Table of Contents

Стандарт объектной модели данных ODMG

Предпосылки и история создания стандарта объектной модели данных

An object database management system (ODBMS, also referred to as object-oriented database management system or OODBMS), is a database management system (DBMS) that supports the modelling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects.

In their influential paper, The Object-Oriented Database System Manifesto, Malcolm Atkinson and others define an OODBMS in 1995 as follows: An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness. Their paper describes each of these features in detail.

There is currently no widely agreed-upon standard for what constitutes an ODBMS and no standard query language to ODBMS equivalent to what SQL is to RDBMS (relational DBMS.) Initiatives by an industry group, the Object Data Management Group (ODMG), to create a standardized Object Query Language (OQL) have been abandoned in 2001. JSR 12's Java Data Objects (JDO) has been another attempt to standardize data access that has not seen adoption by the mainstream. More recent research (2005) by William Cook suggests to use the programming language itself, e.g. Java or .NET, to query objects and thus make a separate OO query standard redundant (Read more about Native Queries).

ODBMS were originally thought of to replace RDBMS because of their better fit with object-oriented programming languages. However, high switching cost, the inclusion of object-oriented features in RDBMS to make them ORDBMS, and the emergence of object-relational mappers (ORMs) have made RDBMS successfully defend their dominance in the data center for server-side persistence.

Object databases are now established as a complement, not a replacement for relational databases. They found their place as embeddable persistence solutions in devices, on clients, in packaged software, in real-time control systems, and to power websites. The open source community has created a new wave of enthusiasm that's now fueling the rapid growth of ODBMS installations. [7, 12]

История

Early 1980s - Orion Research Project at MCC

Won Kim at MCC (Microelectronics and Computer Technology Corporation) in Austin, Texas, begins a research project on ORION.

Two products will later trace their history to ORION: ITASCA (no longer around) and Versant.

Late 1980s - First wave of commercial products

A Lisp-based system, Graphael, appears from the French nuclear regulatory efforts. Eventually, Graphael goes through a re-write and becomes Matisse.

Servo-Logic begins work on GemStone. Servo-Logic is now GemStone Systems.

Start of O2 development at INRIA (France). The founder of O2 is Francois Bencilhon, also from MCC.

Tom Atwood at Ontologic produced Vbase, which supports the proprietary language COP (for C Object

Processor). COP is eventually eclipsed by C++, Ontologic becomes ONTOS, and the database is rewritten to support C++. Tom left Ontologic in the late 1980s and founded Object Design (now part of Progress Software) with ObjectStore (based on C++).

Another product from that time is Objectivity/DB. Drew Wade has been one of the founders of its vendor, Objectivity.

1991 - ODMG

Rick Cattell (SunSoft) initiates the ODMG with 5 major OODBMS vendors. The first standard, ODMG 1.0, was released in 1993. Throughout the 1990s, the ODMG works with the X3H2 (SQL) committee on a common query language. Though no specific goal is achieved, the efforts heavily influence the ODMG OQL (object query language) and, to a lesser extent, SQL:1999.

1995 - The OODBMS Manifesto

Malcolm Atkinson et. al. release "The Object-Oriented Database System Manifesto"

1990s - First Growth Period

Market for commercial ODBMS products grows to some $100M, peaks in 2000 and shrinks since

2001 - Final ODMG 3.0 standards released

A final ODMG 3.0 standards is released. Shortly thereafter, the ODMG submits the ODMG Java Binding to the Java Community Process as a basis for the Java Data Objects (JDO) Specification.

Afterwards, the ODMG disbands.

2004 - Advent of Open Source

db4o released as free, open source ODBMS. In November 2005, db4o is first to implement Native Queries as an object oriented data access API that relies entirely on the programming language (Java/C#) itself.

2005-now. New Data Stores

The world of data management is changing. Service platforms, scalable cloud platforms, analytical data platforms, object databases, object-relational bindings, NoSQL databases and new approaches to concurrency control are all becoming hot topics both in academia and industry.

For up to date information and interviews, please check the "ODBMS Industry Watch Blog." [12, 7]

Введение в объектную модель данных ODMG

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

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

•The basic modeling primitives are the object and the literal. Each object has a unique identifier. A literal has no identifier.

•Objects and literals can be categorized by their types. All elements of a given type have a common range of states (i.e., the same set of properties) and common behavior (i.e., the same set of defined operations). An object is sometimes referred to as an instance of its type.

•The state of an object is defined by the values it carries for a set of properties. These properties can be attributes of the object itself or relationships between the object and one or more other objects. Typically, the values of an object's properties can change over time.

•The behavior of an object is defined by the set of operations that can be executed on or by the object. Operations may have a list of input and output parameters, each with a specified type. Each operation may also return a typed result.

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

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

The application's object model is the ODMS's (logical) schema. The ODMG Object Model is the fundamental definition of an ODMS's functionality. It includes significantly richer semantics than does the relational model, by declaring relationships and operations explicitly. [5]

Компоненты архитектуры ODMG 3.0

The major components of ODMG 3.0 are:

- Object Model. The OMG core model was designed to be a common denominator for object request brokers, object database systems, object programming languages, and other applications. In keeping with the OMG Architecture, ODMS profile had been designed for their model, adding components (e.g.,relationships) to the OMG core object model to support our needs.
- Object Specification Languages. ODL is a specification language used to define the object types that conform to the ODMG Object Model. OIF is a specification language used to dump and load the current state of an ODMS to or from a file or set of files.
- Object Query Language. A declarative (nonprocedural) language for querying and updating ODMS objects had been defined. Relational standard SQL has been used as the basis for OQL, where possible, though OQL supports more powerful capabilities.
- C++ Language Binding
- Smalltalk Language Binding
- Java Language Binding

It is possible to read and write the same database from C++, Smalltalk, and Java, as long as the programmer stays within the common subset of supported datatypes. Note that unlike SQL in relational systems, the ODMG data manipulation languages are tailored to specific application programming languages, in order to provide a single, integrated environment for programming and data manipulation. We don't believe exclusively in a universal DML syntax. We go further than relational systems, as we support a unified object model for sharing data across programming languages, as well as a common query language.

Язык определения объектов ODMG

The Object Definition Language is a specification language used to define the specifications of object types that conform to the ODMG Object Model. ODL is used to support the portability of object schemas across conforming ODMSs.

Several principles have guided the development of the ODL, including the following:

- ODL should support all semantic constructs of the ODMG Object Model.
- ODL should not be a full programming language, but rather a definition language for object specifications.
- ODL should be programming language independent.
- ODL should be compatible with the OMG's Interface Definition Language (IDL).
- ODL should be extensible, not only for future functionality, but also for physical optimizations.
- ODL should be practical, providing value to application developers, while being supportable by the ODMS vendors within a relatively short time frame after publication of the specification.

ODL is not intended to be a full programming language. It is a definition language for object specifications. Database management systems (DBMSs) have traditionally provided facilities that support data definition (using a Data Definition Language or DDL) and data manipulation (using a Data Manipulation Language or DML). The DDL allows users to define their datatypes and interfaces. DML allows programs to create, delete, read, change, and so on, instances of those datatypes. The ODL described in this chapter is a DDL for object types. It defines the characteristics of types, including their properties and operations. ODL defines only the signatures of operations and does not address definition of the methods that implement those operations. The ODMG standard does not provide an OML specification. Chapters 5, 6, and 7 define standard APIs to bind conformant ODMSs to C++, Smalltalk, and Java, respectively.

ODL is intended to define object types that can be implemented in a variety of programming languages. Therefore, ODL is not tied to the syntax of a particular programming language. Users can use ODL to define schema semantics in a programming language–independent way. A schema specified in ODL can be supported by any ODMG-compliant ODMS and by mixed-language implementations.

This portability is necessary for an application to be able to run with minimal modification on a variety of ODMSs. Some applications may in fact need simultaneous support from multiple ODMSs. Others may need to access objects created and stored using different programming languages. ODL provides a degree of insulation for applications against the variations in both programming languages and underlying ODMS products.

The C++, Smalltalk, and Java ODL bindings are designed to fit smoothly into the declarative syntax of their host programming language. Due to the differences inherent in the object models native to these programming languages, it is not always possible to achieve consistent semantics across the programming language–specific versions of ODL.

The syntax of ODL extends IDL—the Interface Definition Language developed by the OMG as part of the Common Object Request Broker Architecture (CORBA). IDL was itself influenced by C++, giving ODL a C++ flavor. ODL adds to IDL the constructs required to specify the complete semantics of the ODMG Object Model.

ODL also provides a context for integrating schemas from multiple sources and applications. These source schemas may have been defined with any number of object models and data definition

languages; ODL is a sort of lingua franca for integration.

For example, various standards organizations like STEP/PDES (Express), INCITS X3H2 (SQL), INCITS X3H7 (Object Information Management), CFI (CAD Framework Initiative), and others have developed a variety of object models and, in some cases, data definition languages. Any of these models can be translated to an ODL specification (Figure 1). This common basis then allows the various models to be integrated with common semantics. An ODL specification can be realized concretely in an object programming language like C++, Smalltalk, or Java.

