Chapter 6

Smalltalk Binding

6.1 Introduction

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 ANSI Smalltalk standards committee. We expect that as standards are agreed upon by that committee and commercial implementations become available that the ODMG Smalltalk binding will evolve to accommodate them. 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.

6.1.1 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. We believe that organizations who specify their objects in ODL will insist that the Smalltalk binding honor those specifications. These principles have several implications that are evident in the design of the binding described in the body of this chapter.

1. There is a unified type system that is shared by Smalltalk and the ODBMS. This type system is ODL as mapped into Smalltalk by the Smalltalk binding.
2. 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.
3. 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.
4. 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.

6.1.2 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. This chapter includes a Smalltalk binding for the Meta Object interfaces defined in Chapter 2.

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 6-1. A conforming implementation must support the Smalltalk output of this binding process; it need not provide automated tools.

6.1.3 Mapping the ODMG Object Model into Smalltalk

Although Smalltalk provides a powerful data model that is close to the one presented in Chapter 2, it remains necessary to precisely describe how the concepts of the ODMG Object Model map into concrete Smalltalk constructions.

6.1.3.1 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.

---

1. OMG Document 94-11-8, November 16, 1994.

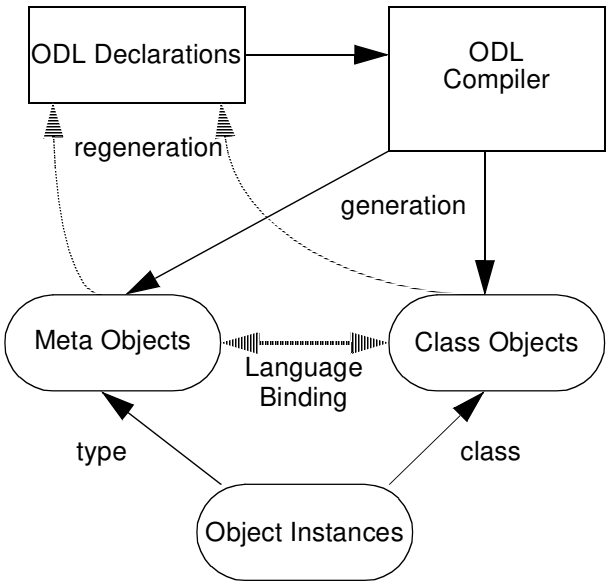

Figure 6-1. Smalltalk Language Binding

6.1.3.2 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.

6.1.3.3 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 which name them. A similar protocol has been defined on the Database class for managing named persistent objects which exist within the database.

6.1.3.4 Extents

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

6.1.3.5 Keys

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

6.1.3.6 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.

6.1.3.7 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.

6.1.3.8 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.

6.2 Smalltalk ODL

6.2.1 OMG IDL Binding Overview

Since the Smalltalk/ODL binding is based upon the OMG Smalltalk/IDL binding, we include here some descriptions of the important aspects of the IDL binding that are needed in order to better understand the ODL binding that follows. These descriptions are not intended to be definitions of these aspects, however, and the reader should consult the OMG binding document directly for the actual definitions.

6.2.1.1 Identifiers

IDL allows the use of underscore characters in its identifiers. Since underscore characters are not allowed in all Smalltalk implementations, the Smalltalk/IDL binding provides a conversion algorithm. To convert an IDL identifier with underscores into a Smalltalk identifier, remove the underscore and capitalize the following letter (if it exists):

```
month_of_year
```

in IDL, becomes in Smalltalk:

```
monthOfYear
```

6.2.1.2 Interfaces

Interfaces define sets of operations that an instance that supports that interface must possess. As such, interfaces correspond to Smalltalk protocols. Implementors are free to map interfaces to classes as required to specify the operations that are supported by a Smalltalk object. In the IDL binding, all objects that have an IDL definition must implement a CORBAName method that returns the fully scoped name of an interface that defines all of its IDL behavior.

```
anObject CORBAName
```

6.2.1.3 Objects

Any Smalltalk object that has an associated IDL definition (by its CORBAName method) may be a CORBA object. In addition, many Smalltalk objects may also represent instances of IDL types as defined below.

6.2.1.4 Operations

IDL operations allow zero or more ***in*** parameters and may also return a functional result. Unlike Smalltalk, IDL operations also allow ***out*** and ***inout*** parameters to be defined, which allow more than a single result to be communicated back to the caller of the method. In the Smalltalk/IDL binding, holders for these output parameters are passed explicitly by the caller in the form of objects that support the CORBAParameter protocol (value, value:).

