considers each of the following aspects of literals:

- types, which includes a description of the types of literals supported by the standard
- copying, which refers to the manner in which literals are copied
- comparing, which refers to the manner in which literals are compared
- equivalence, which includes the method for determining when two literals are equivalent

### 2.4.1 Literal Types

The Object Model supports the following literal types:

- atomic literal
- collection literal
- structured literal

#### 2.4.1.1 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
- long 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};
```

#### 2.4.1.2 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.

### 2.4.1.3 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, for example, address.zip_code = 12345; address.city = "San Francisco". Structure types supported by the ODMG Object Model include

- date
- interval
- time
- timestamp

#### *2.4.1.3.1 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:

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

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.

Each language binding will map the Object Model structures and collections to mechanisms that are provided by the programming language. For example, Smalltalk includes its own Collection, Date, Time, and Timestamp classes.

### 2.4.2 Copying Literals

Literals do not have object identifiers and, therefore, cannot be shared. However, literals do have copy semantics. For example, when iterating through a collection of literals, copies of the elements are returned. Likewise, when returning a literal-valued attribute of an object, a copy of the literal value is returned.

### 2.4.3 Comparing Literals

Since literals do not have object identifiers (not objects), they cannot be compared by identity (i.e., the same_as operation). As a result, they are compared using the equals equivalence operation. This becomes important for collection management. For example, when inserting, removing, or testing for membership in a collection of literals, the equivalence operation equals is used rather than the identity operation same_as.

### 2.4.4 Literal Equivalence

Two literals, **x** and **y**, are considered equivalent (or equal) if they have the same literal type and

- are both atomic and contain the same value
- are both sets, have the same parameter type **t**, and
  - if **t** is a literal type, then for each element in **x**, there is an element in **y** that is equivalent to it, and, for each element in **y**, there is an element in **x** that is equivalent to it
  - if **t** is an Object type, then both **x** and **y** contain the same set of object identifiers
- are both bags, have the same parameter type **t**, and
  - if **t** is a literal type, then for each element in **x**, there is an element in **y** that is equivalent to it, and, for each element in **y**, there is an element in **x** that is equivalent to it. In addition, for each literal appearing more than once in **x**, there is an equivalent literal occurring the same number of times in **y**
  - if **t** is an Object type, then both **x** and **y** contain the same set of object identifiers. In addition, for each object identifier appearing more than once in **x**, there is an identical object identifier appearing the same number of times in **y**

- are both arrays or lists, have the same parameter type **t**, and for each entry **i**
  - if **t** is a literal type, then x[i] is equivalent to y[i] (equal)
  - if **t** is an object type, then x[i] is identical to y[i] (same_as)
- are both dictionary literals, and when considered sets of associations, the two sets are equivalent
- are both structs of the same type, and for each element **j**
  - if the element is a literal type, then **x.j** and **y.j** are equivalent (equal)
  - if the element is an object type, then **x.j** and **y.j** are identical (same_as)

## 2.5 The Full Built-in Type Hierarchy

Figure 2-3 shows the full set of built-in types of the Object Model type hierarchy. Concrete types are shown in nonitalic font and are directly instantiable. Abstract types are shown in italics. In the interest of simplifying matters, both types and type generators are included in the same hierarchy. Type generators are signified by angle brackets (e.g., Set<>).

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, that is, long, short, unsigned long, unsigned short, float, double, boolean, octet, and char. No implicit conversions are provided for structured literals.

*Literal_type*
*Atomic_literal*
**long**
**long long**
**short**
**unsigned long**
**unsigned short**
**float**
**double**
**boolean**
**octet**
**char**
**string**
**enum<>**
*Collection_literal*
**set<>**
**bag<>**
**list<>**
**array<>**
**dictionary<>**
*Structured_literal*
**date**
**time**
**timestamp**
**interval**
**structure<>**
*Object_type*
*Atomic_object*
*Collection_object*
**Set<>**
**Bag<>**
**List<>**
**Array<>**
**Dictionary<>**
*Structured_object*
**Date**
**Time**
**Timestamp**
**Interval**

Figure 2-3. Full Set of Built-in Types

## 2.6 Modeling State—Properties

A class defines a set of properties through which users can access, and in some cases directly manipulate, the state of instances of the class. 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.

### 2.6.1 Attributes

The attribute declarations in a class define the abstract state of its instances. For example, the class Person might contain the following attribute declarations:

```
class 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.

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.

