A JDO Interface to OSI Managed Objects

KEVIN MITCHELL
Agilent Laboratories

An OSI NETeXPERT VSM-managed system consists of hardware network elements that communicate with each other. These network elements are assigned managed *object* names within VSM, grouping objects into *classes* that share characteristics or attributes. Managed objects (MOs) are instances of these object classes. The current state of each instance is stored within a database. The Java Data Objects (JDO) architecture provides a transparent Java-centric view of persistent information to application programmers. This paper explores some of the issues involved in providing a JDO interface to the managed object data maintained by NETeXPERT. Such an interface provides an alternative route for the input or retrieval of managed object data, and can support analysis tasks that might be difficult to implement using the existing rules-processing framework.

Key Words and Phrases: Java, JDO, NETeXPERT, object-relational mapping

1. INTRODUCTION

A network consists of a variety of objects, called devices or network elements. The OSI NETeXPERT VSM platform[OSI 2000a] refers to these devices as *managed objects* (MOs). An MO is any element in a network capable of having a status or condition, typically managed by NETeXPERT VSM. Traditional network devices include communications devices, switching equipment, computers and peripheral equipment, bridges and routers, satellites and other network element management systems. Non-traditional network devices include alarm systems, environmental controls, security devices, operators, and computer applications software.

A basic building block of NETeXPERT VSM systems is the class. A class is a template for MOs, delineating MOs that have the same properties. For example, the class `equipment` can contain a variety of different devices. Each device in the class has a name, a location, an administrative state (locked or unlocked), and an operational state (enabled, disabled, active, or busy). These properties are the class attributes. Differences in the values of these attributes distinguish individual MOs from one another. Classes can be subdivided into more specific groups, i.e. VSM supports (single) inheritance.

An MO is manipulated by an IDEAS server[OSI 2000a] using code written in a simple rule-based scripting language [OSI 2000b]. Although suitable for many event processing tasks, there are also cases where a more powerful manipulation language might be desirable. Suppose a customer wished to develop a Java application to manipulate managed objects. Such an application might be used for bulk object instance creation, report writing, or more sophisticated data analysis for example. NETeXPERT stores the state of each managed object in a relational database, e.g. Oracle. This presents the Java application developer with a number of problems to be solved before an effective application can be created.

The principal problem is that there is likely to be a considerable mismatch between the application model and the database model provided by OSI. The application model will typically be specified in terms of Java classes representing managed object classes, with their corresponding attributes and methods, and with Java inheritance being used to model MO class inheritance. The relational model, as we will see in Section 2, uses a more primitive representation with a single table for all attributes, irrespective of their type or the MO to which they belong. Constructing a Java representation of an MO instance using such a representation requires a non-trivial mapping.

An application-level query is likely to be expressed in terms of MO attributes and should return a collection of matching objects. The scope of a query may well include an MO class and its subclasses. A relational query must be written in terms of tables and columns and will return a set of matching rows. Given the underlying database representation used by OSI, the mapping from application query to database query is non-trivial, potentially involving complex joins.

Java developers typically use JDBC[2001] or SQLJ[1998] to interface to relational databases. Whilst allowing an application to pass SQL statements to the database, such APIs do nothing to hide the relational

Author's address: K. Mitchell, Agilent Labs Scotland, South Queensferry, Scotland EH30 9TG

nature of the database from the application. Clearly we need to create a mapping between the application model and the relational model. If this was a one-off problem then constructing such a mapping by hand might be an acceptable solution. However, the mapping needs to be extended each time a new MO class is defined. Furthermore, allowing user application code to directly manipulate the underlying database tables is potentially error-prone. Ideally we would like to be able to construct this mapping automatically. Java Data Objects (JDO) provides the basis for such a mechanism.

The Java Data Objects (JDO) specification[Russell et al. 2001] is a high level API that defines a standard way for applications to store Java objects in transactional data stores. It allows users to specify their application program logic and queries entirely in Java. Most importantly, it is not necessary for programmers to explicitly fetch and store Java objects from a database: this is done automatically in JDO.

