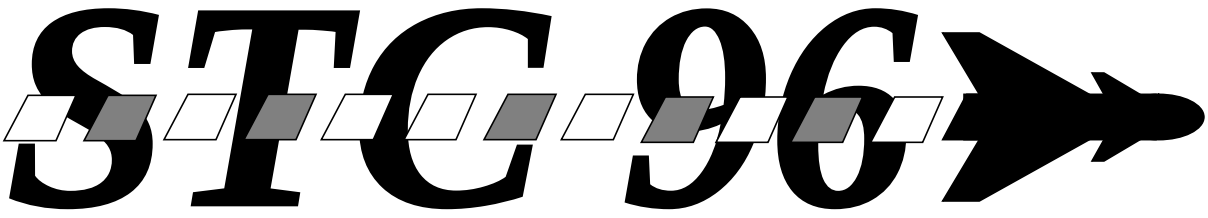

FIRM: An Ada Binding to ODMG-93 1.2

Michael P. Card

E-mail: michael.p.card@lmco.com

Voice: (315)-456-3022

Originally presented Wednesday, April 24, 1996

Updated Thursday, April 16, 1998

Track: 6 (Object-Oriented Development)

Keywords: object-oriented, database, ODMG, Ada, multi-level secure, MLS, real-time

Abstract: Object-oriented database (ODBMS) technology supports a wide range of applications today. ODBMS systems offer better performance than traditional relational databases for those applications which access their objects primarily through inter-object relationships. Data fusion and other advanced avionics applications are in this category. Wright Laboratory created the Functionally Integrated Resource Manager (FIRM) program to develop an ODBMS that can support these real-time avionics applications. Wright Laboratory wants the ODBMS developed for the FIRM program to have the widest possible application, so it has been a goal of the FIRM team to build its ODBMS to be as compliant as possible with the specification published by the Object Data Management Group (ODMG), which is an industry consortium of ODBMS vendors and other parties interested in ODBMS standards. Although the ODMG specification does not include a language binding for Ada, the FIRM team was able to use the ODMG object model and C++ language binding to design an Ada binding for its ODBMS. This paper describes their Ada binding in detail.

Table of Contents for FIRM: An Ada Binding to ODMG-93 1.2

Table of Contents for FIRM: An Ada Binding to ODMG-93 1.2

Table of Contents for FIRM: An Ada Binding to ODMG-93 1.2

Table of Contents for FIRM: An Ada Binding to ODMG-93 1.2

1 INTRODUCTION

1.1 Document Overview

This document provides a detailed description of the application programming interface (API) for an object-oriented database management system (ODBMS). The ODBMS described herein is based upon the object model given in chapter 2 of [ODMG 96]. The API for the ODBMS is designed for maximal ease-of-use by an Ada programmer, so it can be considered a candidate for an Ada binding to [ODMG 96]. The API presented here and its underlying ODBMS are being developed as part of the Functionally Integrated Resource Manager (FIRM) program, which seeks to develop a real-time, multi-level secure ODBMS for use in future avionics systems. The FIRM ODBMS is being developed by Lockheed Martin under contract from Wright Laboratory.

The API consists of eleven Ada packages:

1. **Firm** - contains root type for user-definable types, abstract interface for collections, indices, and relationships
2. **Firm.Atomic_Access** - generic child package which contains operators for getting physical pointers to atomic objects in the database and operations for setting up FIRM storage pools for atomic object types (e.g. Ada types derived from the Firm.Atomic_ Object type)
3. **Firm.Arrays** - generic child package which contains concrete realization of abstract collection interface for the array collection type (see [ODMG 96], section 2.3.5.4)
4. **Firm.Bags** - generic child package which contains concrete realization of abstract collection interface for the bag collection type (see [ODMG 96], section 2.3.5.2)
5. **Firm.Chronos** - generic child package which contains concrete realization of abstract collection interface for the chrono collection type (the chrono collection type is an extension to [ODMG 96]; see section 6 on page 53)
6. **Firm.Lists** - generic child package which contains concrete realization of abstract collection interface for the list collection type (see [ODMG 96], section 2.3.5.3)
7. **Firm.Sets** - generic child package which contains concrete realization of abstract collection interface for the set collection type (see [ODMG 96], section 2.3.5.1)
8. **Firm.Indices** - generic child package for indices on a collection
9. **Firm.Atomic_Relationships** - generic child package for relationships between atomic types
10. **Firm.Collection_Relationships** - generic child package for relationships between atomic types and collection types

These packages are described in section 3 through section 12 in the order listed above. Appendix A, which explains the differences between ODMG-93 version 1.2 and the Ada binding presented here, follows section 12. The other appendices contain source code for the Ada package specifications listed above.

1.2 Design Goals

The ODBMS API presented in this paper (hereinafter referred to as the FIRM ODBMS, the FIRM API, or simply FIRM) was designed with the following goals in mind:

1. Suitability for real-time applications
2. "Minimum visibility" or "seamless binding" to the Ada-95 programming language
3. Tamper-resistant multi-level security (MLS)
4. Consistency with the Object Data Management Group's (ODMG) published standard for ODBMSs, the ODMG-93 version 1.2 specification [ODMG 96]
5. Use of Ada-95 to minimize ODBMS complexity

1.2.1 Suitability for real-time applications

The FIRM bindings allow for the ODBMS to manage databases in main memory[1] as well as in persistent storage. This is absolutely critical for certain real-time embedded applications, since access to main-memory objects is rapid and deterministic (unlike caching objects from secondary storage devices). The FIRM object model and API therefore include objects of "Global" persistence (see see section 3.1.1 on page 14) which are stored in main memory.

Deterministic performance is critical to real-time applications. This means that concurrency control mechanisms which heavily rely on blocking are problematic. The FIRM API is therefore designed to allow alternative concurrency control algorithms such as multi-version mixed method (MVMM, see [Bernstein et. al.] pp. 160-164). Note that allowing multiple versions of an object to exist in the database means that the ODBMS API must not be designed such that a 1:1 correspondence between an object's unique identifier and its physical storage address is required.

Real-time defense applications like trackers, correlators and data fusion algorithms require fast access to historical data. The FIRM ODBMS therefore provides a chrono collection

1. There are several good articles in the literature on main memory databases. One starting point is "Main Memory Database Systems: An Overview" by H. Garcia-Molina and K. Salem in *IEEE Transactions on Knowledge and Data Engineering*, Dec 92, vol. 4 no. 6, pages 509-516.

type (see section 7 on page 65) in addition to the collection types specified in [ODMG 96]. The chrono collection type is optimized for accessing objects by their time of storage, which makes it ideal for history queries.

1.2.2 Seamless binding to Ada-95

This goal could be expressed “Make the ODBMS as user-friendly as possible to an Ada programmer. If possible, make it invisible.” The FIRM ODBMS is intended for use in real-time embedded environments like avionics systems. New software for these systems will likely be written in Ada, so the database application developers that will use the FIRM ODBMS will likely be Ada programmers. These programmers will find an ODBMS easy to use only if it “makes sense” from an Ada programming perspective. “Making sense” in this context means that the ODBMS must be minimally intrusive (e.g. maximally “invisible”) to an Ada program.

Designing an ODBMS to be invisible (e.g. to have a “seamless binding”) means that it “shares the type system of the host programming language and there is a simple mapping between transient and persistent objects” ([Loomis 95], pp. 22-23). Therefore, the creation and manipulation of database objects will look just like the creation and manipulation of non-database objects, since in both cases the same type system is used. This means that transient and persistent objects must share a common set of operators. In addition, an “invisible” ODBMS should have a minimal set of ODBMS-specific functions and methods. This makes the application software easier to read and maintain since it will consist mostly of object manipulations (its purpose) rather than special calls to make the ODBMS do something.

1.2.3 Tamper-resistant multi-level security

The FIRM ODBMS must be able to manage objects of multiple classification levels in accordance with the requirements given in [TCSEC 85]. This means that in addition to maintaining a unique identifier for each object in accordance with [ODMG 96], the FIRM ODBMS must also maintain an appropriate security label for each object. The API for the FIRM ODBMS must be designed to properly handle cases where an Ada task attempts to access an object with a security label that is higher than its own. Such cases could either result in an exception or a “not found” condition; the FIRM ODBMS uses the latter approach.

1.2.4 Consistency with the ODMG standard

The FIRM API is designed to be as compliant as possible with the ODMG standard. It should be recognized, however, that the ODMG standard was developed for commercial object-oriented database management systems which are intended primarily for use in

computer-aided design/computer-aided manufacturing (CAD/CAM) applications and management information systems (MIS). The FIRM ODBMS is being designed for use in military avionics systems. Database applications in these systems require database features that CAD/CAM and MIS database applications do not, such as deterministic real-time transaction execution (see section 1.2.1 on page 7), secure handling of classified data (see section 1.2.3 on page 8), and special collection types to provide real-time access to historical data (see section 1.2.1 on page 7).

The FIRM ODBMS is implemented in Ada. The ODMG specification does not include a language binding for Ada. The FIRM API was therefore developed in accordance with the object model presented in [ODMG 96], chapter 2. Chapter 5 of [ODMG 96], the C++ language binding, was also used for reference. The Ada object model is markedly different from the C++ object model, so some design choices in the FIRM API will look somewhat different from its ODMG C++ counterpart. For a comparison of the object models in FIRM and [ODMG 96], see Appendix A.

1.2.5 Use of Ada-95 to minimize ODBMS complexity

The FIRM ODBMS's API and internals will be designed using the current "best practices" for object-oriented software engineering. We will use the Ada-95 features that support these practices (i.e. encapsulation, abstract classes, class reuse through inheritance and class-wide operators, etc.) to minimize the complexity of the FIRM ODBMS and to simultaneously maximize its maintainability.

2 THE DESIGN AND USE OF THE FIRM API

The FIRM ODBMS interface complies with the ODMG standard wherever possible (see Appendix A on page 81). Both the FIRM source code and this document reference [ODMG 96].

2.1 The FIRM type hierarchy

There are four kinds of objects in the FIRM ODBMS: atomic objects (those which the user derives from the Atomic_Object type), collection references, relationship references, and index references. The FIRM object type hierarchy is shown in Figure 1.

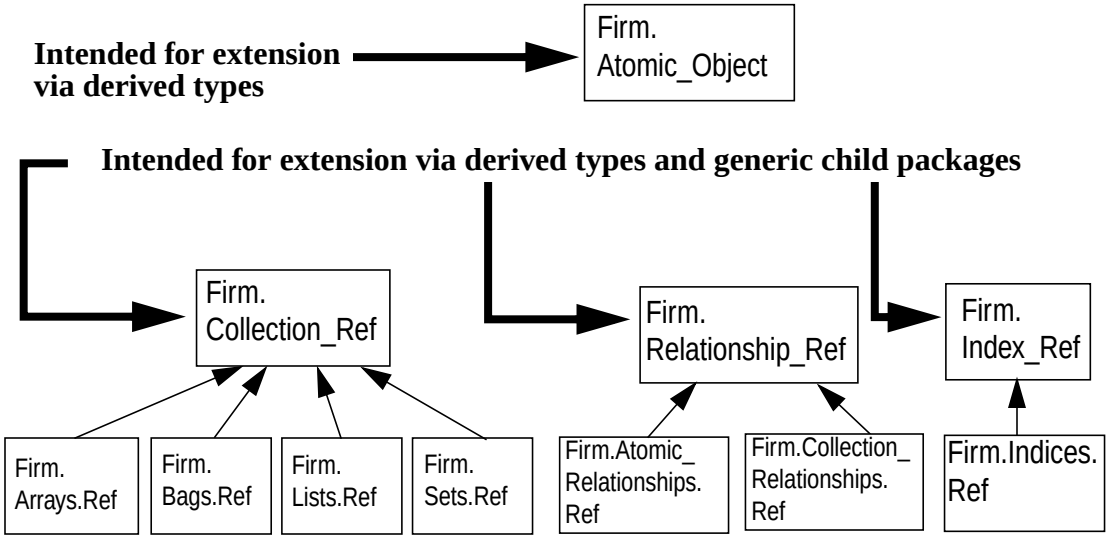

Figure 1: FIRM's object type hierarchy

As Figure 1 shows, all four of these types that are available for extension (e.g. you can inherit from them):

1. **Atomic_Object** - root type for all user-defined objects
2. **Collection_Ref** - root type for collection references
3. **Index_Ref** - root type for index references
4. **Relationship_Ref** - root type for relationship references

The application developer uses the Atomic_Object type to create types whose instances will be managed by the FIRM ODBMS. The amount of storage allocated for user-defined types is specified via a call to Create_Global_Pool or Open_Persistent_Pool (see section 3.3 on page 22). These operations allow the application to define how much storage should be allocated for pools of objects with "Global" (main memory) and "Persistent" persistence (see section 3.1.1 on page 14). Instances of the Atomic_Object type or any type derived

from it are created with “Local” persistence by default, which makes them analogous to an Ada local variable. To get a shareable/more persistent object, the application developer can use the access types whose storage pools were set up in the call to Create_Global_Pool or Open_Persistent_Pool. Invoking the Ada new operator for one of these access types will create an atomic object with Global or Persistent persistence.

The Collection_Ref type is abstract and is provided so that the developers of the FIRM ODBMS can create new kinds of collections for application developers. The Collection_ Ref type and its derivatives are references, which are “logical pointers” to collections (see section 2.2 on page 11).The FIRM ODBMS uses the Collection_Ref type to provide the array, bag, list and set collections specified in [ODMG 96], pp. 17-20 as well as other collection types. The concrete reference types for each collection type are declared in the corresponding generic child package (for example, the reference type for an array collection, Firm.Arrays.Ref, is declared in the Firm.Arrays package).

The Index_Ref type is abstract and is provided so that the developers of the FIRM ODBMS can create new kinds of indices for FIRM’s collections. The Index_Ref type and its derivatives are references, which are “logical pointers” to indices (see section 2.2 on page 11).The FIRM ODBMS uses the Index_Ref type to provide indices (that is, access in key-attribute order) to the array, bag, list and set collections specified in [ODMG 96], pp. 17-20 as well as other collection types. The concrete reference type for indices, Firm.Indices.Ref, is declared in the Firm.Indices package.

The Relationship_Ref type is abstract and is provided so that the developers of the FIRM ODBMS can create new kinds of relationships for application developers. The Relationship_Ref type and its derivatives are references, which are “logical pointers” to relationships (see section 2.2 on page 11).The FIRM ODBMS uses the Relationship_Ref type to provide relationships between atomic objects and between atomic objects and collections. The concrete reference types for each relationship type are declared in the corresponding generic child package (for example, the reference type for a relationship between two atomic object types, Firm.Atomic_Relationships.Ref, is declared in the Firm.Atomic_ Relationships package).

2.2 FIRM’s reference types

A reference is a “logical pointer.” The term logical pointer is used to describe a reference because while a reference is a pointer to an object, it cannot be dereferenced (e.g. translated into a physical address) using Ada-95’s built-in dereferencing operator (“.all”) as can a physical pointer. This is not a problem for the private types supplied by the FIRM ODBMS (i.e. collections), since all of their operations use only the FIRM-supplied reference types (i.e. collection iterators). The only types for which physical access is required are the application-defined types which are derived from the Atomic_Object type.

References allow the ODBMS to control access to the objects in the database. This control occurs when the reference type is dereferenced. By controlling the dereferencing of inter-

object pointers directly, the ODBMS architecture can be made more flexible. For example, by using references instead of physical pointers to connect objects it becomes possible for the ODBMS to use multi-versioning concurrency control. Multi-versioning would be difficult or impossible if all of the objects in the database were linked with physical pointers. Other possible activities that the ODBMS could perform at dereferencing time include:

- security check (mandatory access control (MAC) validation)
- referential integrity check
- data validation checks

These functions cannot be performed if physical pointers are used to connect objects, since the dereferencing of a physical pointer would not invoke any ODBMS validation procedures. This produces a trade-off consideration in the design of an ODBMS interface: reference types provide more semantic information to the ODBMS and thus allow it to be more powerful, but physical pointers provide maximum convenience to the application developer since they are part of the programming language. This topic is covered well in [Loomis 95] on pp. 71-74.

The architectural flexibility that references allow for the ODBMS in conjunction with the additional integrity checks that they make possible are the factors which contributed to our decision to use them in the FIRM ODBMS. The reference types in the FIRM ODBMS and the corresponding types they reference are summarized in Table 1.

Reference type	Corresponding type
Atomic_Access.Ref	Any type derived from type Atomic_Object (e.g. Atomic_Object'class). Such types are application-defined.
Collection_Ref	Reference types derived from Collection_Ref can access their corresponding collection type (e.g. Array_Ref to access array collections, etc.). FIRM collections may only be accessed using types derived from Collection_Ref.
Index_Ref	FIRM's index type, which is not visible to any client of the Firm package. The Index_Ref type is the only way to access an index.
Relationship_Ref	FIRM's relationship type, which is not visible to any client of the Firm package. The Relationship_Ref type is the only way to access a relationship.

Table 1: FIRM reference types

3 THE FIRM PACKAGE

The FIRM ODBMS consists of several Ada packages. The main package, Firm, contains the essential type declarations and class-wide methods for the FIRM ODBMS. Additional functionalities (such as various kinds of collections) are provided in child packages which are described in later sections.

3.1 The Object type hierarchy

The type hierarchy for FIRM's type "Object" is shown in Figure 2, along with all of the other type hierarchies in the FIRM ODBMS.

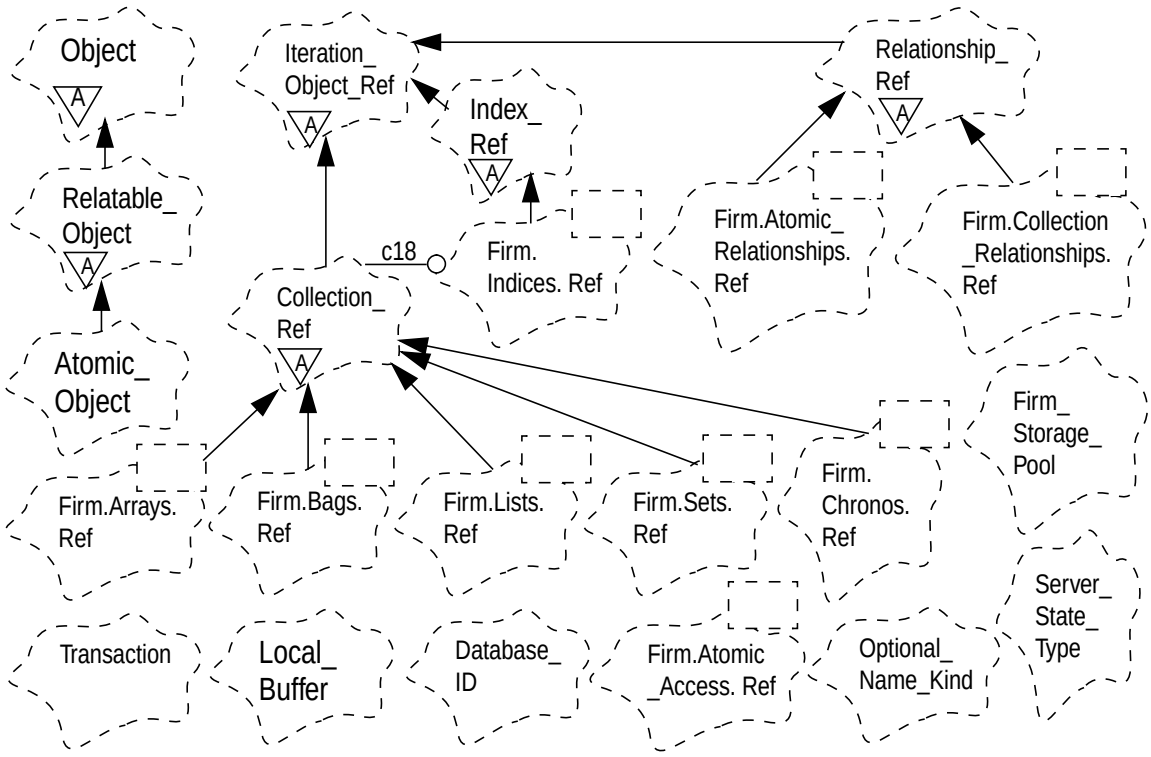

Figure 2: Type hierarchies in the FIRM ODBMS

The following sections contain descriptions of each type.

3.1.1 The Object type

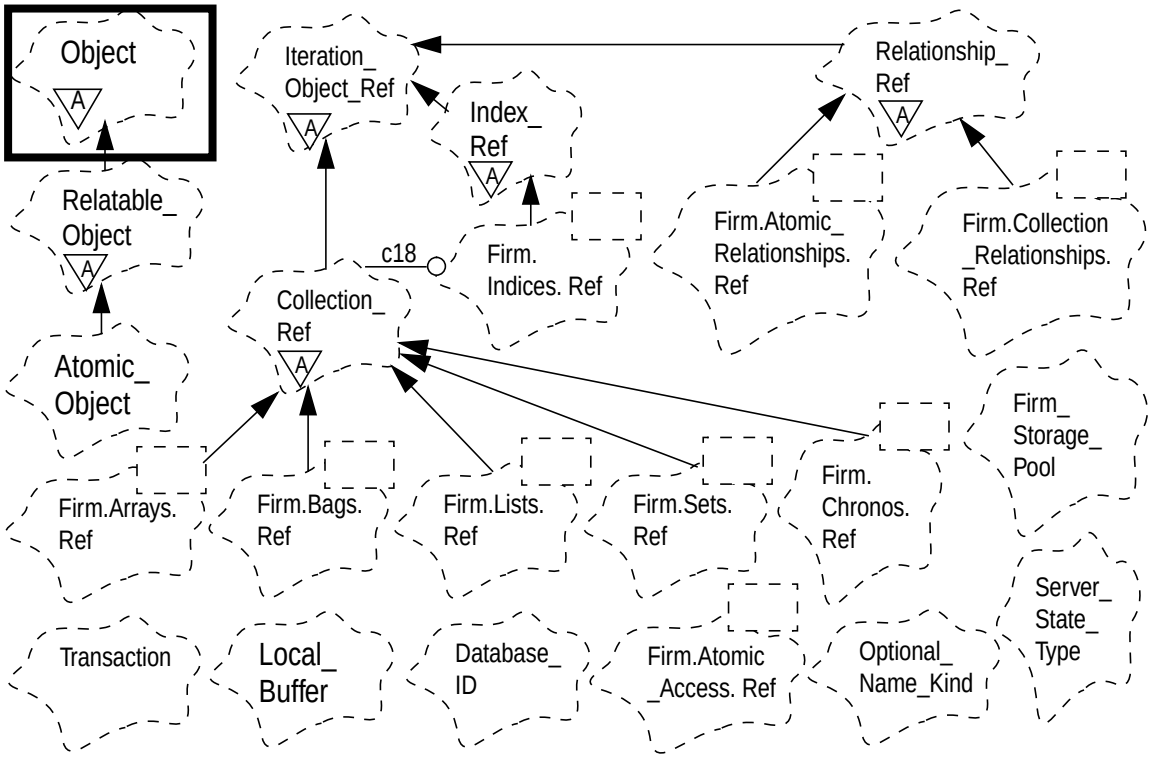

Figure 3: The Object type

The Object type corresponds to “Object_type” in Figure 2-1 on page 30 of [ODMG 96]. The Object type is “abstract”. This means that a FIRM application developer could not create an instance of type Object. Abstract types exist solely to provide inheritable attributes and/or operations for other types, which may or may not be “concrete”. (A concrete type is a type for which actual instances may be created.)

FIRM’s Object type provides a unique object identifier (OID) attribute for all types that are derived from it (e.g. that inherit from it). The OID is used internally by the FIRM ODBMS to find the object’s location in either main memory or secondary storage via an OID lookup. The Object type also provides a persistence indicator. An object’s persistence is fixed at its creation time in the FIRM ODBMS. Table 2 on page 15 describes the kinds of persistence that objects in the FIRM ODBMS can have.

Persistence of the object	Can the object be shared with tasks other than the task that created it?	Is there concurrency control on the object?	Is the object recoverable?	Can the object be created or used outside a transaction?	Life-span of the object
Local	NO	NO	NO	YES	From time of creation to: • out-of-scope • task completion • server shut-down • delete
Global	YES	YES	YES	NO	From time of creation to: • database close • delete
Persistent	YES	YES	YES	NO	From time of creation to: • delete

Table 2: Persistence in the FIRM ODBMS

Local persistence allows the Ada programmer to effectively extend the Ada language to include the FIRM ODBMS's types and operators. That is, an object with Local persistence is the logical equivalent of an Ada local variable except that it can be stored in a collection or participate in a relationship. Local persistence is the equivalent of the "transient" object life-span described in [ODMG 96], paragraph 2.3.3, page 16. Like a local Ada variable, an object with Local persistence is destroyed whenever it is out-of-scope or when the task that created it has completed. Objects with Local persistence are also destroyed when the server is shut down or they are explicitly deleted via a call to the FIRM ODBMS' delete operator (see section 3.1.3.1 on page 19).

Global persistence provides a mechanism for creating objects that are subject to the concurrency control provided by the FIRM ODBMS. This means that these objects are created and used only within database transactions. This in turn means that they may be concurrently accessed by other tasks and that their former state is restored (recovered) when a transaction is aborted. Global objects exist in the database until either the database is closed or until they are explicitly deleted via a call to the FIRM ODBMS' delete operator. Global persistence allows an application to build a main-memory database, which is very

important for real-time applications. The [ODMG 96] specification has no analog to Global persistence since it does not address main-memory databases.

Finally, the “Persistent” category of Persistence allows objects to be created which have all of the properties of objects with Global persistence plus a longer life-span. Objects which have “Persistent” persistence exist from the time they are created until they are explicitly deleted via a call to the FIRM ODBMS’ delete operator. Since these objects can exist even after the database has been closed, it follows that they are kept in some kind of non-volatile storage. “Persistent” persistence is the equivalent of the “persistent” object life-span described in [ODMG 96], paragraph 2.3.3, page 16.

3.1.2 The Relatable_Object type

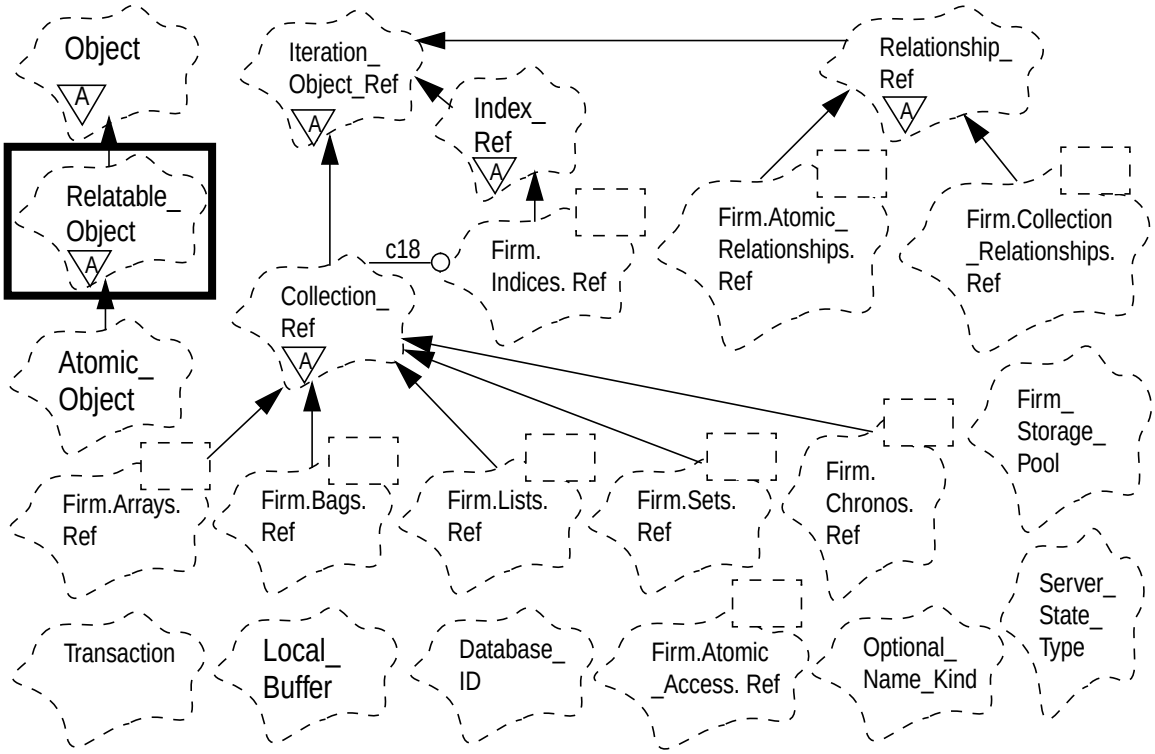

Figure 4: The Relatable_Object type

The abstract Relatable_Object type is derived from the Object type. The Relatable_Object type has no *visible* operations or attributes. This type is the root of the type hierarchy for all types whose instances may participate in a relationship. This means that traversal paths can only be established between instances of types which are derived from Relatable_Object. Like the Object type, this type has an OID attribute as well as other attribute(s) needed to support relationships. The Relatable_Object type has no [ODMG 96] equivalent.