In contrast, attribute declarations in an interface define only abstract behavior of its instances. 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, if the age operation were defined in an interface, the presence of this attribute would not imply state, but rather the ability to compute the age (e.g., from the birthdate of the person). For example:

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

### 2.6.2 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 person can have a one-to-many *parent 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.

A relationship is defined explicitly 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 relationship. For example, a professor *teaches* courses and a course *is taught by* a professor. The teaches traversal path would be defined in the declaration for the Professor type. The is_taught_by traversal path would be defined in the 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:

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

and

```
class 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 ODMS 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 nonexistent 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. While object-valued attributes may be used to implement so-called unidirectional relationships, such constructions are not considered to be true relationships in this standard. Relationships always guarantee referential integrity.

It is important to note that a relationship traversal path is not equivalent to a pointer. A pointer in C++, or an object reference in Smalltalk or Java, has no connotation of a corresponding inverse traversal path that 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 that *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 previously 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.

As in attributes, declarations of relationships that occur within classes define abstract state for storing the relationship and a set of operations for accessing the relationship. Declarations that occur within interfaces define only the operations of the relationship, not the state.

#### 2.6.2.1 Cardinality "One" Relationships

For relationships with cardinality "one" such as

```
relationship    X   Y   inverse Z;
```

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

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

For example, the relationship in the preceding 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)
                    raises(IntegrityError);
void            drop_is_taught_by(in Professor aProfessor)
                    raises(IntegrityError);
```

#### 2.6.2.2 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 that already contains a reference to the given target X. This exception, if it occurs, will also be raised by the form_Y operation that invoked the add_Y. For example:

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

The relationship in the preceding 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)
                        raises(IntegrityError);
void                  add_teaches(in Course aCourse)
                        raises(IntegrityError);
void                  remove_teaches(in Course aCourse)
                        raises(IntegrityError);
```

## 2.7 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++, Smalltalk, and Java programming languages. This consistency enables seamless integration of ODMSs 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 ODMS from taking advantage of multiprocessor support.

### 2.7.1 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. Mappings for exceptions are defined by each language binding. When an exception is raised, information on the cause of the exception is passed back to the exception handler as properties of the exception. 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 runtime system out to the level of the handler. Memory is freed for all objects allocated in intervening stack frames. Any transactions begun within a nested scope, that is, unwound by the runtime 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.

## 2.8 Metadata

Metadata is descriptive information about persistent objects that defines the *schema* of an ODMS. Metadata is used by the ODMS to define the structure of its object storage, and at runtime, guides access to the ODMS's persistent objects. 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* that 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 that programmers can use to correctly construct valid

schemas, several creation, addition, and removal operations are defined that 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 that defines a name scope for the elements of the model.

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

### 2.8.1 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 unbind operation for removing bindings.

```
interface Scope {
    exception          DuplicateName{};
    exception          NameNotFound{string reason; };
    void               bind(in string name, in MetaObject value)
                           raises(DuplicateName);
    MetaObject         resolve(in string name) raises(NameNotFound);
    void               unbind(in string name) raises(NameNotFound);
    list<RepositoryObject>  children();
};
```

### 2.8.2 Visitors

Visitors provide a convenient "double dispatch" mechanism for traversing the meta objects in the repository. To utilize this mechanism, a client must implement a RepositoryObjectVisitor object that responds to the visit_... callbacks in an appropriate manner. Then, by passing this visitor to one of the meta objects in the repository, an appropriate callback will occur that may be used as required by the client object.

```
enum MetaKind {mk_attribute, mk_class, mk_collection, mk_constant,
               mk_const_operand, mk_enumeration, mk_exception,
               mk_expression, mk_interface, mk_literal, mk_member,
               mk_module, mk_operation, mk_parameter, mk_primitive_type,
               mk_relationship, mk_repository, mk_structure,
               mk_type_definition, mk_union, mk_union_case };
interface RepositoryObject {
    void accept_visitor(in RepositoryObjectVisitor a_repository_object_visitor);
    Scope parent();
    readonly attribute MetaKind meta_kind;
};
```