The JDO implementation hides all the details of the persistence mechanism from the application. Two main interfaces are exposed to applications. The main interface for persistent-aware applications is the `PersistenceManager` interface. A persistence manager is responsible for cache management and also provides services such as query management and transaction management. In JDO, objects that are to be made persistent are called persistence-capable objects; they expose the `PersistenceCapable` interface to applications. This provides services such as life cycle state management for persistence-capable classes. A tool called an *enhancer*, provided by the JDO vendor, is used to modify the standard output of a Java compiler to add an implementation of the PersistenceCapable interface to the specified classes.

Of course an application cannot be totally unaware of the persistent nature of the data it is manipulating. Access to the data is typically made within the context of a transaction, and the application must also explicitly indicate when an object instance should be made persistent or transient. A typical JDO application starts by creating a PersistenceManager from a PersistenceManagerFactory.

```
PersistenceManagerFactory pmf = ...
// Initialize factory properties, e.g. database URL and user
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();
```

Suppose we have a Person class, defined by

```
public class Person
{
  public String name;
  public Address address;
  ...
}
```

We can retrieve a persistent instance of this class, and alter it within the context of a transaction, using code similar to the following:

```
txn.begin();
// perform query
Query query = pm.newQuery(Person.class,"name == \"Fred\"");
Collection result = (Collection)query.execute();
Iterator iter = result.iterator();
// iterate over the results
while (iter.hasNext()) {
  Person person = (Person)iter.next();
  // traverse to the address object and update its value
  person.address.street = "1 Park Lane";
}
txn.commit();
```

We can make a transient object persistent using the `makePersistent` method. The reverse operation, making a persistent object transient, uses the `deletePersistent` method.

Person p = **new** *Person*(...);
pm.makePersistent(*p*);
...
Person pp = ...
pm.deletePersistent(*pp*);

Section 7 gives some more examples of JDO code for manipulating NETeXPERT managed objects. Further details of the JDO specification can be found in [Russell et al. 2001].