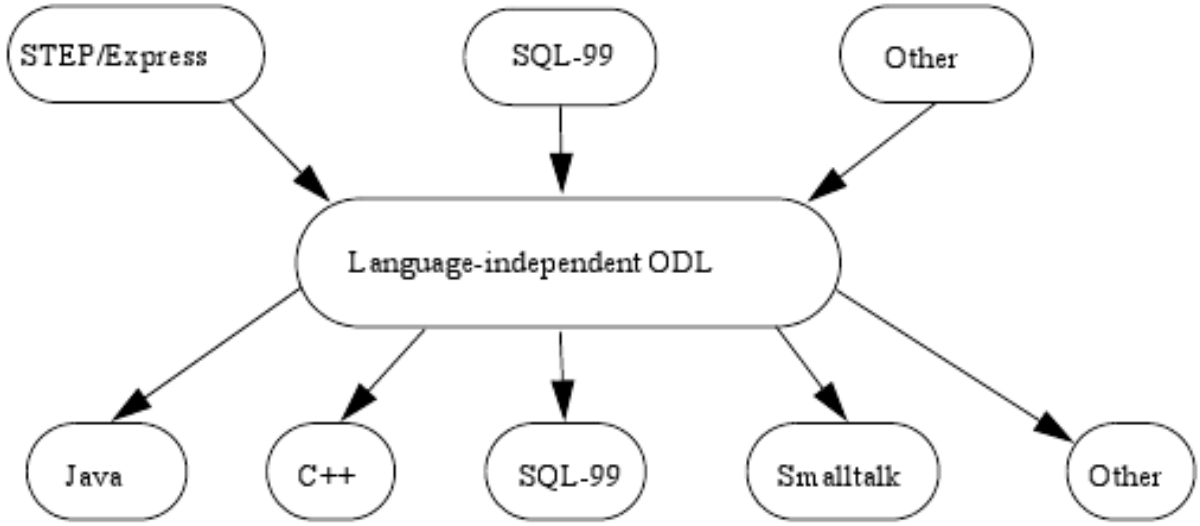

Figure 1.

Specification

A type is defined by specifying its interface or by its class in ODL. The top-level Extended Backus Naur Form (EBNF) for ODL is as follows:

```
 <interface> ::= <interface_dcl>
| <interface_forward_dcl>
 <interface_dcl> ::= <interface_header>
{ [ <interface_body> ] }
 <interface_forward_dcl>::= interface <identifier>
 <interface_header>::= interface <identifier>
[ <inheritance_spec> ]
 <class> ::= <class_dcl> | <class_forward_dcl>
 <class_dcl> ::= <class_header> { <interface_body> }
 <class_forward_dcl> ::= class <identifier>
 <class_header>::= class <identifier>
[ extends <scopedName> ]
[ <inheritance_spec> ]
[ <type_property_list> ]
```

The characteristics of the type itself appear first, followed by lists that define the properties and operations of its interface or class. Any list may be omitted if it is not applicable.

Type Characteristics

Supertype information, extent naming, and specification of keys (i.e., uniqueness constraints) are all characteristics of types, but do not apply directly to the types' instances. The EBNF for type characteristics follows:

```
 <inheritance_spec> ::= : <scoped_name> [ , <inheritance_spec> ]
 <type_property_list> ::= ( [ <extent_spec> ] [ <key_spec> ] )
 <extent_spec>::= extent <string>
 <key_spec> ::= key[s] <key_list>
 <key_list> ::= <key> | <key> , <key_list>
 <key>::= <property_name> | ( <property_list> )
 <property_list>::= <property_name>
 | <property_name> , <property_list>
 <property_name>::= <identifier>
 <scoped_name>::= <identifier>
| :: <identifier>
| <scoped_name> :: <identifier>
```

Each supertype must be specified in its own type definition. Each attribute or relationship traversal path named as (part of) a type's key must be specified in the key_spec of the type definition. The extent and key definitions may be omitted if inapplicable to the type being defined. A type definition should include no more than one extent or key definition.

A simple example for the class definition of a Professor type is

```
 class Professor
 ( extent professors)
 {
 properties
 operations
 };
```

Keywords are highlighted.

Instance Properties

A type's instance properties are the attributes and relationships of its instances. These properties are specified in attribute and relationship specifications. The EBNF is

```
 <interface_body> ::= <export> | <export> <interface_body>
 <export> ::= <type_dcl> ;
| <const_dcl> ;
| <except_dcl> ;
| <attr_dcl> ;
| <rel_dcl> ;
| <op_dcl> ;
```

Attributes

The EBNF for specifying an attribute follows:

```
 <attr_dcl>::= [ readonly ] attribute
<domain_type> <attribute_name>
 [ <fixed_array_size>]
 <attribute_name>::= <identifier>
 <domain_type> ::= <simple_type_spec>
| <struct_type>
| <enum_type>
```

For example, adding attribute definitions to the Professor type's ODL specification:

```
 class Professor
 (extent professors)
 {
attribute string name;
attribute unsigned short faculty_id[6];
attribute long soc_sec_no[10];
attribute Address address;
attribute set<string> degrees;
relationships
operations
 };
```

Note that the keyword attribute is mandatory.

Relationships

A relationship specification names and defines a traversal path for a relationship. A traversal path definition includes designation of the target type and information about the inverse traversal path found in the target type. The EBNF for relationship specification follows:

```
 <rel_dcl> ::= relationship
<target_of_path> <identifier>
 inverse <inverse_traversal_path>
 <target_of_path> ::= <identifier>
| <coll_spec> < <identifier> >
 <inverse_traversal_path>::= <identifier> :: <identifier>
```

Traversal path cardinality information is included in the specification of the target of a traversal path. The target type must be specified with its own type definition. Use of the collection_type option of the EBNF indicates cardinality greater than one on the target side. If this option is omitted, the cardinality on the target side is one. The most commonly used collection types are expected to be Set, for unordered members on the target side of a traversal path, and List, for ordered members on the target side. Bags are supported as well. The inverse traversal path must be defined in the property list of the target type's definition. For example, adding relationships to the Professor type's interface specification:

```
 class Professor
 ( extent professors)
 {
 attribute string name;
 attribute unsigned short faculty_id[6];
 attribute long soc_sec_no[10];
 attribute Address address;
 attribute set<string> degrees;
 relationship set<Student> advises
 inverse Student::advisor;
 relationship set<TA> teaching_assistants
 inverse TA::works_for;
 relationship Department department
 inverse Department::faculty;
 operations
 };
```

The keyword relationship is mandatory. Note that the attribute and relationship specifications can be mixed in the property list. It is not necessary to define all of one kind of property, then all of the other kind.

Operations

```
ODL is compatible with IDL for specification of operations:
 <op_dcl>::= [ <op_attribute>] <op_type_spec>
 <identifier> <parameter_dcls>
 [ <raises_expr> ] [ <context_expr> ]
 <op_attribute>::= oneway
 <op_type_spec> ::= <simple_type_spec>
 | void
 <parameter_dcls>::= ( [ <param_dcl_list> ] )
 <param_dcl_list>::= <param_dcl>
 | <param_dcl> , <param_dcl_list>
 <param_dcl> ::= <param_attribute> <simple_type_spec>
 <declarator>
 <param_attribute> ::= in | out| inout
 <raises_expr> ::= raises ( <scoped_name_list>)
 <context_expr> ::= context ( <string_literal_list> )
 <scoped_name_list>::= <scoped_name>
 | <scoped_name> , <scoped_name_list>
 <string_literal_list> ::= <string_literal>
 | <string_literal> , <string_literal_list>
```

[5]

Объектная модель ODMG, отношение подтипа и наследование. Система встроенных типов ODMG

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

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

Two objects or literals have the same type if and only if they have been declared to be instances of the same named type. Objects or literals that have been declared to be instances of two different types are not of the same type, even if the types in question define the same set of properties and operations. Type compatibility follows the subtyping relationships defined by the type hierarchy. If TS is a subtype of T, then an object of type TS can be assigned to a variable of type T, but the reverse is not possible. No implicit conversions between types are provided by the Object Model. Two atomic literals have the same type if they belong to the same set of literals. Depending on programming language bindings, implicit conversions may be provided between the scalar literal types, that is, long, short, unsigned long, unsigned short, float, double, boolean, octet, and char. No implicit conversions are provided for structured literals.

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

Тип связи в модели ODMG

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

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

A relationship is defined explicitly by declaration of traversal paths that enable applications to use the logical connections between the objects participating in the relationship. Traversal paths are declared in pairs, one for each direction of traversal of the relationship. For example, a professor teaches courses and a course is taught by a professor. The teaches traversal path would be defined in the declaration for the Professor type. The is_taught_by traversal path would be defined in the declaration for the Course type. The fact that these traversal paths both apply to the same relationship is indicated by an inverse clause in both of the traversal path declarations. For example:

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

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

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

The ODMS is responsible for maintaining the referential integrity of relationships.