```
interface RepositoryObjectVisitor {
    void visit_attribute(in Attribute an_attribute);
    void visit_class(in Class a_class);
    void visit_collection(in Collection a_collection);
    void visit_constant(in Constant a_constant);
    void visit_const_operand(in ConstOperand a_const_operand);
    void visit_enumeration(in Enumeration an_enumeration);
    void visit_exception(in Exception an_exception);
    void visit_expression(in Expression an_expression);
    void visit_interface(in Interface an_interface);
    void visit_literal(in Literal a_literal);
    void visit_member(in Member a_member);
    void visit_module(in Module a_module);
    void visit_operation(in Operation an_operation);
    void visit_parameter(in Parameter a_parameter);
    void visit_primitive_type(in PrimitiveType a_primitive_type);
    void visit_relationship(in Relationship a_relationship);
    void visit_repository(in Repository a_repository);
    void visit_structure(in Structure a_structure);
    void visit_type_definition(in TypeDefinition a_type_definition);
    void visit_union(in Union an_union);
    void visit_union_case(in UnionCase an_union_case);
};
```

### 2.8.3 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 that contain other meta object definitions using their defines relationship and that have operations for creating, adding, and removing meta objects within themselves.

```
typedef string ScopedName;

interface MetaObject : RepositoryObject {
    attribute       string    name;
    attribute       string    comment;
    relationship    DefiningScope    definedIn
                        inverse DefiningScope::defines;
    ScopedName      absolute_name();
};
```

```
enum    PrimitiveKind {pk_boolean, pk_char, pk_date, pk_short,
                       pk_unsigned_short, pk_date, pk_time, pk_timestamp,
                       pk_long, pk_unsigned_long, pk_long_long, pk_float,
                       pk_double, pk_octet, pk_interval, pk_void};
enum    CollectionKind {ck_list, ck_array, ck_bag, ck_set, ck_dictionary,
                        ck_sequence, ck_string };
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 primitive_kind);
    Collection        create_collection(in CollectionKind collection_kind,
                          in Operand max_size, in Type sub_type);
    Dictionary        create_dictionary_type(in Type key_type,
                          in Type sub_type);
    Operand           create_operand(in string expression)
                          raises(InvalidExpression);
    Member            create_member(in string member_name,
                          in Type member_type);
    UnionCase         create_union_case(in string case_name,
                          in Type case_type,
                          in list<Operand> caseLabels)
                          raises(DuplicateName, InvalidType);
    Constant          add_constant(in string name, in Type type,
                          in Operand value)
                          raises(DuplicateName);
    TypeDefinition    add_type_definition(in string name, in Type alias)
                          raises(DuplicateName);
    Enumeration       add_enumeration(in string name,
                          in list<string> element_names)
                          raises(DuplicateName, InvalidType);
    Structure         add_structure(in string name, in list<Member> fields)
                          raises(DuplicateName, InvalidType);
    Union             add_union(in string name, In Type switch_type,
                          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_type_definition(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);
};
```

#### 2.8.3.1 Modules

Modules and the Schema Repository itself, which is a specialized module, are DefiningScopes that 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);
    Class       add_class(in string name, in list<Interface> inherits,
                    in Class extender)
                    raises(DuplicateName);
    void        remove_module(in Module object) raises(CannotRemove);
    void        remove_interface(in Interface object) raises(CannotRemove);
    void        remove_class(in Class object) raises(CannotRemove);
};
interface Repository : Module {};
```

#### 2.8.3.2 Operations

Operations model the behavior that application objects support. They maintain a signature list of Parameters and refer to a result type. Operations may raise Exceptions.

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

#### 2.8.3.3 Exceptions

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

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

#### 2.8.3.4 Constants

Constants provide a mechanism for statically associating values with names in the repository. The value is defined by an Operand subclass that 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 that 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                the_Value
                        inverse Operand::value_of;
    relationship    Type                   type
                        inverse Type::constants;
    relationship    set<ConstOperand>      referenced_by
                        inverse ConstOperand::references;
    relationship    Enumeration            enumeration
                        inverse Enumeration::elements;
    Object          value();
};
```

#### 2.8.3.5 Properties

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

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

#### *2.8.3.5.1 Attributes*

Attributes are properties that maintain simple abstract state. They may be read-only, in which case there is no associated accessor for changing their values.