In all these examples note how the Java code is not polluted by any details of the object-relational mapping. The compiled class file for each persistence-capable class is augmented with the necessary mapping code by a vendor-provided enhancer application. But how does the enhancer know which fields to persist, or where to store them? A JDO application defines which attributes of a class are persistent using an XML file with the extension `.jdo`. You can provide such a file either for every persistence-capable class or for each package containing such a class. These files define which fields should be persistent and which are transient. In the case of a field representing a one-to-many or many-to-many relation additional information must also be supplied defining the nature of this relationship. Here is an outline of the jdo file for our `Person` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<jdo>
  <package name="...">
    <class name="Person" ...>
      <field name="name" ...  />
      <field name="address" ...  />
    </class>
  </package>
</jdo>
```

The *.jdo files tell the system what needs to be stored, but not where, or how, to store it. Even for a simple example there are many database schemas that could be used to store the data. We could create one table for each class, with a column for each attribute. This is conceptually simple, but is not efficient when the scope of a query includes subclasses. Or we might create one table for each class hierarchy, rolling each subclass and its attributes into one table. This is efficient for subclass queries, but can potentially result in very wide tables.

Some JDO implementations assume that the database schema is under the control of the implementation. The enhancer not only produces an updated class file but also the SQL code necessary to create the corresponding database tables. When an application has to work with existing data then such an approach is clearly not appropriate. In such cases we need to use a JDO implementation that allows some configurability in the mapping process. One of the ways in which JDO vendors differentiate themselves is in the flexibility of this mapping process.

Unfortunately none of the existing JDO implementations are sufficiently flexible to be able to cope with the representation of managed objects used by OSI, as described in Sections 2 and 3. A specialized enhancer could be developed with hard-wired knowledge of the OSI schema but that would require a non-trivial investment of resources. One of the main aims of this report is to explore how one might bridge the gap between the current database schema and existing JDO implementations. By judicious choice of JDO implementation, coupled with extensive use of database views and triggers, we show how to construct a JDO interface to the MO data. To illustrate this process a prototype has been developed that constructs a Java class for each MO class and attribute type in an OSI database, together with the necessary configuration files to drive a JDO enhancer. Of course such an implementation is unlikely to be as efficient as a solution based on a specialized enhancer, but it allows us to evaluate the utility of such an interface without committing expensive resources.

The rest of the report is structured as follows. Section 2 describes the schema used to represent managed objects in more detail. Section 3 then shows how the different attribute types are encoded in this schema. A brief description of the JDO implementation used in our prototype then follows in Section 4. Section 5 describes how to bridge the gap between database schema and JDO using this implementation. Section 6 discusses some of the finer points of the prototype, and some examples are presented in Section 7. Finally, the conclusions evaluate the utility of the prototype and contains suggestions for future work.

CLASS	NAMINGATTRIBUTE	NAME	TYPE	ISAMGR	PARENT
0	0	UNKNOWN		0	0
1	0	top	Forum	0	0
3	1005	circuit	Forum	0	1
4	1011	contact	Forum	0	1
5	1013	customer	Forum	0	1
6	1015	equipment	Forum	0	1
⋮					
35	1015	router		0	6

Fig. 1. Example `CLASS` table entries

2. THE OSI DATABASE SCHEMA

In this section we describe the tables used to represent managed object instances, classes, attributes and their types. No attempt is made to give a complete description of these tables. Without access to OSI design documents we have had to treat the OSI system as a black box, inferring the semantics of the fields from examples of their use. Whilst sufficient for a prototype, clearly a production-quality implementation would need access to a detailed specification of the schema. The database schema contains many tables in addition to those required to store managed objects and we can (hopefully) safely ignore these. There are ten tables that are of interest to us, `CLASS`, `CLASSATTR`, `ATTRBUTE`, `ABSTRACTTYPE`, `RELATE`, `MO`, `MOATT`, `MORELATE`, `MOM`, and `SURROGATE`, and we describe each of these in turn.

2.1 CLASS

Every managed object class has a corresponding entry in the `CLASS` table. Each row contains, amongst other things, the class name and its superclass.

Column	Type
CLASS	NUMBER(11)
NAMINGATTRIBUTE	NUMBER(11)
NAME	VARCHAR2(254)