IDL operation signatures also differ in syntax from that of Smalltalk selectors, and the IDL binding specifies a mapping rule for composing default selectors from the operation and parameter names of the IDL definition. To produce the default Smalltalk operation selector, begin with the operation name. If the operation has only one parameter, append a colon. If the operation has more than one parameter, append a colon and then append each of the second to last parameter names, each followed by colon. The binding allows default selectors to be explicitly overridden, allowing flexibility in method naming.

```
current();
days_in_year(in ushort year);
from_hmstz(in ushort hour,
    ushort minute,
    in float second,
    in short tz_hour,
    in short tz_minute);
```

in IDL, become in Smalltalk:

```
current
daysInYear:
fromHmstz:minute:second:tzHour:tzMinute:
```

6.2.1.5 Constants

Constants, Exceptions, and Enums that are defined in IDL are made available to the Smalltalk programmer in a global dictionary CORBAConstants, which is indexed by the fully qualified scoped name of the IDL entity.

```
const Time_Zone USpacific = -8;
```

would be accessed by the Smalltalk:

```
(CORBAConstants at: #'::Time::USpacific)
```

6.2.1.6 Types

Since, in Smalltalk, everything is an object, there is no separation of objects and datatypes as exist in other hybrid languages such as C++. Thus it is necessary for some Smalltalk objects to fill dual roles in the binding. Since some objects in Smalltalk are more natural in this role than others, we will describe the simple type mappings first.

Simple Types

IDL allows several basic datatypes that are similar to literal valued objects in Smalltalk. While exact type-class mappings are not specified in the IDL binding for technical reasons, the following mappings comply:

- short, unsigned short, long, unsigned long — An appropriate Integer subclass (SmallInteger, LargePositiveInteger, LargeNegativeInteger, depending upon the value)
- float, double — Float and Double, respectively
- char — Character
- boolean — The Boolean values **true** and **false**
- octet — SmallInteger
- string — An appropriate String subclass
- any — Object (any class that supports CORBAName)

Compound Types

IDL has a number of data structuring mechanisms that have a less intuitive mapping to Smalltalk. The list below describes the *implicit* bindings for these types. Implementors are also free to provide *explicit* bindings for these types that allow other Smalltalk objects to be used in these roles. These explicit bindings are especially important in the ODL binding since the various Collections have an object-literal duality that is not present in IDL (e.g., ODL list sequences also have a List interface).

- Array — An appropriate Array subclass
- Sequence — An appropriate OrderedCollection subclass
- Structure — Implicit: A Dictionary containing the fields of the structure keyed by the structure fields as Symbols
- Structure — Explicit: A class supporting accessor methods to get and set of the structure fields
- Union — Implicit: Object (any class that supports CORBAName)
- Union — Explicit: A class that supports the CORBAUnion protocol (discriminator, discriminator:, value, and value: methods)
- Enum — A class that supports the CORBAEnum protocol (=, <, > methods). Implementations must ensure that the correct ordering is maintained and that instances of different enumeration types cannot be compared.

Binding Examples

```
union Number switch(boolean) {
   case TRUE:                long integerValue;
   case FALSE:               float realValue;
   };
struct Point{Number x; Number y;};
```

The implicit bindings for the above would allow a Point to be represented by a Dictionary instance containing the keys #x and #y and values that are instances of Integer or Float:

```
aPoint := Dictionary with: #x -> 452 with: #y -> 687.44
```

Alternatively, the binding allows the Smalltalk class Point to represent the struct Point{} because it implements the selectors x, x:, y, and y:.

```
enum Weekday{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday};
```

the Smalltalk values for Weekday enumerations would be provided by the implementation and accessed from the CORBAConstants global dictionary, as in:

```
(CORBAConstants at: #'::Date::Wednesday) >
   (CORBAConstants at: #'::Date::Tuesday)
```

6.2.1.7 Exceptions

IDL exceptions are defined within modules and interfaces, and are referenced by the operation signatures that raise them. Each exception may define a set of alternative results that are returned to the caller should the exception be raised by the operation. These values are similar to structures, and Dictionaries are used to represent exception values.

```
exception InvalidDate{};
```

would be raised by the following Smalltalk:

```
(CORBAConstants at: #'::DateFactory::InvalidDate')
   CORBARaiseWith: Dictionary new
```

6.2.2 Smalltalk ODL Binding Extensions

This section describes the binding of ODMG ODL to Smalltalk. ODL provides a description of the database schema as a set of interfaces, including their attributes, relationships, and operations. Smalltalk implementations consist of a set of object classes and their instances. The language binding provides a mapping between these domains.

6.2.2.1 Interfaces and Classes

In ODL, interfaces are used to represent the *abstract behavior* of an object and classes are used to model the *abstract state* of objects. Both types may be implemented by Smalltalk classes. In order to maintain the independence of ODMG ODL and OMG IDL type bindings, all uses of the method CORBAName in the IDL binding are replaced by the method ODLName in this definition. This method will return the name of the ODL interface or class that is bound to the object in the schema repository.

```
aDate ODLName
```

returns the string '::Date', which is the name of its ODL interface. Similarly, all uses of the CORBAConstants dictionary for constants, enums, and exceptions will be replaced by a global dictionary named ODLConstants in this definition. For example:

```
(ODLConstants at: #'::Date::Monday')
```

is a Weekday enum,

```
(ODLConstants at: #'::Time::USpacific')
```

equals -8, and

```
(ODLConstants at: #'::DateFactory::InvalidDate')
```

is an exception.

6.2.2.2 Attribute Declarations

Attribute declarations are used to define pairs of accessor operations that *get* and *set* attribute values. Generally, there will be a one-to-one correspondence between attributes defined within an ODL class and instance variables defined within the corresponding Smalltalk class, although this is not required. ODL attributes define the *abstract state* of their object when they appear within class definitions. When attributes appear within interface definitions, as in IDL they are merely a convenience mechanism for introducing get and set accessing operations.