This means that if an object that participates in a relationship is deleted, then any traversal path to that object must also be deleted. For example, if a particular Course instance is deleted, then not only is that object’s reference to a Professor instance via the is_taught_by traversal path deleted, but also any references in Professor objects to the Course instance via the teaches traversal path must also be deleted. Maintaining referential integrity ensures that applications cannot dereference traversal paths that lead to nonexistent objects.

attribute Studenttop_of_class;

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

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

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

The implementation of relationships is encapsulated by public operations that form and drop members from the relationship, plus public operations on the relationship target classes to provide access and to manage the required referential integrity constraints.

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

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

Cardinality "One" Relationships

For relationships with cardinality "one" such as

```
relationshipXY inverse Z;
```

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

```
attribute X Y;
voidform_Y(in X target) raises(IntegrityError);
voiddrop_Y(in X target) raises (IntegrityError);
```

For example, the relationship in the preceding example interface Course would result

in the following definitions (on the class Course):

```
 attribute Professor is_taught_by;
 voidform_is_taught_by(in Professor aProfessor)
raises(IntegrityError);
 voiddrop_is_taught_by(in Professor aProfessor)
raises(IntegrityError);
```

Cardinality "Many" Relationships

For ODL relationships with cardinality "many" such as

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

we expand the relationship to an equivalent IDL attribute and operations. To convert these definitions into pure IDL, the ODL collection need only be replaced by the keyword sequence. Note that the

add_Y operation may raise an IntegrityError exception in the event that the traversal is a set that already contains a reference to the given target X. This exception, if it occurs, will also be raised by the form_Y operation that invoked the add_Y. For example:

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

The relationship in the preceding example interface Professor would result in the following definitions (on the class Professor):

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

Модель исключений

The ODMG Object Model supports dynamically nested exception handlers, using a termination model of exception handling. Operations can raise exceptions, and exceptions can communicate exception results. Mappings for exceptions are defined by each language binding. When an exception is raised, information on the cause of the exception is passed back to the exception handler as properties of the exception. Control is as follows:

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

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

Представление схемы базы данных в ODMG. Примеры.

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

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

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

```
 interface Database {
 exception DatabaseOpen{};
 exception DatabaseNotFound{};
 exception ObjectNameNotUnique{};
 exception ObjectNameNotFound{};
 voidopen(in string odms_name)
raises(DatabaseNotFound,
DatabaseOpen);
 voidclose() raises(DatabaseClosed,
 TransactionInProgress);
 voidbind(in Object an_object, in string name)
raises(DatabaseClosed,
ObjectNameNotUnique,
TransactionNotInProgress);
 Objectunbind(in string name)
raises(DatabaseClosed,
ObjectNameNotFound,
TransactionNotInProgress);
 Objectlookup(in string object_name)
raises(DatabaseClosed,
ObjectNameNotFound,
TransactionNotInProgress);
 ODLMetaObjects::Module schema()
raises(DatabaseClosed,
TransactionNotInProgress);
 };
```

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

The lookup operation finds the identifier of the object with the name supplied as the argument to the operation. This operation is defined on the Database type, because the scope of object names is the ODMS. The names of objects in the ODMS, the names of types in the ODMS schema, and the extents of types instantiated in the ODMS are global. They become accessible to a program once it has opened the ODMS. Named objects are convenient entry points to the ODMS. A name is bound to an object using the bind operation. Named objects may be unnamed using the unbind operation. The schema operation accesses the root meta object that defines the schema of the ODMS. The schema of an ODMS is contained within a single Module meta object. Meta objects contained within the schema may be located via navigation of the appropriate relationships or by using the resolve operation with a scoped name as the argument. A scoped name is defined by the syntax of ODL and uses double colon (::) delimiters to specify a search path composed of meta object names that uniquely identify each meta object by its location within the schema.

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

Объектный язык запросов ODMG

An object query language named OQL is described, which supports the ODMG data model. It is complete and simple. It deals with complex objects without privileging the set construct and the select-

from-where clause.

Design is based on the following principles and assumptions:

- OQL relies on the ODMG Object Model.
- OQL is very close to SQL-92. Extensions concern object-oriented notions, like complex objects, object identity, path expressions, polymorphism, opeation invocation, and late binding.
- OQL provides high-level primitives to deal with sets of objects but is not restricted to this collection construct. It also provides primitives to deal with structures, lists, and arrays and treats such constructs with the same efficiency.
- OQL is a functional language where operators can freely be composed, as long as the operands respect the type system. This is a consequence of the fact that the result of any query has a type that belongs to the ODMG type model and thus can be queried again.
- OQL is not computationally complete. It is a simple-to-use query language.
- Based on the same type system, OQL can be invoked from within programming languages for which an ODMG binding is defined. Conversely, OQL can invoke operations programmed in these languages.
- OQL does not provide explicit update operators but rather invokes operations defined on objects for that purpose, and thus does not breach the semantics of an object model, which, by definition, is managed by the “methods” defined on the objects.
- OQL provides declarative access to objects. Thus, OQL queries can be easily optimized by virtue of this declarative nature.
- The formal semantics of OQL can easily be defined.

Общая форма запросов

As a stand-alone language, OQL allows querying denotable objects starting from their names, which act as entry points into a database. A name may denote any kind of object, that is, atomic, structure, collection, or literal.

As an embedded language, OQL allows querying denotable objects that are supported by the native language through expressions yielding atoms, structures, collections, and literals. An OQL query is a function that delivers an object whose type may be inferred from the operator contributing to the query expression. This point is illustrated with two short examples.

Assume a schema that defines the types Person and Employee as follows. These types have the extents Persons and Employees, respectively. One of these persons is the chairman (and there is an entry-point Chairman to that person). The type Person defines the name, birthdate, and salary as attributes and the operation age. The type Employee, a subtype of Person, defines the relationship subordinates and the operation seniority.

```
 select distinct x.age
 from Persons x
 where x.name = "Pat"
```

This selects the set of ages of all persons named Pat, returning a literal of type

```
set<integer>.
 select distinct struct(a: x.age, s: x.sex)
 from Persons x
 where x.name = "Pat"
```

This does about the same, but for each person, it builds a structure containing age and sex. It returns a literal of type set<struct>.

```
 select distinct struct(name: x.name, hps: (select y
from x.subordinates as y
where y.salary >100000))
 from Employees x
```

This is the same type of example, but now we use a more complex function. For each employee we build a structure with the name of the employee and the set of the employee's highly paid subordinates. Notice we have used a select-from-where clause in the select part. For each employee x, to compute hps, we traverse the relationship subordinates and select among this set the employees with a salary superior to $100,000.

The result of this query is therefore a literal of the type set<struct>, namely:

```
set<struct (name: string, hps: bag<Employee>)>
```

We could also use a select operator in the from part:

```
 select struct (a: x.age, s: x.sex)
 from (select y from Employees y where y.seniority = "10") as x
 where x.name = "Pat"
```

Of course, you do not always have to use a select-from-where clause:

```
 Chairman
retrieves the Chairman object.
 Chairman.subordinates
retrieves the set of subordinates of the Chairman.
 Persons
gives the set of all persons.
```

Выражения путей. Особенности вызова методов.

You can enter a database through a named object, but more generally as long as you get an object, you need a way to navigate from it and reach the right data. To do this in OQL, we use the "." (or indifferently "->") notation, which enables us to go inside complex objects, as well as to follow simple relationships. For example, we have a Person p and we want to know the name of the city where this person's spouse lives.

Example:

```
p.spouse.address.city.name
```

This query starts from a Person, gets his/her spouse, a Person again, goes inside the complex attribute of type Address to get the City object, whose name is then accessed. This example treated a 1-1 relationship; let us now look at n-p relationships. Assume we want the names of the children of the person p. We cannot write p.children.name because children is a list of references, so the interpretation of the result of this query would be undefined. Intuitively, the result should be a collection of names, but we need an unambiguous notation to traverse such a multiple relationship, and we use the select-from-where clause to handle collections just as in SQL.

Example:

```
select c.name
from p.children c
```

The result of this query is a value of type bag<string>. If we want to get a set, we simply drop duplicates, like in SQL, by using the distinct keyword.

Example:

```
select distinct c.name
from p.children c
```

Now we have a means to navigate from an object to any object following any relationship and entering any complex subvalues of an object. For instance, we want the set of addresses of the children of each Person of the database. We know the collection named Persons contains all the persons of the database. We now have to traverse two collections: Persons and Person.children. Like in SQL, the select-from operator allows us to query more than one collection. These collections then appear in the from part. In OQL, a collection in the from part can be derived from a previous one by following a path that starts from it.

Example:

```
select c.address
from Persons p,
p.children c
```

This query inspects all children of all persons. Its result is a value whose type is `bag<Address>.`

Predicate

Of course, the where clause can be used to define any predicate, which then serves to select only the data matching the predicate. For example, we want to restrict the previous result to the people living on Main Street and having at least two children. Moreover, we are only interested in the addresses of the children who do not live in the same city as their parents.