3.1.3 The Atomic_Object type

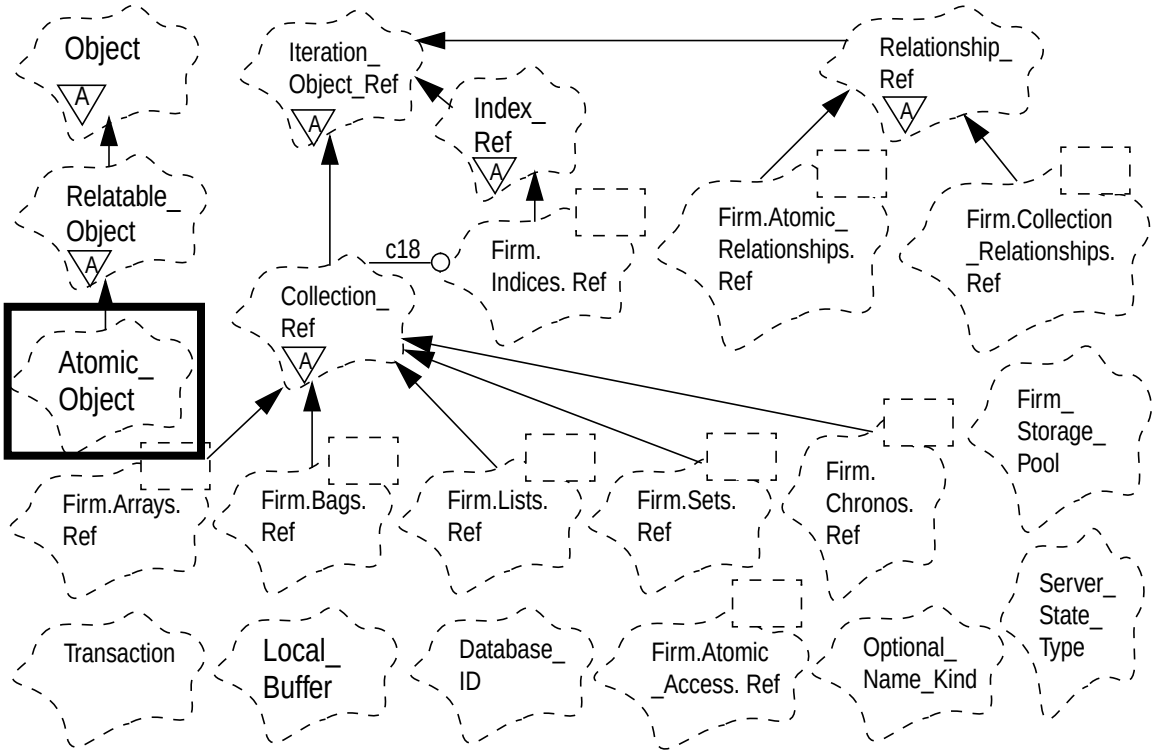

Figure 5: The Atomic_Object type

In order for the FIRM ODBMS to provide seamless integration into the Ada programming language, a mechanism must be provided which allows application developers to define their own Ada types whose instances are controlled by the FIRM ODBMS. To this end, the FIRM ODBMS provides a user-extensible type hierarchy for which the FIRM ODBMS automatically performs initialization and finalization. (Initialization is the process of creating an object, assigning its OID, etc. and finalization is the process of deleting an object and removing all references to it so as to maintain referential integrity in the database). The Atomic_Object type is the root of this user-extensible type hierarchy.

The Atomic_Object type is derived from the Relatable_Object type, and it corresponds to the Atomic_object type in Figure 2-1 on page 30 of [ODMG 96]. Unlike the Relatable_ Object type, however, it is concrete. The Atomic_Object type extends the set of attributes provided by the Relatable_Object type by adding the following new attributes:

1. A security label, which contains both the classification (for example UNCLASSIFIED, CONFIDENTIAL, SECRET or TOP SECRET) of the object as well as additional non-hierarchical security categories.
2. An optional name

Atomic objects may be inserted into collections or be linked via relationships to other Atomic objects or collections. In addition, Atomic objects may be given a name. This optional name could then be used later to retrieve the object (see [ODMG 96], paragraph 2.3.2, pg. 16).

3.1.3.1 Operations on the Atomic_Object type

Table 3 summarizes the operations available for the Atomic_Object type in the Firm package.

Name	Description
Bind	Procedure to give an object of type Atomic_Object'class a name ([ODMG 96], paragraph 2.9, pg. 33)
Delete	Procedure to delete an object of type Atomic_Object'class from the database ([ODMG 96], paragraph 2.3, pg. 15)
Lookup	Given an object's name (and an optional transaction), returns a pointer to an Atomic_Object

Table 3: Operations on the Atomic_Access.Ref type

The operations in Table 3 correspond to those specified in paragraphs 2.3 (or 2.9) in [ODMG 96]. All of these operations require a valid transaction when they are used on objects with Global or Persistent persistence.

3.2 The Database_ID type

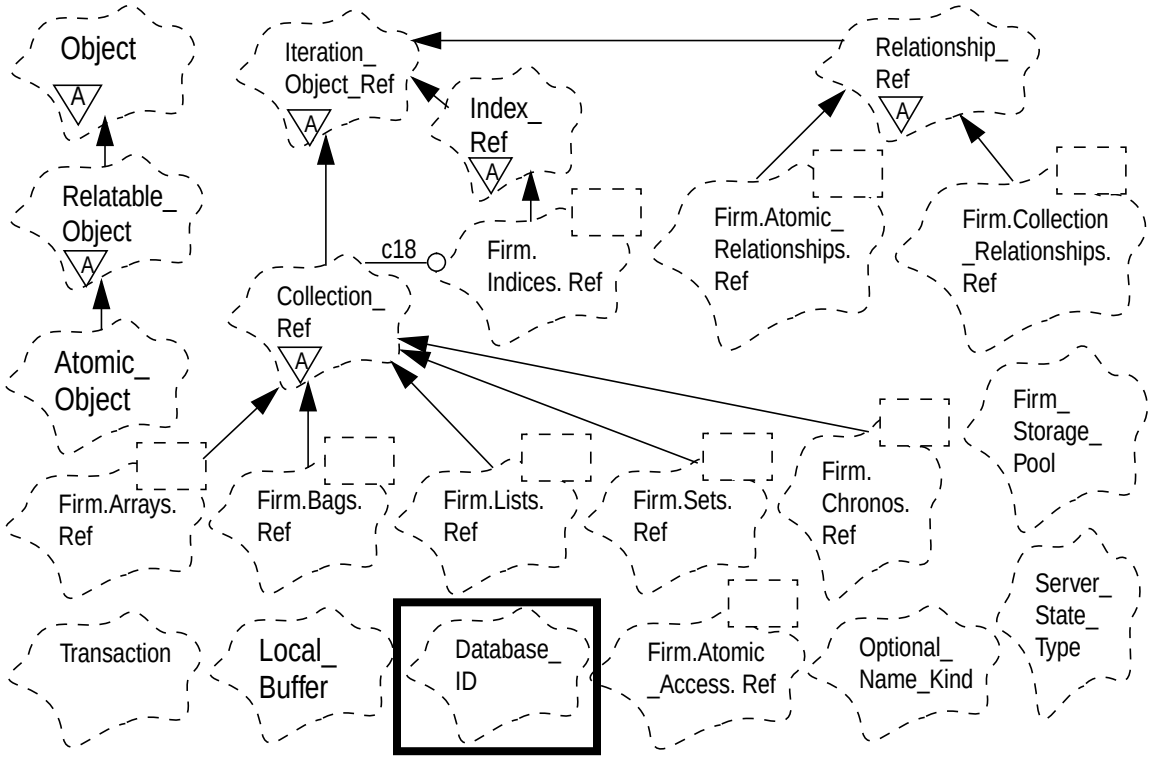

Figure 6: The Database_ID type

Paragraph 2.9, page 33 of [ODMG 96] says that the [ODMG 96] object model requires that an application be allowed to have at least one database open; an implementation may also allow more than one database at a time to be open. FIRM allows multiple databases to be open simultaneously, and a transaction may access objects in one or more databases. The Database_ID type is used to uniquely identify a database residing on a FIRM server.

A FIRM database may be viewed as a set of FIRM storage pools (see section 3.3 on page 22). Thus, when a FIRM database is opened all of its associated storage pools are also opened and prepared for use. Likewise, when a FIRM database is closed all of its associated storage pools are closed and are no longer available for use. Note that “closing” a GLOBAL storage pool is tantamount to deleting it, since GLOBAL storage pools do not reside on a persistent storage medium. Thus, all GLOBAL objects (see page 15) in a database are deleted when the database is closed. PERSISTENT objects, however, will still be accessible if the database is opened later. Objects with LOCAL persistence (see page 15) are kept in storage which is managed by the Ada run-time, not FIRM. The Ada run-time’s storage area is of course not effected by any Database_ID operation.

To ensure referential integrity, FIRM does not support inter-database references. That is, you cannot put an object in one database into a collection in another. Also, you cannot establish a traversal path between objects in different databases. A FIRM database is therefore said to be *self-contained.*

3.2.1 Operations on the Database_ID type

Table 4 summarizes the operations for the Database_ID type. These operations include those specified in paragraph 2.9 of [ODMG 96] which do not involve object names. The object name operations are grouped with the appropriate reference type (i.e. the Lookup and Bind operations for the Atomic_Ref type are given in Table 3 on page 19).

Operation	Description
create	Accepts a string containing a database's name and a database ID. The specified database is created and prepared for on-line use (i.e. the required files are created, storage is allocated, etc.). If the database name or ID is already in use, the OML_Error exception is raised.
db_access	Accepts a database ID. Returns an instance of the enumerated type Firm.Db_Access_Type, which may have the value SOR or REPLICANT. SOR is returned if the specified database is the System Of Record (SOR) copy, otherwise REPLICANT is returned.
open	Accepts a string containing a database's name, a database ID, and instance of Db_Access_Type which indicates the kind of access desired (SOR or REPLICANT), and a pointer to a callback procedure which is invoked if the specified database is opened as a REPLICANT copy but later becomes the System Of Record (SOR) copy via a "transfer SOR" operation. The specified database is prepared for on-line use (i.e. the required files are opened, storage is allocated, etc.). If the database name or ID is already in use, the OML_Error exception is raised.
close	Accepts as input a Database_ID. The specified database is then taken off-line (i.e. the cache is flushed, all relevant files are closed, etc.). No objects in a closed database can be accessed until the database is opened again.

Table 4: Operations for the Database_ID type

3.3 The FIRM_Storage_Pool type

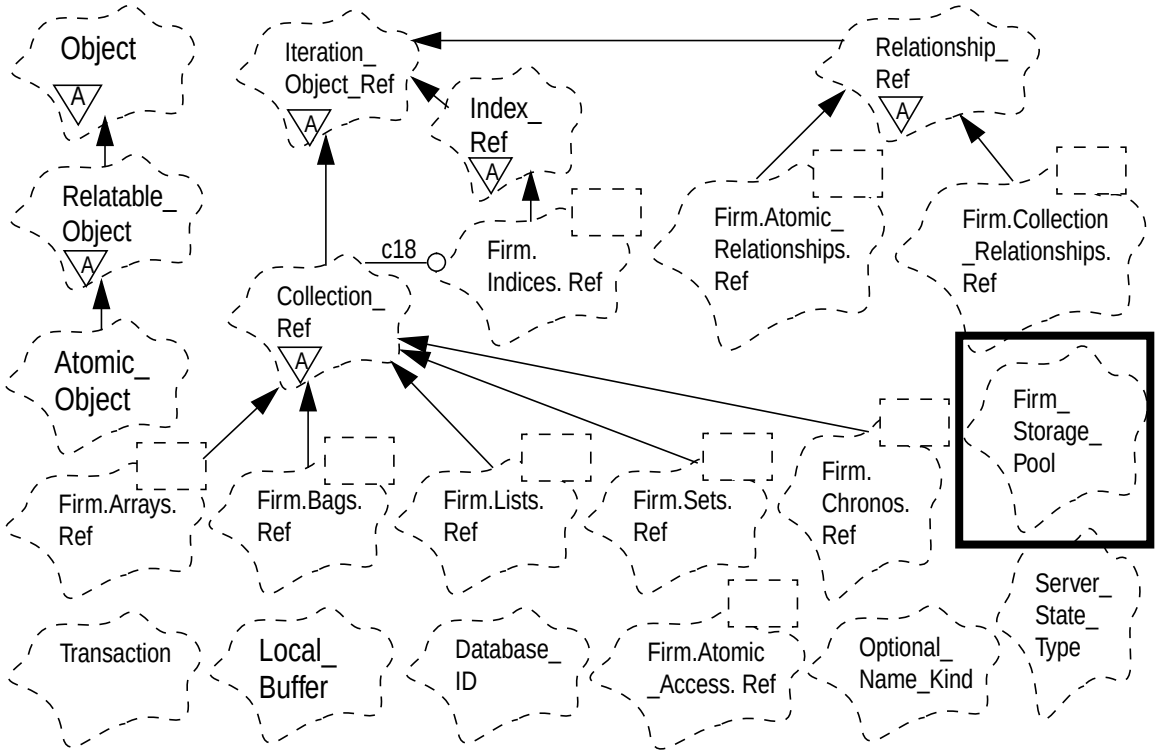

Figure 7: The FIRM_Storage_Pool type

The FIRM ODBMS allows the application developer to preallocate storage for all of the objects in the database. Preallocation of storage makes precise resource budgeting possible. It also allows the storage management algorithms to be faster and more deterministic than those used for dynamic storage management. Providing the ODBMS with a contiguous storage area that it manages also eliminates “memory leakage” problems.

Preallocation of storage for FIRM’s internal types (i.e. collection objects, index objects, relationship objects, etc.) is done by the FIRM ODBMS itself at elaboration time. Configuration information is used to control the storage allocations for these objects. Preallocation of storage for user-defined objects (e.g. instances of types derived from the Atomic_Object type) is accomplished through the Create_Global_Pool and Open_Persistent_Pool operators in the Firm.Atomic_Access package (see section 4 on page 54).

3.4 The Collection_Ref type hierarchy

The Collection_Ref type provides logical pointers to collection objects. The Collection_ Ref type is abstract; concrete types for collection references are provided via child packages. For example, the Firm.Arrays package provides the Ref type to access array collection objects.

All of the property functions and operations which can be performed on a FIRM collection use references to access the collection. That is, these functions and procedures accept descendants of Collection_Ref as their collection parameters rather than the actual collection objects.

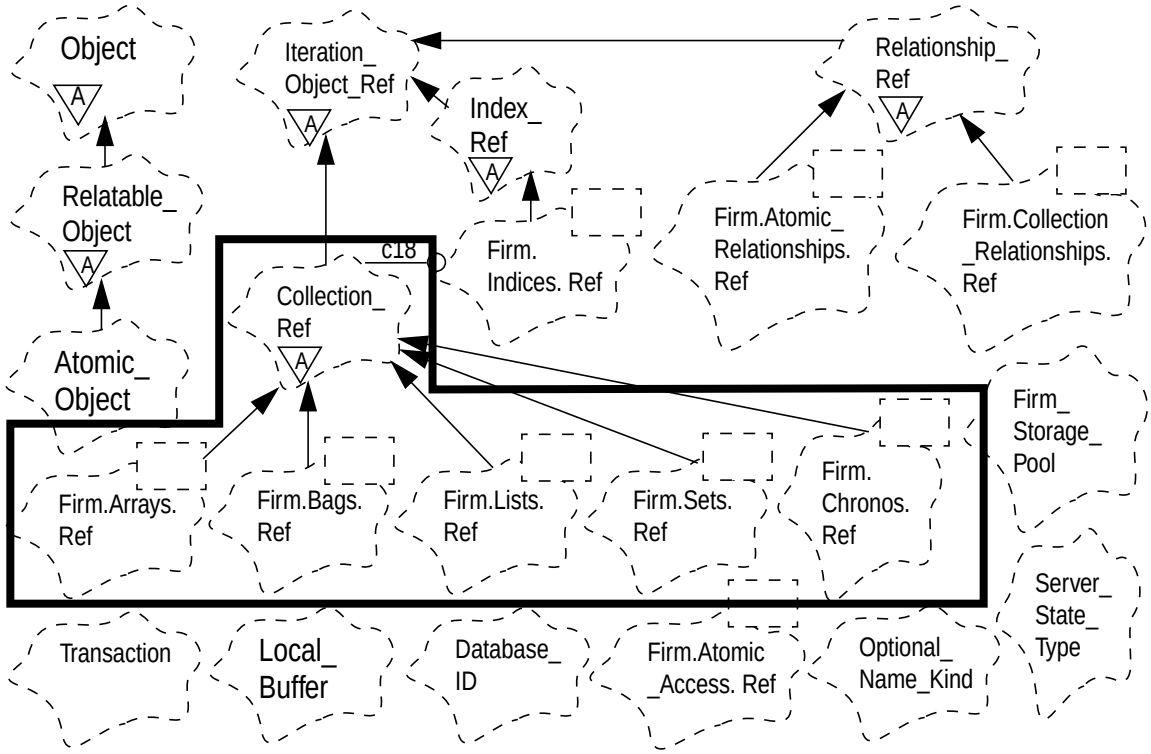

Figure 8: The Collection_Ref type hierarchy

The type hierarchy for the Collection_Ref type is shown in Figure 8 on page 23. Keep in mind that the Firm package only contains the specification for the Collection_Ref type; its descendant types are declared in **generic** child packages. The type hierarchy is presented here only for illustration. Note in Figure 8 on page 23 that the Collection_Ref type is derived from the abstract Iteration_Object_Ref type. FIRM's iterator operations (First, Last, Next, Prior, etc.) are defined for the Iteration_Object_Ref type and therefore any type derived from it must provide these operations. This ensures that FIRM's collections, relationships and indices all provide the basic iteration operations.

The generic child packages which implement FIRM's collections are each instantiated with a "root" type for the collection, and each contains a Ref type which is derived from the Collection_Ref type. The use of a generic package here allows a collection to contain objects of the "root" instantiation type *as well as* any type derived from it. FIRM's collections are therefore **class-wide**.

To ensure a consistent interface for all collection types in FIRM, the Firm package defines several abstract functions and procedures which implement the operations specified in paragraph 2.3.5 of [ODMG 96]. These operations are defined upon the Collection_Ref type. The property functions defined in paragraph 2.3.5 of [ODMG 96] are implemented in the Firm package as class-wide functions for the Collection_Ref type (e.g. they accept parameters of type Collection_Ref'class). The following sections describe these operations and property functions.

3.4.1 Collection properties

In accordance with the properties that all FIRM collections have, the FIRM package provides the following abstract functions which return property information for any collection reference derived from the Collection_Ref type:

Function Name	Description
Cardinality	Function which accepts a collection reference and an optional transaction and returns the number of objects in the collection
Is_Empty	Function which accepts a collection reference and an optional transaction and returns TRUE if there are no objects in the collection, FALSE otherwise
Is_Indexed ⌘	Function which accepts a collection reference and an optional transaction and returns TRUE if an index has been created for the collection, FALSE otherwise
Persistence ⌘	Function which accepts a collection reference and an optional transaction and returns the collection's persistence (LOCAL, GLOBAL, or PERSISTENT).

Table 5: Collection properties

Function Name	Description
Get_Tag ⌘	Function which accepts a collection reference and returns the tag (type Ada.Tags.Tag) of the “root” type that the collection was instantiated with.

Table 5: Collection properties

3.4.2 Collection operations

In addition to the properties mentioned in section 3.4.1, the following abstract operations are provided for the Collection_Ref type:

Name	Description
Bind	Procedure to give a collection a name ([ODMG 96], paragraph 2.9, pg. 33)
Create	Function which accepts a database ID, a desired persistence (see section 3.1.1 on page 14) and possibly other parameters. Returns a collection reference.
Copy	Procedure which accepts two collection references. The “destination” collection is vacated and the objects in the “source” collection are inserted into the destination collection so that both collections will contain the same objects.
Delete	Procedure which accepts a collection reference. The specified collection is deleted from the database
Insert_Element	Procedure which accepts an atomic object reference and a collection reference. The object is inserted into the specified collection right after the current task’s iterator unless the semantics of the collection are otherwise. (For example, objects are always inserted at the end of a chrono regardless of the position of the iterator). You can insert an object which is either a) the same type as the type that the collection package was instantiated with (e.g. the “root” type for the collection) or b) any type derived from the “root” type
Remove_Element	Procedure which accepts a collection reference and an atomic object reference. The specified atomic object is removed from the collection
Contains_Element	Function which accepts a collection reference and an atomic object reference. Returns boolean TRUE if the specified object is in the collection, FALSE otherwise
Lookup	Given an object’s name, returns a Collection_Ref’class for the collection object (inherited in accordance with paragraph 2.9 of [ODMG 96]).
Vacate ⌘	Procedure which accepts a collection reference. The collection is emptied of all its objects. The objects themselves are not deleted from the database, they are simply removed from the collection

Table 6: Collection operations

Name	Description
First ⌘	Iterator operator which accepts a collection reference and returns a pointer (Firm.Atomic_Access.Ptr) to the first object in the collection
Last ⌘	Iterator operator which accepts a collection reference and returns a pointer (Firm.Atomic_Access.Ptr) to the last object in the collection
Next	Iterator operator which accepts a collection reference and returns a pointer (Firm.Atomic_Access.Ptr) to the next object in the collection. Equivalent to the **next** operator in ODMG Iterator class (see [ODMG 96], paragraph 2.3.5, pg. 18).
Prior ⌘	Iterator operator which accepts a collection reference and returns a pointer (Firm.Atomic_Access.Ptr) to the prior object in the collection
Reset	Iterator operator which accepts a collection reference and resets the iterator for the current task so that it is logically pointing to a location just prior to the first object in the collection. Equivalent to the **reset** operator in ODMG Iterator class (see [ODMG 96], paragraph 2.3.5, pg. 18).
Get_Element	Function which accepts a collection reference and returns a pointer (Firm.Atomic_Access.Ptr) to the object currently pointed to by the collection's iterator (each concurrent task has its own iterator). Equivalent to the **get_element** operator in ODMG Iterator class (see [ODMG 96], paragraph 2.3.5, pg. 18).
First ⌘	Iterator operator which accepts a relationship reference and returns a reference (Collection_Ref'class) to the first collection object on the "other side" of the relationship from the atomic object specified in the most recent call to Set_Iterator (see section 3.7 on page 38)
Last ⌘	Iterator operator which accepts a relationship reference and returns a reference (Collection_Ref'class) to the last collection object on the "other side" of the relationship from the atomic object specified in the most recent call to Set_Iterator (see section 3.7 on page 38)
Next ⌘	Iterator operator which accepts a relationship reference and returns a reference (Collection_Ref'class) to the next collection object on the "other side" of the relationship from the atomic object specified in the most recent call to Set_Iterator (see section 3.7 on page 38)
Prior ⌘	Iterator operator which accepts a relationship reference and returns a reference (Collection_Ref'class) to the prior collection object on the "other side" of the relationship from the atomic object specified in the most recent call to Set_Iterator (see section 3.7 on page 38)

Table 6: Collection operations

Name	Description
Get_Element ⌘	Function which accepts a relationship reference and returns a reference (Collection_Ref'class) to the collection object currently pointed to by the relationship's iterator (each concurrent task has its own iterator; see section 3.7 on page 38)

Table 6: Collection operations

The operations in Table 5 and Table 6 correspond to those specified in paragraph 2.3.5 in [ODMG 96], except those followed by a ⌘ which are extra operations provided by the FIRM ODBMS. Collection iterators in FIRM have a different interface than the Iterator class defined in [ODMG 96] (see paragraph 2.3.5, pg. 18). Iterators in FIRM are not first-class objects as they are in [ODMG 96]; instead, they are built into the FIRM types which support iteration (collections, relationships, and indices). Each concurrent task gets one iterator (cursor) for each instance of an "iteratable" type. FIRM iterators are therefore cursors. All of the operations listed in Table 6 require a valid transaction when they are used with objects of Global or Persistent persistence.

3.4.3 Nesting collections

The ODMG object model allows collections to contain not only atomic objects but also other collections (see [ODMG 96], section 2.3.5 on page 17). The FIRM object model restricts collections such that they may only contain atomic objects. If you want to "nest" collections inside each other, you can do so by defining an atomic object with a collection reference attribute. For example:

```
type Thing is new Firm.Atomic_Object with
   record
      Stuff : Integer; -- whatever the attributes of a Thing are
   end record;

package Thing_Access is new Firm.Atomic_Access(Thing);

-- Instantiate Sets package for the "Thing" type
package Sets_Of_Things is new Firm.Sets(
   Member_Type => Thing,
   Member_Access => Thing_Access);

-- Now define an atomic type which has a set reference as its attribute
type A_Set_Of_Things is new Firm.Atomic_Object with
   record
      The_Set : Sets_Of_Things.Ref := Sets_Of_Things.Null_Ref;
   end record;

package A_Set_Of_Things_Access is new Firm.Atomic_Access(
   A_Set_Of_Things);

-- Instantiate Sets package for the "A_Set_Of_Things" type
package Sets_Of_Sets_Of_Things is new Firm.Sets(
   Member_Type => A_Set_Of_Things,
   Member_Access => A_Set_Of_Things_Access);
```

This approach does not offer the same degree of referential integrity as a truly nested collection, however. This is because the deletion of an "inner" collection would not result in the deletion of the atomic object which points to it. This means that the "intermediate" atomic object would still be left in the "outer" collection. Referential integrity must therefore be provided by the application when nesting collections in this manner.

The preferred approach for realizing multi-dimensional collections with FIRM is to use FIRM's spatial indices, which will be added later in 1998.