For example:

```
attribute Enum Rank {full, associate, assistant} rank;
```

yields Smalltalk methods:

```
rank
rank: aProfessorRank
```

6.2.2.3 Relationship Declarations

Relationships define sets of accessor operations for adding and removing associations between objects. As with attributes, relationships are a part of an object's *abstract state*. The Smalltalk binding for relationships results in public methods to *form* and *drop* members from the relationship, plus public methods on the relationship target classes to provide access and private methods to manage the required referential integrity constraints. We begin the relationship binding by applying the Chapter 2 mapping rule from ODL relationships to equivalent IDL constructions, and then illustrate with a complete example.

Single-Valued Relationships

For single-value relationships such as

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

we expand first to the IDL attribute and operations:

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

which results in the following Smalltalk selectors:

```
Y
formY:
dropY:
Y:                                  "private"
```

For example, from Chapter 3:

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

yields Smalltalk methods (on the class Course):

```
formIsTaughtBy: aProfessor
dropIsTaughtBy: aProfessor
isTaughtBy
isTaughtBy:                         "private"
```

Multivalued Relationships

For a multivalued ODL relationship such as

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

we expand first to the IDL attribute and operations:

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

which results in the following Smalltalk selectors:

```
Y
formY:
dropY:
addY:           "private"
removeY:        "private"
```

For example, from Chapter 3:

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

yields Smalltalk methods (on class Professor):

```
formTeaches: aCourse
dropTeaches: aCourse
teaches
addTeaches: aCourse             "private"
removeTeaches: aCourse          "private"
```

Finally, to form the above relationship, the programmer could write:

```
| professor course |
professor := Professor new.
course := Course new.
professor formTeaches: course.
   -or-
course formIsTaughtBy: professor.
```

6.2.2.4 Collections

Chapter 2 introduced several new kinds of Collections that extend the IDL sequence to deal with the special needs of ODBMS users. The following shows the Smalltalk method selector that this binding defines for each of the Collection interfaces. Where possible, we have explicitly bound operations to commonly available Smalltalk80 selectors when the default operation binding rules would not produce the desired selector.

“interface Collection”

size	“unsigned long cardinality()”
isEmpty	“boolean is_empty()”
isOrdered	“boolean is_ordered()”
allowsDuplicates	“boolean allows_duplicates()”
add: anObject	“void insert_element(...)”
remove: anObject	“void remove_element(...)”
includes: anObject	“boolean contains_element(...)”
createIterator: aBoolean	“Iterator create_iterator(...)”
createBidirectionalIterator: aBoolean	“BidirectionalIterator create_bidirectional_iterator(...)”

“interface Iterator”

isStable	“boolean is_stable()”
atEnd	“boolean at_end()”
reset	“boolean reset()”
getElement	“any get_element()”
nextPosition	“void next_position()”
replaceElement: anAny	“void replace_element(...)”

“interface BidirectionalIterator”

atBeginning	“boolean at_beginning()”
previousPosition	“void previous_position()”

“interface Set”

createUnion: aSet	“Set create_union(...)”
createIntersection: aSet	“Set create_intersection(...)”
createDifference: aSet	“Set create_difference(...)”
isSubsetOf: aSet	“boolean is_subset_of(...)”
isProperSubsetOf: aSet	“boolean is_proper_subset_of(...)”
isSupersetOf: aSet	“boolean is_superset_of(...)”
isProperSupersetOf: aSet	“boolean is_proper_superset_of(...)”

“interface CollectionFactory”

new: aLong	“Collection new_of_size(...)”

“interface Bag”

occurrencesOf: anAny	“unsigned long occurrences_of(...)”
createUnion: aBag	“Bag create_union(...)”
createIntersection: aBag	“Bag create_intersection(...)”
createDifference: aBag	“Bag create_difference(...)”

“interface List”

at: aULong put: anObject	“void replace_element_at(...)”
removeElementAt: aULong	“void remove_element_at(...)”
retrieveElementAt: aULong	“any retrieve_element_at(...)”
add: anObject after: aULong	“void insert_element_after(...)”
add: anObject before: aULong	“void insert_element_before(.)”
addFirst: anObject	“void insert_element_first(...)”
addLast: anObject	“void insert_element_last(...)”
removeFirst	“void remove_first_element()”
removeLast	“void remove_last_element()”
first	“any retrieve_first_element()”
last	“any retrieve_last_element()”
concat: aList	“List concat(...)”
append: aList	“void append(...)”

```
"interface Array"
   replaceElementAt: aULong element: anAnyObject
                                        "void replace_element_at(...)"
   removeElementAt: aULong              "void remove_element_at(...)"
   retrieveElementAt: aULong            "any retrieve_element_at(...)"
   resize: aULong                       "void resize(...)"
"interface Dictionary"
   at: anObject put: anObject1          "void bind(...)"
   removeKey: anObject                  "void unbind(...)"
   at: anObject                         "any lookup(...)"
   includesKey: anObject                "boolean contains_key(...)"
```