Example:

```
select c.address
from Persons p,
p.children c
where p.address.street = "Main Street" and
count(p.children) >= 2 and
c.address.city != p.address.city
```

Boolean Operators

The where clauses of queries contain atomic or complex predicates. Complex predicates are built by combining atomic predicates with boolean operators and, or, and not. OQL supports special versions of and and or, namely, andthen and orelse. These two operators enable conditional evaluation of their second operand and also dictate that the first operand be evaluated first. Let X and Y be boolean expressions in the following two cases:

```
X andthen Y
X orelse Y
```

In the first case, Y is only evaluated if X has already evaluated to true. In the second case, Y is only evaluated if X has already evaluated to false. Note that you cannot introduce one of the operators andthen and orelse, without at the same time introducing the other. This is so, because the one shows up whenever an expression involving the other is negated. For example:

```
not (X1 andthen X2)
```

is equivalent to

```
not X1 orelse not X2
```

The following is an example OQL query that exploits the andthen operator:

```
select p.name
from Persons p
where p.address != nil
andthen p.address.city = Paris
```

It retrieves objects of type Person (or any of its subtypes) that live in Paris. The andthen operator makes sure the predicate on address.city is only evaluated for instances in Persons that have a not nil address.

Join

In the from clause, collections that are not directly related can also be declared. As in SQL, this allows computation of joins between these collections. This example selects the people who bear the name of a flower, assuming there exists a set of all flowers called Flowers.

Example:

```
select p
from Persons p,
 Flowers f
where p.name = f.name
```

Выражения над коллекциями объектов

Universal Quantification

If x is a variable name, e1 and e2 are expressions, e1 denotes a collection, and e2 is an expression of type boolean, then for all x in e1: e2 is an expression of type boolean. It returns true if all the elements of collection e1 satisfy e2; it returns false if any element of e1 does not satisfy e2, and it returns UNDEFINED otherwise.

Example:

for all x in Students: x.student_id > 0

This returns true if all the objects in the Students set have a positive value for their student_id attribute.

Existential Quantification

If x is a variable name, e1 and e2 are expressions, e1 denotes a collection, and e2 is an expression of type boolean, then exists x in e1: e2 is an expression of type boolean. It returns true if there is at least one element of collection e1 that satisfies e2; it returns false if no element of collection e1 satisfies e2; and it returns UNDEFINED otherwise.

Example:

exists x in Doe.takes: x.taught_by.name = "Turing"

This returns true if at least one course Doe takes is taught by someone named Turing. If e is a collection expression, then exists(e) and unique(e) are expressions that return a boolean value. The first one returns true if there exists at least one element in the collection, while the second one returns true if there exists only one element in the collection.

Note that these operators accept the SQL syntax for nested queries like select ... from col where exists (select ... from col1 where predicate)

The nested query returns a bag to which the operator exists is applied. This is of course the task of an optimizer to recognize that it is useless to compute effectively the inter mediate bag result.

Membership Testing

If e1 and e2 are expressions, e2 is a collection, and e1 is an object or a literal having the same type or a subtype as the elements of e2, then e1 in e2 is an expression of type boolean. It returns true if element e1 is not UNDEFINED and belongs to collection e2, it returns false if e1 is not UNDEFINED and does not belong to collection e2, and it returns UNDEFINED if e1 is UNDEFINED.

Example:

Doe in Students
This returns true.
Doe in TA
This returns true if Doe is a teaching assistant.

Aggregate Operators

If e is an expression that denotes a collection, if <op> is an operator from {min, max, count, sum, avg}, then <op>(e) is an expression.

Example:

max (select salary from Professors)

This returns the maximum salary of the professors.

If e is of type collection(t), where t is integer or float, then <op>(e), where <op> is an aggregate operator different from count, is an expression of type t. If any of the elements in e is UNDEFINED, then <op>(e) returns UNDEFINED.

If e is of type collection(t), then count(e) is an expression of type integer. UNDEFINED elements, if any, are counted by operator count.

Example:

count({"Paris", "Palo Alto", UNDEFINED})

This returns 3. [5]

Связывание с объектными языками программирования

C++ Binding

This chapter defines the C++ binding for ODL/OML. ODL stands for Object Definition Language. It is the declarative portion of C++ ODL/OML. The C++ binding of ODL is expressed as a library that provides classes and functions to implement the concepts defined in the ODMG Object Model. OML stands for Object Manipulation Language. It is the language used for retrieving objects from the database and modifying them. The C++ OML syntax and semantics are those of standard C++ in the context of the standard class library.

ODL/OML specifies only the logical characteristics of objects and the operations used to manipulate them. It does not discuss the physical storage of objects. It does not address the clustering or memory management issues associated with the stored physical representation of objects or access structures like indices used to accelerate object retrieval. In an ideal world, these would be transparent to the programmer. In the real world, they are not. An additional set of constructs called physical pragmas is defined to give the programmer some direct control over these issues, or at least to enable a programmer to provide “hints” to the storage management subsystem provided as part of the object data management system (ODMS) runtime. Physical pragmas exist within the ODL and OML. They are added to object type definitions specified in ODL, expressed as OML operations, or shown as optional arguments to operations defined within OML. Because these pragmas are not in any sense a stand-alone language, but rather a set of constructs added to ODL/OML to address implementation issues, they are included within the relevant subsections of this chapter.

Language Design Principles

The programming language–specific bindings for ODL/OML are based on one basic principle: The programmer feels that there is one language, not two separate languages with arbitrary boundaries between them. This principle has two corollaries that are evident in the design of the C++ binding

defined in the body of this chapter:

1. There is a single unified type system across the programming language and the database; individual instances of these common types can be persistent or transient.
2. The programming language–specific binding for ODL/OML respects the syntax and semantics of the base programming language into which it is being inserted.

Language Binding

The C++ binding maps the Object Model into C++ by introducing a set of classes that can have both persistent and transient instances. These classes are informally referred to as “persistence-capable classes” in the body of this chapter. These classes are distinct from the normal classes defined by the C++ language, all of whose instances are transient; that is, they don’t outlive the execution of the process in which they were created. Where it is necessary to distinguish between these two categories of classes, the former are called “persistence-capable classes”; the latter are referred to as “transient classes.”

The C++ to ODMS language binding approach described by this standard is based on the smart pointer or “Ref-based” approach. For each persistence-capable class T, an ancillary class d_Ref<T> is defined. Instances of persistence-capable classes are then referenced using parameterized references, for example,

1. d_Ref<Professor>profP;
2. d_Ref<Department> deptRef;
3. profP−>grant_tenure();
4. deptRef = profP−>dept;

Statement (1) declares the object profP as an instance of the type d_Ref<Professor>. Statement (2) declares deptRef as an instance of the type d_Ref<Department>. Statement (3) invokes the grant_tenure operation defined on class Professor, on the instance of that class referred to by profP. Statement (4) assigns the value of the dept attribute of the professor referenced by profP to the variable deptRef.

Instances of persistence-capable classes may contain embedded members of C++ built-in types, user-defined classes, or pointers to transient data. Applications may refer to such embedded members using C++ pointers (∗) or references (&) only during the execution of a transaction.

In this chapter, we use the following terms to describe the places where the standard is formally considered undefined or allows for an implementor of one of the bindings to make implementation-specific decisions with respect to implementing the standard. The terms are

Undefined: The behavior is unspecified by the standard. Implementations have complete freedom (can do anything or nothing), and the behavior need not be documented by the implementor or vendor.

Implementation-defined: The behavior is specified by each implementor vendor. The implementor/vendor is allowed to make implementation-specific decisions about the behavior. However, the behavior must be well defined and fully documented and published as part of the vendor's implementation of the standard.

Figure 3 shows the hierarchy of languages involved, as well as the preprocess, compile, and link steps that generate an executable application.

Mapping the ODMG Object Model into C++

Although C++ provides a powerful data model that is close to the one presented in previous chapters, it is worth trying to explain more precisely how concepts introduced in before map into concrete C++

constructs.

Object and Literal: An ODMG object type maps into a C++ class. Depending on how a C++ class is instantiated, the result can be an ODMG object or an ODMG literal. A C++ object embedded as a member within an enclosing class is treated as an ODMG literal. This is explained by the fact that a block of memory is inserted into the enclosing object and belongs entirely to it. For instance, you cannot copy the enclosing object without getting a copy of the embedded one at the same time. In this sense, the embedded object cannot be considered as having an identity since it acts as a literal.

Structure: The Object Model notion of a structure maps into the C++ construct struct or class embedded in a class.

Implementation: C++ has implicit the notion of dividing a class definition into two parts: its interface (public part) and its implementation (protected and private members and function definitions). However, in C++ only one implementation is possible for a given class.