3.5 The Optional_Name_Kind type

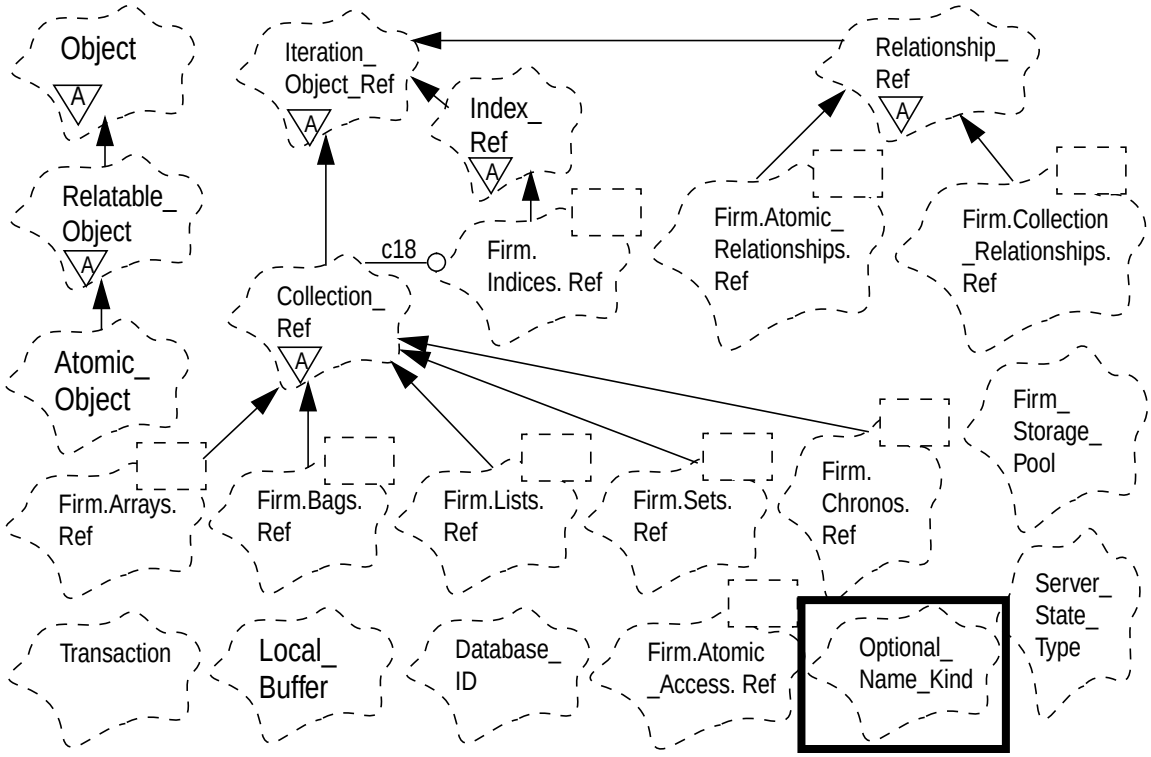

Figure 9: The Optional_Name_Kind type

In the FIRM ODBMS, all of the four major object types (atomic objects, collections, indices, and relationships) can have names. In accordance with [ODMG 96], section 2.9, FIRM allows atomic objects and collection objects to be assigned names via their respective Bind operators (see page 19 and page 25, respectively). As in [ODMG 96], these names are optional. The Optional_Name_Kind type is an enumerated type which lists the types of names in FIRM which are optional[1]. The Optional_Name_Kind type is shown in Figure 9.

3.5.1 Operations on the Optional_Name_Kind type

Table 7 summarizes the operation provided for the Optional_Name_Kind type.

1. Not all object names are optional in FIRM. FIRM's index and relationship objects are assigned a single unique name at creation time which cannot be unbound (except by deleting the object). Each index or relationship object may have only one name (the one assigned to it at the time of its creation). This is not at variance with [ODMG 96], since [ODMG 96] does not have first-class index or relationship objects.

Operation	Description
Unbind	Accepts as input a database ID, a string containing an object name, an optional transaction, and the kind of name to be unbound (type Optional_Name_Kind, value is either Atomic_Name or Collection_Name). The specified name is removed from the specified kind of object in the specified database.

Table 7: Operations on the Optional_Name_Kind type

The Unbind operation is not provided in [ODMG 96] for object names; it has been added for FIRM.

3.6 The Index_Ref type hierarchy

The Index_Ref type provides logical pointers to index objects. The Index_Ref type is abstract; concrete types for index references are provided by the Firm.Indices generic child package. The Firm.Indices package provides a Ref type for accessing index objects.

All of the operations which can be performed on a FIRM index use references to access the index. That is, these functions and procedures accept descendants of Index_Ref as their index parameters rather than the actual index objects.

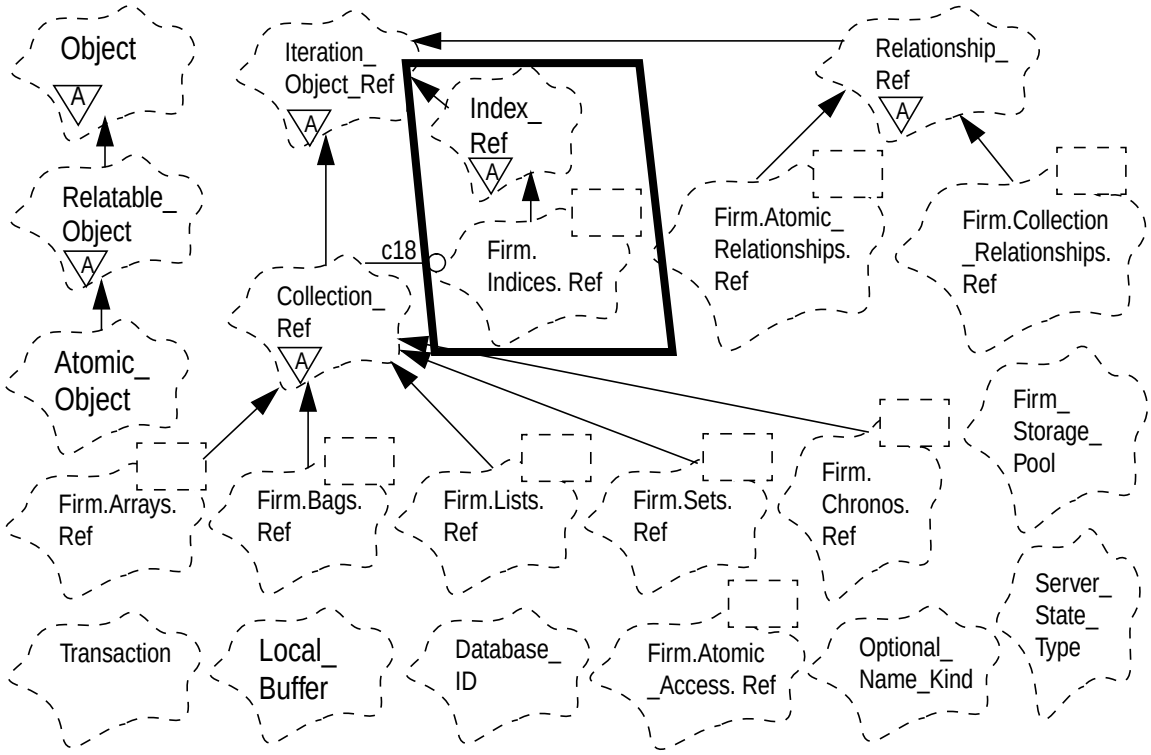

Figure 10: The Index_Ref type hierarchy

The type hierarchy for the Index_Ref type is shown in Figure 10. Keep in mind that the Firm package only contains the specification for the Index_Ref type; its descendant types are declared in **generic** child packages. The type hierarchy is presented here only for illustration. Note in Figure 10 on page 31 that the Index_Ref type is derived from the abstract Iteration_Object_Ref type. FIRM's iterator operations (First, Last, Next, Prior, etc.) are defined for the Iteration_Object_Ref type and therefore any type derived from it must provide these operations. This ensures that FIRM's collections, relationships and indices all provide the basic iteration operations.

The generic child package which implements FIRM's indices (Firm.Indices) is instantiated with a "root" type for the index, an equality operator for the root type's class, and a less-than operator for the root type's class. This package provides a Ref type which is derived from the Index_Ref type. The use of a generic package here allows the index to order objects of the "root" instantiation type *as well as* any type derived from it. FIRM's indices are therefore **class-wide**.

Note that Figure 10 on page 31 shows a collaboration named "c18" between the Index_Ref and Collection_Ref types. This collaboration implies that when an index is created on a collection, the collection's Get_Tag property function (see section 3.4.1, page 25) is called by the index's Create operation. This allows the index to check the type that the collection package was instantiated with to be sure it's the same as the type that the index package was instantiated with. This is necessary to ensure that all of the objects that will ever be inserted into the collection can be properly sorted by the index.

3.6.1 Operations on the Index_Ref type

To ensure a consistent interface for all index types in FIRM, the Firm package defines several abstract functions and procedures which implement the index operations. These operations are summarized in Table 8.

The [ODMG 96] specification does not give much detail about the use of indices in an ODBMS. The semantics of object keys are described in paragraph 2.2.3 on page 14 of [ODMG 96], but no indexing operations are specified. Therefore, all of the operations listed in Table 8 are additions for FIRM. All of the operations in Table 8 require a valid transaction when they are used with objects of Global or Persistent persistence.

Operation	Description
Create_Global_Pool	Procedure to preallocate GLOBAL storage for indices with GLOBAL persistence. Storage is allocated by providing the desired database ID and number of instances.
Create	Function which accepts a collection reference, a name string, and a Boolean parameter which indicates whether or not the index should allow duplicate keys. Returns a reference to an index which will sort the objects in the collection using the "<" and "=" functions provided for the generic instantiation. (The index is created with the same persistence as its collection)
Delete	Procedure which accepts an index reference. Removes the specified index from its collection

Table 8: Operations on the Index_Ref type

Operation	Description
Find_Match	Function which accepts an index reference and an object with its "key" attributes set (the "key" attributes are those used by the index's "<" and "=" functions to sort the objects). Returns a reference for an object with matching key attributes in the specified index. If there is no matching object, Null_ Atomic_Ref is returned and the iterator is left pointing at the object just prior to where the matching object would have been.
Lookup	Function which accepts an ASCII string containing an index name. Returns a reference for the index with the specified name (inherited in accordance with paragraph 2.9 of [ODMG 96]).
First	Iterator operator which accepts an index reference and returns a pointer (Firm.Atomic_Access.Ptr) to the first object in the index (e.g. the object with the least key)
Last	Iterator operator which accepts an index reference and returns a pointer (Firm.Atomic_Access.Ptr) to the last object in the index (e.g. the object with the greatest key)
Next	Iterator operator which accepts an index reference and returns a pointer (Firm.Atomic_Access.Ptr) to the next object in the index (e.g. the object with the next greater key)
Prior	Iterator operator which accepts an index reference and returns a pointer (Firm.Atomic_Access.Ptr) to the prior object in the index (e.g. the object with the next lesser key)
Reset	Iterator operator which accepts an index reference and resets the iterator for the current task so that it is logically pointing to a location just prior to the first object in the index
Get_Element	Function which accepts an index reference and returns a pointer (Firm.Atomic_Access.Ptr) to the object currently pointed to by the index's iterator (each concurrent task has its own iterator)

Table 8: Operations on the Index_Ref type

[ODMG 96] provides no method for searching an index. The FIRM ODBMS uses the Find_Match operation, which accepts as its parameters an Index_Ref and an object with its key attributes set to the desired values. This function will return a pointer (Firm.Atomic_ Access.Ptr) to the object in the indexed collection which has matching key attributes. Null is returned if no such object exists in the collection.

The key attributes of an object are those attributes which are used to order the index. These attributes are used by the sorting functions ("Equal_To" and "Less_Than") that the Firm. Indices package is instantiated with. For example, to create an index for objects of type

Person, the application programmer would have to decide how Person objects were to be sorted. It might be desirable to sort Person objects both by surname and age. In this case, the key attributes of a Person object would be the surname and age attributes. The application developer could then create two indices for a collection of Persons. For one index, the programmer would instantiate Firm.Indices with “=” and “<” operators which compare Person surnames. For the other index, the programmer would instantiate Firm.Indices with “=” and “<” operators which compare Person ages.

Note that it is necessary for the application programmer to write these sorting operators because the FIRM ODBMS itself does not have any knowledge about the internals of any user-defined types (e.g. the types derived from the Atomic_Object type).

3.6.2 An index example

As an example, suppose that we wish to create two indices for a collection of objects of the Person type. One of these indices will allow iteration of Person objects in surname order, the other in birth date (age) order. The declaration of the Person type and a collection package for it might look like this:

```
package Persons is

   type Person is new Firm.Atomic_Object with private;
   function Equal_To_By_Surname (L,R: in Person'Class) return Boolean;
   function Less_Than_By_Surname (L,R: in Person'Class) return Boolean;
   function Equal_To_By_Birthdate (L,R: in Person'Class) return Boolean;
   function Less_Than_By_Birthdate (L,R: in Person'Class) return Boolean;

private ...
end Persons;

package Person_Access is new Firm.Atomic_Access(Persons.Person);

package Person_Sets is new Firm.Sets (
   Member_Type => Persons.Person,
   Member_Access => Person_Access );
```

The declarations above give us an atomic object type which represents persons and a package which can create and manage sets of Person objects. (Here, *set* refers to the [ODMG 96] set collection type; see [ODMG 96], section 2.3.5.1). We can also create packages which create and manage indices on collections of Person objects:

```
package Persons_By_Surname is new Firm.Indices (
   Keyed_Atomic_Object => Persons.Person,
   Member_Access => Person_Access,
   Equal_To => Persons.Equal_To_By_Surname,
   Less_Than => Persons.Less_Than_By_Surname );

package Persons_By_Birthdate is new Firm.Indices (
   Keyed_Atomic_Object => Persons.Person,
   Member_Access => Person_Access,
   Equal_To => Persons.Equal_To_By_Birthdate,
   Less_Than => Persons.Less_Than_By_Birthdate );
```

If we now wanted to create a set of Persons and index it by surname and birthdate, we could do it like this:

```
MULTIPLE_INDEX_BLOCK:
declare
   My_Set : Person_Sets.Create (
      DB => My_DB,
      Persistence => Firm.LOCAL );
   Person_Ptr : Person_Access.Ptr;
begin

   -- Create an index on the set of Person objects which orders the
   -- Person objects by surname and allows no duplicate keys
   Persons_By_Surname.Create (
      C => My_Set,
      Name => “My persons by surname”,
      Duplicate_Keys => FALSE );

   -- Create an index on the set of Person objects which orders the
   -- Person objects by birthdate and which allows duplicate keys
   Persons_By_Birthdate.Create (
      C => My_Set,
      Name => “My persons by birthdate”,
      Duplicate_Keys => TRUE ); -- allows 2 persons in set to have same BD

   -- Now insert Persons into the set. They will be ordered automatically
   My_Set.Insert_Element (.....);

   -- Iterate set of persons in surname order
   ITERATE_BY_SURNAME:
   loop
      Person_Ptr := Persons_By_Surname.Next (...);
   end loop ITERATE_BY_SURNAME;

   -- Iterate set of persons in birthdate order
   ITERATE_BY_BIRTHDATE:
   loop
      Person_Ptr := Persons_By_Birthdate.Next (...);
   end loop ITERATE_BY_BIRTHDATE;

end MULTIPLE_INDEX_BLOCK;
```

3.6.3 Error handling for indices

The FIRM ODBMS will handle errors that occur during index operations in accordance with Table 9.

Operation	Condition	Result
Create	The specified collection was created from a collection package which was instantiated with a different type than the index package	Raise OML_Error exception
First	Collection that index was created for is empty	Set task's iterator to null and return null; do not raise an exception.
Last	Collection that index was created for is empty	Set task's iterator to null and return null; do not raise an exception.
Next	Task's iterator is pointing to the last object in the index	Set task's iterator to null and return null; do not raise an exception.
Prior	Task's iterator is pointing to the first object in the index	Set task's iterator to null and return null; do not raise an exception.
Find_Match	There is no object whose key attributes match those of the object supplied in the call to Find_Match	Return null, do not raise an exception
Get_Element	Task's iterator is null	Return null, do not raise an exception

Table 9: Error handling in the FIRM ODBMS for indices

3.7 The Relationship_Ref type hierarchy

The Relationship_Ref type provides logical pointers to relationship objects. The Relationship_Ref type is abstract; concrete types for relationship references are provided by the Firm.Atomic_Relationships and Firm.Collection_Relationships generic child packages. These packages each provide a Ref type for accessing relationship objects.

All of the operations which can be performed on a FIRM relationship use references to access the relationship. That is, these functions and procedures accept descendants of Relationship_Ref as their relationship parameters rather than the actual relationship objects.

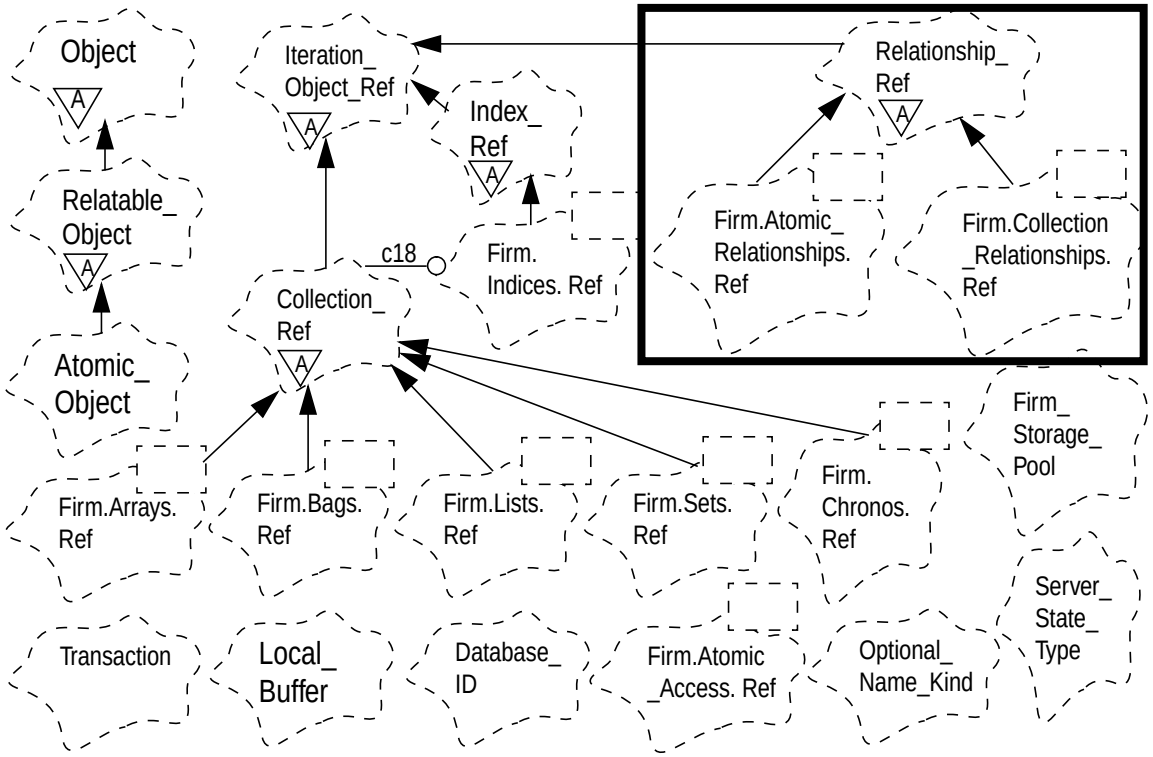

Figure 11: Relationship_Ref type hierarchy

The type hierarchy for the Relationship_Ref type is shown in Figure 11. This type definition is the FIRM ODBMS' implementation of the relationship types defined in [ODMG 96], paragraph 2.5.2, pp.25-27. Keep in mind that the Firm package only contains the specification for the Relationship_Ref type; its descendant types are declared in **generic** child packages. The type hierarchy is presented here only for illustration. Note in Figure 11 on page 38 that the Relationship_Ref type is derived from the abstract Iteration_Object_ Ref type. FIRM's iterator operations (First, Last, Next, Prior, etc.) are defined for the Iteration_Object_Ref type and therefore any type derived from it must provide these

operations. This ensures that FIRM's collections, relationships and indices all provide the basic iteration operations.

The generic child packages which implement FIRM's relationships (Firm.Atomic_Relationships and Firm.Collection_Relationships) are instantiated with the types which are to be related. The generic formal parameters of the Firm.Atomic_Relationships package allow two types derived from Firm.Atomic_Object to be related. The generic formal parameters of the Firm.Collection_Relationships package allow an atomic type (e.g. a type derived from Firm.Atomic_Object) to be related to a collection type. The use of generic packages here allows relationship traversal paths to be established between objects of the "root" instantiation types *as well as* any types derived from them. FIRM's relationships are therefore **class-wide**.

In the FIRM ODBMS, relationships are "first-class" objects (that is, they have OIDs and unique names). In this respect, the FIRM ODBMS has already incorporated one of the possible future changes to the ODMG object model (see [ODMG 96], paragraph 2.10.4, item 1). Making relationships first-class objects rather than trying to make them part of the type definition for a class (see page 26 of [ODMG 96]) has the following benefits:

1. **Dynamic relationships** - Making relationships objects allows them to be created and deleted as needed, thus giving an application more flexibility.
2. **Better class normalization** - Attempting to make a relationship specification part of a type definition could result in relationship attributes being replicated in every instance of a class. It would not be desirable for every object in a class to contain a relationship's cardinality, etc. since this information is not unique to the object's OID and thus violates good class normalization principles (see [Lee 95], pg. 26). Of course, instead of including all of the information about a relationship in each object you could instead give each object a pointer to a place where all of this information is kept. This approach would at least reduce the amount of duplicated information, but in fact doing this is much like making the relationship a first-class object.
3. **No need for a preprocessor** - Since relationships are not part of Ada-95, any attempt to include a relationship efficiently in a tagged type definition will probably require some "extra keywords" and an Ada preprocessor. This obviously violates our stated goal of providing a maximally seamless binding to Ada-95 (see section 1.2.2 on page 8).

The FIRM ODBMS only supports unordered relationships. That is, the traversal paths from one object to another are iterated in an order determined by FIRM, not the application.

While the [ODMG 96] object model only supports relationships between two distinct atomic object types (a "binary" relationship), FIRM allows relationships to be created

between one (“unary”) or two atomic object types or between an atomic object type and a collection type. Unary relationships allow instances of the same type to be related to each other, i.e. a Friends relationship could be created for the Person type so that links could be established between those persons who were friends with each other. Allowing a relationship to be created between an atomic object and collection type allows atomic objects to be related to *groups* of other atomic objects. For example, a ComposedFrom relationship could be established between FusedTrack objects and sets of RadarDetection and InfraredDetection objects. FIRM’s relationships are more flexible so that they can be used to construct a database schema that is as similar as possible to the “real world.”

3.7.1 One-to-one relationships

A one-to-one relationship may be created in FIRM by invoking the Firm.Create operator with its “Rel_Type” parameter set to Firm.ONE_TO_ONE. A one-to-one relationship “A” between two types “U” and “V” is a relationship where any object of type “U” may have one traversal path in “A” linking it to one object of type “V” (see Figure 12). Note that there may be several relationships defined between two classes, so an object may have many traversal paths which belong to many different relationships.

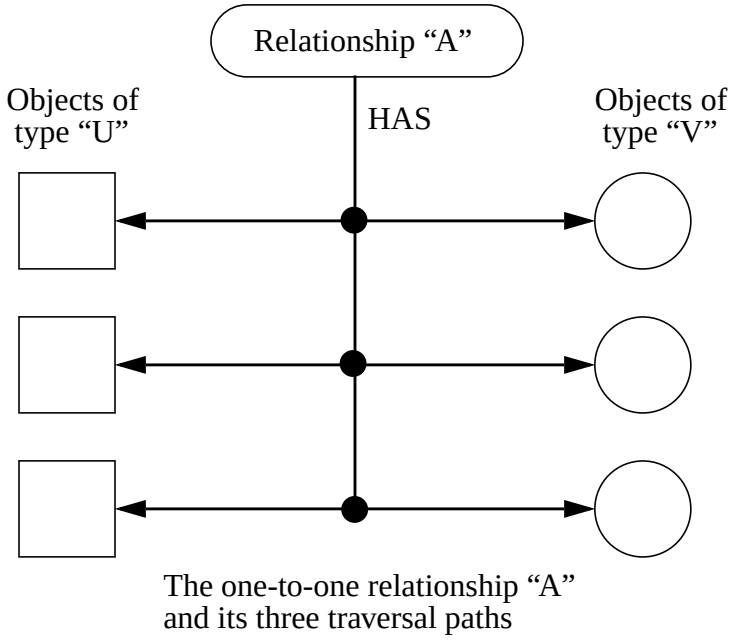

Figure 12: Example of a one-to-one relationship

3.7.2 One-to-many relationships

A one-to-many relationship may be created in FIRM by invoking the Firm.Create operator with its “Rel_Type” parameter set to Firm.ONE_TO_MANY. A one-to-many relationship “B” between two types “W” and “X” is a relationship where any object of type “W” may have N traversal paths in “B” linking it to N objects of type “X”. Each object of type “X”, however, may be linked to only one object of type “W”. See Figure 13.

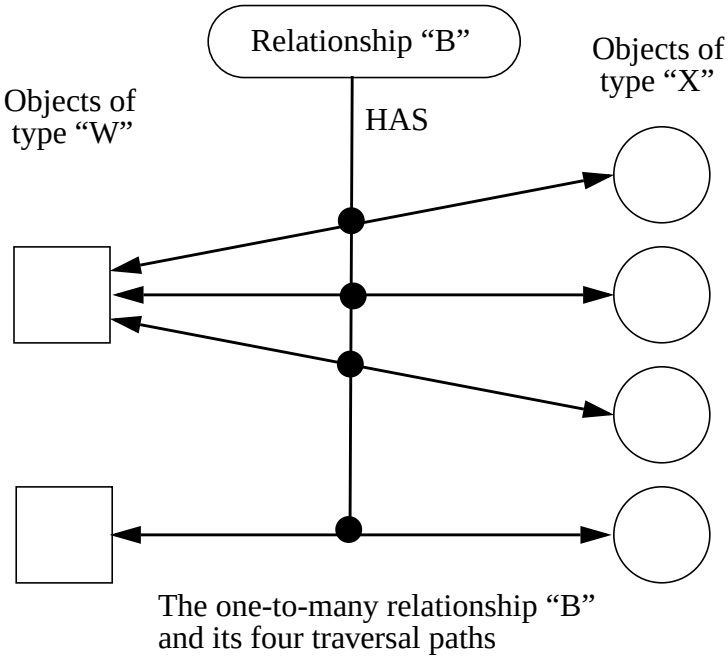

Figure 13: Example of a one-to-many relationship

3.7.3 Many-to-many relationships

A many-to-many relationship may be created in FIRM by invoking the Firm.Create operator with its “Rel_Type” parameter set to Firm.MANY_TO_MANY. A many-to-many relationship “C” between two types “Y” and “Z” is a relationship where any object of type “Y” may have N traversal paths in “C” linking it to N objects of type “Z”, and each object of type “Z” may in turn have M traversal paths in “C” linking it to M objects of type “Y”. See Figure 14. Note how similar this looks to Figure 13 on page 42; by adding one traversal path, we see now that one of the “Z” objects has two traversal paths to two “Y” objects. This makes the relationship many-to-many rather than one-to-many, since now it is possible to go to more than one “Y” object from a “Z” object. Thus, a many-to-many relationship looks like one-to-many relationships that “overlap”.