TYPE	VARCHAR2(10)
ISAMGR	NUMBER(7)
PARENT	NUMBER(11)

Every managed object class is derived, directly or indirectly, from the class "`top`" in this table. Unfortunately this class itself is derived from the "`UNKNOWN`" class which has itself as the parent. This makes the relationship between table rows and Java classes slightly more messy than it needs to be. Figure 1 contains a fragment of a class table illustrating how the class `router` is a subclass of `equipment`, which is itself a subclass of `top`.

2.2 CLASSATTR

The `CLASS` table defines the names, and inheritance hierarchy, of the classes but says nothing about the attributes defined for each class. This information is provided by the `CLASSATTR` table.

Name	Type
CLASS	NUMBER(11)
ORDR	NUMBER(7)
ATTRBUTE	NUMBER(11)
RELATEDCLASS	NUMBER(11)
RELATIONSHIP	NUMBER(11)
MANDATORY	NUMBER(7)
STATUS	NUMBER(7)

For each class there will be a row in this table for every attribute associated with the class, *excluding* inherited attributes. Figure 2 contains a fragment of a class attribute table. This example shows that the class with key `3`, which represents the `circuit` class in Figure 1, has three attributes with keys `1000`, `1002` and `1001`.

CLASS	ORDR	ATTRBUTE	RELATED CLASS	RELATION SHIP	MANDATORY	STATUS
3	1	1000	0	0	1	0
3	2	1002	0	0	1	0
3	3	1001	0	0	1	0
⋮						

Fig. 2. Example `CLASSATTR` table entries

ATTRBUTE	NAME	TYPE	DEFVALID
1000	administrativeState	105	949
1001	bandwidthType	106	null
1002	aEndpointName	5	0
⋮			

Fig. 3. Example `ATTRBUTE` table entries

TYPE	SEGMENT	REFERENCE	NOTATION
5	0	MO	MO
105	0	administrativeStateEnum	Enumerated{"locked"(0),...}
106	0	bandwidthTypeEnum	Enumerated{"digital"(0),...}
⋮			

Fig. 4. Example `ABSTRACTTYPE` table entries

2.3 ATTRBUTE

The `CLASSATTR` table defines which attributes are associated with which classes, but not what the attributes themselves mean. In particular we need to know the name and type corresponding to each attribute key. This information is provided by the `ATTRBUTE` table.

Name	Type
ATTRBUTE	NUMBER(11)
NAME	VARCHAR2(254)
TYPE	NUMBER(11)
DEFVALID	NUMBER(11)

Figure 3 contains a fragment of an attribute table. This example shows that the three attributes associated with the `circuit` class are called `administrativeState`, `bandwidthType` and `aEndpointName`, with types whose keys are `105`, `106` and `5` respectively.

2.4 ABSTRACTTYPE

Attributes in the NETeXPERT framework can have many different types. Some attributes hold primitive values like integers and strings whilst others represent structured data such as sequences, disjoint unions and records. The `ABSTRACTTYPE` table describes the types currently in use by attributes in the database.

Name	Type
TYPE	NUMBER(11)
SEGMENT	NUMBER(7)
REFERENCE	VARCHAR2(254)
NOTATION	VARCHAR2(254)

The `REFERENCE` column contains the type's name, whilst the `NOTATION` column defines its meaning. In the case of a primitive type this is the same as the type name. But for other types the `NOTATION` column describes how the type is constructed from more primitive elements. This column has a relatively limited size and some complex types may have descriptions that exceed this limit. The `SEGMENT` column is used in such cases to break the description into multiple segments, ordered by the index in this column. Figure 4 contains a fragment of a type table. This shows that the `aEndPointName` attribute contains a managed object reference, whereas both the `administrativeState` and `bandwidthType` attributes have enumerated types for their values. Section 3 discusses the `NOTATION` column in more detail.

RELATE	NAME	ONAME	GENERIC
0	NONE	null	0
1	ConnectUp	ConnectDown	1
2	ConnectDown	ConnectUp	1
3	ContainedIn	Contains	1
4	Contains	ContainedIn	1
5	ManagedBy	Manages	1
6	Manages	ManagedBy	1
7	bsc_site_down	bsc_site_up	0
8	bsc_site_up	bsc_site_down	0
⋮			

Fig. 5. Example `RELATE` table entries

MO	CLASS	NAME	PARENT	MOM	REPORTEDAS
30	35	Switch_DA_SYB_MO	0	0	null
31	36	EFDgeneral	0	0	null
90	43	2DOMCR1_TAIPEI2	408	0	null
⋮					

Fig. 6. Example `MO` table entries

2.5 RELATE

The `RELATE` table keeps track of the different kinds of relationships defined between managed objects.

Name	Type
RELATE	NUMBER(11)
NAME	VARCHAR2(30)
ONAME	VARCHAR2(30)
GENERIC	NUMBER(7)

The `RELATE` column contains the primary key for the table. Relationships are typically named in pairs, corresponding to the two ends of the link. The `NAME` column contains the name of a relationship, whilst the `ONAME` column contains the name for the opposite end of the link, as illustrated in Figure 5. The `GENERIC` column seems to be set to `1` for the three built-in relations and `0` for everything else. The NETeXPERT documentation states that relations should be defined in pairs. In the test database provided by OSI there are some relations where this is not true. It is not clear whether this is an error, or whether it has semantic significance.

2.6 MO

The tables discussed so far define the static state of the system, i.e. the classes, attributes and types used to define managed objects. We now turn our attention to the representation of the instances of these classes. The `MO` table contains a row for each such instance.