6.2.2.5 Structured Literals

Chapter 2 defined structured literals to represent Date, Time, Timestamp, and Interval values that must be supported by each language binding. The following section defines the binding from each operation to the appropriate Smalltalk selector.

```
"interface Date"
   year                                 "ushort year()"
   month                                "ushort month()"
   day                                  "ushort day()"
   dayOfYear                            "ushort day_of_year()"
   monthOfYear                          "Month month_of_year()"
   dayOfWeek                            "Weekday day_of_week()"
   isLeapYear                           "boolean is_leap_year()"
   = aDate                              "boolean is_equal(...)"
   > aDate                              "boolean is_greater(...)"
   >= aDate                             "boolean is_greater_or_equal(...)"
   < aDate                              "boolean is_less(...)"
   <= aDate                             "boolean is_less_or_equal(...)"
   isBetween: aDate and: aDate1         "boolean is_between(...)"
   next: aWeekday                       "Date next(...)"
   previous: aWeekday                   "Date previous(...)"
   addDays: along                       "Date add_days(...)"
   subtractDays: aLong                  "Date subtract_days(...)"
   subtractDate: aDate                  "Date subtract_date(...)"
"interface DateFactory"
   julianDate: aUShort
      julianDay: aUShort1               "Date julian_date(...)"
   calendarDate: aUShort
      month: aUShort1
      day: aUShort2                     "Date calendar_date(...)"
   isLeapYear: aUShort                  "boolean is_leap_year(...)"
   isValidDate: aUShort
      month: aUShort1
      day: aUShort2                     "boolean is_valid_date(...)"
   daysInYear: aUShort                  "unsigned short days_in_year(...)"
   daysInMonth: aUShort
      month: aMonth                     "unsigned short days_in_month(...)"
   today                                "Date current()"
```

“interface Interval”

day	“ushort day()”
hour	“ushort hour()”
minute	“ushort minute()”
second	“ushort second()”
millisecond	“ushort millisecond()”
isZero	“boolean is_zero()”
plus: anInterval	“Interval plus(...)”
minus: anInterval	“Interval minus(...)”
product: aLong	“Interval product(...)”
quotient: aLong	“Interval quotient(...)”
isEqual: anInterval	“boolean is_equal(...)”
isGreater: anInterval	“boolean is_greater(...)”
isGreaterOrEqual: anInterval	“boolean is_greater_or_equal(...)”
isLess: anInterval	“boolean is_less(...)”
isLessOrEqual: anInterval	“boolean is_less_or_equal(...)”

“interface Time”

hour	“ushort hour()”
minute	“ushort minute()”
second	“ushort second()”
millisecond	“ushort millisecond()”
timeZone	“Time_Zone time_zone()”
tzHour	“ushort tz_hour()”
tzMinute	“ushort tz_minute()”
= aTime	“boolean is_equal(...)”
> aTime	“boolean is_greater(...)”
>= aTime	“boolean is_greater_or_equal(...)”
< aTime	“boolean is_less(...)”
<= aTime	“boolean is_less_or_equal(...)”
isBetween: aTime and: aTime1	“boolean is_between(...)”
addInterval: anInterval	“Time add_interval(...)”
subtractInterval: anInterval	“Time subtract_interval(...)”
subtractTime: aTime	“Interval subtract_time(...)”

“interface TimeFactory”

defaultTimeZone	“Time_Zone default_time_zone()”
setDefaultTimeZone	“void setDefault_time_zone(...)”
fromHms: aUShort minute: aUShort1 second: aFloat	“Time from_hms(...)”
fromHmstz: aUShort minute: aUShort1 second: aFloat tzhour: aShort tzminute: aShort1	“Time from_hmstz(...)”
current	“Time current(...)”

“interface Timestamp”

getDate	“Date get_date()”
getTime	“Time get_time()”
year	“ushort year()”
month	“ushort month()”

```
    day                                 "ushort day()"
    hour                                "ushort hour()"
    minute                              "ushort minute()"
    second                              "ushort second()"
    millisecond                         "ushort millisecond()"
    tzHour                              "ushort tz_hour(...)"
    tzMinute                            "ushort tz_minute(...)"
    plus: anInterval                    "Interval plus(...)"
    minus: anInterval                   "Interval minus(...)"
    isEqual: aTimestamp                 "boolean is_equal(...)"
    isGreater: aTimestamp               "boolean is_greater(...)"
    isGreaterOrEqual: aTimestamp        "boolean is_greater_or_equal(...)"
    isLess: aTimestamp                  "boolean is_less(...)"
    isLessOrEqual: aTimestamp           "boolean is_less_or_equal(...)"
    isBetween: aTimestamp
        bStamp: aTimestamp1             "boolean is_between(...)"
"interface TimestampFactory"
    current                             "Timestamp current()"
    create: aDate aTime: aTime          "Timestamp create(...)"
```

6.3 Smalltalk OML

The Smalltalk Object Manipulation Language (OML) consists of a set of method additions to the classes Object and Behavior, plus the classes Database and Transaction. The guiding principle in the design of Smalltalk OML is that the syntax used to create, delete, identify, reference, get/set property values, and invoke operations on a persistent object should be no different from that used for objects of shorter lifetimes. A single expression may thus freely intermix references to persistent and transient objects. All Smalltalk OML operations are invoked by sending messages to appropriate objects.