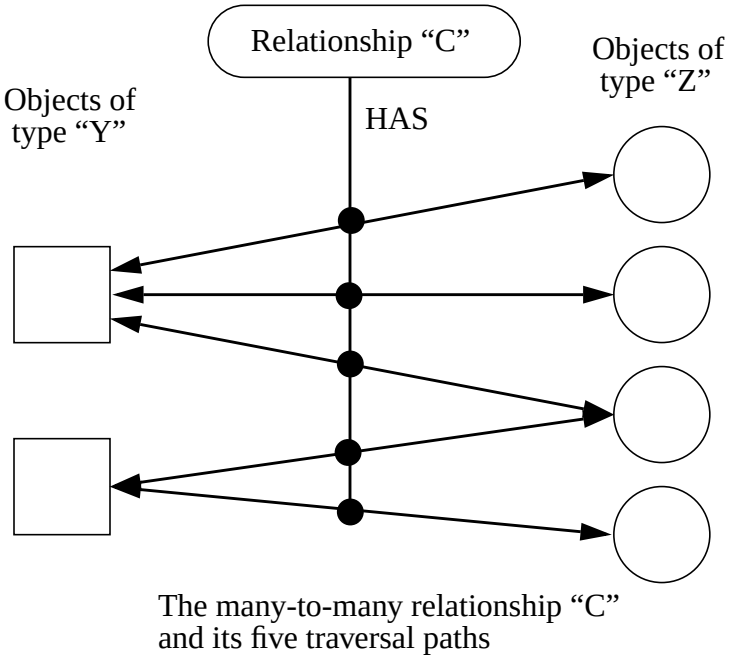

Figure 14: Example of a many-to-many relationship

3.7.4 Operations on relationships

Table 10 summarizes the abstract operations that all FIRM relationship types must provide. These abstract operations are in the Firm package. The generic child packages for relationships may add other operations. The object model in the [ODMG 96] specification does not specify operators for relationships, so those in Table 10 are provided by FIRM. All of the operations in Table 10 require a valid transaction when they are used with relationships of Global or Persistent persistence.

Operation	Description
Create	Function which accepts a database ID, a relationship type (one of three enumeration values: ONE_TO_ONE, ONE_TO_MANY, MANY_TO_MANY), a name, and the desired persistence (see section 3.1.1 on page 14). Returns a reference for a relationship with the cardinality specified in the relationship type parameter. The relationship created will allow traversal paths to be added between objects of the types specified in the generic instantiation (or any type derived from these types).
Delete	Procedure which accepts a relationship reference. All traversal paths that are part of this relationship are deleted, as is the relationship itself
Lookup	Function which accepts a string containing a name and returns a reference to a relationship. Inherited as specified in paragraph 2.9 of [ODMG 96].
First	Iterator operator which accepts a relationship reference and returns a pointer (Firm.Atomic_Access.Ptr) to the first atomic object on the “other side” of the relationship from the object that is the iteration “base” object
Last	Iterator operator which accepts a relationship reference and returns a pointer (Firm.Atomic_Access.Ptr) to the last atomic object on the “other side” of the relationship from the object that is the iteration “base” object
Next	Iterator operator which accepts a relationship reference and returns a pointer (Firm.Atomic_Access.Ptr) to the next atomic object on the “other side” of the relationship from the object that is the iteration “base” object
Prior	Iterator operator which accepts a relationship reference and returns a pointer (Firm.Atomic_Access.Ptr) to the prior atomic object on the “other side” of the relationship from the object that is the iteration “base” object

Table 10: Operations on the Relationship_Ref type

Operation	Description
Reset	Iterator operator which accepts a relationship reference and resets the iterator for the current task so that it is logically pointing to a location just prior to the first object on the “other side” of the relationship from the object that is the iteration “base” object
Get_Element	Function which accepts a relationship reference and returns a pointer (Firm.Atomic_Access.Ptr) to the atomic object currently pointed to by the relationship’s iterator (each concurrent task has its own iterator)

Table 10: Operations on the Relationship_Ref type

The iterator operators which return references for collections that are on the “other side” of a relationship are in Table 6 on page 25.

3.7.5 A relationship example

As an example, suppose that we wish to create a many-to-many relationship between student objects and course objects in a FIRM ODBMS application. This could be done as follows:

```
with Students; with Student_Access;
with Courses; with Course_Access;
with Firm.Atomic_Relationships;
package Student_Course_Relationships is new Firm.Atomic_Relationships (
   From_Type => Students.Student,
   From_Access => Student_Access,
   To_Type => Courses.Course,
   To_Access => Course_Access );
....
CREATE_ENROLLMENT_RELATIONSHIP:
declare
   EE_Enrollment_Ref : Student_Course_Relationships.Ref;
begin
   EE_Enrollment_Ref := Student_Course_Relationships.Create (
      DB => My_Database_ID,
      Persistence => Firm.GLOBAL,
      Name => “EE Enrollment”,
      Rel_Type => Firm.MANY_TO_MANY );
end CREATE_ENROLLMENT_RELATIONSHIP;
```

If we wanted have the student object named Mike enrolled in two classes, EE101 and EE102, and the student object named John enrolled in EE201, the FIRM ODBMS application code could look like this:

```
ENROLL_STUDENTS_BLOCK:
declare
   -- Get reference for student/courses relationship
   EE_Enrollment_Ref : Student_Course_Relationships.Ref :=
      Student_Course_Relationships.Lookup(“EE Enrollment”);
begin
   -- Add Mike to EE101 and EE102
   Student_Course_Relationships.Add_Traversal_Path (
      From => Firm.Lookup(“Mike”), --Get ref for student Mike
      To => Firm.Lookup(“EE101”), -- Get ref for course EE101
      R => EE_Enrollment_Ref,
      T => My_Transaction );

   Student_Course_Relationships.Add_Traversal_Path (
      From => Firm.Lookup(“Mike”),
      To => Firm.Lookup(“EE102”),
      R => EE_Enrollment_Ref,
```

```
      T => My_Transaction );
   -- Add John to EE201
   Student_Course_Relationships.Add_Traversal_Path (
      From => Firm.Lookup("John"),
      To => Firm.Lookup("EE201"),
      R => EE_Enrollment_Ref,
      T => My_Transaction );
end ENROLL_STUDENTS_BLOCK;
```

Now, suppose that we wanted to print out all of the students enrolled in the course named EE101. Presuming the existence of a Display_Student_Info procedure, we could simply use an iterator as follows:

```
PRINT_EE101_ENROLLMENT:
declare
   -- Get reference for student/courses relationship
   EE_Enrollment_Ref : Student_Course_Relationships.Ref :=
      Student_Course_Relationships.Lookup("EE Enrollment");
   -- Reference for a student in EE101
   Student_Ptr : Student_Access.Ptr;
   Local_Buffer : Firm.Local_Buffer_Ptr := new Firm.Local_Buffer(1000);
begin
   -- Set iterator on EE101 course object
   Student_Course_Relationships.Set_Iterator (
      R => EE_Enrollment_Ref,
      On => Firm.Lookup(Db => My_Database_ID, Name => "EE101" ...).all,
      T => My_Transaction );
   -- Now iterate through all of the students in EE101
   loop
      -- Iterate to next student. The first time the Next operator is
      -- called, it will go to the first student.
      Student_Ptr := Student_Course_Relationships.Next (
         R => EE_Enrollment_Ref,
         Buffer => Local_Buffer,
         T => My_Transaction );
      -- exit loop after last student
      exit when Student_Ptr = null;

      Display_Student_Info (Student_Ptr.all);
   end loop;
end PRINT_EE101_ENROLLMENT;
```

3.7.6 Error handling for relationships

The FIRM ODBMS will handle errors that occur during relationship operations in accordance with Table 11.

Operation	Condition	Result
First, Last, Next, or Prior	Set_Iterator has not been called to establish which object to iterate from	Raise OML_Error exception (see section 3.11 on page 53)
First, Last, Next, or Prior	There are no traversal paths from the current object that are in the specified relationship	Set task's iterator to null and return null; do not raise an exception.
Get_Element	Task's iterator is null	Return null, do not raise an exception

Table 11: Error handling in the FIRM ODBMS for relationships

3.8 The Transaction type

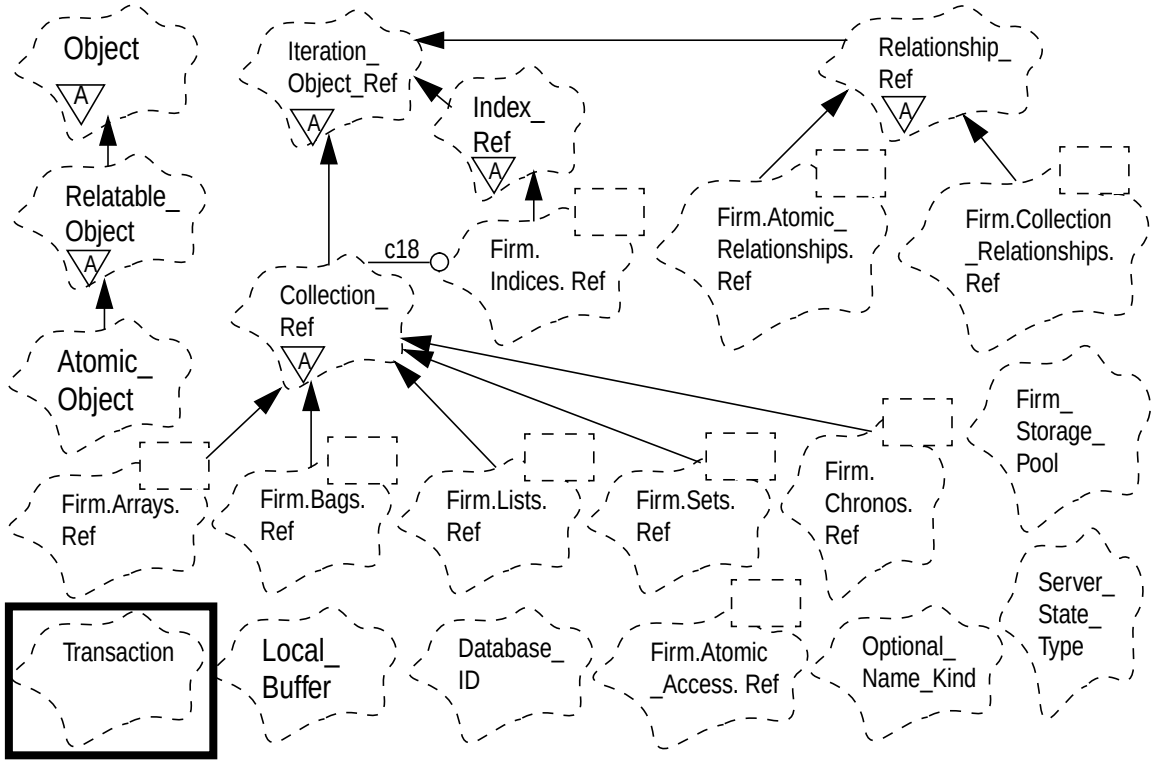

Figure 15: The Transaction type

As specified in paragraph 2.8, page 31 of [ODMG 96], the FIRM ODBMS allows objects with Global or “Persistent” persistence (see section 3.1.1 on page 14) to be created, deleted, updated or read only within the context of a transaction. A transaction is the smallest unit of work that a database application task can perform on one or more such objects such that the result of this work is usable by other tasks. For example, if we were updating the address and telephone information for an Employee object, we would define the bounds of the transaction to include the updates for the street, city, phone number, etc. since we wouldn’t want anyone to use the object until all of this information was correct. A FIRM application developer indicates the bounds of a transaction using the begin_transaction and commit operations. Thus, a FIRM application program for our example might look like this:

```
package Employee_Access is new Firm.Atomic_Access(Employee);
My_Transaction := Firm.Begin_Transaction(Firm.UPDATE); -- start of trans.

Employee_Ptr := Employee_Access.Update_Object( -- This call uses the
```

```
   Db => My_Db_ID, Name => “John Doe”,-- object name instead of
   T => My_Transaction); -- a reference

Employee_Ptr.Street := “1305 James Street”; -- Update the address
Employee_Ptr.City := “Syracuse”;
Employee_Ptr.State := “NY”;
Employee_Ptr.ZIP := 13206;
Employee_Ptr.Phone := “315-456-1000”; -- Update the phone #

Firm.Commit(My_Transaction); -- end of the transaction
```

Thus, the Transaction type is used by the FIRM ODBMS to indicate the bounds of database transactions. Only one task may be active within a transaction. This means that an Ada program must not create a task or rendezvous with another task within a transaction.

The Transaction type corresponds to the Transaction type specified in [ODMG 96], paragraph 2.8.2, pp. 32-33. A Null_Transaction constant is used as the default for all FIRM database operations where a transaction is required. This allows FIRM’s operations to work on objects with Local persistence (which do not require a transaction) as well as objects with Global or Persistent persistence (which do require a transaction).

3.8.1 Operations on the Transaction type

Table 12 summarizes the operations for the Transaction type; these operations are those specified in paragraph 2.8.2 of [ODMG 96].

Operation	Description
begin_transaction	Start a new transaction. FIRM allows an optional time limit to be specified. If the transaction exceeds this time limit, the Time_Error exception will be raised (see section 3.11 on page 53).
abort_transaction	Abandon an in-process transaction, return all modified objects to their state prior to the corresponding begin_transaction call
checkpoint	Save the current state of all modified objects into the database but do not end the transaction
commit	Save the current state of all modified objects into the database and end the transaction

Table 12: Operations for the Transaction type

3.9 The Server_State_Type type

The Server_State_Type type is an enumerated type that indicates which of three possible states a FIRM server may be in:

1. **MASTER** - The FIRM server is running on the primary host and a “standby” (i.e. backup) server is being automatically kept up-to-date on a secondary (i.e. backup) host. This dual-redundant configuration is used for systems which require fault tolerance. The application running on a MASTER server is capable of accessing any database for read or write or performing any FIRM operation.
2. **STANDBY** - The FIRM server is running on the secondary (i.e. backup) host, and as such the databases are automatically kept up-to-date by the FIRM server running on the primary host. In general, the application program should suspend itself if the FIRM server is running on the secondary host. This is so the application will not interfere with the mirroring or journalling of the databases or with any other activity occurring on the MASTER server. (NOTE: any “interfering” activity by the application running on the STANDBY server will result in an OML_Error exception). It is possible for the application running on the STANDBY server to do read transactions safely, so that the STANDBY server can be used to off-load some retrieval processing from the MASTER if desired.
3. **SOLO** - The FIRM server is running without a standby server. There is therefore no dual-redundant fault tolerance in this configuration. The application running on a SOLO server is capable of accessing any database for read or write or performing any FIRM operation.

3.9.1 Operations on the Server_State_Type type

Table 13 summarizes the server control operations provided in the Firm package.

Operation	Description
startup	Activates FIRM’s message logging capability and prepares the server for on-line use. Databases may be opened after startup has been invoked. This procedure has an optional parameter for a pointer to a parameterless “callback” procedure. If supplied, the callback procedure will be invoked when the server changes state from STANDBY to SOLO or STANDBY to MASTER.
block_until	Suspends the invoking task until the FIRM server is in the specified state.
shutdown	Closes all open databases, turns off FIRM’s message logging and prepares the FIRM server for being taken off-line (i.e. power-down for maintenance, etc.). If the server is running with a dual-redundant STANDBY server, the STANDBY will be shut down as well.

Table 13: Operations for the Server_State_Type type

3.10 The Firm.Msg_Log package

Nested within the public part of the Firm package is the Msg_Log package. This package encapsulates the operations and types associated with logging messages. The FIRM ODBMS itself uses the operations in this package to log its own messages. The Msg_Log package allows FIRM application programs to use the same message handling facility as FIRM itself.

FIRM's message handling facility is implemented as a component, which means that its implementation may be changed using the Flexible Architecture Builder (FAB). This allows FIRM's error handling facility to be integrated with an existing error handling facility in an embedded system.

3.10.1 Error logging operations

Table 14 summarizes the message logging operations available to FIRM applications.

Operation	Description
Log_Msg	Logs a message for later post-mission analysis. Accepts as input a transaction and a message category type. The message category type is an enumerated type which has five values: Fatal, Error, Informational, Warning, and Security. (The Security category is used for MLS configurations of the FIRM ODBMS to provide the audit trails required for B1 MLS). Log_Msg also accepts as input a string containing the name of the sender, a "what" string that provides a brief description of what happened, and an optional "where" string which provides the name of the Ada unit in which is logging the error. In addition, three optional "why" strings may be passed in to provide extra information.
Display_Last_Msg	This procedure accepts as input a transaction and displays the last error message logged by the transaction. Different implementations of this component can be generated so that the "display" functionality will match the capabilities of the system. In a software development environment, for example, this operation might simply use Ada.Text_IO to display the message to an active window on a workstation. Although this procedure can also be used outside of a transaction (the input transaction id is null), this is discouraged since the displayed message may not be the one expected.

Table 14: Error logging operations in the FIRM API

3.11 Exceptions

Table 15 summarizes the exceptions provided in the Firm package and the conditions under which the exceptions are raised.

Exception	Conditions
Allocation_Error	Raised when an attempt is made to create a GLOBAL or PERSISTENT object and its corresponding storage pool has no free space or, in the case of a storage pool for discriminated types, not enough **contiguous** free space.
Cache_Error	Raised when there is insufficient cache for a persistent storage pool
Configuration_Error	Raised when a maximum value specified in a FIRM constants package or configuration file is exceeded, e.g. as when an attempt is made to create a global storage pool and the maximum number of global pools (as specified by Firm_Config_Constants.Max_Global_Pools) have already been created.
Deadlock_Error	Raised when a transaction deadlocks. When this exception is caught, the current transaction should be aborted and restarted.
Internal_Error	Raised when a software error occurs within the FIRM ODBMS.
OML_Error	Raised when an object is manipulated improperly, i.e. when an application attempts to create or manipulate an object with Global or “Persistent” persistence (see section 3.1.1 on page 14) outside a transaction.
System_Error	Raised when an operating system call returns a bad status (i.e. cannot read from device).
Time_Error	Raised when a transaction exceeds the time limit it declared in the call to begin_transaction.

Table 15: Exceptions provided by the Firm package

4 THE FIRM.ATOMIC_ACCESS PACKAGE

The Firm.Atomic_Access package is a generic package which is instantiated with a non-discriminated type derived from the Firm.Atomic_Object type. The Firm.Atomic_Access package contains operations for creating storage pools for objects with GLOBAL or PERSISTENT persistence as well as operations for getting a physical pointer (e.g. Ada access type) to an object given its Atomic_Ref.

4.1 Operations on the FIRM_Storage_Pool type

The FIRM_Storage_Pool type is shown in Figure 16.

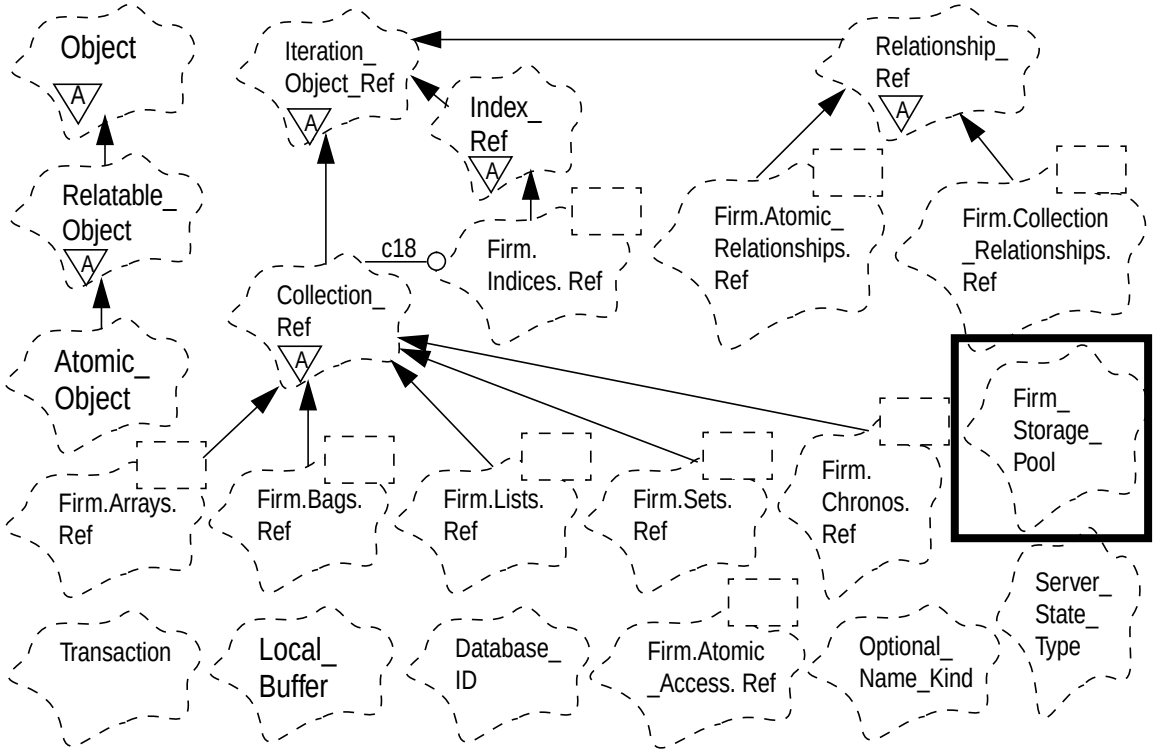

Figure 16: The FIRM_Storage_Pool type

The Create_Global_Pool and Open_Persistent_Pool operators are used to preallocate storage for an atomic object type. These operators initialize an instance of the FIRM_Storage_Pool type. After one of them has been invoked on an instance of the FIRM_Storage_Pool type, the instance can be used to access the desired storage pool. For example:

```
My_Global_Pool: Firm.FIRM_Storage_Pool;
My_DB: Firm.Database_ID := Firm.Create(“My database”,1);
```

```
        type My_Atomic_Type is new Firm.Atomic_Object with ...

        package My_Access is new Firm.Atomic_Access(My_Atomic_Type);

        type My_Global_Ptr is access My_Atomic_Type;
        for My_Global_Ptr'storage_pool use My_Global_Pool;
begin
        My_Access.Create_Global_Pool (
                DB => My_DB,
                Instances => 100_000,
                Storage_Pool => My_Global_Pool );
```

After the call to Firm.Create_Global_Pool in this example, Ada's new operator could be called for an instance of My_Global_Ptr. This would result in the creation and initialization of an instance of My_Atomic_Type in FIRM's global storage area.

Therefore, the FIRM_Storage_Pool type is provided to allow application developers to create Global or Persistent atomic objects. If an atomic object is created without using a FIRM storage pool, the object will have Local persistence.

Table 16 summarizes the operations available on the FIRM_Storage_Pool type in the Firm.Atomic_Access package.

Name	Description
Create_Global_Pool	Procedure to preallocate GLOBAL storage for a type derived from Atomic_Object. This operations is overloaded so that it may be used for discriminated (variable-size) objects or non-discriminated (all the same size) objects. Storage is allocated by providing the desired database ID and number of instances for non-discriminated objects. For discriminated objects, the database ID, size of the smallest object, and total pool size must be provided.
Open_Persistent_Pool	Procedure to preallocate GLOBAL storage for a type derived from Atomic_Object. This operations is overloaded so that it may be used for discriminated (variable-size) objects or non-discriminated (all the same size) objects. If the persistent storage was previously created, it is opened and readied for use. Otherwise, it is created. Storage is opened or created by providing the desired database ID, number of instances, and number of instances in cache for non-discriminated objects. For discriminated objects, the database ID, size of the smallest object, total pool size and total cache size must be provided.

Table 16: Operations on the FIRM_Storage_Pool type

4.2 Operations on the Firm.Atomic_Access.Ref type

The Firm.Atomic_Access.Ref type is shown in Figure 17.

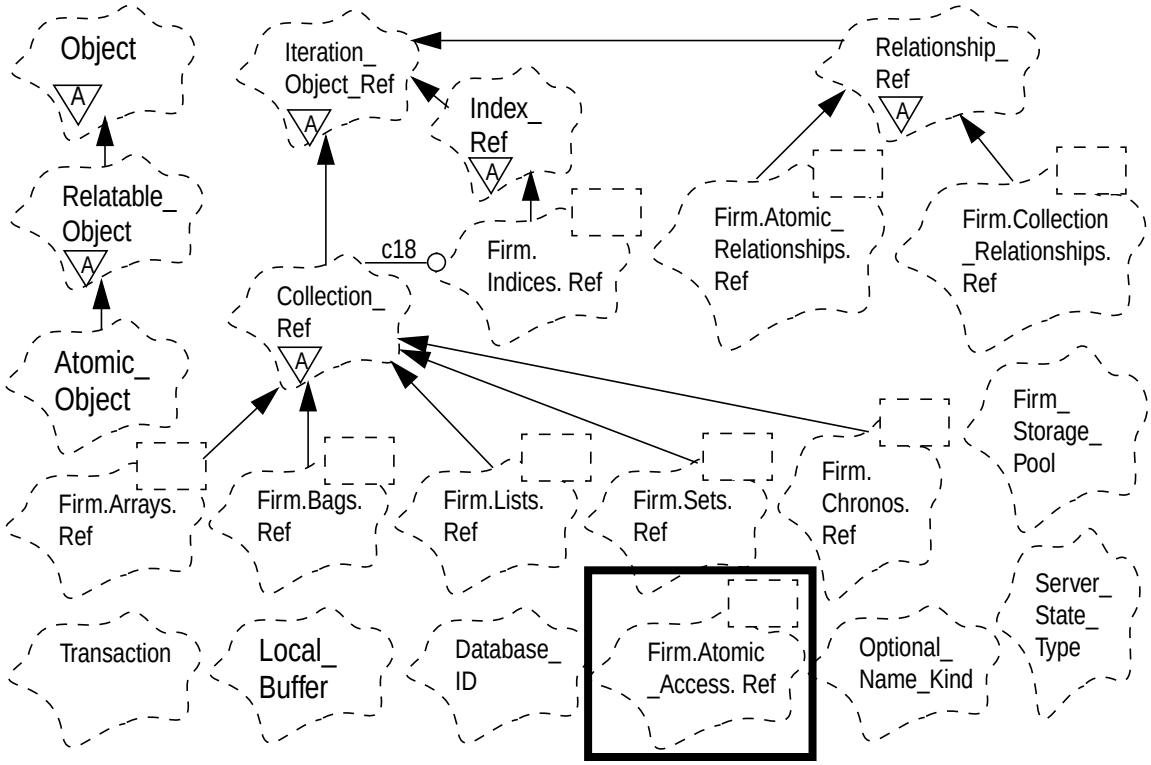

Figure 17: The Atomic_Ref type

Table 17 summarizes the operations available for the Ref type in the Firm.Atomic_Access package.

Name	Description
Update_Object ⌘	Function to return a pointer to a new, updateable copy of an object already stored in the FIRM ODBMS. This operator is overloaded- it accepts as input either an atomic object reference (Atomic_Ref) or database ID and object name in conjunction with an optional transaction. Returns a pointer to an updateable copy of the object.

Table 17: Operations on the Atomic_Ref type

The ⌘ symbol in Table 17 indicates an extra operation provided by the FIRM ODBMS that is not part of [ODMG 96]. The Update_Object operation has no exact counterpart in [ODMG 96]; it is used in FIRM for object modification in lieu of the ODMG C++ binding

mark_modified operator (see [ODMG 96], paragraph 5.3.4, pg. 118). This operation requires a valid transaction when it is used on an object with GLOBAL or PERSISTENT persistence.

4.3 The Local_Buffer type

The Local_Buffer type is shown in Figure 18.