Name	Type
MO	NUMBER(11)
CLASS	NUMBER(11)
NAME	VARCHAR2(254)
PARENT	NUMBER(11)
MOM	NUMBER(11)
REPORTEDAS	VARCHAR2(254)

Figure 6 contains a fragment of an MO table. Note that all managed objects, irrespective of the class to which they belong, are stored in single table.

2.7 MOATT

The `MO` table defines which instances of which classes currently exist in the system, but does not define the values of the attributes in these instances. This is the role of the `MOATT` table.

MOID	ATTRID	COMPID	STRVALUE	...	LOWINT VALUE	SYNTAX
2	1000		unlocked			3
2	1012	0000000000			4	6
2	1026		active			3
2	1353		critical			3
3	1012	0000000000			4	6
3	1018		v-h-Coordinates			3
3	1019		10			12
3	1021		SQF			12
3	1023				2	4
3	1027	0000000000	123 George Street			12
3	1027	0000000001	Glasgow			12
5	1000		shuttingDown			3
5	1012	0000000000			4	6
5	1357	0				12
5	1357	1	Ascending			3
6	1000		unlocked			3
6	1012	0000000000			4	6
6	1015		Test			12
6	1022				3	6
6	1026		busy			3
6	1358	0.0000000001.0000000000			42	4
6	1358	0.0000000001.0000000002			43	4
6	1358	0.0000000002.0000000000			0	4
6	1358	1.0000000001.0	Fred Bloggs			12
6	1358	1.0000000001.1			0	1
6	1358	2	1.000000000000000e+00			9
6	1359	0000000000.0000000000			18	4
6	1359	0000000000.0000000001			181	4
6	1359	0000000002.0000000000			0	4
⋮						

Fig. 7. Example `MOATT` table entries

Name	Type
MOID	NUMBER(11)
ATTRID	NUMBER(11)
COMPID	VARCHAR2(240)
STRVALUE	VARCHAR2(254)
REALVALUE	FLOAT(126)
HIGHINTVALUE	NUMBER(11)
LOWINTVALUE	NUMBER(11)
SYNTAX	NUMBER(7)

Consider the case of a simple string attribute. There will be one row in this table for each instance of a class containing this attribute. Each of these rows will contain the attribute value in the `STRVALUE` column, with the `REALVALUE`, `HIGHINTVALUE` and `LOWINVALUE` columns being unused. The situation is more complex in the case of an attribute with a structured type such as a sequence or record. In such a case there will be one row for each leaf in the tree representing the structured object, with the `COMPID` column indicating the path from the root to this leaf. This process is illustrated in Figure 7 which contains a fragment of an MO attribute table. Section 3 describes the encoding of the `COMPID` column in more detail. At this point it is sufficient to note that all attributes of all managed objects are held in a single table, and the values of compound attributes are spread across multiple rows within the table. The difficulties involved in mapping from the application model to the database model should start to become apparent.

2.8 MORELATE

The `MORELATE` table keeps track of the relationships between managed objects. Note that the `ManagedBy` and `ContainedIn` information is also duplicated in the `MO` table, presumably for efficiency reasons.

Name	Type
MO1	NUMBER(11)
PRIORITY1	NUMBER(7)
MO2	NUMBER(11)
PRIORITY2	NUMBER(7)
STRATEGY	NUMBER(7)
KIND	NUMBER(7)

The `KIND` column is a foreign key to the `RELATE` column of the `RELATE` table. It is not clear what the priority and strategy columns are used for in this table. They appear to be always set to `0` in the test database. When there is a relationship between managed objects `A` and `B` it can usually be specified in two different ways as each NETeXPERT relation typically has an inverse. The test database doesn't appear to use a canonical form for expressing such relations. So to find out if `A` is "connected up" to `B` one may need to check for rows corresponding to both (`A`,`B`) in the `ConnectUp` relation and (`B`,`A`) in the `ConnectDown` relation.

2.9 MOM

The `MOM` table contains a row for each managed object manager. [1] [2]

Name	Type
MOM	NUMBER(11)
TIMEZONE	VARCHAR2(10)

2.10 SURROGATE

The `SURROGATE` table is used to allocate fresh keys for the various objects used in the system.

Name	Type
TID	NUMBER(11)
ID	NUMBER(11)
LOCKING	NUMBER(11)

The row with `TID` = `1` is used for managed object instances. A JDO implementation will need to use this table to allocate fresh MO keys that do not clash with those generated by the IDEAS server.

3. OSI DATATYPE ENCODING

In this section we describe in more detail how datatypes are represented in the `NOTATION` column of the `ABSTRACTTYPE` table, and how the corresponding values are stored within the `MOATT` table.

3.1 Primitive types

The VSM platform defines a number of primitive types, such as `String`, `Boolean`, `Integer32`, `Integer64` and `MO`, with the obvious semantics. They can be distinguished from other types in the database as the `NOTATION` column just contains the name of the type in these cases. The `STRVALUE`, `REALVALUE`, `HIGHINTVALUE` or `LOWINVALUE` column in the `MOATT` table contains the corresponding value depending on the type concerned. In the case of 64-bit quantities the value is spread across the `HIGHINTVALUE` and `LOWINVALUE` columns. Figure 8 summarizes these encodings.

3.2 Enumerations

An enumeration type, as illustrated in Figure 4, contains a string starting with "`Enumerated{`". Following this are the different cases in the enumeration, in the format "*name*"(*n*), followed by a terminating brace. So a complete example, for the `administrativeStateEnum` type of Figure 4, would look like