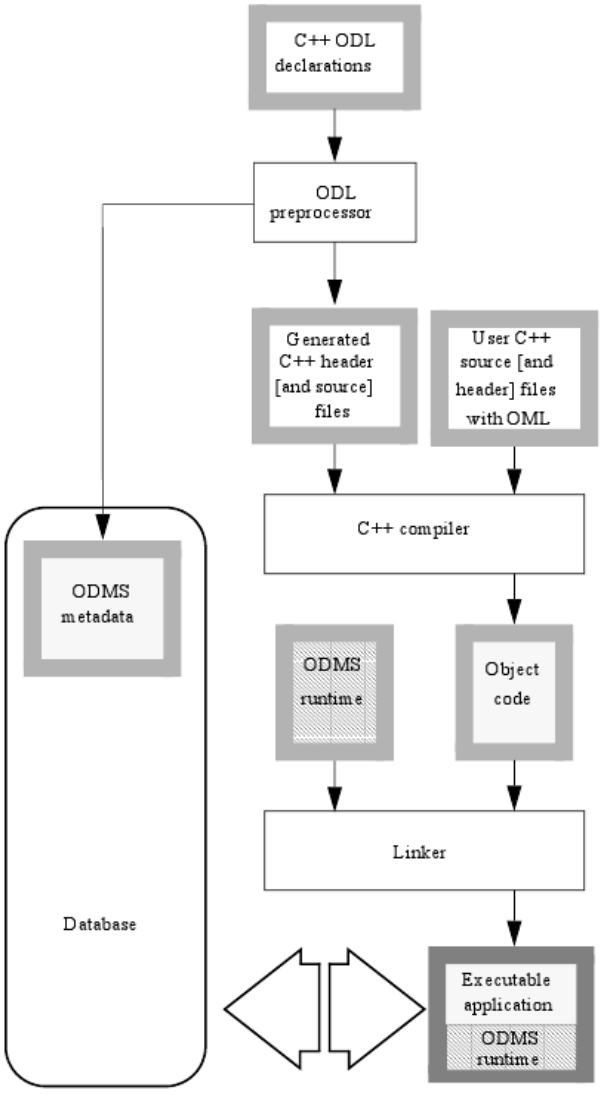

Figure 5-1. Language Hierarchy

Figure 3.

Collection Classes: The ODMG Object Model includes collection type generators, collection types, and collection instances. Collection type generators are represented as template classes in C++. Collection types are represented as collection classes, and collection instances are represented as instances of these collection classes. To illustrate these three categories:

```
template<class T> class d_Set : public d_Collection<T> { ... };
class Ship { ... };
d_Set<d_Ref<Ship> > Cunard_Line;
```

d_Set<T> is a collection template class. d_Set<d_Ref<Ship> > is a collection class. Cunard_Line is a particular collection, an instance of the class d_Set<d_Ref<Ship> >. The subtype-supertype hierarchy of collection types defined in the ODMG Object Model is directly carried over into C++. The type d_Collection<T> is an abstract class in C++ with no direct instances. It is instantiable only through its derived classes. The only differences between the collection classes in the C++ binding and their counterparts in the Object Model are the following:

- Named operations in the Object Model are mapped to C++ function members.
- For some operations, the C++ binding includes both the named function and an overloaded infix operation, for example, d_Set::union_with also has the form operator+=. The statements s1.union_with(s2) and s1 += s2 are functionally equivalent.
- Operations that return a boolean in the Object Model are modeled as function members that return a d_Boolean in the C++ binding.
- The create and delete operations defined in the Object Model have been replaced with C++ constructors and destructors.

Array: C++ provides a syntax for creating and accessing a contiguous and indexable sequence of objects. This has been chosen to map partially to the ODMG array collection. To complement it, a d_Varray C++ class is also provided, which implements an array whose upper bound may vary.

Relationship: Relationships are not directly supported by C++. Instead, they are supported in ODMG by including instances of specific template classes that provide the maintenance of the relationship.

The relation itself is implemented as a reference (one-to-one relation) or as a collection (one-to-many relation) embedded in the object.

Extents: The class d_Extent<T> provides an interface to the extent for a persistence-capable class T in the C++ binding.

Keys: Key declarations are not supported by C++.

Names: An object can have multiple names. The bind operation in the Object Model is implemented in C++ with the set_object_name and rename_object methods to maintain backward compatibility with previous releases of the C++ binding.

Use of C++ Language Features

Prefix

The global names in the ODMG interface will have a prefix of d_. The intention is to avoid name collisions with other names in the global name space. The ODMG will keep the prefix even after C++ name spaces are generally available.

Name Spaces

The name space feature added to C++ did not have generally available implementations at the time this specification was written.

Exception Handling

When error conditions are detected, an instance of class d_Error is thrown using the standard C++ exception mechanism. Class d_Error is derived from the class exception defined in the C++ standard.

Preprocessor Identifier

A preprocessor identifier is defined for conditional compilation. With ODMG 3.0, the following symbol

#define __ ODMG_ _ 30

is defined. The value of this symbol indicates the specific ODMG release, for example, 20 (release 2.0), 21 (release 2.1), or 30 (release 3.0). The preprocessor identifier for the original ODMG-93 release was _ _ODMG_93_ _ .

Implementation Extensions

Implementations must provide the full function signatures for all the interface methods specified in the chapter and may provide variants on these methods, with additional parameters. Each additional parameter must have a default value. This allows applications that do not use the additional parameters to be portable.

Example

This section gives a complete example of a small C++ application. This application manages records about people. A Person may be entered into the database. Then special events can be recorded: marriage, the birth of children, moving to a new address. The application comprises two transactions: The first one populates the database, while the second consults and updates it.

The next section defines the schema of the database, as C++ ODL classes. The C++ program is given in the subsequent section.

Schema Definition

Here is the C++ ODL syntax:

```
// Schema Definition in C++ ODL
class City; // forward declaration
struct Address {
d_UShortnumber;
d_Stringstreet;
d_Ref<City> city;
Address();
Address(d_UShort, const char*, const d_Ref<City> &);
};
extern const char _spouse [ ], _parents [ ], _children [ ];
class Person : public d_Object {
public:
// Attributes (all public, for this example)
d_String name;
Addressaddress;
// Relationships
d_Rel_Ref<Person, _spouse>spouse;
d_Rel_List<Person, _parents>children;
d_Rel_List<Person, _children> parents;
// Operations
 Person(const char * pname);
void birth(const d_Ref<Person> &child); // a child is born
void marriage(const d_Ref<Person> &to_whom);
```

```
d_Ref<d_Set<d_Ref<Person> > >ancestors() const; // returns ancestors
void move(const Address &); // move to a new address
// Extent
static d_Ref<d_Set<d_Ref<Person> > >people; // a reference to class extent1
static const char * const extent_name;
};
class City : public d_Object {
public:
// Attributes
d_ULong city_code;
d_Stringname;
d_Ref<d_Set<d_Ref<Person> > > population; // the people living in this city
// Operations
City(int, const char*);
// Extension
static d_Ref<d_Set<d_Ref<City> > >cities;// a reference to the class extent
static const char * const extent_name;
};
```

Schema Implementation

We now define the code of the operations declared in the schema. This is written in plain C++. We assume the C++ ODL preprocessor has generated a file, “schema.hxx”, which contains the standard C++ definitions equivalent to the C++ ODL classes.

```
// Classes Implementation in C++
#include "schema.hxx"
const char _spouse [ ] = "spouse";
const char _parents [ ] = "parents";
const char _children [ ] = "children";
// Address structure:
Address::Address(d_UShort pnum, const char* pstreet,
const d_Ref<City> &pcity)
 : number(pnumber),
street(pstreet),
city(pcity)
 {}
Address::Address()
 : number(0),
street(0),
city(0)
 {}
// Person Class:
const char * const Person::extent_name = "people";
Person::Person(const char * pname)
 : name(pname)
 {
people->insert_element(this); // Put this person in the extension
}
void Person::birth(const d_Ref<Person> &child)
 { // Adds a new child to the children list
children.insert_element_last(child);
if(spouse)
 spouse->children.insert_element_last(child);
}
void Person::marriage(const d_Ref<Person> &to_whom)
 { // Initializes the spouse relationship
spouse = with; // with->spouse is automatically set to this person
}
d_Ref<d_Set<d_Ref<Person> > > Person::ancestors()
 {// Constructs the set of all ancestors of this person
d_Ref<d_Set<d_Ref<Person> > > the_ancestors =
 new d_Set<d_Ref<Person> >;
int i;
for( i = 0; i < 2; i++)
 if( parents[i] ) {
```

```
 // The ancestors = parents union ancestors(parents)
 the_ancestors->insert_element(parents[i]);
 d_Ref<d_Set<d_Ref<Person> > > grand_parents= parents[i]->ancestors();
 the_ancestors->union_with(*grand_parents);
 grand_parents.delete_object();
 }
return the_ancestors;
}
void Person::move(const Address &new_address)
 { // Updates the address attribute of this person
if(address.city)
 address.city->population->remove_element(this);
new_address.city->population->insert_element(this);
mark_modified();
 address = new_address;
 }
 // City class:
 const char * const City::extent_name = "cities";
 City::City(d_ULong code, const char * cname)
: city_code(code),
 name(cname)
{
 cities->insert_element(this);// Put this city into the extension
 }
```

An Application

We now have the whole schema well defined and implemented. We are able to populate the database and play with it. In the following application, the transaction Load builds some objects into the database. Then the transaction Consult reads it, prints some reports from it, and makes updates. Each transaction is implemented inside a C++ function.