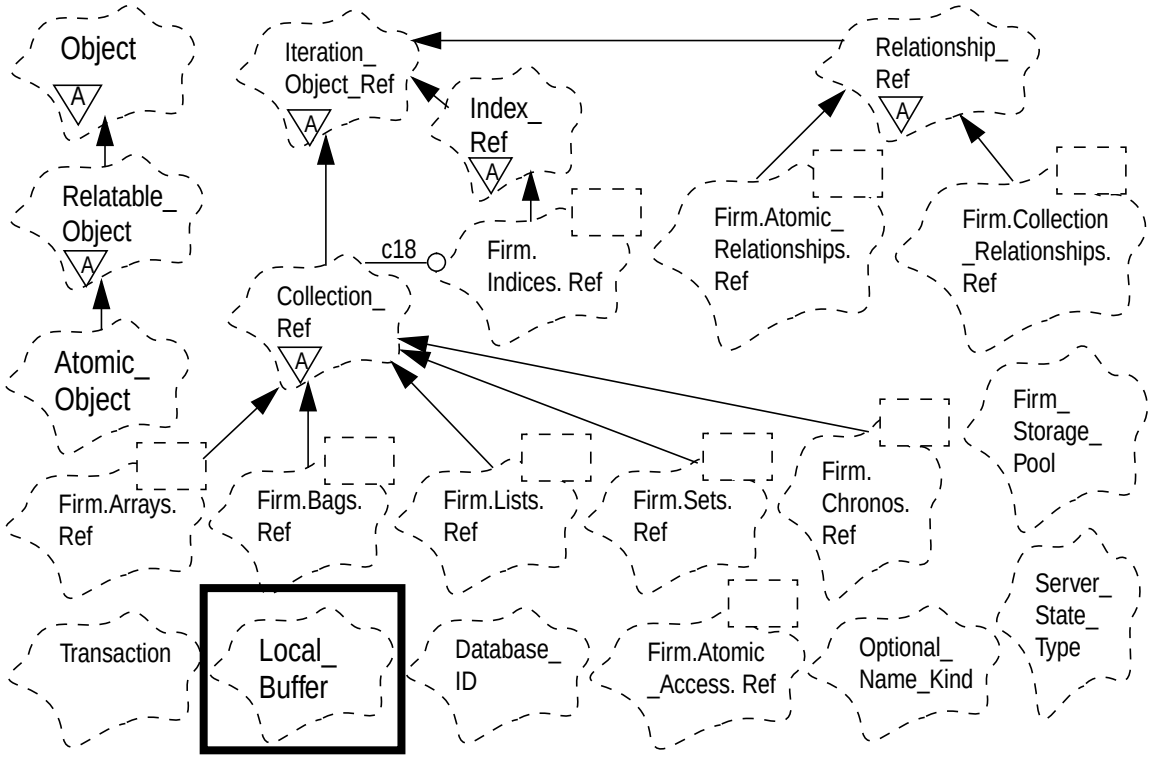

Figure 18: The Local_Buffer type

The Local_Buffer type provides a way for concurrent database tasks to get local copies of database objects for reading. The Local_Buffer type ensures that database applications which request read-only access to an object by calling the Get_Object function (see section 4.3.1 on page 59) are not given physical pointers into the FIRM ODBMS's object cache. Giving an application a physical pointer like this would allow the application to actually update the object by simply using Ada's pointer dereferencing mechanism and changing an attribute (i.e. My_Object_Ptr.all. The_Attribute := New_Value;). It would therefore be possible for an application with a read lock on an object to update it and so corrupt the database. By forcing an application which wants read-only access to an object to provide storage for a copy in a previously-created local buffer, the potential for this type of corruption is eliminated.

There is, however, a performance penalty associated with making these "local copies". From a performance point of view, the best way for an application to read an object would be for Get_Object to ignore any local buffers and simply return a pointer into the object

cache itself without requiring any memory-to-memory copying. To accommodate the need for high performance as well as high database integrity, the Flexible Architecture Builder (FAB) for the FIRM ODBMS will allow the application developer to switch between these two modes of operation. Initially, application code would be developed in the mode where the Get_Object function uses local buffers and returns a pointer to the copy of the object that Get_Object will put into the local buffer. After the application has been thoroughly tested, the developer could switch to the mode where the local buffer is ignored and the Get_Object function returns a pointer into the object cache itself.

4.3.1 Operations on the Local_Buffer type

Table 18 summarizes the operations in the Firm.Atomic_Access package available for the Local_Buffer type.

Operation	Description
Get_Object	Function to get a readable copy of an object and put it into a Local_Buffer. This function is overloaded- it accepts as input either an atomic object reference (Atomic_Ref) or database ID and object name in conjunction with a Local_Buffer_Ptr and an optional transaction. Returns a pointer to the object that has been placed into the Local_Buffer.

Table 18: Operations for the Local_Buffer type

4.4 The Copy operation

FIRM provides a Copy operation for its atomic objects. The Firm.Atomic_Access.Copy operation accepts as input a pointer to a source object, a pointer to a destination object, and an optional transaction. The destination object must be accessible for update, and the source and destination objects must be exactly the same type. The user-defined attributes of the destination object are set to match those in the source object, but the FIRM-specific attributes (e.g. the object's OID, its security label, etc.) are not changed.

5 THE FIRM.ARRAYS PACKAGE

The Firm.Arrays package provides the array collection type for the FIRM ODBMS in accordance with [ODMG 96], paragraph 2.3.5.4 on pages 19-20. An array is a collection of objects whose logical representation is a contiguous one-dimensional array of "cells." Each cell in the array is capable of containing an object. The cells in an array collection may be accessed using an iterator or array index, where the first cell in the array is at index 0. Note here that the term "index" refers to the position of the object in the array (i.e. the index of the first cell in the array is 0), not to an index object (see section 3.6 on page 31).

The Firm.Arrays package is a generic child package which has two generic formal parameters: **Member_Type** and **Member_Access**. An instantiation of Firm.Arrays can use any type derived from Firm.Atomic_Object for the Member_Type parameter. The Member_Access parameter must use an instantiation of Firm.Atomic_Access for the type given in the Member_Type parameter. Array collections are accessed using the Firm.Arrays.Ref type, which inherits from the abstract Firm.Collection_Ref type (see section 3.4 on page 23).

5.1 Array properties

The abstract property functions given in Table 5 on page 24 are supported by the Firm.Arrays.Ref type.

5.1.1 Additional array properties

In addition to the property functions given in Table 5 on page 24, the Firm.Arrays package provides an additional property function for the Firm.Arrays.Ref type which is given in Table 19.

Name	Description
Length	Function which accepts an array reference and returns the length of (e.g. the number of cells in) the array

Table 19: Additional array properties

5.2 Array operations

The abstract operations listed in Table 6 on page 25 are supported by the Firm.Arrays.Ref type. The Create operation has a length parameter which specifies the number of cells in the array.

5.2.1 Additional array operations

In addition to the operations given in Table 6, [ODMG 96] paragraph 2.3.5.4 specifies some additional operations for the array collection type. These are given in Table 20.

Name	Description
Index_Of ⌘	Function which returns the array index pointed to by the current task's iterator.
Remove_Element_At	Procedure which accepts an array reference and an array index. The object at the specified array index is removed from the array.
Replace_Element_At	Procedure which accepts an atomic object, an array reference, and an array index. The object at the specified array index is replaced by the object supplied in the call to this procedure
Retrieve_Element_At	Function which accepts an array reference and an array index. Returns a pointer (Member_Access.Ptr) to the object at the specified array index

Table 20: Operations specific to arrays in [ODMG 96]

The operations in Table 20 correspond to those specified in paragraph 2.3.5.4 in [ODMG 96], except those followed by a ⌘ which are extra operations provided by the FIRM ODBMS. The Resize operator specified in [ODMG 96] for arrays has been omitted from FIRM's array collection for real-time performance reasons.

5.3 Error handling for arrays

The [ODMG 96] specification does not specify what actions a compliant ODBMS should take for error conditions, nor does it define what error conditions are for the collections. The FIRM ODBMS will therefore behave as specified in Table 21.

Operation	Condition	Result
First	Normal iteration	Returns a pointer (Member_Access.Ptr) to the object in the first non-empty cell in the array. Empty cells are skipped over. If all cells in the array are empty, null is returned. The task's iterator is set to point to the cell containing the object or to null if null was returned.
Last	Normal iteration	Returns a pointer (Member_Access.Ptr) to the object in the last non-empty cell in the array. Empty cells are skipped over. If all cells in the array are empty, null is returned. The task's iterator is set to point to the cell containing the object or to null if null was returned.

Table 21: Error handling in the FIRM ODBMS for arrays

Operation	Condition	Result
Next	Normal iteration	Returns a pointer (Member_Access.Ptr) to the object in the next non-empty cell in the array. Empty cells are skipped over. If all the cells after the current one are empty, null is returned. The task's iterator is set to point to the cell containing the object or to null if null was returned.
Prior	Normal iteration	Returns a pointer (Member_Access.Ptr) to the object in the preceding non-empty cell in the array. Empty cells are skipped over. If all cells before the current one are empty, null is returned. The task's iterator is set to point to the cell containing the object or to null if null was returned.
Get_Element	Task's iterator is null	Return null, do not raise an exception
Index_Of	Task's iterator is null	Raise OML_Error exception (see section 3.11 on page 53)
Insert_Element	Normal insertion	Object is inserted into first empty cell in the array, and the task's iterator is set to point to the cell where the object was inserted.
Insert_Element	All of the cells in the array already contain an object	Raise OML_Error exception (see section 3.11 on page 53)
Remove_ Element_At	Cell at specified index is already empty	Set task's iterator to point to specified cell, return successfully.
Remove_ Element_At	Index specified is out-of-range	Raise OML_Error exception (see section 3.11 on page 53)
Replace_ Element_At	Index specified is out-of-range	Raise OML_Error exception (see section 3.11 on page 53)
Retrieve_ Element_At	Cell at specified index is empty	Return null, set task's iterator to point to specified cell.
Retrieve_ Element_At	Index specified is out-of-range	Raise OML_Error exception (see section 3.11 on page 53)

Table 21: Error handling in the FIRM ODBMS for arrays

6 THE FIRM.BAGS PACKAGE

The Firm.Bags package provides the bag collection type for the FIRM ODBMS in accordance with [ODMG 96], paragraph 2.3.5.2 on pages 18-19. A bag is a collection of objects whose logical representation is a linked list of objects. The objects in the bag are not necessarily stored in the order of their insertion (in [ODMG 96] parlance, this means that the bag collection is “unordered”). An object may be inserted multiple times into a bag, so that the bag can logically contain “duplicates” of the object.

The Firm.Bags package is a generic child package which has two generic formal parameters: **Member_Type** and **Member_Access**. An instantiation of Firm.Bags can use any type derived from Firm.Atomic_Object for the Member_Type parameter. The Member_Access parameter must use an instantiation of Firm.Atomic_Access for the type given in the Member_Type parameter. Bag collections are accessed using the Firm.Bags.Ref type, which inherits from the abstract Firm.Collection_Ref type (see section 3.4 on page 23).

6.1 Bag properties

The abstract property functions given in Table 5 on page 24 are supported by the Firm.Bags.Ref type.

6.2 Bag operations

The abstract operations listed in Table 6 on page 25 are supported by the Firm.Bags.Ref type.

6.2.1 Additional bag operations

In addition to the operations given in Table 6, [ODMG 96] paragraph 2.3.5.2 specifies some additional operations for the bag collection type. These are given in Table 22.

Name	Description
Union	Function which accepts two bag references and returns a reference for a bag which contains all of the objects in both of the input bags
Intersection	Function which accepts two bag references and returns a reference for a bag which contains only those objects that the two input bags have in common
Difference	Function which accepts two bag references and returns a reference for a bag which contains only those objects that the two input bags do not have in common

Table 22: Operations specific to bags in [ODMG 96]

6.3 Error handling for bags

The [ODMG 96] specification does not specify what actions a compliant ODBMS should take for error conditions, nor does it define what error conditions are for the collections. The FIRM ODBMS will therefore behave as specified in Table 23.

Operation	Condition	Result
First	Bag is empty	Set task's iterator to null and return null; do not raise an exception.
Last	Bag is empty	Set task's iterator to null and return null; do not raise an exception.
Next	Task's iterator is pointing to the last object in the bag	Set task's iterator to null and return null; do not raise an exception.
Prior	Task's iterator is pointing to the first object in the bag	Set task's iterator to null and return null; do not raise an exception.
Get_Element	Task's iterator is null	Return null, do not raise an exception

Table 23: Error handling in the FIRM ODBMS for bags

7 THE FIRM.CHRONOS PACKAGE

Applications like trackers, correlators and data fusion algorithms require historical data. For example, (a,b,g) trackers use previous state variables for tracks to predict the motion of the targets being tracked. If an ODBMS is to be used as the data store for these kinds of applications the ODBMS must have collection types designed for fast access to temporal data. The FIRM ODBMS' chrono collection type addresses this need.

A chrono is a fixed-size circular queue whose objects are stored in the order of their insertion. This means that the objects are ordered by their time of storage, so that the first object in the chrono can be considered the "oldest" since it was inserted first. Similarly, the last object in the chrono can be considered the newest since it would by definition be the one most recently inserted. The time of storage is kept for each object so the chrono can be accessed by time as well as by iteration. When the chrono is full (e.g. the number of objects in the chrono is equal to its size), the next object inserted will overwrite (and therefore delete) the oldest object in the chrono. A chrono therefore "wraps around" as shown in Figure 19. Chronos do not contain duplicates.

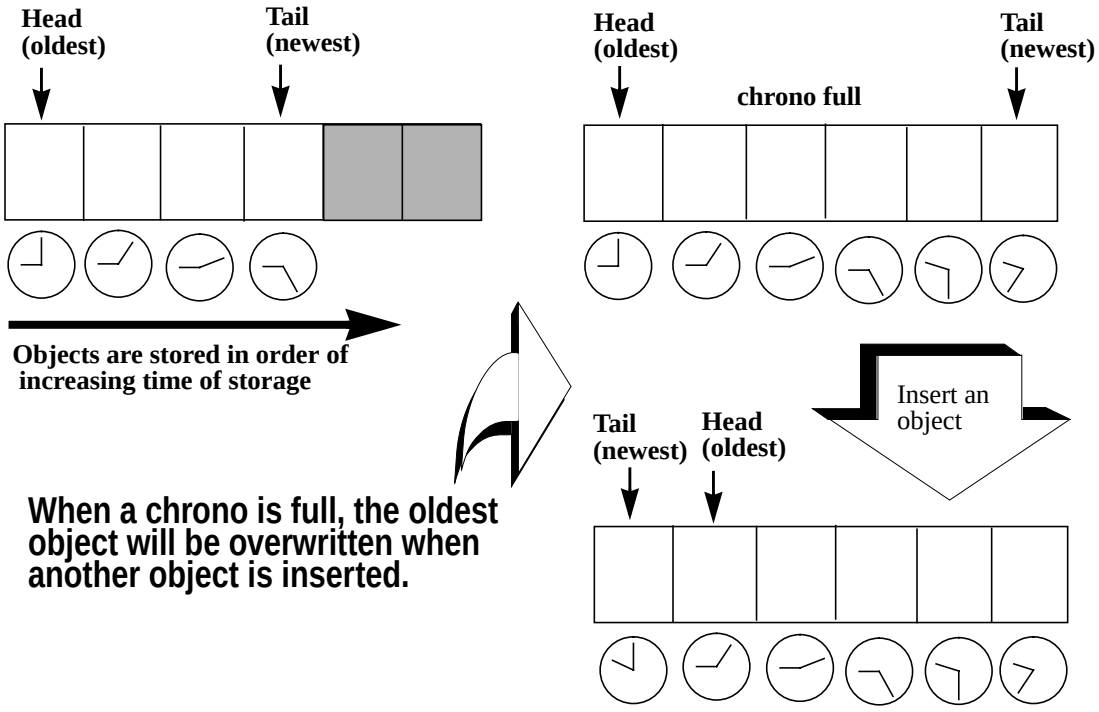

Figure 19: The chrono collection type

The Firm.Chronos package is a generic child package which has two generic formal parameters: **Member_Type** and **Member_Access**. An instantiation of Firm.Chronos can use any type derived from Firm.Atomic_Object for the Member_Type parameter. The

Member_Access parameter must use an instantiation of Firm.Atomic_Access for the type given in the Member_Type parameter. Chrono collections are accessed using the Firm.Chronos.Ref type, which inherits from the abstract Firm.Collection_Ref type (see section 3.4 on page 23).

7.1 Chrono properties

The abstract property functions given in Table 5 on page 24 are supported by the Firm.Chronos.Ref type.

7.1.1 Additional chrono properties

In addition to the property functions given in Table 5, the Firm.Chronos package provides two additional property functions for the Firm.Chronos.Ref type which are given in Table 24.

Name	Description
Length	Function which accepts a chrono reference and returns the length of (e.g. the number of cells in) the chrono
Timespan	Function which accepts a chrono reference and returns the expected minimum time that an object will be stored in the chrono (e.g. how long can an object remain in the chrono before the chrono “wraps around” and overwrites the object?)

Table 24: Additional chrono properties

7.2 Chrono operations

The abstract operations listed in Table 6 on page 25 are supported by the Firm.Chronos. Ref type. The Create operation has a length parameter which specifies the maximum number of objects that can be inserted into the chrono. The Create operation also has a timespan parameter which specifies the minimum time that should elapse between the storage of an object in the chrono and the time the chrono “wraps around” and thus overwrites the object. If the chrono wraps around before this timespan has elapsed, a warning message will be logged via the system’s error processing services.

7.2.1 Additional chrono operations

In addition to the operations given in Table 6, there are additional operations for the chrono collection type. These are given in Table 25 on page 67.

Name	Description
Newest	This function is simply a rename of the **last** iterator operator. (The last object in a chrono is the one which was most recently stored, so it's the newest).
Oldest	This function is simply a rename of the **first** iterator operator. (The first object in a chrono is the one which was stored first, so it's the oldest).
Retrieve_Element_At	Function which accepts a chrono reference and a time of storage (type is Calendar.Time). Returns a pointer (Member_Access.Ptr) to the object stored in the chrono at the specified time. If no object was stored at the specified time, null is returned
Retrieve_Element_At_Or_After	Function which accepts a chrono reference and a time of storage (type is Calendar.Time). Returns a pointer (Member_Access.Ptr) to the object stored in the chrono at the specified time. If no object was stored at the specified time, a pointer to the first object stored after the specified time is returned. If no such object exists, null is returned
Retrieve_Element_At_Or_Before	Function which accepts a chrono reference and a time of storage (type is Calendar.Time). Returns a pointer (Member_Access.Ptr) to the object stored in the chrono at the specified time. If no object was stored at the specified time, a pointer to the last object stored before the specified time is returned. If no such object exists, null is returned
Time_Of_Storage	Function which accepts a chrono reference and returns the time of storage (type Calendar.Time) of the object currently pointed to by the chrono's iterator (each concurrent task has its own iterator)

Table 25: Operations specific to chronos

7.3 Error handling for chronos

The [ODMG 96] specification does not specify what actions a compliant ODBMS should take for error conditions, nor does it define what error conditions are for the collections. The FIRM ODBMS will therefore behave as specified in Table 26.

Operation	Condition	Result
First (Oldest)	Chrono is empty	Set task's iterator to null and return null; do not raise an exception.
Last (Newest)	Chrono is empty	Set task's iterator to null and return null; do not raise an exception.
Next	Task's iterator is pointing to the last object in the chrono	Set task's iterator to null and return null; do not raise an exception.
Prior	Task's iterator is pointing to the first object in the chrono	Set task's iterator to null and return null; do not raise an exception.
Get_Element	Task's iterator is null	Return null, do not raise an exception
Insert_Element	Normal insertion	Object is inserted at the end of the chrono right after the most recently inserted object. Task's iterator is set to point to the inserted object.
Time_Of_ Storage	Task's iterator is null	Raise OML_Error exception (see section 3.11 on page 53)

Table 26: Error handling in the FIRM ODBMS for chronos

8 THE FIRM.LISTS PACKAGE

The Firm.Lists package provides the list collection type for the FIRM ODBMS in accordance with [ODMG 96], paragraph 2.3.5.3 on page 19. A list is a collection of objects whose logical representation is a linked list of objects. The objects in the list are stored in the order of their insertion by default (in [ODMG 96] parlance, this means that the list collection is "ordered"). An object may be inserted multiple times into a list, so that the list can logically contain "duplicates" of the object.

The Firm.Lists package is a generic child package which has two generic formal parameters: **Member_Type** and **Member_Access**. An instantiation of Firm.Lists can use any type derived from Firm.Atomic_Object for the Member_Type parameter. The Member_Access parameter must use an instantiation of Firm.Atomic_Access for the type given in the Member_Type parameter. List collections are accessed using the Firm.Lists.Ref type, which inherits from the abstract Firm.Collection_Ref type (see section 3.4 on page 23).

8.1 List properties

The abstract property functions given in Table 5 on page 24 are supported by the Firm.Lists.Ref type.

8.2 List operations

The abstract operations listed in Table 6 on page 25 are supported by the Firm.Lists.Ref type.

8.2.1 Additional list operations

In addition to the operations given in Table 6, [ODMG 96] paragraph 2.3.5.3 on page 19 specifies some additional operations for the list collection type. These are given in Table 27.

Name	Description
Insert_Element_After	Procedure which accepts an atomic object and a list reference. The object is inserted into the list immediately after the current position of the transaction's iterator. Same as Insert_Element operation.
Insert_Element_Before	Procedure which accepts an atomic object and a list reference. The object is inserted into the list immediately before the current position of the transaction's iterator
Insert_Element_First	Procedure which accepts an atomic object and a list reference. The object is inserted at the head of the list.

Table 27: Operations specific to lists in [ODMG 96]

Name	Description
Insert_Element_Last	Procedure which accepts an atomic object and a list reference. The object is inserted at the end of the list.
Remove_First_Element	Procedure which accepts a list reference. The object at the head of the list is removed from the list. An overloaded version of this procedure returns a pointer (Member_Access.Ptr) to the removed object. The overloaded version can therefore be used like the stack “POP” operation.
Remove_Last_Element	Procedure which accepts a list reference. The object at the end of the list is removed from the list. An overloaded version of this procedure returns a pointer (Member_Access.Ptr) to the removed object. The overloaded version can therefore be used like the stack “POP” operation.
Retrieve_First_Element	Function which accepts a list reference. Returns a pointer (Member_Access. Ptr) to the object at the head of the list. Same as First operation.
Retrieve_Last_Element	Function which accepts a list reference. Returns a pointer (Member_Access. Ptr) to the object at the end of the list. Same as Last operation.
Concat	Function which accepts two list references. Returns a reference to a new list which is the concatenation of the two input lists.
Append	Procedure which accepts two list references. The objects in the second list are appended to the end of the first list.

Table 27: Operations specific to lists in [ODMG 96]

The positional operations specified for lists in [ODMG 96] (e.g. Remove_Element_At, Replace_Element_At, and Retrieve_Element_At) have been omitted from FIRM’s list collection for real-time performance reasons.

8.3 Error handling for lists

The [ODMG 96] specification does not specify what actions a compliant ODBMS should take for error conditions, nor does it define what error conditions are for the collections. The FIRM ODBMS will therefore behave as specified in Table 28.

Operation	Condition	Result
First	List is empty	Set task's iterator to null and return null; do not raise an exception.
Last	List is empty	Set task's iterator to null and return null; do not raise an exception.
Next	Task's iterator is pointing to the last object in the list	Set task's iterator to null and return null; do not raise an exception.
Prior	Task's iterator is pointing to the first object in the list	Set task's iterator to null and return null; do not raise an exception.
Get_Element	Task's iterator is null	Return null, do not raise an exception
Insert_Element	Normal insertion	Object is inserted into the list right after object pointed to by task's iterator. If task's iterator is null, object is inserted at beginning of list, as if Insert_Element_First had been used. Task's iterator is set to point to the inserted object.
Remove_First_ Element_	List is empty	Return successfully, do not raise an exception
Remove_Last_ Element_	List is empty	Return successfully, do not raise an exception
Retrieve_First_ Element_	List is empty	Return null, do not raise an exception
Retrieve_Last_ Element_	List is empty	Return null, do not raise an exception

Table 28: Error handling in the FIRM ODBMS for lists

9 THE FIRM.SETS PACKAGE

The Firm.Sets package provides the set collection type for the FIRM ODBMS in accordance with [ODMG 96], paragraph 2.3.5.1 on page 18. A set is a collection of objects whose logical representation is a linked list of objects. The objects in the set are not necessarily stored in the order of their insertion (in [ODMG 96] parlance, this means that the set collection is “unordered”). An object may not be inserted multiple times into a set. Thus, a set never contains “duplicates” of an object.

The Firm.Sets package is a generic child package which has two generic formal parameters: **Member_Type** and **Member_Access**. An instantiation of Firm.Sets can use any type derived from Firm.Atomic_Object for the Member_Type parameter. The Member_Access parameter must use an instantiation of Firm.Atomic_Access for the type given in the Member_Type parameter. Set collections are accessed using the Firm.Sets.Ref type, which inherits from the abstract Firm.Collection_Ref type (see section 3.4 on page 23).

9.1 Set properties

The abstract property functions given in Table 5 on page 24 are supported by the Firm.Sets.Ref type.

9.2 Set operations

The abstract operations listed in Table 6 on page 25 are supported by the Firm.Sets.Ref type.

9.2.1 Additional set operations

In addition to the operations given in Table 6, [ODMG 96] paragraph 2.3.5.1 specifies some additional operations for the set collection type. These are given in Table 29.

Name	Description
Union	Function which accepts two set references and returns a reference for a set which contains all of the objects in both of the input sets
Intersection	Function which accepts two set references and returns a reference for a set which contains only those objects that the two input sets have in common
Difference	Function which accepts two set references and returns a reference for a set which contains only those objects that the two input sets do not have in common

Table 29: Operations specific to sets in [ODMG 96]

Name	Description
Is_Subset	Function which accepts two set references and returns TRUE if the “left” set is a subset of the “right” set, FALSE otherwise
Is_Proper_Subset	Function which accepts two set references and returns TRUE if the “left” set is a proper subset of the “right” set, FALSE otherwise
Is_Superset	Function which accepts two set references and returns TRUE if the “left” set is a superset of the “right” set, FALSE otherwise
Is_Proper_Superset	Function which accepts two set references and returns TRUE if the “left” set is a proper superset of the “right” set, FALSE otherwise

Table 29: Operations specific to sets in [ODMG 96]

9.3 Error handling for sets

The [ODMG 96] specification does not specify what actions a compliant ODBMS should take for error conditions, nor does it define what error conditions are for the collections. The FIRM ODBMS will therefore behave as specified in Table 30.

Operation	Condition	Result
First	Set is empty	Set task’s iterator to null and return null; do not raise an exception.
Last	Set is empty	Set task’s iterator to null and return null; do not raise an exception.
Next	Task’s iterator is pointing to the last object in the set	Set task’s iterator to null and return null; do not raise an exception.
Prior	Task’s iterator is pointing to the first object in the set	Set task’s iterator to null and return null; do not raise an exception.
Get_Element	Task’s iterator is null	Return null, do not raise an exception
Insert_Element	Object is already in the set	Raise OML_Error exception (see section 3.11 on page 53)

Table 30: Error handling in the FIRM ODBMS for sets

10 THE FIRM.INDICES PACKAGE

The Firm.Indices package provides FIRM's index capability (see section 3.6 on page 31).

The Firm.Indices package is a generic child package which has four generic formal parameters:

1. **Keyed_Atomic_Type** - Any type derived from Firm.Atomic_Object
2. **Member_Access** - An instantiation of Firm.Atomic_Access for the type given in the Keyed_Atomic_Type parameter
3. **Equal_To** - A function which accepts a "left" and a "right" instance of type Keyed_Atomic_Type'Class and returns Boolean TRUE if the two instances have identical key attributes.
4. **Less_Than** - A function which accepts a "left" and a "right" instance of type Keyed_Atomic_Type'Class and returns Boolean TRUE if the key attributes of the "left" instance are less than those of the "right" instance.