6.3.1 Object Protocol

Since all Smalltalk objects inherit from class Object, it is natural to implement some of the ODMG language binding mechanisms as methods on this class. The following text defines the Smalltalk binding for the common operations on all objects defined in Chapter 2.

```
"interface Object"
    == anObject                         "boolean same_as(...)"
    copy                                "Object copy()"
    lock: aLockType                     "void lock(...)"
    tryLock: aLockType                  "boolean try_lock(...)"
"interface ObjectFactory"
    new                                 "Object new()"
```

6.3.1.1 Object Persistence

Persistence is not limited to any particular subset of the class hierarchy, nor is it determined at object creation time. A transient object that participates in a relationship with a persistent object will become persistent when a transaction commit occurs. This approach is called *transitive persistence*. Named objects (see “Database Names” on page 216) are the roots from which the Smalltalk binding’s transitive persistence policy is computed.

6.3.1.2 Object Deletion

In the Smalltalk binding, as in Smalltalk, there is no notion of explicit deletion of objects. An object is removed from the database during garbage collection if that object is not referenced by any other persistent object. The delete() operation from interface Object is not supported.

6.3.1.3 Object Locking

Objects activated into memory acquire the default lock for the active concurrency control policy. Optionally, a lock can be explicitly acquired on an object by sending the appropriate locking message to it. Two locking mode enumeration values are required to be supported: read and write. The OMG Concurrency service’s LockSet interface is the source of the following method definitions.

To acquire a lock on an object that will block the process until success, the syntax would be

```
anObject lock: aLockMode.
```

To acquire a lock without blocking, the syntax would be

```
anObject tryLock: aLockMode. "returns a boolean indicating
                              success or failure"
```

In these methods, the receiver is locked in the context of the current transaction. A lockNotGrantedSignal is raised by the lock: method if the requested lock cannot be granted. Locks are released implicitly at the end of the transaction, unless an option to retain locks is used.

6.3.1.4 Object Modification

Modified persistent Smalltalk objects will have their updated values reflected in the ODBMS at transaction commit. Persistent objects to be modified must be sent the message markModified. MarkModified prepares the receiver object by setting a write lock (if it does not already have a write lock) and marking it so that the ODBMS can detect that the object has been modified.

```
anObject markModified
```

It is conventional to send the markModified message as part of each method that sets an instance variable's value. Immutable objects, such as instances of Character and SmallInteger and instances such as **nil**, **true**, and **false**, cannot change their intrinsic values. The markModified message has no effect on these objects. Sending markModified to a transient object is also a null operation.

6.3.2 Database Protocol

An object called a Database is used to manage each connection with a database. A Smalltalk application must open a Database before any objects in that database are accessible. A Database object may only be connected to a single database at a time; however, a vendor may allow many concurrent Databases to be open on different databases simultaneously.

```
"interface Database"
    open: aString                          "void open(...)"
    close                                  "void close()"
    bind: anObject name: aString           "void bind(...)"
    inbind:  aString                       "Object unbind(...)"
    lookup: aString                        "Object lookup(...)"
    schema                                 "Module schema()"
```

6.3.2.1 Opening a Database

To open a new database, send the open: method to an instance of the Database class.

```
database := Database new.
... set additional parameters as required ...
database open: aDatabaseName
```

If the connection is not established, a connectionFailedSignal will be raised.

6.3.2.2 Closing a Database

To close a database, send the close message to the Database.

```
aDatabase close
```

This closes the connection to the particular database. Once the connection is closed, further attempts to access the database will raise a notConnectedSignal. A Database that has been closed may be subsequently reopened using the open method defined above.

6.3.2.3 Database Names

Each Database manages a persistent name space that maps string names to objects or collections of objects, which are contained in the database. The following paragraphs describe the methods that are used to manage this name space. In addition to being assigned an object identifier by the ODBMS, an individual object may be given a name that is meaningful to the programmer or end user. Each database provides methods for

associating names with objects and for determining the names of given objects. Named objects become the roots from which the Smalltalk binding's transitive persistence policy is computed.

The bind:name: method is used to name any persistent object in a database.

```
aDatabase bind: anObject name: aString
```

The lookup:ifAbsent: method is used to retrieve the object that is associated with the given name. If no such object exists in the database, the absentBlock will be evaluated.

```
aDatabase lookup: aString ifAbsent: absentBlock
```

6.3.2.4 Schema Access

The schema of a database may be accessed by sending the schema method to a Database instance. This method returns an instance of a Module that contains (perhaps transitively) all of the meta objects that define the database's schema.