```
interface Attribute : Property {
    attribute       boolean         is_read_only;
};
```

#### *2.8.3.5.2 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 {
    relationship    Relationship    traversal
                        inverse Relationship::traversal;
    Cardinality     get_cardinality();
};
```

### 2.8.3.6 Types

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

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

Type meta objects are used to represent information about datatypes. They participate in a number of relationships with the other meta objects that 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<Dictionary>     dictionaries
                        inverse Dictionary::key_type;
    relationship    set<Specifier>      specifiers
                        inverse Specifier::type;
    relationship    set<Union>          unions
                        inverse Union::switch_type;
    relationship    set<Operation>      operations
                        inverse Operation::result;
    relationship    set<Property>       properties
                        inverse Property::type;
    relationship    set<Constant>       constants
                        inverse Constant::type;
    relationship    set<TypeDefinition> type_defs
                        inverse TypeDefinition::alias;
};
interface PrimitiveType : Type {
    readonly attribute PrimitiveKind        primitive_kind;
};
```

#### *2.8.3.6.1 Interfaces*

Interfaces are the most important types in the repository. Interfaces define the abstract behavior of application objects and contain operations for creating and removing Attributes, Relationships, and Operations within themselves in addition to the operations inherited from DefiningScope. 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, except Modules and Interfaces.

```
interface Interface : Type, DefiningScope {
    struct ParameterSpec {
        string      param_name;
        Direction   param_mode;
        Type        param_type; };
    relationship    set<Interface>      inherits
                        inverse Interface::derives;
    relationship    set<Interface>      derives
                        inverse Interface::inherits;
    exception       BadParameter{string reason; };
    exception       BadRelationship{string reason; };
```

```
        Attribute       add_attribute(in string attr_name, in Type attr_type)
                            raises(DuplicateName);
        Relationship    add_relationship(in string rel_name,
                                in Type rel_type,
                                in Relationship rel_traversal)
                            raises(DuplicateName, BadRelationship);
        Operation       add_operation(in string op_name,
                                in Type op_result,
                                in list<ParameterSpec> op_params,
                            in list<Exception> op_raises)
                            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);
};
```

#### *2.8.3.6.2 Classes*

Classes are a subtype of Interface whose properties define the abstract state of objects stored in an ODMS. Classes are linked in a single inheritance hierarchy whereby state and behavior are inherited from an extender class. Classes may define keys and extents over their instances.

```
interface Class : Interface {
    attribute       list<string>        extents;
    attribute       list<string>        keys;
    relationship    Class               extender
                      inverse Class::extensions;
    relationship    set<Class>          extensions
                      inverse Class::extender;
};
```

#### *2.8.3.6.3 Collections*

Collections are types that 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 {
    readonly attribute CollectionKind       collection_kind;
    relationship    Operand                 max_size
                        inverse Operand::size_of;
    relationship    Type                    subtype
                        inverse Type::collections;
    boolean         is_ordered();
    unsigned long   bound();
};
interface Dictionary : Collection {
    relationship    Type        key_type
                        inverse Type::dictionaries;
};
```

#### 2.8.3.6.4 Constructed Types

Some types contain named elements that themselves refer to other types and are said to be *constructed* from those types. The ScopedType interface is an abstract class that consolidates these mechanisms for its subclasses Enumeration, Structure, and Union. Enumerations contain Constants, Structures contain Members, and Unions contain UnionCases. Unions, in addition, have a relationship with a switch_type that 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           exception_result
                        inverse Exception::result;
};
interface Union : ScopedType {
    relationship    Type                switch_type
                        inverse Type::unions;
    relationship    list<UnionCase>     cases
                        inverse UnionCase::union_type;
};
```

### 2.8.4 Specifiers

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

```
interface Specifier : RepositoryObject {
    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>   case_labels
                        inverse Operand::case_in;
};
enum Direction {mode_in, mode_out, mode_inout } ;
interface Parameter : Specifier {
    attribute       Direction       parameter_mode;
    relationship    Operation       operation
                        inverse Operation::signature;
};
```

### 2.8.5 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 that 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 : RepositoryObject {
    relationship    Expression      operand_in
                        inverse Expression::the_operands;
    relationship    Constant        value_of
                        inverse Constant::the_value;
    relationship    Collection      size_of
                        inverse Collection::max_size;
    relationship    UnionCase       case_in
                        inverse UnionCase::case_labels;
    Object          value();
};
interface Literal : Operand {
    attribute    Object   literal_value;
};
interface ConstOperand : Operand {
    relationship    Constant    references
                        inverse Constant::referenced_by;
};
```