The Equal_To and Less_Than sorting operations must be supplied to instantiate Firm.Indices because *the FIRM ODBMS itself has no knowledge of the attributes a user defines for a type derived from Firm.Atomic_Object.* Indices are accessed using the Firm.Indices.Ref type, which inherits from the abstract Firm.Index_Ref type (see section 3.6 on page 31).

10.1 Index operations

The abstract operations listed in Table 8 on page 32 are supported by the Firm.Indices.Ref type.

10.2 Error handling for indices

Error handling for indices is described in Table 9 on page 37.

11 THE FIRM.ATOMIC_RELATIONSHIPS PACKAGE

The Firm.Atomic_Relationships package provides FIRM's relationship capability for relationships between atomic object types (see section 3.7 on page 38).

The Firm.Atomic_Relationships package is a generic child package which has four generic formal parameters:

1. **From_Type** - Any type derived from Firm.Atomic_Object. Instances of this type will be on the "From" side of the relationship
2. **From_Access** - An instantiation of the Firm.Atomic_Access package for the type given in the From_Type parameter
3. **To_Type** - Any type derived from Firm.Atomic_Object. Instances of this type will be on the "To" side of the relationship
4. **To_Access** - An instantiation of the Firm.Atomic_Access package for the type given in the To_Type parameter

Relationships between atomic object types are accessed using the Firm.Atomic_Relationships.Ref type, which inherits from the abstract Firm.Relationship_Ref type (see section 3.7 on page 38).

11.1 Atomic relationship operations

The abstract operations listed in Table 10 on page 44 are supported by the Firm.Atomic_Relationships.Ref type.

11.1.1 Additional atomic relationship operations

In addition to the operations given in Table 10, FIRM provides some additional operations for relationships between atomic object types. These are given in Table 31.

Operation	Description
Add_Traversal_Path	Procedure which accepts two atomic objects and a relationship reference. If the two objects are in the right classes for the relationship and it is OK to add a traversal path between the objects, a traversal path is established between them
Remove_Traversal_Path	Procedure which accepts two atomic objects and a relationship reference. If the two objects are in the right classes for the relationship, the traversal path between them is deleted
Remove_All_Paths	Procedure which accepts an atomic object and a relationship reference. All traversal paths from the atomic object that are part of the specified relationship are deleted

Table 31: Operations on the Firm.Atomic_Relationships.Ref type

Operation	Description
Set_Iterator	Procedure which accepts a relationship reference and an atomic object that is on one of the relationship's traversal paths. Sets the current task's iterator for the relationship onto the specified (e.g. base) object so that the application can iterate through the objects on the "other side" of the relationship from the specified object

Table 31: Operations on the Firm.Atomic_Relationships.Ref type

11.2 Error handling for atomic relationships

In addition to the error handling for relationships that is defined in Table 11 on page 48, Table 32 contains error handling for the operations given in Table 31.

Operation	Condition	Result
Set_Iterator	The object from which to perform subsequent traversal path iteration is not of the right type for the relationship	Raise OML_Error exception (see section 3.11 on page 53)
Add_Traversal_Path	A traversal path for the specified relationship already exists between the two specified objects	Raise OML_Error exception (see section 3.11 on page 53)
Add_Traversal_Path	One or more of the objects on the desired path is not the right type for the relationship	Raise OML_Error exception (see section 3.11 on page 53)
Remove_Traversal_Path	There is no traversal path between the specified objects that is in the specified relationship, although the two objects are of the right types for the specified relationship	Return successfully, do not raise an exception
Remove_Traversal_Path	One or more of the objects on the desired path is not the right type for the relationship	Raise OML_Error exception (see section 3.11 on page 53)
Remove_All_Paths	There are no traversal paths in the specified relationship (e.g. it is "empty")	Return successfully, do not raise an exception

Table 32: Error handling in the FIRM ODBMS for atomic relationships

12 THE FIRM.COLLECTION_RELATIONSHIPS PACKAGE

The Firm.Collection_Relationships package provides FIRM's relationship capability for relationships between atomic object types and collection types (see section 3.7 on page 38).

The Firm.Collection_Relationships package is a generic child package which has three generic formal parameters:

1. **From_Type** - Any type derived from Firm.Atomic_Object. Instances of this type will be on the "From" side of the relationship
2. **From_Access** - An instantiation of the Firm.Atomic_Access package for the type given in the From_Type parameter
3. **To_Type** - Any type derived from Firm.Collection_Ref. Instances of this type will be on the "To" side of the relationship

Relationships between atomic object types and collection types are accessed using the Firm.Collection_Relationships.Ref type, which inherits from the abstract Firm.Relationship_Ref type (see section 3.7 on page 38).

12.1 Collection relationship operations

The abstract operations listed in Table 10 on page 44 are supported by the Firm.Collection_Relationships.Ref type.

12.1.1 Additional collection relationship operations

In addition to the operations given in Table 10, FIRM provides some additional operations for relationships between atomic object types. These are given in Table 33.

Operation	Description
Add_Traversal_Path	Procedure which accepts an atomic object, a reference for a collection (type To_Type), and a relationship reference. If the two objects are in the right classes for the relationship and it is OK to add a traversal path between the objects, a traversal path is established between them
Remove_Traversal_Path	Procedure which accepts an atomic object, a reference for a collection (type To_Type), and a relationship reference. If the two objects are in the right classes for the relationship, the traversal path between them is deleted
Remove_All_Paths	Procedure which accepts a either an atomic object or a collection reference (type To_Type) and a relationship reference. All traversal paths from the collection object that are part of the specified relationship are deleted

Table 33: Operations on the Firm.Collection_Relationships.Ref type

Operation	Description
Set_Iterator	Procedure which accepts a relationship reference and an atomic object or a collection reference (type To_Type) that is on one of the relationship's traversal paths. Sets the current task's iterator for the relationship onto the specified (e.g. base) object so that the application can iterate through the objects on the "other side" of the relationship from the specified object

Table 33: Operations on the Firm.Collection_Relationships.Ref type

12.2 Error handling for collection relationships

In addition to the error handling for relationships that is defined in Table 11 on page 48, Table 34 contains error handling for the operations given in Table 33.

Operation	Condition	Result
Set_Iterator	The object from which to perform subsequent traversal path iteration is not of the right type for the relationship	Raise OML_Error exception (see section 3.11 on page 53)
Add_Traversal_Path	A traversal path for the specified relationship already exists between the two specified objects	Raise OML_Error exception (see section 3.11 on page 53)
Add_Traversal_Path	One or more of the objects on the desired path is not the right type for the relationship	Raise OML_Error exception (see section 3.11 on page 53)
Remove_Traversal_Path	There is no traversal path between the specified objects that is in the specified relationship, although the two objects are of the right types for the specified relationship	Return successfully, do not raise an exception
Remove_Traversal_Path	One or more of the objects on the desired path is not the right type for the relationship	Raise OML_Error exception (see section 3.11 on page 53)
Remove_All_Paths	There are no traversal paths in the specified relationship (e.g. it is "empty")	Return successfully, do not raise an exception

Table 34: Error handling in the FIRM ODBMS for atomic relationships

13 BIBLIOGRAPHY

13.1 Government Documents

[TCSEC 85] Trusted Computer System Evaluation Criteria, DoD 5200.28-STD (More commonly known as “The Orange Book”), National Computer Security Center, Alexandria VA, December 1985.

13.2 Non-Government Documents

[AdaLRM95] *Ada 95 Reference Manual*, International Standard ANSI/ISO/IEC-8652:1995, Infometrics, Cambridge Mass, 1995

[AdaRtnl95] *Ada 95 Rationale*, Infometrics, Cambridge Mass, 1995

[Bernstein et. al.] Bernstein, Philip A., Hadzilacos, Vassos and Goodman, Nathan, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading Mass., 1987.

[Gamma et. al.] Gamma, Erich, et. al. *Design Patterns*, Addison-Wesley, Reading Mass, 1995

[HPL-95-11] Stepanov, Alexander A. and Lee, Meng, *The Standard Template Library*, Hewlett-Packard, February 1995.

[ODMG 96] Cattell, R. G. G, *The Object Database Standard: ODMG-93 Release 1.2*, Morgan Kaufmann, San Francisco CA, 1996

[Lee 95] Lee, Byung S, “Normalization in OODB Design,” *SIGMOD Record*, vol. 24, no. 3, pp. 23-27.

[Loomis 95] Loomis, Mary E. S., *Object Databases*, Addison-Wesley, Reading Mass, 1995

[Wirfs-Brock 90] Wirfs-Brock, Rebecca et. al., *Designing Object-Oriented Software*, Prentice-Hall, 1990.

FIRM: An Ada Binding to ODMG-93 1.2

APPENDIX A: A COMPARISON OF ODMG-93 1.2 AND FIRM

Table 35 contains a feature-by-feature comparison of the ODMG-93 version 1.2 object model (see [ODMG 96], chapter 2) and its realization in FIRM. The “Compare / Contrast” column explains any differences between the two or provides clarification as needed.

Feature	ODMG-93 1.2 Par.	This paper	Compare / Contrast
Types with visible interfaces and hidden internal details	2.2		This is achieved using Ada-95’s private types (see [AdaLRM95], section 7.3 and [AdaRtnl95] chapter 7).
Types can be extended via inheritance	2.2.1		This is achieved using Ada-95’s type extension mechanism (see [AdaLRM95], sections 3.9.1 and 7.3, as well as [AdaRtnl95] section 3.6.1 and the example in page II-4).
Types can be abstract	2.2.1		Provided by Ada’s abstract type mechanism (see [AdaLRM95], section 3.9.3).
Extents	2.2.2		ODMG-style extents (e.g. built-in set collections) incur too much overhead for a real-time ODBMS, so they are not included in FIRM.
Keys	2.2.3	section 3.6	FIRM provides an index type for use on its collections.
same_as operator	2.3	section 2.2	Ada “=” operator for FIRM’s reference types
copy operator	2.3	section 4.4	Provided for Atomic and Collection objects, but not for Indices and Relationships.
delete operator	2.3	section 3.1.3.1 section 3.6.1 section 3.7.4	Provided for all FIRM objects, same as ODMG-93 1.2.
Object identifiers	2.3.1	section 3.1.1	FIRM provides unique identifiers to all instances of Object’class, same as ODMG-93 1.2.
Object names	2.3.2	section 3.1.3.1 section 3.6.1 section 3.7.4	All FIRM objects can be named. The Bind operator is used to name Atomic and Collection objects, while Index and Relationship objects are named in their Create operation.
Object lifetimes	2.3.3	section 3.1.1	FIRM supports all ODMG-93 1.2 lifetimes, plus adds “Global” persistence for main memory databases.

Table 35: Comparison of ODMG 1.2 and FIRM object models

Feature	ODMG-93 1.2 Par.	This paper	Compare / Contrast
User-defined object types	2.3.4	section 2.1 section 3.1.3	Same as ODMG-93 1.2 (Atomic_object type)
collection objects	2.3.5	section 3.4.1	Same as ODMG-93 1.2, except FIRM allows only atomic objects within collections. To nest collections, see section 3.4.3.
cardinality operator	2.3.5	section 3.4.1	Same as ODMG-93 1.2
is_empty operator	2.3.5	section 3.4.1	Same as ODMG-93 1.2
insert_element operator	2.3.5	section 3.4.2	Same as ODMG-93 1.2
remove_element operator	2.3.5	section 3.4.2	Same as ODMG-93 1.2
contains_element operator	2.3.5	section 3.4.2	Same as ODMG-93 1.2
create_iterator operator	2.3.5	section 3.4.2	FIRM's iterators are built into the iteratable objects; they are cursors. Therefore, this operation is not provided.
Empty exception	2.3.5	section 5.3 section 8.3 section 3.7.6	FIRM's iterators do not raise an exception when the collection is empty since this is not an exceptional condition. Instead, iteration on an empty collection returns a null reference. This incurs less run-time overhead than exception processing.
NoMoreElements exception	2.3.5	section 6.3 section 8.3 section 9.3 section 3.7.6	FIRM's iterators do not raise an exception when the end of a collection is reached since this is not an exceptional condition. Iteration from the end of a collection returns a null reference. This incurs less run-time overhead than exception processing.
not_done operator	2.3.5	section 5.3 section 8.3 section 9.3 section 3.7.6	FIRM's iterators do not have a not_done operator. You know you have completed iteration of a collection, index, or relationship when a null reference is returned.
next operator	2.3.5	section 3.4.2	Same as ODMG-93 1.2

Table 35: Comparison of ODMG 1.2 and FIRM object models

Feature	ODMG-93 1.2 Par.	This paper	Compare / Contrast
advance operator	2.3.5		Not provided in FIRM because FIRM's iterators do not raise exceptions when the end of a collection is reached. See "not_done" operator. FIRM's iterators provide prior, first and last operators for greater flexibility.
get_element operator	2.3.5	section 3.4.2	Same as ODMG-93 1.2
reset operator	2.3.5	section 3.4.2	Same as ODMG-93 1.2
set object	2.3.5.1	section 9	Same as ODMG-93 1.2
bag object	2.3.5.2	section 6	Same as ODMG-93 1.2
list object	2.3.5.3	section 8	Same as ODMG-93 1.2, except that FIRM omits the Remove_Element_At, Replace_Element_At, and Retrieve_Element_At operators
array object	2.3.5.4	section 5	Same as ODMG-93 1.2, except that FIRM omits the Resize operator
atomic literals	2.4.1		Provided by Ada-95; see [AdaLRM95]
collection literals	2.4.2		Not provided by FIRM because FIRM's collections only contain objects derived from the FIRM Atomic_Object type. (This is necessary because all objects in FIRM must have labels for MLS).
time of day functions	2.4.3.1, 2.4.3.2, 2.4.3.3, 2.4.3.4		Time of day functionality for FIRM is provided by Ada-95. See [AdaLRM95], sections 9.6 and D.8.
user-defined structures	2.4.3.5		Provided by Ada-95 because Ada-95 allows objects (tagged record types) to be nested. See [AdaLRM95], section 3.8 (section 3.7 has examples of records and arrays nested within each other).
attributes	2.5.1		Provided by Ada-95 via the fields in a tagged record type (see [AdaLRM95], section 3.9)

Table 35: Comparison of ODMG 1.2 and FIRM object models

Feature	ODMG-93 1.2 Par.	This paper	Compare / Contrast
relationships	2.5.2	section 3.7	Provided by FIRM's relationship types. FIRM relationships provide all ODMG-93 1.2 functionality except ordered relationships; FIRM relationships may only be unordered. FIRM's relationships are first-class objects which provide the following capabilities in addition to those in ODMG-93 1.2: • Unary relationships • Relationships between atomic objects and collections
operations	2.6		Provided by Ada-95's dispatching operations (see [AdaLRM95], section 3.9.2 and [AdaRtnl95] section II.2). Note that Ada-95 also provides non-dispatching operations which are not bound to a single type.
exception model	2.6.1		FIRM uses the Ada-95 exception model (see [AdaLRM95] section 11) in lieu of the ODMG-93 1.2 exception model. The models are identical except that Ada-95 does not allow the exception type to be subtyped (e.g. no inheritance from it).
metadata	2.7		FIRM uses Ada-95's RTTI for its metadata needs (see [AdaLRM95], section 3.9). Note that Ada-95's RTTI could be augmented by use of the Ada Semantic Interface Specification (ASIS).
type hierarchy	2.7.1	section 3.1	This hierarchy is supported by FIRM in conjunction with Ada-95 except for collection literals, which FIRM does not support.
type compatibility rules	2.7.2		Supported by Ada-95
null value	2.7.3		FIRM provides null reference values for all of its object types, and Ada-95 provides null values for its access types.
table type	2.7.4		This could be provided in FIRM via a Bag of Row, where Row is: type Row is new Firm.Atomic_Object with record a1:t1; a2:t2; ... end record;
transaction model	2.8		Same as ODMG-93 1.2

Table 35: Comparison of ODMG 1.2 and FIRM object models

Feature	ODMG-93 1.2 Par.	This paper	Compare / Contrast
concurrency control	2.8.1		FIRM will use multi-version mixed-method rather than two-phase locking.
transaction operations	2.8.2	section 3.8	Same as ODMG-93 1.2, except that FIRM allows an optional time limit to be specified in the call to begin_transaction to support real-time processing
open operator	2.9	section 3.2.1	Does not create a new database if specified database does not exist; the create operation is used to create new databases. Otherwise, same as ODMG-93 1.2.
close operator	2.9	section 3.2.1	Same as ODMG-93 1.2
bind operator	2.9	section 3.1.3.1 section 3.4.2	Same as ODMG-93 1.2 for Atomic objects and Collections. Bind is an inherent part of the Create operation for Indices and Relationships in FIRM (e.g. they are always named).
lookup operator	2.9	section 3.1.3.1 section 3.4.2 section 3.6.1 section 3.7.4	Same as ODMG-93 1.2