6.3.3 Transaction Protocol

"interface Transaction"	
begin	"void begin()"
commit	"void commit()"
abort	"void abort()"
checkpoint	"void checkpoint()"
isOpen	"boolean isOpen()"
join	"void join()"
leave	"void leave()"
"interface TransactionFactory"	
current	"Transaction current()"

6.3.3.1 Transactions

Transactions are implemented in Smalltalk using methods defined on the class Transaction. Transactions are dynamically scoped and may be started, committed, aborted, checkpointed, joined, and left. The default concurrency policy is pessimistic concurrency control (see Locking, above), but an ODBMS may support additional policies as well. With the pessimistic policy all access, creation, modification, and deletion of persistent objects must be done within a transaction.

A transaction may be started by invoking the method begin on a Transaction instance.

```
aTransaction begin
```

A transaction is committed by sending it the commit message. This causes the transaction to commit, writing the changes to all persistent objects that have been modified within the context of the transaction to the database.

```
aTransaction commit
```

Transient objects are not subject to transaction semantics. Committing a transaction does not remove transient objects from memory, nor does aborting a transaction restore the state of modified transient objects. The method for executing block-scoped transactions (below) provides a mechanism to deal with transient objects.

A transaction may also be checkpointed by sending it the checkpoint message. This is equivalent to performing a commit followed by a begin, except that all locks are retained and the transaction's identity is preserved.

```
aTransaction checkpoint
```

Checkpointing can be useful in order to continue working with the same objects while ensuring that intermediate logical results are written to the database.

A transaction may be aborted by sending it the abort message. This causes the transaction to end, and all changes to persistent objects made within the context of that transaction will be rolled back in the database.

```
aTransaction abort
```

A transaction is open if it has received a begin but not a commit or a abort message. The open status of a particular Transaction may be determined by sending it the isOpen message.

```
aTransaction isOpen
```

A process thread must explicitly create a transaction object or associate itself with an existing transaction object. The join message is used to associate the current process thread with the target Transaction.

```
aTransaction join
```

The leave message is used to drop the association between the current process thread and the target Transaction.

```
aTransaction leave
```

The current message is defined on the Transaction class and is used to determine the transaction associated with the current process thread. The value returned by this method may be **nil** if there is no such association.

```
Transaction current
```

6.3.3.2 Block-Scoped Transactions

A transaction can also be scoped to a Block to allow for greater convenience and integrity. The following method on class Transaction evaluates aBlock within the context of a new transaction. If the transaction commits, the commitBlock will be evaluated after the commit has completed. If the transaction aborts, the abortBlock will be evaluated after the rollback has completed. The abortBlock may be used to undo any side effects of the transaction on transient objects.

```
Transaction perform: aBlock
   onAbort: abortBlock
   onCommit: commitBlock
```

Within the transaction block, the checkpoint message may be used without terminating the transaction.

6.3.3.3 Transaction Exceptions

Several exceptions that may be raised during the execution of a transaction are defined:

- The noTransactionSignal is raised if an attempt is made to access persistent objects outside of a valid transaction context.
- The inactiveSignal is raised if a transactional operation is attempted in the context of a transaction that has already committed or aborted.
- The transactionCommitFailedSignal is raised if a commit operation is unsuccessful.

6.4 Smalltalk OQL

Chapter 4 defined the Object Query Language. This section describe how OQL is mapped to the Smalltalk language. The current Smalltalk OQL binding is a loosely coupled binding modeled after the OMG Object Query Service Specification. A future binding may include one that is more tightly integrated with the Smalltalk language.

6.4.1 Query Class

Instances of the class Query have four attributes: queryResult, queryStatus, queryString, and queryParameters. The queryResult holds the object that was the result of executing the OQL query. The queryStatus holds the status of query execution. The queryString is the OQL query text to be executed. The queryParameters contains variable/value pairs to be bound to the OQL query at execution.

The Query class supports the following methods:

```
create: aQueryString params: aParameterList     "returns a Query"
evaluate: aQueryString params: aParameterList   "returns query result"
complete                                        "returns enum complete"
incomplete                                      "returns enum incomplete"
```

Instances of the Query class support the following methods:

```
prepare: aParameterList     "no result"
execute: aParameterList     "no result"
getResult                   "returns the query result"
getStatus                   "returns a QueryStatus"
```

The execute: and prepare: methods can raise the QueryProcessingError signal if an error in the query is detected. The queryString may include parameters specified by the form $variable, where variable is a valid Smalltalk integer. Parameter lists may be partially specified by Dictionaries and fully specified by Arrays or OrderedCollections.

Example:

Return all persons older than 45 who weigh less than 150. Assume there exists a collection of people called AllPeople.

```
| query result |
query := Query
    create: 'select name from AllPeople where age > $1 and weight < $2'
    params: #(45 150).
query execute: Dictionary new.
[query getStatus = Query complete] whileFalse: [Processor yield].
result := query getResult.
```

To return all persons older than 45 that weigh less than 170, the same Query instance could be reused. This would save the overhead of parsing and optimizing the query again.

```
query execute: (Dictionary with: 2->170).
[query getStatus = Query complete] whileFalse: [Processor yield].
result := query getResult.
```

The following example illustrates the simple, synchronous form of querying an OQL database. This query will return the bag of the names of customers from the same state as aCustomer.

```
Query
    evaluate: 'select c.name from AllCustomers c where c.address.state = $1'
    params: (Array with: aCustomer address state)
```

6.5 Schema Access

Chapter 2 defined Metadata that define the operations, attributes, and relationships between the meta objects in a database schema. The following text defines the Smalltalk binding for these interfaces.