```
Enumerated{"locked"(0),"unlocked"(1),"shuttingDown"(2)}
```

In the `MOATT` table the corresponding values are stored as strings rather than integers. So an instance of an attribute of type `administrativeStateEnum` representing the `unlocked` state would contain "`unlocked`" in the `STRVALUE` value rather than `1` in the `LOWINTVALUE` column.

[1] This table also contains rows for objects that don't correspond to managed object managers. It's not clear what these represent.
[2] The `TIMEZONE` column is always null in the sample database.
[3] In our sample database the `REALVALUE` column does not appear to be used for any values.

Type	Stored as	Syntax
`Null`	"Null" in `STRVALUE` column	7
`Boolean`	`0` or `1` in `LOWINTVALUE` column (`0` = `false`)	1
`Integer32`	Value in `LOWINTVALUE` column	4
`UnsignedInteger32`	Value in `LOWINTVALUE` column	13
`Integer64`	Value in `HIGHINTVALUE` and `LOWINTVALUE` columns	5
`UnsignedInteger64`	Value in `HIGHINTVALUE` and `LOWINTVALUE` columns	14
`Real`	Value stored as string in `STRVALUE` column.[3]	9
`String`	Value in `STRVALUE` column	12
`OctetString`	Value in `STRVALUE` column in the form "`0x6672656465...`"	8
`MO`	Managed object ID stored in `LOWINTVALUE` column	6

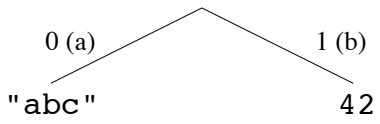
Fig. 8. Encoding of Primitive Types

3.3 Record types

For simplicity, consider the case of a record type, or sequence type in OSI terminology, `RT` containing two fields, `a` and `b`. Furthermore, assume that field `a` has type `String` whilst field `b` has type Integer32. Such a type would be represented in the `ABSTRACTTYPE` table by a row with `REFERENCE` set to `RT` and a `NOTATION` column having the value