Table 35: Comparison of ODMG 1.2 and FIRM object models

APPENDIX B: FIRM PACKAGE SPECIFICATION

```
-- NOTE: The following differences exist between the FIRM object model and
--       that presented in chapter 2 of ODMG-93 release 1.2:
--
--  1) OQL - OQL is not supported due to the deterministic, real-time
--     requirements for FIRM.
--
--  2) Iterators - FIRM's iterators are built-in to the "iteratable" objects
--     in FIRM, which include collections, relationships, and indices. This
--     is different from the ODMG approach, where iterators are created and
--     maintained by the user.  The FIRM approach bounds the number of
--     iterators to one per concurrent task which makes iterator maintenance
--     deterministic.
--
--  3) Collections - In FIRM,collections may contain only atomic objects, not
--     literals or other collections as in ODMG-93 release 1.2. FIRM will
ultimately`
--     use spacial data indices to realize multi-dimensional data structures
rather
--     than nested collections.
--
--  4) Relationships - In FIRM, relationships are first-class objects.  This
--     allows for dynamic creation/deletion of relationships and also allows
--     property information about a relationship to be efficiently stored.
--     That is, this approach minimizes the amount of information about its
--     relationships that an object must contain.
--
--     Also, FIRM supports ONLY unordered relationships.
--
--  5) Literals are not stored in the FIRM ODBMS since literals by definition
--     are not database objects (e.g. they cannot be modified and they have no
--     unique identifer; see ODMG-93 release 1.2, sections 2.1 and 2.3.1).
--    Literals in FIRM are provided by the Ada programming language. Therefore,
--     FIRM does not provide the collection literals in section 2.4.2 of
--     ODMG-93 release 1.2.
--
--  6) The structured literals in section 2.4.3 are not implemented directly
--     in FIRM; this functionality is provided by the Ada.Calendar and
--     Ada.Real_Time.
--
--  7) The Ada exception model is substituted for the one in section 2.6.1.
--     Functionally, they are equivalent except that Ada does not allow
--     subtyping of the exception type.
--
with Ada.Finalization;
with Ada.Real_Time;
with Ada.Tags;
```

```
with Ada.Unchecked_Conversion;
with Ada.Unchecked_Deallocation;
with Firm_Config_Constants;
with Firm_Msgs.Msg_Archives;
with Firm_Msgs.Cntl_Stat;
with System.Address_To_Access_Conversions;
with System.Storage_Elements;
with System.Storage_Pools;
package Firm is

    -- --------------- --
    -- Site operations --
    -- --------------- --

    subtype Server_State_Type is Firm_Msgs.Cntl_Stat.Set_Odbms_Type;
    -- FIRM is designed to support dual-redundant servers. The master server is
the
    -- one used to run applications while the standby server is a backup in case
the
    -- master fails. The standby is kept up-to-date automatically by the master
    -- server. The FIRM server may be in one of three states:
    --
    -- MASTER:  The server is running on the primary host and is servicing
applications
    -- STANDBY: The server is running on the backup host and is being kept up-
to-date
    --          by the MASTER
    -- SOLO:    The server is running on the primary host and is servicing
    --          applications, but there is no STANDBY server to fail over to

    -- Function which returns current state of the FIRM server.
    function Get_State return Server_State_Type;

    type Callback_Ptr is access procedure;
    -- Type for a pointer to a callback procedure which is invoked when the FIRM
    -- server changes state from standby to master. This transaition occurs when
the
    -- master site has failed and the standby site is taking over as the master.

    -- Procedure to prepare a FIRM server for use. This procedure must be
    -- called after Shutdown in order to get a FIRM server ready to use again.
    -- The callback procedure (if specified) will be invoked when the FIRM server
    -- changes state from standby to master, i.e. upon a failover.
    procedure Startup (Callback : Callback_Ptr := null);

    -- Procedure to block (i.e. suspend) the invoking task until the FIRM server
is in
```

```
    -- the desired state.
    procedure Block_Until (Continue_State : in Server_State_Type);

    -- Procedure to shut down a FIRM server. This procedure gets the databases
    -- on the server in a quiescent state so that the server may be powered
    -- down, etc. and later restarted.
    procedure Shutdown;

    -- ------------------- --
    -- Database operations --
    -- ------------------- --

    type Database_Id is range 0 .. Firm_Config_Constants.Max_Databases;
    for Database_Id'Size use Firm_Config_Constants.Database_Id_Size;
    -- Type for database identifiers

    Null_Db : constant Database_Id := Database_Id'First;
    -- Null database constant (the null database is used to contain objects
    -- with LOCAL persistence)

    subtype Valid_Database_Id is
       Database_Id range 1 .. Firm_Config_Constants.Max_Databases;
    -- Non-null database IDs. A non-null database contains objects of GLOBAL or
    -- PERSISTENT persistence

    type Db_Access_Type is (Sor, Replicant);
    -- Type to define the kinds of access to a database an application has. A
    -- replicant database may not be updated, as it resides at another FIRM
server and
   -- the local instance is an automatically-maintained copy. A System Of Record
(SOR)
    -- database resides on the local server and is the only update-able copy.
Another
    -- FIRM server may open a replicant copy of a SOR database.

   -- Procedure to create a new database. The user must specify a unique name and
   -- identifier. This procedure will always create a new System Of Record (SOR)
    -- database on the FIRM server which calls it.
    procedure Create (Db_Name : in String; Db_Id : in Database_Id);

    -- These operations are specified in ODMG-93 release 1.2, page 33,
    -- section 2.9, pg. 33. The user must supply a unique name and identifier
to open
    -- a database.
    procedure Open
                (Db_Name : in String;
                 Db_Id : in Database_Id;
```

```
                  Db_Access : in Db_Access_Type;
                  -- Type of access desired
                  Xfer_Sor_Callback : in Callback_Ptr := null
               -- Callback procedure which is invoked when a database initially
               -- opened as a replicant becomes the System Of Record (SOR) copy
                  -- via a "transfer SOR" operation
                  );

    procedure Close (Db : in Database_Id);

    -- Function to indicate what type of access is available to the specified
database
    function Db_Access (Db : in Database_Id) return Db_Access_Type;

    -- ------------------------------ --
    -- Types for database transactions --
    -- ------------------------------ --

    type Transaction is private;
    Null_Transaction : constant Transaction;

    type Transaction_Type is (Read, Update);

    type Consistency_Policy is (Strict, Relaxed);
    -- The strict consistency policy is used to assure that when an update
   -- transaction is committed, all replicant copies of the effected database(s)
    -- receive their journal messages. The relaxed consistency policy does not
   -- guarantee this, but a transaction using relaxed consistency will take less
   -- time to perform commit processing. Thus, relaxed consistency can be used
for
    -- time-critical transactions on databases whose replicants will not be
harmed
    -- by a lost update.

    -- -------------------------- --
    -- Operations on transactions --
    -- -------------------------- --

    -- These operations are as specified in the ODMG-93 specification
    -- release 1.2, section 2.8.2, page 32.  The Begin_Transaction
    -- operation has an extra parameter to allow real-time
    -- deadlines to be specified.  The default transaction execution time
    -- allotment, 0.0, is used to indicate that the transaction is "soft
    -- real-time", meaning that there is no fixed deadline.  A value other
    -- than 0.0 denotes a "hard real-time" transaction which must be
    -- completed within the allotted time.  If the transaction is not
    -- finished within the allotted time, the Time_Error exception
```

```
-- will be raised.  The operations begin and abort have _transaction
-- appended to their names because begin and abort are reserved words
-- in Ada.
function Begin_Transaction
            (T_Type : in Transaction_Type;
             Name : in String;
             Time_Allotted : in Ada.Real_Time.Time_Span :=
                Ada.Real_Time.Time_Span_Zero;
            Consistency : Consistency_Policy := Strict) return Transaction;
procedure Commit (T : in out Transaction);
procedure Checkpoint (T : in Transaction);
procedure Abort_Transaction (T : in out Transaction);

-- -------------------------- --
-- FIRM object class hierarchy --
-- -------------------------- --

type Persistence_Type is (Local, Global, Persistent);
-- Persistence settings for objects.  LOCAL is the equivalent of ODMG-93
-- release 1.2 "transient" objects (see section 2.3.3, pg. 16). PERSISTENT
-- is same as ODMG-93 "persistent" (see ref. above). GLOBAL objects are
-- unique to FIRM and are used to provide main-memory databases. GLOBAL
-- objects exist from creation time to the time the database is closed or
-- until deleted.  They are therefore "coterminous with database session".
-- The SAME value here is provided for the copy operation so that the new
-- copy will have the same level of persistence as the original.

subtype Storage_Pool_Persistence is
   Persistence_Type range Global .. Persistent;
-- Persistence levels for FIRM storage pools. FIRM storage pools do not
-- contain objects of LOCAL persistence; these are contained in storage
-- managed by Ada (stack or heap).

type Object is abstract tagged limited private;
-- Root type for all objects with OIDs, including atomic objects,
-- collections, indices, and relationships

type Relatable_Object is abstract new Object with private;
-- Root type for all objects which can participate in relationships

type Atomic_Object is new Relatable_Object with private;
-- Root class for user-defined objects

package Atomic_Object_Conversions is
   new System.Address_To_Access_Conversions (Atomic_Object'Class);

subtype Atomic_Ptr is Atomic_Object_Conversions.Object_Pointer;
```

```
    -- ------------------------------------ --
    -- Storage allocation for atomic objects --
    -- ------------------------------------ --

    type Firm_Storage_Pool is new
       System.Storage_Pools.Root_Storage_Pool with private;
    -- Type for a FIRM storage pool. This type is used to specify FIRM's
    -- Global or Persistent storage management for access types which access
    -- a type derived from Atomic_Object.  The storage pools for FIRM's
    -- Global and Persistent storage areas for a type are returned from a
    -- a call to Firm.Atomic_Access.Create_Global_Pool or
    -- Firm.Atomic_Access.Create_Persistent_Pool.

    -- ------------------------------- --
    -- Local buffer type and operations --
    -- ------------------------------- --

   -- The Local_Buffer type is used for storing a copy of an object of GLOBAL or
    -- PERSISTENT persistence which is returned by a retrieval operation. The
copy in the
   -- buffer is for read purposes only; any updates to it will not be reflected
in the
    -- database.
    type Local_Buffer is new System.Storage_Elements.Storage_Array;

    type Local_Buffer_Ptr is access all Local_Buffer;

    -- ------------------------------------- --
    -- Class-wide operations on atomic objects --
    -- ------------------------------------- --

    -- Lookup operation specified in ODMG-93 v. 1.2, section 2.9 pg. 33
    function Lookup (Db : in Database_Id;
                     Name : in String;
                     Buffer : in Local_Buffer_Ptr;
                    T : in Transaction := Null_Transaction) return Atomic_Ptr;

    -- ODMG-93 v. 1.2 Bind operation for Atomic (user-defined) objects.
    -- See par 2.9, pg 33
    procedure Bind (Obj : in Atomic_Object'Class;
                    Name : in String;
                    T : in Transaction := Null_Transaction);

    -- This operation corresponds to the delete operation specified in the
    -- ODMG-93 specification release 1.2, section 2.3, pg. 15
    procedure Delete (Obj : in out Atomic_Object'Class;
```

```
                    T : in Transaction := Null_Transaction);

    -- ---------------------------------------------------- --
    -- FIRM abstract type for objects which can be iterated --
    -- ---------------------------------------------------- --
    type Iteration_Object_Ref is abstract tagged private;

    -- The following methods are for the integrated iterators that
    -- FIRM provides with its iteration objects. All of the iteration objects
    -- in FIRM provide methods like those shown below. The Next, Reset, and
    -- Get_Element operations are part of the ODMG-93 release 1.2
    -- for iterators (section 2.3.5, pg. 18). The other operations are
    -- added for FIRM. The pointer type returned by these methods is not
    -- Atomic_Ptr, but is instead the "Member_Access.Ptr" type for the
collection,
    -- index or relationship generic package.
    --
    -- function First (O : in Iteration_Object_Ref;
    --              Buffer : in Local_Buffer_Ptr;
    --              T : in Transaction := Null_Transaction) return
Member_Access.Ptr;
    --
    -- function Last (O : in Iteration_Object_Ref;
    --             Buffer : in Local_Buffer_Ptr;
   --          T : in Transaction := Null_Transaction) return Member_Access.Ptr;
    --
    -- function Next (O : in Iteration_Object_Ref;
    --             Buffer : in Local_Buffer_Ptr;
   --          T : in Transaction := Null_Transaction) return Member_Access.Ptr;
    --
    -- function Prior (O : in Iteration_Object_Ref;
    --              Buffer : in Local_Buffer_Ptr;
    --              T : in Transaction := Null_Transaction) return
Member_Access.Ptr;
    --
    -- procedure Reset (O : in Iteration_Object_Ref;
    --               T : in Transaction := Null_Transaction);
    --
    -- This function returns ref to object currently pointed to by the
    -- object's iterator.
    -- function Get_Element
    --          (O : in Iteration_Object_Ref;
    --           Buffer : in Local_Buffer_Ptr;
   --         T : in Transaction := Null_Transaction) return Member_Access.Ptr;

    -- --------------------------------- --
    -- FIRM abstract type for Collections --
```

```
-- ---------------------------------- --
type Collection_Ref is abstract new Iteration_Object_Ref with private;

-- -------------------------- --
-- Properties for collections --
-- -------------------------- --

-- The following properties are specified for all collections in
-- ODMG-93 release 1.2, section 2.3.5, pg. 17.
function Cardinality
            (C : in Collection_Ref; T : in Transaction := Null_Transaction)
            return Natural is abstract;

function Is_Empty
            (C : in Collection_Ref; T : in Transaction := Null_Transaction)
            return Boolean is abstract;


-- The following properties are not in ODMG-93 release 1.2; they have been
-- added for FIRM.
function Persistence
            (C : in Collection_Ref; T : in Transaction := Null_Transaction)
            return Persistence_Type is abstract;

function Is_Indexed
            (C : in Collection_Ref; T : in Transaction := Null_Transaction)
            return Boolean is abstract;

-- This function returns the tag of the type that a collection was
-- instantiated with. The collection may contain an instance of this type
-- or any type derived from it.
function Get_Tag (C : in Collection_Ref) return Ada.Tags.Tag is abstract;

-- ------------------------ --
-- Operations on collections --
-- ------------------------ --

-- The following operations are specified as available on all types
-- of collections in the ODMG-93 release 1.2 spec,section 2.3.5,
-- pp. 17-18.  Note that the Copy and Delete operations are inherited
-- from the ODMG-93 Object type (section 2.3, pg. 15).

-- Bind operation for collection objects (see ODMG-93 release 1.2,
-- section 2.9, pg. 33).
procedure Bind (C : in Collection_Ref;
                Name : in String;
                T : in Transaction := Null_Transaction) is abstract;
```

```
procedure Copy (From_C : in Collection_Ref;
                To_C : in out Collection_Ref;
                T : in Transaction := Null_Transaction) is abstract;

procedure Delete (C : in out Collection_Ref;
                  T : in Transaction := Null_Transaction) is abstract;

-- All of FIRM's collections provide insertion and removal methods like
-- those shown below.
--
-- procedure Insert_Element (C : in out Collection_Ref;
--                           Obj : in Member_Type'Class;
--                           T : in Transaction := Null_Transaction);
--
-- procedure Remove_Element (C : in out Collection_Ref;
--                           Obj : in Member_Type'Class;
--                           T : in Transaction := Null_Transaction);
--
-- function Contains_Element
--          (C : in Collection_Ref;
--           Obj : in Member_Type'Class;
--           T : in Transaction := Null_Transaction) return Boolean;

-- This operation is not in ODMG-93 release 1.2.  It has been added
-- for FIRM.
procedure Vacate (C : in out Collection_Ref;
                  T : in Transaction := Null_Transaction) is abstract;

-- Lookup operation for collections (ODMG-93 release 1.2, section 2.9,
-- page 33).
function Lookup (Db : in Database_Id;
                 Name : in String;
                 T : in Transaction := Null_Transaction)
                return Collection_Ref is abstract;

-- ------------------------------------------------------------- --
-- Operations on strings (names of atomic objects and collections) --
-- ------------------------------------------------------------- --

-- Type for various kinds of object names
type Name_Kind is
   (Atomic_Name,       -- name for atomic       object
    Collection_Name,   -- "    "   collection   "
    Index_Name,        -- "    "   index        "
    Relationship_Name, -- "    "   relationship "
    Internal_Name      -- "    "   internal object not visible to user
```

```
    );

-- Type for those objects which may or may not have names. Note that some
-- object types must have one unique name which cannot be unbound (i.e.
-- indices and relationships).
subtype Optional_Name_Kind is Name_Kind
                                range Atomic_Name .. Collection_Name;

-- Unbind operation for names of atomic and collection objects.
-- This operation is not part of ODMG-93 v. 1.2, it was added for FIRM.
procedure Unbind (Db : in Database_Id;
                  Name : in String;
                  Kind : in Optional_Name_Kind;
                  T : in Transaction := Null_Transaction);

-- ------------------------------------- --
-- Abstract type for indices on collections --
-- ------------------------------------- --

type Index_Ref is abstract new Iteration_Object_Ref with private;

function Create (C : in Collection_Ref'Class;
                 Name : in String;
                Duplicate_Keys : in Boolean) return Index_Ref is abstract;

procedure Delete (I : in out Index_Ref;
                  T : in Transaction := Null_Transaction) is abstract;

-- All FIRM indices provide a key match method "Find_Match" like the one
-- below.
--
-- function Find_Match
--          (I : in Index_Ref;
--           Obj : in Member_Type'Class;
--           Buffer : in Local_Buffer_Ptr;
--         T : in Transaction := Null_Transaction) return Member_Access.Ptr;

function Lookup (Db : in Database_Id;
                 Name : in String;
                T : in Transaction := Null_Transaction) return Index_Ref is
   abstract;

-- ------------------------------------------------------------------ --
-- Type decalarations for FIRM relationships.  In FIRM, relationships --
-- are first-class objects as per 2.10.4 of ODMG-93 release 1.2.      --
-- ------------------------------------------------------------------ --
type Relationship_Type is (One_To_One, One_To_Many, Many_To_Many);
```

```
type Relationship_Ref is abstract new Iteration_Object_Ref with private;

function Create (Db : in Database_Id;
                 Persistence : in Persistence_Type;
                 Rel_Type : in Relationship_Type;
                 Name : in String) return Relationship_Ref is abstract;

procedure Delete (R : in out Relationship_Ref;
                  T : in Transaction := Null_Transaction) is abstract;

-- Lookup operation for relationships, see ODMG-93 release 1.2, section
-- 2.9, pg. 33.
function Lookup (Db : in Database_Id;
                 Name : in String;
                 T : in Transaction := Null_Transaction)
                return Relationship_Ref is abstract;

-- Iteration operators for iterating traversal paths to collections; these
-- must be provided by each collection type
function First (R : in Relationship_Ref'Class;
                T : in Transaction := Null_Transaction)
               return Collection_Ref is abstract;

function Last (R : in Relationship_Ref'Class;
               T : in Transaction := Null_Transaction)
              return Collection_Ref is abstract;

function Next (R : in Relationship_Ref'Class;
               T : in Transaction := Null_Transaction)
              return Collection_Ref is abstract;

function Prior (R : in Relationship_Ref'Class;
                T : in Transaction := Null_Transaction)
               return Collection_Ref is abstract;

function Get_Element (R : in Relationship_Ref'Class;
                      T : in Transaction := Null_Transaction)
                     return Collection_Ref is abstract;

-- ------------------------- --
-- Exceptions raised by FIRM --
-- ------------------------- --

Deadlock_Error : exception;
-- This exception is raised whenever a transaction becomes deadlocked.
-- If this exception is caught, the current transaction should be aborted
```

```
-- and restarted.

Cache_Error : exception;
-- This exception is raised when an attempt is made to access an object
-- of PERSISTENT persistence and there is not enough cache storage for
-- the object.

Allocation_Error : exception;
-- This exception is raised whenever an object of GLOBAL or PERSISTENT
-- persistence cannot be created because
--   (1) there is not enough storage available in its storage pool or
--   (2) there is not enough CONTIGUOUS storage available in its storage
--       pool (this applies to variable-size objects).

Internal_Error : exception;
-- This exception is raised whenever there is a failure within the FIRM
-- ODBMS that was not caused by any application programming errors.

Oml_Error : exception;
-- This exception is raised whenever there is a problem with the way
-- the user has tried to use a FIRM object.

Configuration_Error : exception;
-- This exception is raised when the user has exceeded a maximum
-- specified in the Firm_Config_Constants package. For example, if an
-- application attempts to call Specify_Max_Instances or
-- Specify_Max_Storage and the maximum number of GLOBAL storage pools
-- have already been created, this exception will be raised. To correct
-- the problem, the configuration of FIRM must be changed
-- (e.g. Firm_Config_Constants.MAX_GLOBAL_POOLS must be increased, etc)
-- and the application must be recompiled & relinked.

System_Error : exception;
-- This exception is raised whenever a call to an operating system
-- function fails (i.e. error reading a device, etc)

Time_Error : exception;
-- This exception is raised whenever a transaction fails to complete
-- within its time allotment.

package Msg_Log is

   type Msg_Category_Type is new Firm_Msgs.Msg_Archives.Msg_Category_Type;

    -- This is the external interface to FIRM's  portable message logging
    -- package.  It is anticapated that the embedded systems which FIRM
    -- will be used in will have their own message logging and error
```

```
        -- reporting (MELER) subsystems.  This package provides a standard
        -- interface to whatever the surrounding system's MELER subsystem may
        -- consist of.  When FIRM is ported into a new environment, the
        -- implementation of this package will need to customized to feed into
        -- the surronding system's MELER but the rest of FIRM's error and
        -- message reporting mechanism can stay the same.
        --
        -- Two functions are provided:
        --
        -- 1.) Logging messages.  Our expectation is that the encapsulating
        --     system will have have sequential message log of some kind for
        --     post-mission analysis.  There may also be an on-board display
        --     for scrolling through past messages (more likely in a submarine
        --     or ground-based radar than in a one-man fighter plane).  The
        --     Log_Msg procedure found in this package spec is intended to
        --     be a general purpose interface to a sequential message logging
        --     system.
        --
        -- 2.) Displaying an alert message.  Examples include, "INSUFFICENT
        --     STORAGE", or "MASTER PROCESSOR FAILED".  The Display_Last_Msg
        --     procedure may be set up send all or part of the last message
        --     logged by the task to the system's alert display.
        --

        -- The Log_Msg procedure logs a message for later post-mission
        -- analysis.
        procedure Log_Msg
                    (Who : in Transaction     -- Id of Transaction logging
                        := Null_Transaction;  -- the message
                     Sender_Name : in String :=
                        "";   -- Name of Transaction or
                              -- program logging the message.
                     -- NOTE: If a valid transaction is identified through the
"Who"
                    -- parameter, the Sender_Name parameter is ignored and the
name
                    -- provided on the Begin_Transaction function is used.  The
                     -- Sender_Name parameter may be used when the program
sending
                     -- the message is not a database transaction.
                     What : in String;         -- brief message
                     Where : in String := "";   -- Unit_Name
                     Why_1 : in String := "";   -- up to 3 lines of text
                     Why_2 : in String := "";
                     Why_3 : in String := "";
                     Msg_Cat : Msg_Category_Type := Informational);
```

```
        -- The Display_Last_Msg procedure causes all or part of the last
        -- message logged by transaction to be displayed as an alert.  This
        -- capability should only be invoked from within a transaction.  If
        -- it is used with a Null_Transaction identifier, it can only be
        -- guaranteed to display the last message logged with a null
        -- transaction identifier.  That message may have been logged
        -- by some other program than the intended one.  For this reason,
        -- Display_Last_Msg should only be used within a valid transaction.

        procedure Display_Last_Msg (Who : in Transaction);

    end Msg_Log;

private

end Firm;
```

APPENDIX C: FIRM.ATOMIC_ACCESS PACKAGE SPECIFICATION

```
with System.Address_To_Access_Conversions;
generic
    -- The box symbol means that the type may be discriminated
    type New_Atomic_Object (<>) is new Atomic_Object with private;
package Firm.Atomic_Access is

    -- This package contains all of the functions which return accesses
    -- (pointers) to atomic objects. It also contains the procedures needed
    -- to set up storage pools so that Ada access types can be used to
    -- create objects in FIRM-managed storage.

    -- ------------------------------------------------------------- --
    -- Pointer type used to access objects of all persistence levels and --
    -- to create objects on the Ada heap                              --
    -- ------------------------------------------------------------- --
    package New_Atomic_Object_Conversions is
       new System.Address_To_Access_Conversions (New_Atomic_Object'Class);

    subtype Ptr is New_Atomic_Object_Conversions.Object_Pointer;

    -- Procedure for specifying how many instances of the type derived from
    -- Atomic_Object will be stored in a GLOBAL storage pool. This procedure
should
    -- be used for types which are NOT discriminated.
    procedure Create_Global_Pool
                (Db : in Database_Id;
                 -- Database the pool belongs to
                 Instances : in Natural;
                -- Number of instances that will have the desired persistence.
This
                 -- count must include all versions of each instance if multi-
                 -- version concurrency control is in use.
                 Storage_Pool : in out Firm_Storage_Pool
                -- Storage pool that can be used to create objects with GLOBAL
                 -- persistence.
                 );

    -- Procedure for specifying how much GLOBAL storage will be needed in the
   -- database for all of the instances of the discriminated type derived from
    -- Atomic_Object.
    procedure Create_Global_Pool
                (Db : in Database_Id;
                 -- Database the pool belongs to
                 Smallest_Size : in System.Storage_Elements.Storage_Count;
                 -- Size of the smallest instance of the type
                 Storage : in System.Storage_Elements.Storage_Count;
```

```
                    -- Amount of storage needed for all instances. This total
                   -- must include all versions of each instance if multi-version
                    -- concurrency control is in use.
                    Storage_Pool : in out Firm_Storage_Pool
                   -- Storage pool that can be used to create objects with desired
                    -- persistence. By default, instances of Atomic_Object'class
                    -- are created with Local persistence.
                    );

    -- Procedure for specifying how many instances of the type derived from
   -- Atomic_Object will be stored in a PERSISTENT storage pool. This procedure
    -- should be used for types which are NOT discriminated.
   -- Note that if the pool has not yet been created, a call to this procedure
    -- will create it. Otherwise, the existing pool will be opened.
    procedure Open_Persistent_Pool
                  (Db : in Database_Id;
                   -- Database the pool belongs to
                   Instances : in Natural;
                   -- Number of instances that will have the desired persistence.
This
                   -- count must include all versions of each instance if multi-
                   -- version concurrency control is in use.
                   Cached_Instances : in Natural;
                   -- The total number of instances that the pool's cache must be
                   -- able to contain
                   Storage_Pool : in out Firm_Storage_Pool
                   -- Storage pool that can be used to create objects with GLOBAL
                   -- persistence.
                   );

   -- Procedure for specifying how much PERSISTENT storage will be needed in the
    -- database for all of the instances of the discriminated type derived from
    -- Atomic_Object.
   -- Note that if the pool has not yet been created, a call to this procedure
    -- will create it. Otherwise, the existing pool will be opened.
    procedure Open_Persistent_Pool
                  (Db : in Database_Id;
                   -- Database the pool belongs to
                   Smallest_Size : in System.Storage_Elements.Storage_Count;
                   -- Size of the smallest instance of the type
                   Storage : in System.Storage_Elements.Storage_Count;
                   -- Amount of storage needed for all instances. This total
                   -- must include all versions of each instance if multi-version
                   -- concurrency control is in use.
                   Cache : in System.Storage_Elements.Storage_Count;
                   -- Total size of the pool's cache
                   Storage_Pool : in out Firm_Storage_Pool
```

```
                -- Storage pool that can be used to create objects with desired
                -- persistence. By default, instances of Atomic_Object'class
                -- are created with Local persistence.
                );

-- -------------- --
-- Reference type --
-- -------------- --

type Ref is private;
-- Type for a "smart pointer" to an object. Must be dereferenced using
-- either the Get_Object or Update_Object functions. References to GLOBAL
-- or PERSISTENT objects may only be dereferenced inside a transaction, but
-- they remain valid after a transaction has committed or aborted. Pointers
-- obtained by Get_Object or Update_Object are valid only within their
-- enclosing transaction; after the transaction commits or aborts, the
-- pointers are invalid.

Null_Ref : constant Ref;
-- Reference equivalent of a null pointer

-- Function to get a reference for an object
function Get_Ref (Obj : in New_Atomic_Object'Class) return Ref;

-- --------------------------- --
-- Object retrieval operations --
-- --------------------------- --

-- These operations allow the user to fetch a local copy of an object from
-- the database.  These operations are not specified in ODMG-93 release 1.2.
function Get_Object (Obj : in Ref;
                     Buffer : in Local_Buffer_Ptr;
                     T : in Transaction := Null_Transaction) return Ptr;

function Get_Object (Db : in Database_Id;
                     Name : in String;
                     Buffer : in Local_Buffer_Ptr;
                     T : in Transaction := Null_Transaction) return Ptr;

-- ------------------------ --
-- Object update operations --
-- ------------------------ --

-- These operations return a pointer to a new version of the object which
-- the user can update.  The changes will be saved at commit time.
-- These operations are not specified in ODMG-93 release 1.2.
function Update_Object
```

```
                (Obj : in Ref; T : in Transaction := Null_Transaction)
                return Ptr;

    function Update_Object (Db : in Database_Id;
                            Name : in String;
                           T : in Transaction := Null_Transaction) return Ptr;

    -- This procedure is called when an an UPDATE transaction gets a pointer
    -- to an object from Get_Object or a retrieval operation (First, Last, etc)
    -- and subsequently wishes to update the retrieved object. This procedure
    -- will get the correct lock on the object and return a pointer to it which
    -- is suitable for updating its attributes.
    procedure Update_Object (Obj_Ptr : in out Ptr;
                             T : in Transaction := Null_Transaction);

    -- -------------- --
    -- Copy operation --
    -- -------------- --

    -- Copy operator specified in ODMG-93 v. 1.2, par. 2.3, pg. 15
    procedure Copy (From : in Ptr;
                    To : in out Ptr;
                    T : in Transaction := Null_Transaction);

private
end Firm.Atomic_Access;
```

APPENDIX D: FIRM.ARRAYS PACKAGE SPECIFICATION

```
with Ada.Tags;
with Firm.Atomic_Access;
generic
    type Member_Type (<>) is new Atomic_Object with private;
    with package Member_Access is new Firm.Atomic_Access (Member_Type);
package Firm.Arrays is

    -- NOTE: The "Resize" operation was removed from FIRM's implementation
    --       of the ODMG-93 Release 1.2 Array collection for real-time
    --       performance reasons.

    -- ------------------------------------------------------------------ --
    -- Type decalarations for the ODMG-93 release 1.2 "array" collection  --
    -- (see section 2.3.5.4, pg. 19).                                     --
    -- ------------------------------------------------------------------ --
    type Ref is new Collection_Ref with private;

    -- -------------------- --
    -- Properties of arrays --
    -- -------------------- --

    -- The following properties are specified for all collections in
    -- ODMG-93 release 1.2, section 2.3.5, pg. 17.
    function Cardinality (A : in Ref; T : in Transaction := Null_Transaction)
                         return Natural;

    function Is_Empty (A : in Ref; T : in Transaction := Null_Transaction)
                      return Boolean;


    -- The following properties are not in ODMG-93 release 1.2; they have been
    -- added for FIRM.
    function Persistence (A : in Ref; T : in Transaction := Null_Transaction)
                         return Persistence_Type;

    function Is_Indexed (A : in Ref; T : in Transaction := Null_Transaction)
                        return Boolean;

    -- This property was added for arrays; it is not in ODMG-93 release 1.2.
    function Length (A : in Ref; T : in Transaction := Null_Transaction)
                    return Positive;

    -- This function returns the tag of "Type_In_Collection"
    function Get_Tag (A : in Ref) return Ada.Tags.Tag;

    -- -------------------- --
```

```
-- Operations on arrays --
-- -------------------- --

-- The following operations are specified as available on all types
-- of collections in the ODMG-93 release 1.2 spec,section 2.3.5,
-- pp. 17-18.  They are refined here for the array collection type.

function Create (Db : in Database_Id;
                 Persistence : in Persistence_Type;
                 Length : in Positive) return Ref;

-- Bind operation for collection objects (see ODMG-93 release 1.2,
-- section 2.9, pg. 33).
procedure Bind (A : in Ref;
                Name : in String;
                T : in Transaction := Null_Transaction);

procedure Copy (From_A : in Ref;
                To_A : in out Ref;
                T : in Transaction := Null_Transaction);

procedure Delete (A : in out Ref; T : in Transaction := Null_Transaction);

-- ODMG-93 release 1.2 does not specify the behavior of this operation
-- for array collections.  FIRM assumes that what is desired is sequential
-- insertion, so the new element is inserted into the first empty cell
-- after the current position.
procedure Insert_Element (A : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

procedure Remove_Element (A : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

function Contains_Element
            (A : in Ref;
             Obj : in Member_Type'Class;
             T : in Transaction := Null_Transaction) return Boolean;

-- This operation is not in ODMG-93 release 1.2.  It has been added
-- for FIRM.
procedure Vacate (A : in out Ref; T : in Transaction := Null_Transaction);

-- Lookup operation for collections (ODMG-93 release 1.2, section 2.9,
-- page 33).
function Lookup (Db : in Database_Id;
```

```
                Name : in String;
                T : in Transaction := Null_Transaction) return Ref;

-- The following methods are for the integrated iterators that
-- FIRM provides with its array collections.
function First (A : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

function Last (A : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Next (A : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Prior (A : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

procedure Reset (A : in Ref; T : in Transaction := Null_Transaction);

function Get_Element (A : in Ref;
                      Buffer : in Local_Buffer_Ptr;
                      T : in Transaction := Null_Transaction)
                     return Member_Access.Ptr;

-- The following operations are specified as available for all array
-- collections in the ODMG-93 release 1.2 spec (see section 2.3.5.4,
-- pg. 19).
procedure Replace_Element_At (A : in out Ref;
                              Obj : in Member_Type'Class;
                              Index : in Natural;
                              T : in Transaction := Null_Transaction);

procedure Remove_Element_At (A : in out Ref;
                             Index : in Natural;
                             T : in Transaction := Null_Transaction);

function Retrieve_Element_At (A : in Ref;
                              Index : in Natural;
                              Buffer : in Local_Buffer_Ptr;
```

```
                                    T : in Transaction := Null_Transaction)
                                  return Member_Access.Ptr;

    -- The following operation is not part of the ODMG-93 Release 1.2 spec; it
    -- has been added for FIRM.

    -- Operation to return array index being pointed to by the iterator
    function Index_Of (A : in Ref; T : in Transaction := Null_Transaction)
                      return Natural;

   -- ------------------------------------------------------------------------
--
   -- Iterator operations for relationships between atomic objects and arrays --
   -- ------------------------------------------------------------------------
--

    function First (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Last (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Next (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Prior (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Get_Element (R : in Relationship_Ref'Class;
                          T : in Transaction := Null_Transaction) return Ref;

    Null_Ref : constant Ref;

private

end Firm.Arrays;
```

APPENDIX E: FIRM.BAGS PACKAGE SPECIFICATION

```
with Ada.Tags;
with Firm.Atomic_Access;
generic
    type Member_Type (<>) is new Atomic_Object with private;
    with package Member_Access is new Firm.Atomic_Access (Member_Type);
package Firm.Bags is

    -- ------------------------------------------------------------- --
    -- Type decalarations for the ODMG-93 release 1.2 "bag" collection  --
    -- (section 2.3.5.2, pp. 18-19).                                 --
    -- ------------------------------------------------------------- --
    type Ref is new Collection_Ref with private;

    -- ------------------ --
    -- Properties of bags --
    -- ------------------ --

    -- The following properties are specified for all collections in
    -- ODMG-93 release 1.2, section 2.3.5, pg. 17.
    function Cardinality (B : in Ref; T : in Transaction := Null_Transaction)
                         return Natural;

    function Is_Empty (B : in Ref; T : in Transaction := Null_Transaction)
                       return Boolean;


    -- The following properties are not in ODMG-93 release 1.2; they have been
    -- added for FIRM.
    function Persistence (B : in Ref; T : in Transaction := Null_Transaction)
                         return Persistence_Type;

    function Is_Indexed (B : in Ref; T : in Transaction := Null_Transaction)
                        return Boolean;

    -- This function returns the tag of "Type_In_Collection"
    function Get_Tag (B : in Ref) return Ada.Tags.Tag;

    -- ------------------ --
    -- Operations on bags --
    -- ------------------ --

    -- The following operations are specified as available on all types
    -- of collections in the ODMG-93 release 1.2 spec (see section 2.3.5,
    -- pp. 17-18).  They are refined here for the bag collection type.

    function Create (Db : in Database_Id; Persistence : in Persistence_Type)
```

```
                return Ref;

-- Bind operation for collection objects (see ODMG-93 release 1.2,
-- section 2.9, pg. 33).
procedure Bind (B : in Ref;
                Name : in String;
                T : in Transaction := Null_Transaction);

procedure Copy (From_B : in Ref;
                To_B : in out Ref;
                T : in Transaction := Null_Transaction);

procedure Delete (B : in out Ref; T : in Transaction := Null_Transaction);

procedure Insert_Element (B : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

procedure Remove_Element (B : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

function Contains_Element
            (B : in Ref;
             Obj : in Member_Type'Class;
             T : in Transaction := Null_Transaction) return Boolean;

-- This operation is not in ODMG-93 release 1.2.  It has been added
-- for FIRM.
procedure Vacate (B : in out Ref; T : in Transaction := Null_Transaction);

-- Lookup operation for collections (ODMG-93 release 1.2, section 2.9,
-- page 33).
function Lookup (Db : in Database_Id;
                 Name : in String;
                 T : in Transaction := Null_Transaction) return Ref;


-- The following methods are for the integrated iterators that
-- FIRM provides with its bag collections.

function First (B : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

function Last (B : in Ref;
```

```
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

function Next (B : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Prior (B : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

procedure Reset (B : in Ref; T : in Transaction := Null_Transaction);

function Get_Element (B : in Ref;
                      Buffer : in Local_Buffer_Ptr;
                      T : in Transaction := Null_Transaction)
                     return Member_Access.Ptr;

-- The following operations are specified as available  for all bag
-- collections in the ODMG-93 release 1.2 spec (see section 2.3.5.2,
-- pg. 19).

function Union (B1 : in Ref;
                B2 : in Ref;
                T : in Transaction := Null_Transaction) return Ref;

function Intersection (B1 : in Ref;
                       B2 : in Ref;
                       T : in Transaction := Null_Transaction) return Ref;

function Difference (B1 : in Ref;
                     B2 : in Ref;
                     T : in Transaction := Null_Transaction) return Ref;

-- ------------------------------------------------------------------- --
-- Iterator operations for relationships between atomic objects and bags --
-- ------------------------------------------------------------------- --

function First (R : in Relationship_Ref'Class;
                T : in Transaction := Null_Transaction) return Ref;

function Last (R : in Relationship_Ref'Class;
               T : in Transaction := Null_Transaction) return Ref;
```

```
    function Next (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Prior (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Get_Element (R : in Relationship_Ref'Class;
                          T : in Transaction := Null_Transaction) return Ref;

    Null_Ref : constant Ref;

private

end Firm.Bags;
```