"interface MetaObject"	
name	"attribute name"
name: aString	
comment	"attribute comment"
comment: aString	
formDefinedIn: aDefiningScope	"relationship definedIn"
dropDefinedIn: aDefiningScope	
definedIn	
definedIn: aDefiningScope	
"interface Scope"	
bind: aString value: aMetaObject	"void bind(...)"
resolve: aString	"MetaObject resolve(...)"
unBind: aString	"MetaObject un_bind(...)"
"interface DefiningScope"	
formDefines: aMetaObject	"relationship defines"

```
    dropDefines: aMetaObject
    defines
    addDefines: aMetaObject
    removeDefines: aMetaObject
    createPrimitiveType: aPrimitiveKind
                                    "PrimitiveType create_primitive_type()"
    createCollectionType: aCollectionKind
        maxSize: anOperand
        subType: aType          "Collection create_collection_type(...)"
    createOperand: aString             "Operand create_operand(...)"
    createMember: aString
        memberType: aType              "Member create_member(...)"
    createCase: aString
        caseType: aType
        caseLabels: aCollection        "UnionCase create_case(...)"
    addConstant: aString
        value: anOperand               "Constant add_constant(...)"
    addTypedef: aString alias: aType   "TypeDefinition add_typedef(...)"
    addEnumeration: aString
        elementNames: aCollection      "Enumeration add_enumeration(...)"
    addStructure: aString
        fields: anOrderedCollection    "Structure add_structure(...)"
    addUnion: aString
        switchType: aType
        cases: aCollection             "Union add_union(...)"
    addException: aString
        result: aStructure             "Exception add_exception(...)"
    removeConstant: aConstant          "void remove_constant(...)"
    removeTypedef: aTypeDefinition     "void remove_typedef(...)"
    removeEnumeration: anEnumeration   "void remove_enumeration(...)"
    removeStructure: aStructure        "void remove_structure(...)"
    removeUnion: aUnion                "void remove_union(...)"
    removeException: anException       "void remove_exception(...)"
"interface Module"
    addModule: aString                 "Module add_module(...)"
    addInterface: aString
        inherits: aCollection          "Interface add_interface(...)"
    removeModule: aModule              "void remove_module(...)"
    removeInterface: anInterface       "void remove_interface(...)"
"interface Operation"
    formSignature: aParameter          "relationship signature"
    dropSignature: aParameter
    signature
    addSignature: aParameter
    removeSignature: aParameter
    formResult: aType                  "relationship result"
    dropResult: aType
    result
    result: aType
```

formExceptions: anException “relationship exceptions”
dropExceptions: anException
exceptions
addExceptions: anException
removeExceptions: anException

“interface Exception”

formResult: aStructure “relationship result”
dropResult: aStructure
result
result: aStructure

formOperations: anOperation “relationship operations”
dropOperations: anOperation
operations
addOperations: anOperation
removeOperations: anOperation

“interface Constant”

formHasValue: anOperand “relationship hasValue”
dropHasValue: anOperand
hasValue
hasValue: anOperand

formType: aType “relationship type”
dropType: aType
type
type: aType

formReferencedBy: aConstOperand “relationship referencedBy”
dropReferencedBy: aConstOperand
referencedBy
addReferencedBy: aConstOperand
removeReferencedBy: aConstOperand

formEnumeration: anEnumeration ““relationship enumeration”
dropEnumeration: anEnumeration
enumeration
enumeration: anEnumeration

value “any value(...)”

“interface Property”

formType: aType “relationship type”
dropType: aType
type
type: aType

“interface Attribute”

isReadOnly “attribute isReadOnly”
isReadOnly: aBoolean

“interface Relationship”

formTraversal: aRelationship “relationship traversal”
dropTraversal: aRelationship
traversal
traversal: aRelationship

getCardinality “Cardinality getCardinality(...)”

“interface TypeDefinition”

formAlias: aType	“relationship alias”
dropAlias: aType	
alias	
alias: aType	

“interface Type”

formCollections: aCollection	“relationship collections”
dropCollections: aCollection	
collections	
addCollections: aCollection	
removeCollections: aCollection	
formSpecifiers: aSpecifier	“relationship specifiers”
dropSpecifiers: aSpecifier	
specifiers	
addSpecifiers: aSpecifier	
removeSpecifiers: aSpecifier	
formUnions: aUnion	“relationship unions”
dropUnions: aUnion	
unions	
addUnions: aUnion	
removeUnions: aUnion	
formOperations: anOperation	“relationship operations”
dropOperations: anOperation	
operations	
addOperations: anOperation	
removeOperations: anOperation	
formProperties: aProperty	“relationship properties”
dropProperties: aProperty	
properties	
addProperties: aProperty	
removeProperties: aProperty	
formConstants: aConstant	“relationship constants”
dropConstants: aConstant	
constants	
addConstants: aConstant	
removeConstants: aConstant	
formTypeDefs: aTypeDefinition	“relationship typeDefs”
dropTypeDefs: aTypeDefinition	
typeDefs	
addTypeDefs: aTypeDefinition	
removeTypeDefs: aTypeDefinition	