The database is opened by the main program, which then starts the transactions.

```
 #include <iostream.h>
 #include "schema.hxx"
 static d_Database dbobj;
 static d_Database * database = &dbobj;
 void Load()
{// Transaction that populates the database
 d_Transaction load;
 load.begin();
 // Create both persons and cities extensions, and name them.
 Person::people = new(database) d_Set<d_Ref<Person> >;
 City::cities = new(database) d_Set<d_Ref<City> >;
 database->set_object_name(Person::people, Person::extent_name);
database->set_object_name(City::cities, City::extent_name);
// Construct 3 persistent objects from class Person.
d_Ref<Person> God, Adam, Eve;
God = new(database, "Person") Person("God");
Adam = new(database, "Person") Person("Adam");
Eve = new(database, "Person") Person("Eve");
// Construct an Address structure, Paradise, as (7 Apple Street, Garden),
// and set the address attributes of Adam and Eve.
Address Paradise(7, "Apple", new(database, "City") City(0, "Garden"));
Adam->move(Paradise);
Eve->move(Paradise);
// Define the family relationships
God->birth(Adam);
Adam->marriage(Eve);
Adam->birth(new(database, "Person") Person("Cain"));
Adam->birth(new(database, "Person") Person("Abel"));
load.commit();// Commit transaction, putting objects into the database
}
static void print_persons(const d_Collection<d_Ref<Person> >& s)
 { // A service function to print a collection of persons
```

```
d_Ref<Person> p;
d_Iterator<d_Ref<Person> > it = s.create_iterator();
while( it.next(p) ) {
cout << "--- " << p->name << " lives in ";
if (p->address.city)
cout << p->address.city->name;
else
cout << "Unknown";
cout << endl;
}
}
void Consult()
 { // Transaction that consults and updates the database
 d_Transaction consult;
 d_List<d_Ref<Person> >list;
 d_Bag<d_Ref<Person>>bag;
 consult.begin();
 // Static references to objects or collections must be recomputed
 // after a commit
 Person::people = database->lookup_object(Person::extent_name);
 City::cities = database->lookup_object(City::extent_name);
 // Now begin the transaction
 cout << "All the people ....:" << endl;
 print_persons(*Person::people);
 cout << "All the people sorted by name ....:" << endl;
 d_oql_execute("select p from people order by name", list);
 print_persons(list);
 cout << "People having 2 children and living in Paradise ...:" << endl;
 d_oql_execute(list, "select p from p in people\
 where p.address.city.name = \"Garden\"\
 and count(p.children) = 2", bag);
 print_persons(bag);
 // Adam and Eve are moving ...
 Address Earth(13, "Macadam", new(database, "City") City(1, "St-Croix"));
 d_Ref<Person> Adam;
 d_oql_execute("element(select p from p in people\
 where p.name = \"Adam\")", Adam);
 Adam->move(Earth);
 Adam->spouse->move(Earth);
 cout << "Cain's ancestors ...:" << endl;
 d_Ref<Person> Cain = Adam->children.retrieve_element_at(0);
 print_persons(*(Cain->ancestors()));
 consult.commit();
}
main()
 {
 database->open("family");
 Load();
 Consult();
 database->close();
}
```

Smalltalk Binding

This chapter defines the Smalltalk binding for the ODMG Object Model, ODL, and OQL. While no Smalltalk language standard exists at this time, ODMG member organizations participate in the X3J20 INCITS Smalltalk standards committee. In the interests of consistency and until an official Smalltalk standard exists, we will map many ODL concepts to class descriptions as specified by Smalltalk80.

Language Design Principles

The ODMG Smalltalk binding is based upon two principles: It should bind to Smalltalk in a natural way that is consistent with the principles of the language, and it should support language interoperability consistent with ODL specification and semantics.

- There is a unified type system that is shared by Smalltalk and the ODMS. This type system is ODL as mapped into Smalltalk by the Smalltalk binding.
- The binding respects the Smalltalk syntax, meaning the Smalltalk language will not have to be modified to accommodate this binding. ODL concepts will be represented using normal Smalltalk coding conventions.
- The binding respects the fact that Smalltalk is dynamically typed. Arbitrary Smalltalk objects may be stored persistently, including ODL-specified objects that will obey the ODL typing semantics.
- The binding respects the dynamic memory management semantics of Smalltalk. Objects will become persistent when they are referenced by other persistent objects in the database and will be removed when they are no longer reachable in this manner.

Language Binding

The ODMG binding for Smalltalk is based upon the OMG Smalltalk IDL binding.1 As ODL is a superset of IDL, the IDL binding defines a large part of the mapping required by this document. This chapter provides informal descriptions of the IDL binding topics and more formally defines the Smalltalk binding for the ODL extensions, including relationships, literals, and collections. The ODMG Smalltalk binding can be automated by an ODL compiler that processes ODL declarations and generates a graph of meta objects, which model the schema of the database. These meta objects provide the type information that allows the Smalltalk binding to support the required ODL type semantics. The complete set of such meta objects defines the entire schema of the database and would serve much in the same capacity as an OMG Interface Repository.

In such a repository, the meta objects that represent the schema of the database may be programmatically accessed and modified by Smalltalk applications, through their standard interfaces. One such application, a binding generator, may be used to generate Smalltalk class and method skeletons from the meta objects. This binding generator would resolve the type-class mapping choices that are inherent in the ODMG Smalltalk binding.

The information in the meta objects is also sufficient to regenerate the ODL declarations for the portions of the schema that they represent. The relationships between these components are illustrated in Figure 4. A conforming implementation must support the Smalltalk output of this binding process; it need not provide automated tools.

Mapping the ODMG Object Model into Smalltalk

Although Smalltalk provides a powerful data model that is close to the one presented in previous chapters, it remains necessary to precisely describe how the concepts of the ODMG Object Model map into concrete Smalltalk constructions.

Object and Literal

An ODMG object type maps into a Smalltalk class. Since Smalltalk has no distinct notion of literal objects, both ODMG objects and ODMG literals may be implemented by the same Smalltalk classes.

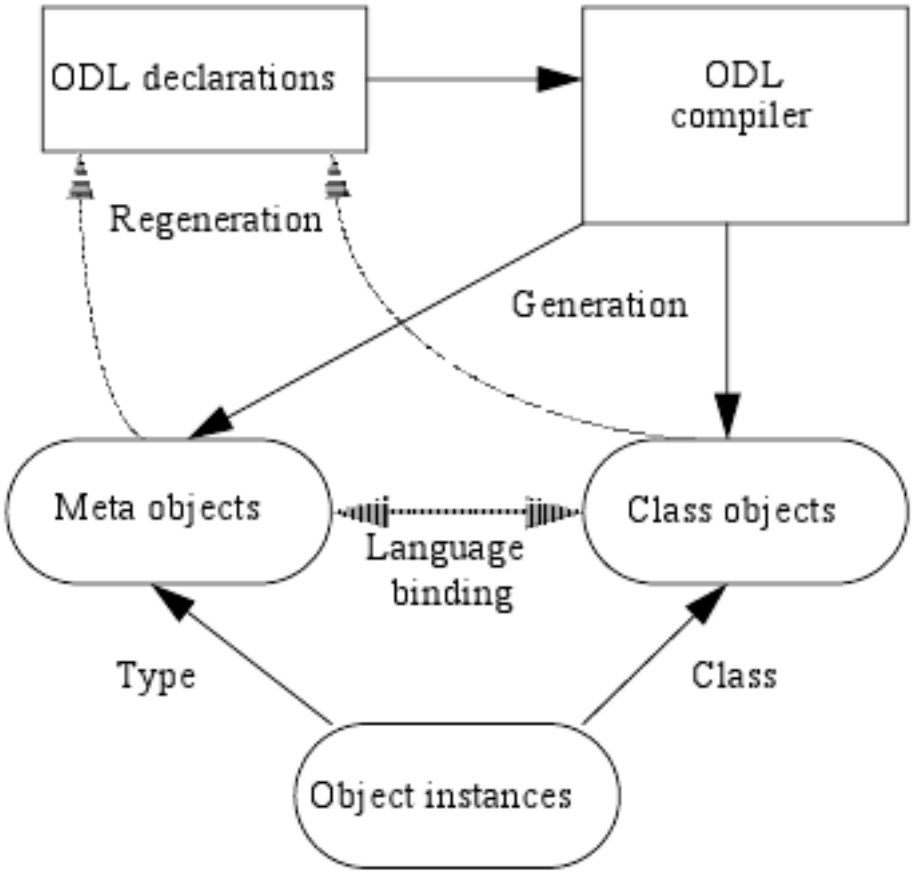

Figure 6-1 . Smalltalk Language Binding

Figure 4.