APPENDIX F: FIRM.CHRONOS PACKAGE SPECIFICATION

```
with Ada.Real_Time;
with Ada.Tags;
with Firm.Atomic_Access;
generic
    type Member_Type (<>) is new Atomic_Object with private;
    with package Member_Access is new Firm.Atomic_Access (Member_Type);
package Firm.Chronos is

    -- ------------------------------------------------------------- --
    -- Type decalarations for the FIRM "chrono" collection, which is a --
    -- circular queue (FIFO) with objects ordered by their time of     --
    -- storage.                                                        --
    -- ------------------------------------------------------------- --
    type Ref is new Collection_Ref with private;

    -- ---------------------- --
    -- Properties for chronos --
    -- ---------------------- --

    -- The following properties are specified for all collections in
    -- ODMG-93 release 1.2, section 2.3.5, pg. 17.
    function Cardinality (Ch : in Ref; T : in Transaction := Null_Transaction)
                         return Natural;

    function Is_Empty (Ch : in Ref; T : in Transaction := Null_Transaction)
                      return Boolean;


    -- The following properties are not in ODMG-93 release 1.2; they have been
    -- added for FIRM.
    function Persistence (Ch : in Ref; T : in Transaction := Null_Transaction)
                         return Persistence_Type;

    function Is_Indexed (Ch : in Ref; T : in Transaction := Null_Transaction)
                        return Boolean;

    -- Length of the circular queue (e.g. maximum number of objects that can
    -- be inserted)
    function Length (Ch : in Ref; T : in Transaction := Null_Transaction)
                    return Positive;

    -- Expected interval between time of storage of the oldest and newest
    -- objects in the chrono. This value is specified in the Create operation.
    function Timespan (Ch : in Ref; T : in Transaction := Null_Transaction)
                      return Ada.Real_Time.Time_Span;
```

```
-- This function returns the tag of "Type_In_Collection"
function Get_Tag (Ch : in Ref) return Ada.Tags.Tag;

-- --------------------- --
-- Operations on chronos --
-- --------------------- --

-- The following operations are specified as available on all types
-- of collections in the ODMG-93 release 1.2 spec, section 2.3.5,
-- pp. 17-18.  They are refined here for the chrono collection type.

function Create (Db : in Database_Id;
                 Persistence : in Persistence_Type;
                 Length : in Positive;
                 Timespan : in Ada.Real_Time.Time_Span;
                 Delete_Removed_Members : in Boolean) return Ref;

-- Bind operation for collection objects (see ODMG-93 release 1.2,
-- section 2.9, pg. 33).
procedure Bind (Ch : in Ref;
                Name : in String;
                T : in Transaction := Null_Transaction);

procedure Copy (From_Ch : in Ref;
                To_Ch : in out Ref;
                T : in Transaction := Null_Transaction);

procedure Delete (Ch : in out Ref; T : in Transaction := Null_Transaction);

procedure Insert_Element (Ch : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

procedure Remove_Element (Ch : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

function Contains_Element
            (Ch : in Ref;
             Obj : in Member_Type'Class;
             T : in Transaction := Null_Transaction) return Boolean;

-- This operation is not in ODMG-93 release 1.2.  It has been added
-- for FIRM.
procedure Vacate (Ch : in out Ref; T : in Transaction := Null_Transaction);

-- Lookup operation for collections (ODMG-93 release 1.2, section 2.9,
```

```
-- page 33).
function Lookup (Db : in Database_Id;
                 Name : in String;
                 T : in Transaction := Null_Transaction) return Ref;

-- The following methods are for the integrated iterators that
-- FIRM provides with its chrono collections.
function First (Ch : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

function Last (Ch : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Next (Ch : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Prior (Ch : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

procedure Reset (Ch : in Ref; T : in Transaction := Null_Transaction);

function Get_Element (Ch : in Ref;
                      Buffer : in Local_Buffer_Ptr;
                      T : in Transaction := Null_Transaction)
                     return Member_Access.Ptr;

-- The following renames for the first and last iterator functions
-- are provided for convenience since they represent the "age" of the
-- objects in the chrono (e.g. how long it has been since insertion)
function Oldest (Ch : in Ref;
                 Buffer : in Local_Buffer_Ptr;
                 T : in Transaction := Null_Transaction)
                return Member_Access.Ptr renames First;

function Newest (Ch : in Ref;
                 Buffer : in Local_Buffer_Ptr;
                 T : in Transaction := Null_Transaction)
                return Member_Access.Ptr renames Last;
```

```
    -- The following operations are provided for chrono collections in
    -- addition to the standard ODMG-93 release 1.2 operations for all
    -- collections.
    function Retrieve_Element_At (Ch : in Ref;
                                  Time : in Ada.Real_Time.Time;
                                  Buffer : in Local_Buffer_Ptr;
                                  T : in Transaction := Null_Transaction)
                                 return Member_Access.Ptr;

    function Retrieve_Element_At_Or_After
                (Ch : in Ref;
                 Time : in Ada.Real_Time.Time;
                 Buffer : in Local_Buffer_Ptr;
                 T : in Transaction := Null_Transaction)
                return Member_Access.Ptr;

    function Retrieve_Element_At_Or_Before
                (Ch : in Ref;
                 Time : in Ada.Real_Time.Time;
                 Buffer : in Local_Buffer_Ptr;
                 T : in Transaction := Null_Transaction)
                return Member_Access.Ptr;

    -- Get the time of storage for the current object in the chrono
    function Time_Of_Storage (Ch : in Ref;
                              T : in Transaction := Null_Transaction)
                             return Ada.Real_Time.Time;

    -- ------------------------------------------------------------------------
- --
    -- Iterator operations for relationships between atomic objects and chronos
--
    -- ------------------------------------------------------------------------
- --

    function First (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Last (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Next (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Prior (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;
```

```
    function Get_Element (R : in Relationship_Ref'Class;
                          T : in Transaction := Null_Transaction) return Ref;

    Null_Ref : constant Ref;

private

end Firm.Chronos;
```

APPENDIX G: FIRM.LISTS PACKAGE SPECIFICATION

```
with Ada.Tags;
with Firm.Atomic_Access;
generic
    type Member_Type (<>) is new Atomic_Object with private;
    with package Member_Access is new Firm.Atomic_Access (Member_Type);
package Firm.Lists is

    -- NOTE: The positional operations (i.e. Remove_Element_At) were removed
    --       from FIRM's implementation of the ODMG-93 Release 1.2 List
    --       collection for real-time performance reasons.

    -- ------------------------------------------------------------------ --
    -- Type decalarations for the ODMG-93 release 1.2 "list" collection  --
    -- (section 2.3.5.3, pg. 19).                                        --
    -- ------------------------------------------------------------------ --
    type Ref is new Collection_Ref with private;

    -- ------------------- --
    -- Properties of lists --
    -- ------------------- --

    -- The following properties are specified for all collections in
    -- ODMG-93 release 1.2, section 2.3.5, pg. 17.
    function Cardinality (L : in Ref; T : in Transaction := Null_Transaction)
                         return Natural;

    function Is_Empty (L : in Ref; T : in Transaction := Null_Transaction)
                      return Boolean;


    -- The following properties are not in ODMG-93 release 1.2; they have been
    -- added for FIRM.
    function Persistence (L : in Ref; T : in Transaction := Null_Transaction)
                         return Persistence_Type;

    function Is_Indexed (L : in Ref; T : in Transaction := Null_Transaction)
                        return Boolean;

    -- This function returns the tag of "Type_In_Collection"
    function Get_Tag (L : in Ref) return Ada.Tags.Tag;

    -- ------------------- --
    -- Operations on lists --
    -- ------------------- --

    -- The following operations are specified as available on all types
```

```
-- of collections in the ODMG-93 release 1.2 spec (see section 2.3.5,
-- pp. 17-18).  They are refined here for the list collection type.

function Create (Db : in Database_Id; Persistence : in Persistence_Type)
                return Ref;

-- Bind operation for collection objects (see ODMG-93 release 1.2,
-- section 2.9, pg. 33).
procedure Bind (L : in Ref;
                Name : in String;
                T : in Transaction := Null_Transaction);

procedure Copy (From_L : in Ref;
                To_L : in out Ref;
                T : in Transaction := Null_Transaction);

procedure Delete (L : in out Ref; T : in Transaction := Null_Transaction);

procedure Insert_Element (L : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

procedure Remove_Element (L : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

function Contains_Element
            (L : in Ref;
             Obj : in Member_Type'Class;
             T : in Transaction := Null_Transaction) return Boolean;

-- This operation is not in ODMG-93 release 1.2.  It has been added
-- for FIRM.
procedure Vacate (L : in out Ref; T : in Transaction := Null_Transaction);

-- Lookup operation for collections (ODMG-93 release 1.2, section 2.9,
-- page 33).
function Lookup (Db : in Database_Id;
                 Name : in String;
                 T : in Transaction := Null_Transaction) return Ref;


-- The following methods are for the integrated iterators that
-- FIRM provides with its list collections.

function First (L : in Ref;
                Buffer : in Local_Buffer_Ptr;
```

```
                T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Last (L : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
             return Member_Access.Ptr;

function Next (L : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
             return Member_Access.Ptr;

function Prior (L : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

procedure Reset (L : in Ref; T : in Transaction := Null_Transaction);

function Get_Element (L : in Ref;
                      Buffer : in Local_Buffer_Ptr;
                      T : in Transaction := Null_Transaction)
                    return Member_Access.Ptr;


-- The following operations are specified as available  for all list
-- collections in the ODMG-93 release 1.2 spec (see section 2.3.5.3,
-- pg. 19).

procedure Insert_Element_After (L : in out Ref;
                                Obj : in out Member_Type'Class;
                                T : in Transaction := Null_Transaction)
   renames Insert_Element;

procedure Insert_Element_Before (L : in out Ref;
                                 Obj : in out Member_Type'Class;
                                 T : in Transaction := Null_Transaction);

procedure Insert_Element_First (L : in out Ref;
                                Obj : in out Member_Type'Class;
                                T : in Transaction := Null_Transaction);

procedure Insert_Element_Last (L : in out Ref;
                               Obj : in out Member_Type'Class;
                               T : in Transaction := Null_Transaction);
```

```
    procedure Remove_First_Element (L : in out Ref;
                                    T : in Transaction := Null_Transaction);

    procedure Remove_First_Element (L : in out Ref;
                                    Obj : out Member_Access.Ptr;
                                    T : in Transaction := Null_Transaction);

    procedure Remove_Last_Element (L : in out Ref;
                                   T : in Transaction := Null_Transaction);

    procedure Remove_Last_Element (L : in out Ref;
                                   Obj : out Member_Access.Ptr;
                                   T : in Transaction := Null_Transaction);

    function Retrieve_First_Element (L : in Ref;
                                     Buffer : in Local_Buffer_Ptr;
                                     T : in Transaction := Null_Transaction)
                                    return Member_Access.Ptr renames First;

    function Retrieve_Last_Element (L : in Ref;
                                    Buffer : in Local_Buffer_Ptr;
                                    T : in Transaction := Null_Transaction)
                                   return Member_Access.Ptr renames Last;

    function Concat (Left : in Ref;
                     Right : in Ref;
                     T : in Transaction := Null_Transaction) return Ref;

    procedure Append (Left : in out Ref;
                      Right : in Ref;
                      T : in Transaction := Null_Transaction);

   -- ------------------------------------------------------------------- -
-
    -- Iterator operations for relationships between atomic objects and lists --
    -- ------------------------------------------------------------------- -
-

    function First (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Last (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Next (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;
```

```
    function Prior (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Get_Element (R : in Relationship_Ref'Class;
                          T : in Transaction := Null_Transaction) return Ref;

    Null_Ref : constant Ref;

private

end Firm.Lists;
```

APPENDIX H: FIRM.SETS PACKAGE SPECIFICATION

```
with Ada.Tags;
with Firm.Atomic_Access;
with System.Address_To_Access_Conversions;
generic
    type Member_Type (<>) is new Atomic_Object with private;
    with package Member_Access is new Firm.Atomic_Access (Member_Type);
package Firm.Sets is

    -- ------------------------------------------------------------- --
    -- Type decalarations for the ODMG-93 release 1.2 "set" collection  --
    -- (section 2.3.5.1, pg. 18).                                    --
    -- ------------------------------------------------------------- --
    type Ref is new Collection_Ref with private;

    -- ------------------ --
    -- Properties of sets --
    -- ------------------ --

    -- The following properties are specified for all collections in
    -- ODMG-93 release 1.2, section 2.3.5, pg. 17.
    function Cardinality (S : in Ref; T : in Transaction := Null_Transaction)
                         return Natural;

    function Is_Empty (S : in Ref; T : in Transaction := Null_Transaction)
                      return Boolean;


    -- The following properties are not in ODMG-93 release 1.2; they have been
    -- added for FIRM.
    function Persistence (S : in Ref; T : in Transaction := Null_Transaction)
                         return Persistence_Type;

    function Is_Indexed (S : in Ref; T : in Transaction := Null_Transaction)
                        return Boolean;

    -- This function returns the tag of "Member_Type"
    function Get_Tag (S : in Ref) return Ada.Tags.Tag;

    -- ------------------ --
    -- Operations on sets --
    -- ------------------ --

    -- The following operations are specified as available on all types
    -- of collections in the ODMG-93 release 1.2 spec (see section 2.3.5,
    -- pp. 17-18).  They are refined here for the set collection type.
```

```
function Create (Db : in Database_Id; Persistence : in Persistence_Type)
                return Ref;

-- Bind operation for collection objects (see ODMG-93 release 1.2,
-- section 2.9, pg. 33).
procedure Bind (S : in Ref;
                Name : in String;
                T : in Transaction := Null_Transaction);

procedure Copy (From_S : in Ref;
                To_S : in out Ref;
                T : in Transaction := Null_Transaction);

procedure Delete (S : in out Ref; T : in Transaction := Null_Transaction);

procedure Insert_Element (S : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

procedure Remove_Element (S : in out Ref;
                          Obj : in out Member_Type'Class;
                          T : in Transaction := Null_Transaction);

function Contains_Element
            (S : in Ref;
             Obj : in Member_Type'Class;
             T : in Transaction := Null_Transaction) return Boolean;

-- This operation is not in ODMG-93 release 1.2.  It has been added
-- for FIRM.
procedure Vacate (S : in out Ref; T : in Transaction := Null_Transaction);

-- Lookup operation for collections (ODMG-93 release 1.2, section 2.9,
-- page 33).
function Lookup (Db : in Database_Id;
                 Name : in String;
                 T : in Transaction := Null_Transaction) return Ref;


-- The following methods are for the integrated iterators that
-- FIRM provides with its set collections.

function First (S : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;
```

```
function Last (S : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Next (S : in Ref;
               Buffer : in Local_Buffer_Ptr;
               T : in Transaction := Null_Transaction)
              return Member_Access.Ptr;

function Prior (S : in Ref;
                Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction)
               return Member_Access.Ptr;

procedure Reset (S : in Ref; T : in Transaction := Null_Transaction);

function Get_Element (S : in Ref;
                      Buffer : in Local_Buffer_Ptr;
                      T : in Transaction := Null_Transaction)
                     return Member_Access.Ptr;

-- The following operations are specified as available  for all set
-- collections in the ODMG-93 release 1.2 spec (see section 2.3.5.1,
-- pg. 18).

function Union (S1 : in Ref;
                S2 : in Ref;
                T : in Transaction := Null_Transaction) return Ref;

function Intersection (S1 : in Ref;
                       S2 : in Ref;
                       T : in Transaction := Null_Transaction) return Ref;

function Difference (S1 : in Ref;
                     S2 : in Ref;
                     T : in Transaction := Null_Transaction) return Ref;

function Is_Subset (Left : in Ref;
                    Right : in Ref;
                    T : in Transaction := Null_Transaction) return Boolean;

function Is_Proper_Subset
            (Left : in Ref;
             Right : in Ref;
             T : in Transaction := Null_Transaction) return Boolean;
```

```
    function Is_Superset
                (Left : in Ref;
                 Right : in Ref;
                 T : in Transaction := Null_Transaction) return Boolean;

    function Is_Proper_Superset
                (Left : in Ref;
                 Right : in Ref;
                 T : in Transaction := Null_Transaction) return Boolean;

   -- ------------------------------------------------------------------- --
    -- Iterator operations for relationships between atomic objects and sets --
   -- ------------------------------------------------------------------- --

    function First (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Last (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Next (R : in Relationship_Ref'Class;
                   T : in Transaction := Null_Transaction) return Ref;

    function Prior (R : in Relationship_Ref'Class;
                    T : in Transaction := Null_Transaction) return Ref;

    function Get_Element (R : in Relationship_Ref'Class;
                          T : in Transaction := Null_Transaction) return Ref;

    Null_Ref : constant Ref;

private

end Firm.Sets;
```

APPENDIX I: FIRM.INDICES PACKAGE SPECIFICATION

```
with Firm.Atomic_Access;
generic
    type Keyed_Atomic_Object (<>) is new Atomic_Object with private;
    with package Member_Access is new Firm.Atomic_Access (Keyed_Atomic_Object);
    with function Equal_To (L, R : Keyed_Atomic_Object'Class) return Boolean;
    with function Less_Than (L, R : Keyed_Atomic_Object'Class) return Boolean;
package Firm.Indices is

    type Ref is new Index_Ref with private;

    -- Procedure for specifying how many index objects may be created for each
    -- instantiation of this package.
    procedure Create_Global_Pool
                (Db : in Database_Id;
                 -- Database the pool belongs to
                 Instances : in Natural
                -- Number of instances that will have the desired persistence.
This
                 -- count must include all versions of each instance if multi-
                 -- version concurrency control is in use.
                 );

    -- The ODMG-93 release 1.2 specification does not provide an interface
    -- for indices, although section 2.2.3 does give semantics for keys.
    -- Indices are named by default, so the Bind operation is not provided
    -- for indices.
    function Create (C : in Collection_Ref'Class;
                     Name : in String;
                     Duplicate_Keys : in Boolean) return Ref;

    procedure Delete (I : in out Ref; T : in Transaction := Null_Transaction);

    function Find_Match (I : in Ref;
                         Obj : in Atomic_Object'Class;
                         Buffer : in Local_Buffer_Ptr;
                         T : in Transaction := Null_Transaction)
                        return Member_Access.Ptr;

    function Lookup (Db : in Database_Id;
                     Name : in String;
                     T : in Transaction := Null_Transaction) return Ref;

    -- -------------------- --
    -- Iterators on indices --
    -- -------------------- --
```

```
    -- The following methods are for the integrated iterators that
    -- FIRM provides with its collection objects and their indices.

    function First (I : in Ref;
                    Buffer : in Local_Buffer_Ptr;
                    T : in Transaction := Null_Transaction)
                   return Member_Access.Ptr;

    function Last (I : in Ref;
                   Buffer : in Local_Buffer_Ptr;
                   T : in Transaction := Null_Transaction)
                  return Member_Access.Ptr;

    function Next (I : in Ref;
                   Buffer : in Local_Buffer_Ptr;
                   T : in Transaction := Null_Transaction)
                  return Member_Access.Ptr;

    function Prior (I : in Ref;
                    Buffer : in Local_Buffer_Ptr;
                    T : in Transaction := Null_Transaction)
                   return Member_Access.Ptr;

    procedure Reset (I : in Ref; T : in Transaction := Null_Transaction);

    function Get_Element (I : in Ref;
                          Buffer : in Local_Buffer_Ptr;
                          T : in Transaction := Null_Transaction)
                         return Member_Access.Ptr;

    -- Null index constant
    Null_Ref : constant Ref;

private

end Firm.Indices;
```

APPENDIX J: FIRM.ATOMIC_RELATIONSHIPS PACKAGE SPEC.

```
with Firm.Atomic_Access;
generic
    type From_Type (<>) is new Atomic_Object with private;
    with package From_Access is new Firm.Atomic_Access (From_Type);
    type To_Type (<>) is new Atomic_Object with private;
    with package To_Access is new Firm.Atomic_Access (To_Type);
package Firm.Atomic_Relationships is

    type Ref is new Relationship_Ref with private;

    -- The ODMG-93 release 1.2 specification does not specify operations for
    -- relationships, although section 2.5.2 on pp. 25-26 does specify the
    -- semantics of relationships.  In FIRM, relationships are first-class
    -- objects (see section 2.10.4, item 1). Relationships in FIRM are
    -- always named, so a Bind operation is not provided.

    -- ALSO, FIRM supports ONLY unordered relationships.

    function Create (Db : in Database_Id;
                     Persistence : in Persistence_Type;
                     Rel_Type : in Relationship_Type;
                     Name : in String) return Ref;

    procedure Delete (R : in out Ref; T : in Transaction := Null_Transaction);

    -- Lookup operation for relationships, see ODMG-93 release 1.2, section
    -- 2.9, pg. 33.
    function Lookup (Db : in Database_Id;
                     Name : in String;
                     T : in Transaction := Null_Transaction) return Ref;

    procedure Add_Traversal_Path (From : in out From_Type'Class;
                                  To : in out To_Type'Class;
                                  R : in out Ref;
                                  T : in Transaction := Null_Transaction);

    procedure Remove_Traversal_Path (From : in out From_Type'Class;
                                     To : in out To_Type'Class;
                                     R : in out Ref;
                                     T : in Transaction := Null_Transaction);

    procedure Remove_All_Paths (From : in out Atomic_Object'Class;
                                R : in out Ref;
                                T : in Transaction := Null_Transaction);

    procedure Set_Iterator (R : in Ref;
```

```
                        On : in Atomic_Object'Class;
                        T : in Transaction := Null_Transaction);

-- Returns pointer to iteration "base", i.e. object from which traversal path
 -- iteration is performed.
 function Get_Iterator
             (R : in Ref;
              Buffer : in Local_Buffer_Ptr;
              T : in Transaction := Null_Transaction) return Atomic_Ptr;

 function First (R : in Ref;
                  Buffer : in Local_Buffer_Ptr;
                  T : in Transaction := Null_Transaction)
                 return To_Access.Ptr;

 function Last (R : in Ref;
                 Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction) return To_Access.Ptr;

 function Next (R : in Ref;
                 Buffer : in Local_Buffer_Ptr;
                T : in Transaction := Null_Transaction) return To_Access.Ptr;

 function Prior (R : in Ref;
                  Buffer : in Local_Buffer_Ptr;
                  T : in Transaction := Null_Transaction)
                 return To_Access.Ptr;

 function Get_Element (R : in Ref;
                       Buffer : in Local_Buffer_Ptr;
                       T : in Transaction := Null_Transaction)
                      return To_Access.Ptr;

 function First (R : in Ref;
                  Buffer : in Local_Buffer_Ptr;
                  T : in Transaction := Null_Transaction)
                 return From_Access.Ptr;

 function Last (R : in Ref;
                 Buffer : in Local_Buffer_Ptr;
                 T : in Transaction := Null_Transaction)
                return From_Access.Ptr;

 function Next (R : in Ref;
                 Buffer : in Local_Buffer_Ptr;
                 T : in Transaction := Null_Transaction)
                return From_Access.Ptr;
```

```
    function Prior (R : in Ref;
                    Buffer : in Local_Buffer_Ptr;
                    T : in Transaction := Null_Transaction)
                   return From_Access.Ptr;

    function Get_Element (R : in Ref;
                          Buffer : in Local_Buffer_Ptr;
                          T : in Transaction := Null_Transaction)
                         return From_Access.Ptr;

    procedure Reset (R : in Ref; T : in Transaction := Null_Transaction);

    Null_Ref : constant Ref;

private

end Firm.Atomic_Relationships;
```

APPENDIX K: FIRM.COLLECTION_RELATIONSHIPS PACKAGE SPEC.

```
with Firm.Atomic_Access;
generic
    type From_Type (<>) is new Atomic_Object with private;
    with package From_Access is new Firm.Atomic_Access (From_Type);
    type To_Type is new Collection_Ref with private;
package Firm.Collection_Relationships is

    type Ref is new Relationship_Ref with private;

    -- The ODMG-93 release 1.2 specification does not specify operations for
    -- relationships, although section 2.5.2 on pp. 25-26 does specify the
    -- semantics of relationships.  In FIRM, relationships are first-class
    -- objects (see section 2.10.4, item 1). Relationships in FIRM are
    -- always named, so a Bind operation is not provided.

    -- ALSO, FIRM supports ONLY unordered relationships.

    function Create (Db : in Database_Id;
                     Persistence : in Persistence_Type;
                     Rel_Type : in Relationship_Type;
                     Name : in String) return Ref;

    procedure Delete (R : in out Ref; T : in Transaction := Null_Transaction);

    -- Lookup operation for relationships, see ODMG-93 release 1.2, section
    -- 2.9, pg. 33.
    function Lookup (Db : in Database_Id;
                     Name : in String;
                     T : in Transaction := Null_Transaction) return Ref;

    procedure Add_Traversal_Path (From : in From_Type'Class;
                                  To : in To_Type;
                                  R : in out Ref;
                                  T : in Transaction := Null_Transaction);

    procedure Remove_Traversal_Path (From : in From_Type'Class;
                                     To : in To_Type;
                                     R : in out Ref;
                                     T : in Transaction := Null_Transaction);

    procedure Remove_All_Paths (From : in From_Type'Class;
                                R : in out Ref;
                                T : in Transaction := Null_Transaction);

    procedure Remove_All_Paths (To : in To_Type;
                                R : in out Ref;
```

```
                                T : in Transaction := Null_Transaction);

    -- This method sets the iterator to null and positions it on the
    -- specified object. Iteration then returns references to the objects
    -- on the "other end" of the relationship's traversal paths.
    procedure Set_Iterator (R : in Ref;
                            On : in From_Type'Class;
                            T : in Transaction := Null_Transaction);

    procedure Set_Iterator (R : in Ref;
                            On : in To_Type;
                            T : in Transaction := Null_Transaction);

    function First (R : in Ref;
                    Buffer : in Local_Buffer_Ptr;
                    T : in Transaction := Null_Transaction)
                   return From_Access.Ptr;

    function Last (R : in Ref;
                   Buffer : in Local_Buffer_Ptr;
                   T : in Transaction := Null_Transaction)
                  return From_Access.Ptr;

    function Next (R : in Ref;
                   Buffer : in Local_Buffer_Ptr;
                   T : in Transaction := Null_Transaction)
                  return From_Access.Ptr;

    function Prior (R : in Ref;
                    Buffer : in Local_Buffer_Ptr;
                    T : in Transaction := Null_Transaction)
                   return From_Access.Ptr;

    function Get_Element (R : in Ref;
                          Buffer : in Local_Buffer_Ptr;
                          T : in Transaction := Null_Transaction)
                         return From_Access.Ptr;

    procedure Reset (R : in Ref; T : in Transaction := Null_Transaction);

    Null_Ref : constant Ref;

private

end Firm.Collection_Relationships;
```