“interface PrimitiveType”

kind	“attribute kind”
kind: aPrimitiveKind	

“interface Interface”

formInherits: anInheritance	“relationship inherits”
dropInherits: anInheritance	
inherits	

```
    addInherits: anInheritance
    removeInherits: anInheritance
    formDerives: anInheritance                  "relationship derives"
    dropDerives: anInheritance
    derives
    addDerives: anInheritance
    removeDerives: anInheritance
    addAttribute: aString attrType: aType       "Attribute add_attribute(...)"
    addRelationship: aString
        relType: aType
        relTraversal: aRelationship             "Relationship add_relationship(...)"
    addOperation: aString
        opResult: aType
        opParams: anOrderedCollection
        opRaises: anOrderedCollection1          "Operation add_operation(...)"
    removeAttribute: anAttribute                "void remove_attribute(...)"
    removeRelationship: aRelationship           "void remove_relationship(...)"
    removeOperation: anOperation                "void remove_operation(...)"
"interface Inheritance"
    formDerivesFrom: anInterface                "relationship derivesFrom"
    dropDerivesFrom: anInterface
    derivesFrom
    derivesFrom: anInterface
    formInheritsTo: anInterface                 "relationship inheritsTo"
    dropInheritsTo: anInterface
    inheritsTo
    inheritsTo: anInterface
"interface Class"
    extents                                     "attribute extents"
    extents: anOrderedCollection
    formExtender: aClass                        "relationship extender"
    dropExtender: aClass
    extender
    extender: aClass
    formExtensions: aClass                      "relationship extensions"
    dropExtensions: aClass
    extensions
    addExtensions: aClass
    removeExtensions: aClass
"interface Collection"
    kind                                        "attribute kind"
    kind: aCollectionKind
    formMaxSize: anOperand                      "relationship maxSize"
    dropMaxSize: anOperand
    maxSize
    maxSize: anOperand
    formSubtype: aType                          "relationship subtype"
    dropSubtype: aType
```

subtype subtype: aType	
isOrdered bound	“boolean isOrdered(...)” “unsigned long bound(...)”
“interface Enumeration”	
formElements: aConstant dropElements: aConstant elements addElements: aConstant removeElements: aConstant	“relationship elements”
“interface Structure”	
formFields: aMember dropFields: aMember fields addFields: aMember removeFields: aMember	“relationship fields”
formExceptionResult: anException dropExceptionResult: anException exceptionResult exceptionResult: anException	“relationship exceptionResult”
“interface Union”	
formSwitchType: aType dropSwitchType: aType switchType switchType: aType	“relationship switchType”
formCases: aUnionCase dropCases: aUnionCase cases addCases: aUnionCase removeCases: aUnionCase	“relationship cases”
“interface Specifier”	
name name: aString	“attribute name”
formType: aType dropType: aType type type: aType	“relationship type”
“interface Member”	
formStructureType: aStructure dropStructureType: aStructure structureType structureType: aStructure	“relationship structure_type”
“interface UnionCase”	
formUnionType: aUnion dropUnionType: aUnion unionType unionType: aUnion	“relationship union_type”

formCaseLabels: anOperand dropCaseLabels: anOperand caseLabels addCaseLabels: anOperand removeCaseLabels: anOperand	“relationship caseLabels”
“interface Parameter”	
parameterMode parameterMode: aDirection	“attribute parameterMode”
formOperation: anOperation dropOperation: anOperation operation operation: anOperation	“relationship operation”
“interface Operand”	
formOperandIn: anExpression dropOperandIn: anExpression operandIn operandIn: anExpression	“relationship OperandIn”
dropValueOf: aConstant valueOf valueOf: aConstant	“relationship valueOf”
formSizeOf: aCollection dropSizeOf: aCollection sizeOf sizeOf: aCollection	“relationship sizeOf”
formCaseIn: aUnionCase dropCaseIn: aUnionCase caseIn caseIn: aUnionCase	“relationship caseIn”
value	“any value(...)”
“interface Literal”	
literalValue literalValue: anAnyObject	“attribute literalValue”
“interface ConstOperand”	
formReferences: aConstant dropReferences: aConstant references references: aConstant	relationship references”
“interface Expression”	
operator operator: aString	“attribute operator”
formHasOperands: anOperand dropHasOperands: anOperand hasOperands addHasOperands: anOperand removeHasOperands: anOperand	“relationship hasOperands”

6.6 Future Directions

Many people believe that keys and extents are an essential ingredient of database query processing. Implicit extents and keys would be preferable to explicit mechanisms involving named Collections, yet there are challenging engineering issues that must be faced to rationalize these capabilities with the notions of transitive persistence and dynamic storage management herein presented.

A uniform set of Database administration operations would facilitate application portability and allow system administration tools to be constructed that could work uniformly across multiple vendors' database products.

This binding has only touched upon the need for interface regeneration mechanisms. Such mechanisms would allow programmers with existing applications utilizing language-specific and even database-specific ODL mechanisms to produce the interface definitions that would insulate them from the differences between these mechanisms.