Relationship

This concept is not directly supported by Smalltalk and must be implemented by Smalltalk methods that support a standard protocol. The relationship itself is typically implemented either as an object reference (one-to-one relation) or as an appropriate Collection subclass (one-to-many, many-to-many relations) embedded as an instance variable of the object. Rules for defining sets of accessor methods are presented that allow all relationships to be managed uniformly.

Names

Objects in Smalltalk have a unique object identity, and references to objects may appear in a variety of naming contexts. The Smalltalk system dictionary contains globally accessible objects that are indexed by symbols that name them. A similar protocol has been defined on the Database class for managing named persistent objects that exist within the database.

Extents

Extents are not supported by this binding. Instead, users may use the database naming protocol to explicitly register and access named Collections.

Keys

Key declarations are not supported by this binding. Instead, users may use the database naming protocol to explicitly register and access named Dictionaries.

Implementation

Everything in Smalltalk is implemented as an object. Objects in Smalltalk have instance variables that are private to the implementations of their methods. An instance variable refers to a single Smalltalk object, the class of which is available at runtime through the class method. This instance object may itself refer to other objects.

Collections

Smalltalk provides a rich set of Collection subclasses, including Set, Bag, List, Dictionary, and Array classes. Where possible, this binding has chosen to use existing methods to implement the ODMG Collection interfaces. Unlike statically typed languages, Smalltalk collections may contain heterogeneous elements whose type is only known at runtime. Implementations utilizing these collections must be able to enforce the homogeneous type constraints of ODL.

Database Administration

Databases are represented by instances of Database objects in this binding, and a protocol is defined for creating databases and for connecting to them. Some operations regarding database administration are not addressed by this binding and represent opportunities for future work.

Java Binding

This chapter defines the binding between the ODMG Object Model (ODL and OML) and the Java programming language as defined by the Java™ 2 Platform. It is designed to be compatible with the OMG Persistence Service.

Language Design Principles

The ODMG Java binding is based on one fundamental principle: The programmer should perceive the binding as a single language for expressing both database and programming operations, not two separate languages with arbitrary boundaries between them. This principle has several corollaries evident throughout the definition of the Java binding in the body of this chapter:

- There is a single unified type system shared by the Java language and the database; individual instances of these common types can be persistent or transient.
- The binding respects the Java language syntax, meaning that the Java language will not have to be modified to accommodate this binding.
- The binding respects the automatic storage management semantics of Java.

Objects will become persistent when they are referenced by other persistent objects in the database. Additionally, database storage may be explicitly managed by the application program.

Note that the Java binding provides persistence by reachability, like the ODMG Smalltalk binding (this has also been called transitive persistence). On database commit, all objects reachable from database root objects are stored in the database.

Language Binding

The Java binding provides two ways to declare persistence-capable Java classes:

- Existing Java classes can be made persistence-capable.
- Java class declarations (as well as a database schema) may automatically be generated by a preprocessor for ODMG ODL.

One possible ODMG implementation that supports these capabilities would be a postprocessor that

takes as input the Java .class file (bytecodes) produced by the Java compiler and produces new modified bytecodes that support persistence. Another implementation would be a preprocessor that modifies Java source before it goes to the Java compiler. Another implementation would be a modified Java interpreter.

We want a binding that allows all of these possible implementations. Because Java does not have all hooks we might desire, and the Java binding must use standard Java syntax, it is necessary to distinguish special classes understood by the database system.

These classes are called persistence-capable classes. They can have both persistent and transient instances. Only instances of these classes can be made persistent. Because a Java class definition does not contain all the object modeling information required, it is necessary to augment the class definition with a property file. A class is persistence-capable if the class name or its package name is specified in the property file with the key-value persistent=capable.

Use of Java Language Features

Name Spaces and Interfaces

The ODMG Java API is defined in the package org.odmg. The entire API consists of interfaces, rather than classes, so that it can be shared without change by all vendors.

In order to bootstrap the implementation, the ODMG vendor needs to provide an Implementation object that includes factories for ODMG implementation classes.1

This approach permits more than one ODMG implementation in the same JVM.

However, we require that all instances of each persistence-capable class belong to the same implementation to allow for an efficient and practical implementation.

Implementation Bootstrap Object

The only vendor-dependent line of code required in an ODMG application is the one that retrieves an ODMG implementation object from the vendor. The implementation

```
object implements the org.odmg.Implementation interface:
 public interface Implementation {
 public Transaction newTransaction(); // Create transaction object and
// associate it with the current thread
 public Transaction currentTransaction(); // Get current transaction for thread,
// or null if none
 public Database newDatabase(); // Create database object
 public OQLQuery newOQLQuery(); // Create query object
 public DListnewDList();// Factories for Collections
 public DBag newDBag();
 public DSet newDSet();
 public DArray newDArray();
 public DMap newDMap();
 public String getObjectId(Object obj); // Get a string representation of
// the object’s identifier
 public Database getDatabase(Object obj); // Get database of an object
 }
```

Implementation Extensions

Implementations must provide the full function signatures for all the interface methods specified in the chapter, but may also provide variants on these methods with different types or additional parameters.

Mapping the ODMG Object Model into Java

The Java language provides a comprehensive object model comparable to the one presented in Chapter 2. This section describes the mapping between the two models and the extensions provided by the Java

binding. The following features are not yet supported by the Java binding: relationships, extents, keys, and access to the metaschema.

Object and Literal: An ODMG object type maps into a Java object type. The ODMG atomic literal types map into their equivalent Java primitive types. There are no structured literal types in the Java binding.

Structure: The Object Model definition of a structure maps into a Java class.

Implementation: The Java language supports the independent definition of interface from implementation. Interfaces and abstract classes cannot be instantiated and therefore are not persistence-capable.

Collection Interfaces: The collection objects described in Section 2.3.6 specify collection behavior, which may be implemented using many different collection representations such as hash tables, trees, chained lists, and so on. The Java binding provides the following interfaces and at least one implementation for each of these collection objects:

```
public interface DCollectionextends java.util.Collection { ... }
public interface DSet extends DCollection, java.util.Set { ... }
public interface DBag extends DCollection { ... }
public interface DListextends DCollection, java.util.List { ... }
public interface DArray extends DCollection, java.util.List { ... }
public interface DMap extends java.util.Map { ... }
```

Array: Java provides a syntax for creating and accessing a contiguous and indexable sequence of objects, and a separate class, Vector, for extensible sequences. The ODMG Array collection maps into either the primitive array type, the Java Vector class, or the ODMG DArray interface, depending on the desired level of capability.

Relationship: The ODMG Java binding supports binary relationships. Two persistence-capable classes are involved in the definition of a relationship. The class fields representing the roles of the relationship are referred to as traversal paths.

The cardinality of a traversal path might be either one or many, identified by being just a single reference or by being a collection type field. In the latter case, the element type of the collection is specified in the property file (class name follows the keyword refersTo). Beyond DCollection the interfaces DBag, DSet, DList, and DArray are also valid attribute types for traversal paths of a many cardinality.

The traversal paths and cardinality of a relationship are declared within the Java classes (normal class fields), while the inverse traversal path and the element type are defined within a property file (see Section 7.5).

Extents: Extents are not yet supported by the Java binding. The programmer is responsible for defining a collection to serve as an extent and writing methods to maintain it.

Keys: Key declarations are not yet supported by the Java binding.

Names: Objects may be named using methods of the Database interface defined in the Java OML. The root objects of a database are the named objects; root objects and any objects reachable from them are persistent.

Exception Handling: When an error condition is detected, an exception is thrown using the standard Java exception mechanism. The following standard exception types are defined; some are thrown from specific ODMG interfaces and are thus subclasses of ODMGException, others may be thrown in the course of using persistent objects and are thus subclasses of ODMGRuntimeException, and others are related to query processing and are just subclasses of QueryException, which in turn is a subclass of

ODMGException. [5]

Примеры СУБД, следующих стандарту ODMG

ODBMSs are being used in a wide variety of applications where they provide the best solution. They are used in embedded or real-time applications where their performance or rich data model are needed. They are used in web servers for caching or as an alternative to O/R mapping, using JDO or JPA with extensions, where there is no existing SQL database or where an ODBMS's in-memory performance is needed for some of the data. They are used in engineering, scientific, or business applications where an ODBMS's speed at reference-following is needed, e.g., expanding a design's subcomponents or a large bill of materials.

As the ODBMSs have matured, they have become another "off the shelf" data management option for application developers, along with traditional RDBMSs, text search engines, O/R mapping, indexed files, and other tools.

OODBMS are being used across the board in most vertical markets. They end up in the more difficult applications where models are increasingly complex with deep graphs, many to many and recursive relationships. There are certain areas where you find more of these types of applications than others i.e. Telecommunications and Defense. However, really they are showing up everywhere for scalability, ease of use and performance reasons. In Telecom our customers use us in every aspect of business including: network management, operational support systems, billing, softswitch, content management, provisioning, and more. For Defense, it is also a broad spectrum including: simulation, battle management, intelligence analytics, communications, target tracking, planning, etc.