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>       the_operands
                        inverse Operand::operand_in;
};
```

## 2.9 Locking and Concurrency Control

The ODMG Object Model uses a conventional lock-based approach to concurrency control. This approach provides a mechanism for enforcing shared or exclusive access to objects. The ODMS supports the property of serializability by monitoring requests for locks and granting a lock only if no conflicting locks exist. As a result, access to persistent objects is coordinated across multiple transactions, and a consistent view of the ODMS is maintained for each transaction.

The ODMG Object Model supports traditional pessimistic concurrency control as its default policy, but does not preclude an ODMS from supporting a wider range of concurrency control policies.

### 2.9.1 Lock Types

The following locks are supported in the ODMG Object Model:

- read
- write
- upgrade

*Read* locks allow shared access to an object. *Write* locks indicate exclusive access to an object. Readers of a particular object do not conflict with other readers, but writers conflict with both readers and writers. *Upgrade* locks are used to prevent a form of deadlock that occurs when two processes both obtain read locks on an object and then attempt to obtain write locks on that same object. Upgrade locks are compatible with read locks, but conflict with upgrade and write locks. Deadlock is avoided by initially obtaining upgrade locks, instead of read locks, for all objects that intend to be modified. This avoids any potential conflicts when a write lock is later obtained to modify the object.

These locks follow the same semantics as those defined in the OMG Concurrency Control Service.

### 2.9.2 Implicit and Explicit Locking

The ODMG Object Model supports both implicit and explicit locking. Implicit locks are locks acquired during the course of the traversal of an object graph. For example, read locks are obtained each time an object is accessed and write locks are obtained each time an object is modified. In the case of implicit locks, no specific operation is executed in order to obtain a lock on an object. However, explicit locks are acquired by expressly requesting a specific lock on a particular object. These locks are obtained using the lock and try_lock operations defined in the Object interface. While read and write locks can be obtained implicitly or explicitly, upgrade locks can only be obtained explicitly via the lock and try_lock operations.

### 2.9.3 Lock Duration

By default, all locks (read, write, and upgrade) are held until the transaction is either committed or aborted. This type of lock retention is consistent with the SQL-92 definition of transaction isolation level 3. This isolation level prevents dirty reads, nonrepeatable reads, and phantoms.

## 2.10 Transaction Model

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

A transaction is a unit of logic for which an ODMS 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 ODMS from one internally consistent state to another internally consistent state. There may be times during the transaction when the ODMS is inconsistent. However, *isolation* guarantees that no other user of the ODMS sees changes made by a transaction until that transaction commits. Concurrent users always see an internally consistent ODMS. *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 ODMS 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 persistent storage and made visible to other users of the ODMS. When a transaction aborts, none of the changes made by it are installed in the persistent storage, including any changes made prior to the time of abort. The execution of concurrent transactions must yield results that are indistinguishable from results that would have been obtained if the transactions had been executed serially. This property is sometimes called *serializability*.

### 2.10.1 Distributed Transactions

Distributed transactions are transactions that span multiple processes and/or that span more than one database, as described 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 because it is not visible to the programmers but used only by transaction monitors.Vendors are not required to support distributed transactions, but if they do, their implementations must be XA-compliant.

### 2.10.2 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 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 ODMS. Note that a single logical ODMS may be implemented as one or more physical persistent stores, 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 ODMS.

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.

### 2.10.3 Transaction Operations

There are two types that are defined to support transaction activity within an ODMS: TransactionFactory and Transaction.

The TransactionFactory type is used to create transactions. The following operations are defined in the TransactionFactory interface:

```
interface TransactionFactory {
    Transaction     new();
    Transaction     current();
};
```

The new operation creates Transaction objects. The current operation returns the Transaction that is associated with the current thread of control. If there is no such association, the current operation returns *nil*.

Once a Transaction object is created, it is manipulated using the Transaction interface. The following operations are defined in the Transaction interface:

```
interface Transaction {
    void            begin() raises(TransactionInProgress,
                        DatabaseClosed);
    void            commit() raises(TransactionNotInProgress);
    void            abort() raises(TransactionNotInProgress);
    void            checkpoint() raises(TransactionNotInProgress);
    void            join() raises(TransactionNotInProgress);
    void            leave() raises(TransactionNotInProgress);
    boolean         isOpen();
};
```

After a Transaction object is created, it is initially closed. An explicit begin operation is required to open a transaction. If a transaction is already open, additional begin operations raise the TransactionInProgress exception.

The commit operation causes all persistent objects created or modified during a transaction to be written to the ODMS and to become accessible to other Transaction objects running against that ODMS. All locks held by the Transaction object are released. Finally, it also causes the Transaction object to complete and become closed. The TransactionNotInProgress exception is raised if a commit operation is executed on a closed Transaction object.

The abort operation causes the Transaction object to complete and become closed. The ODMS is returned to the state it was in prior to the beginning of the transaction. All locks held by the Transaction object are released. The TransactionNotInProgress exception is raised if an abort operation is executed on a closed Transaction object.

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 ODMS, and it retains all locks held by the Transaction object. The Transaction object remains open. The TransactionNotInProgress exception is raised if a checkpoint operation is executed on a closed Transaction object.

ODMS operations are always executed in the context of a transaction. Therefore, to execute any operations on persistent objects, an active Transaction object must be associated with the current thread. The join operation associates the current thread with a Transaction object. If the Transaction object is open, persistent object operations may be executed; otherwise a TransactionNotInProgress exception is raised.

If an implementation allows multiple active Transaction objects to exist, the join and leave operations allow a thread to alternate between them. To associate the current thread with another Transaction object, simply execute a join on the new Transaction object. If necessary, a leave operation is automatically executed to disassociate the current thread from its current Transaction object. Moving from one Transaction object to another does not commit or abort a Transaction object. When the current thread has no current Transaction object, the leave operation is ignored.

After a Transaction object is completed, to continue executing operations on persistent objects, either another open Transaction object must be associated with the current thread or a begin operation must be applied to the current Transaction object to make it open 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. The transaction completes when any one of the threads executes a commit or abort operation against the Transaction object.

In order to begin a transaction, a Database object must be opened. During the processing of a transaction, any operation executed on a Database object is *bound* to that transaction. A Database object may be bound to any number of transactions. All Database objects, bound to transactions in progress, must remain open until those transactions have completed via either a commit or a rollback. If a close operation is called on the Database object prior to the completion of all transactions, the TransactionInProgress exception is raised and the Database object remains open.

## 2.11 Database Operations

An ODMS may manage one or more logical ODMSs, each of which may be stored in one or more physical persistent stores. Each logical ODMS is an instance of the type Database, which is supplied by the ODMS. Instances of type Database are created using the DatabaseFactory interface:

```
interface DatabaseFactory {
    Database        new();
};
```

Once a Database object is created by using the new operation, it is manipulated using the Database interface. The following operations are defined in the Database interface:

```
interface Database {
    exception DatabaseOpen{};
    exception DatabaseNotFound{};
    exception ObjectNameNotUnique{};
    exception ObjectNameNotFound{};
    void        open(in string odms_name)
                    raises(DatabaseNotFound,
                           DatabaseOpen);
    void        close() raises(DatabaseClosed,
                               TransactionInProgress);
    void        bind(in Object an_object, in string name)
                    raises(DatabaseClosed,
                           ObjectNameNotUnique,
                           TransactionNotInProgress);
    Object      unbind(in string name)
                    raises(DatabaseClosed,
                           ObjectNameNotFound,
                           TransactionNotInProgress);
    Object      lookup(in string object_name)
                    raises(DatabaseClosed,
                           ObjectNameNotFound,
                           TransactionNotInProgress);
    ODLMetaObjects::Module  schema()
                    raises(DatabaseClosed,
                           TransactionNotInProgress);
};
```

The open operation must be invoked, with an ODMS name as its argument, before any access can be made to the persistent objects in the ODMS. The Object Model requires

only a single ODMS to be open at a time. Implementations may extend this capability, including transactions that span multiple ODMSs. The close operation must be invoked when a program has completed all access to the ODMS. When the ODMS closes, it performs necessary cleanup operations, and if a transaction is still in progress, raises the TransactionInProgress exception. Except for the open and close operations, all other Database operations must be executed within the scope of a Transaction. If not, a TransactionNotInProgress exception will be raised.

The lookup operation finds 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 ODMS. The names of objects in the ODMS, the names of types in the ODMS schema, and the extents of types instantiated in the ODMS are global. They become accessible to a program once it has opened the ODMS. Named objects are convenient entry points to the ODMS. A name is bound to an object using the bind operation. Named objects may be unnamed using the unbind operation.

The schema operation accesses the root meta object that defines the schema of the ODMS. The schema of an ODMS is contained within a single Module meta object. Meta objects contained within the schema may be located via navigation of the appropriate relationships or by using the resolve operation with a scoped name as the argument. A scoped name is defined by the syntax of ODL and uses double colon (::) delimiters to specify a search path composed of meta object names that uniquely identify each meta object by its location within the schema. For example, using examples defined in Chapter 3, the scoped name "Professor::name" resolves to the Attribute meta object that represents the name of class Professor.

The Database type may also support operations designed for ODMS administration, for example, create, delete, 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.