```
Sequence{a String,b Integer32}
```

An instance of this type can be viewed as a tree with two branches.

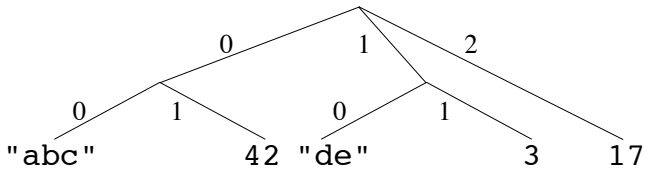


Each leaf in such a structured value is stored as a separate row in the `MOATT` table, with the path from the root to the leaf stored in the corresponding `COMPID` column. For compactness, a positional index is used instead of the field name. So in the case of an attribute of this type the `COMPID` column will contain `0` for the `String` component and `1` for the `Integer32` component. If a record has more than ten attributes then the single digit indices are padded with a leading `0`. Presumably a record type with more than a hundred components would pad everything to three digits with leading zeros, and so on.

Now consider a slightly more complex case where the field types are themselves structured. The type

```
Sequence{f1 RT,f2 RT,f3 Integer32}
```

illustrates such a scenario. An instance of this type would look like



An attribute with this type would require five rows to represent each instance, corresponding to the five leaves in the tree. The row containing the value `42` would have the path `0.1` in the `COMPID` column.

The example illustrates one of the main problems in developing a JDO mapping for this schema. JDO implementations typically map persistent classes to tables or views. Consider the problem of constructing a view containing all instances of the `RT` type. Each row will contain columns for the `a` and `b` fields. Such rows will have to be assembled by querying the `MOATT` table. But this table is indexed by attribute, not type, and so we must examine the `COMPID` column to resolve which rows correspond to fields of the `RT` type. If records were the only compound type then we could construct an auxiliary table to assist in this task. The table would contain (type,attribute,component) triples, indicating in which attributes a given type could be found, and the component id(s) associated with these occurrences. However, the `COMPID` component is also used for arrays, choices, and the `Any` type. As we will see in the following sections, these uses greatly

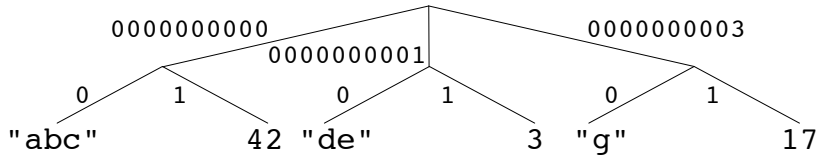
complicate the construction of such a view. Section 5.4 considers this problem in more detail, and describes the solution adopted in the prototype implementation.

3.4 Arrays

Like records, arrays can also be viewed as trees where each branch is labelled with the array index. OSI uses this approach with the minor wrinkle that each index is 0-padded to ten digits. The arrays can also be sparse, i.e. the first index does not have to be `0000000000`, and there may be "gaps" in the branch labels. An array of `RT` types would be represented by a row in the `ABSTRACTTYPE` table with a `NOTATION` column having the value