Some organizations use ODBMS in Finance for risk management, online trading, batch clearing, ticker systems, transportation for planning, rail management, airline/car/hotel reservations, communications, logistics and more. Further, some organizations use an ODBMS in areas such as rich media content management, work force tracking, home improvement planning, legal document processing, SCADA systems, online gaming, website store fronts, embedded in medical devices, on mobile phones, just so many areas all over the map. So, is the domain of OODB use changing, yes, it is growing in all visible directions.

The main markets appear to be: Defense and the Intelligence Community; telecom, medical and process control equipment; scientific research; online communities; front end caching of complex IT data; advanced financial systems and manufacturing. Each manufacturer's web site gives concrete examples of applications, ranging from military reconnaissance and targeting systems to space debris tracking and operating theater applications. There seems to have been a shift from engineering applications to complex IT and real-time systems over the past twenty years.

There are applications that benefit so much from the use of ODBMS that they are the only wise choice. This is when either the object domain model is so complex that a mapping to a relational databases is unaffordable or when the performance requirements to store objects are so high that a relational database can't keep up with the speed.

For almost any application ODBMS can be used to improve development speed by avoiding O-R mapping.

Let me give you two examples of db4o use:

- Indra Sistemas caches real-time data of train systems in db4o. They need the performance to store objects.
- Ricoh uses db4o for it's next copier platform. They have low-resource constraints on their

machines.[13]

Перспективы развития стандарта объектных систем баз данных в OMG

In February 2006, the Object Management Group (OMG) in Needham, MA, has decided to develop the "4th generation" standard for object databases in order to facilitate broader adoption of standards-based object database technology. To this end, the OMG set up the Object Database Technology Working Group (ODBT WG) and acquired the rights to develop new OMG specifications based on the works of the disbanded Object Data Management Group (ODMG), which issued the last ODMG 3.0 standard in 2001.

The ODBT WG was formed to advance the standards for object databases and, through better standards, encourage broader adoption of object database technology. Chaired by Michael P. Card from Syracuse Research Corporation and Char Wales from Mitre Corporation, members of the ODBT WG include users and vendors affiliated with companies such as Boeing, RTI, Objectivity, Promia, Raytheon, Sparx Systems, and db4objects. The Object Database Technology Working Group (ODBT WG) is part of the OMG's Middleware and Related Services (MARS) Platform Task Force (PTF). [10]

July 27, 2009 -- took place a discussion at the ICOODB 09 conference on July 2, 2009 in Zurich. There are some questions discussed there.

Current market for ODBMS

The object database market has continued to grow and mature. There has been some consolidation, which is to be expected given the number of vendors that were trying to share the market, but as a result all of the vendors are now profitable; it seems that most markets can support only 3 major vendors.

The market has been quite persistent. I (Rick Cattell) was surprised myself to see significant revenue figures for the ODBMS market after years of nay-sayers who compared RDBMS and ODBMS revenues and extrapolated impending doom for the latter. But comparing ODBMSs to RDBMSs is like comparing the market for pickup trucks to the market for passenger cars; they are for different purposes and markets. It is now generally accepted that "one size does not fit all". ODBMSs are being used in applications where RDBMSs simply don't work.

Most innovative features (if any) added/ or improved to/for ODBMS in the last years

The recent introduction of an open source ODBMS (db4o) is a significant development: a much

larger audience is now considering an ODBMS option as a result.

Scalability and performance are the core value areas of the OODB. So, we have had tremendous focus on the internals of Versant in order to deliver performance and scalability on N-Core processors. It is hard to describe all of the work in detail,but it can best be described as introducing parallel algorithmic processing in the core kernel. Additionally, we made improvements in areas such as online reorganization, which is an area known to bring most relational databases offline in order to address. The architecture of Versant is uniquely suited to move objects around physically so that slow performance due to fragmentation is completely eliminated. Another of the most improved areas have been with the query execution engine. There was a lot of work to add advanced index types and provide set based query operators, aggregations, ordered results, etc.

These improvements make it easier to support existing tooling and provide a means of relational

minded personnel to get information from the database.

As an extension of the query work, innovative additions have been in the area of .NET and support for LINQ as a native query language. Architecturally, we keep moving closer to the cluster database concept where the application is completely agnostic to the fact that the database is physically distributed over potentially 1000's of machines and the configuration can be dynamically changed while applications are online.

Several features were added to ObjectStore to enable better support for our customers who use it as a cache for relational data. ObjectStore was integrated with DataXtend Semantic Integrator (DXSI), one of the products in the Progress portfolio.

DXSI provides a GUI-based OR mapping tool. The integration is useful because it enables and facilitates the caching of multiple heterogeneous relational databases.

Standards: Why have standardization activities for ODBMSs not progressed?
ODBMS Java binding and ODBMS query language standardization have continued to progress through JDO, JPA, and LINQ. -As far as the rest of the ODMG standard is concerned (the object model and the Smalltalk and C++ bindings), I believe standardization has not progressed further because the vendors believe we are now adequately "done," as far as is a priority at this time.

Standartization activities don't progress because it does not make good business sense to the vendors. Further, adopting standardization now would not be following the natural process. The process is:

innovation, adoption, consolidation, standardization. This is how it has happened in every industry since the industrial revolution. While there has been limited consolidation out of necessity, there has not been sufficient adoption and therefore no business justification for standardization (i.e. it will not help the vendors be more successful to have a standard for such a limited number of users). Unlike innovation, standardization itself does not lead to adoption.

Of course, even in the RDBMs space, there is no true standardization. You cannot unplug an application using Oracle and expect it to work with Sybase or MySQL. Probably more importantly, object persistence has moved past the ODBMS space into the RDBMS space via ORM and manifested itself as a new problem space know as Persistent Object Lifecycle Management. I think this is where Craig Russell with JDO has revolutionized the persistence field and should be recognized for his contribution over time. It was due to efforts such as JDO, to examine cross cutting concerns of object persistence, independent of the storage mechanism, which has lead to a broader understanding of transactional requirements of persistent objects. If the OODBMS vendors can ever take advantage of a standardization initiative, it should be to get onboard with what is happening in this area of standardization which is also directly applicable to our space.

There isn't a large user community demanding standards beyond the ones that exist. Most of the ODBMSs also support SQL/ODBC too. However, we did make progress last year by committing to support for LINQ.

Progress invested in providing a JDO API for ObjectStore. Progress Software were one of the first vendors to support this standard. Based on our experience, however, we have not really seen a strong adoption of this standard in the market place. More recently, there have been several other emerging API standards and frameworks including JSR-220, JPA, and Hibernate. These still appear to be in a state of flux. We continue to weigh in on which standards to invest in that would provide the best returns.[13]

Литература:

1) **The Object-Oriented Database System Manifesto, Malcolm Atkinson University of Glasgow, Francois Bancilhon Altair, David DeWitt University of Wisconsin, Klaus Dittrich University of Zurich, David Maier Oregon Graduate Center, Stanley Zdonik Brown University, Proceedings of the First Deductive and Object-oriented Databases (DOOD) Conference, 1989, http://www.kybele.etsii.urjc.es/docencia/BD/extra/oo-manifesto.pdf**
2) **Rick Cattell Object Database Tutorial, International Conference on Object-Oriented Databases, Zurich, 2009, http://synthesis.ipi.ac.ru/synthesis/student/oodb/essayRef/TutorialCattell**

3) **OODB Tutorial, by ETH, 2008http://synthesis.ipi.ac.ru/synthesis/student/oodb/essayRef/OODBtutorial**
4) **Object-oriented database languages, G.Ullman http://synthesis.ipi.ac.ru/synthesis/student/oodb/essayRef/UllmanAdditionalODLOQL**
5) **The Object Data Management Standard: ODMG 3.0 Edited by R. G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez,Morgan Kaufmann, 2000, ISBN 1-55860-647-5, 260 pages http://synthesis.ipi.ac.ru/synthesis/student/oodb/essayRef/odmg30**
6) **Object Database Management Systems Portal http://www.odbms.org/odmg/**
7) **Introduction to ODBMS http://www.odbms.org/Introduction/**
8) **Free Downloads and Links http://www.odbms.org/downloads.aspx**
9) **OODB: Manifestos, Foundations, Standards + ODMG + SQL1999+2003 Pages: 55 – 92, http://synthesis.ipi.ac.ru/synthesis/student/oodb/objretut.ps**
10) **Next Generation Object Standard http://www.odbms.org/odmg/ng.aspx**
11) **Object Database Technology, Request For Information, OMG, June, 2006http://synthesis.ipi.ac.ru/synthesis/student/oodb/essayRef/ODTrfi**
12) **ODBMS Portal, http://www.odbms.org/**
13) **A New Renaissance for ODBMSs?, Panel, July 2009, http://www.odbms.org/Download/Panel.Renaissance.ODBMS.PDF**