```
SequenceOf RT
```

An instance of this type would look like



Note that finding all rows in the `MOATT` table corresponding to the `b` field of the `RT` class involves pattern matching, rather than simple string matching, when instances of this type can be embedded inside arrays.

3.5 Type aliases

Sometimes it is convenient to create a type with exactly the same structure as an existing type. If the value in the `NOTATION` column of an `ABSTRACTTYPE` row is equal to the name of an existing type then the type introduced by this row is treated as a type alias. Such aliases introduce some interesting problems when mapping abstract types to Java classes. These issues are discussed in Section 5.6.

3.6 Choice types

The NETeXPERT platform supports choice, or disjoint union, types. The type

```
Choice{anInt Integer32, aString String, aBool Boolean, anRT RT}
```

represents objects that can contain an integer, string, boolean or RT value. At any point in time such an object can contain exactly one of these choices, but which one? The `STRVALUE` column provides this information. An example will probably help here. Suppose an attribute with id `3036` has this choice type. The `MOATT` table might contain the following rows to represent an instance of this attribute.

MOID	ATTRID	COMPID	STRVALUE	LOWINTVALUE
659	3036		3	
659	3036	3.0	abc	
659	3036	3.1		42

This example represents an instance of the choice type holding an `RT` value. The value `3` in the first row indicates the value contains the fourth choice, 0-based, and the next two rows describe the `RT` value.

3.7 The `Any` type

The NETeXPERT system supports an `Any` type. An attribute of this type can store any value of any type. We can also build sequences of `Any` types to store heterogeneous collections, and store such types as record components. The `STRVALUE` column indicates the type of the associated value. The type name is embedded inside "`0<...>`" brackets, as illustrated in the following example.

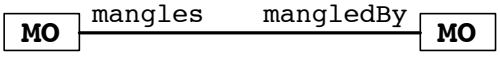
MOID	ATTRID	COMPID	STRVALUE	LOWINTVALUE
659	3037		0<RT>	
659	3037	0<RT>.0	abc	
659	3037	0<RT>.1		42

3.8 Relationships

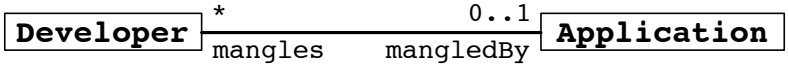
Relations are probably the most complex aspect of the OSI schema. Part of the complexity is due to the inherent power of associations. The relational features of the system have also evolved over time, leading to some unnecessary semantic complexity, ambiguity, and duplication of effort. In this section we try to present a reconstruction of these aspects of the system. However, given the difficulties already mentioned, it is difficult to be completely sure that this account faithfully describes how relations are supposed to work in the current system.

A non-zero `RELATEDCLASS` value in the `CLASSATTR` table is used for two distinct purposes. If the corresponding `RELATIONSHIP` column is zero, indicating no relationship, then the value constrains the type of the associated attribute. For example, suppose the attribute type was `MO`, but the `RELATEDCLASS` entry contained (the key for) the `equipment` class. All instances of the attribute for this particular class would then be restricted to equipment values. This mechanism illustrates some of the tension between attribute types and managed object types within IDEAS. Although types can contain managed object references, they cannot limit the type to a particular MO subclass. The `RELATEDCLASS` mechanism provides a simple way of achieving such constraints, but only for attributes of type `MO` and `SequenceOfMO`. Curiously, the test database contains non-zero `RelatedClass` entries for record types containing no managed objects. It doesn't appear to be possible to construct such examples using the current NETeXPERT GUI and so we assume such entries are historical relics. The system defines an `UNKNOWN` managed object class, with a key of 0, from which all other classes inherit. The test database contains examples of attributes of type `SequenceOfMO`, with a type constraint, where instances of the attribute contain `UNKNOWN` objects. It is not clear whether such occurrences are intentional, requiring special treatment of the `UNKNOWN` class when enforcing type constraints, or whether this is another example of legacy data.

Now consider the more interesting case where the `RELATIONSHIP` column contains a non-zero value. Relations in NETeXPERT provide a way of creating associations between managed objects. This is best illustrated with an example. We assume the `RELATE` table has been populated with two relations, `mangles` and its inverse, `mangledBy`. The relations can be thought of as *roles* on associations, using UML terminology [Booch et al. 1999].



Note that at this stage the classes that can participate in this association have not been specified. The multiplicity of the roles is also unknown. Can an object be mangled by many other objects, or at most one? Can an object be responsible for mangling many different objects? A particular use of this association might relate developers and applications, as in



This diagram represents the situation where a developer can mangle at most one application, and an application can be mangled by zero or more developers. How do such instantiations of an association get established in the NETeXPERT framework? And how do individual objects get related?

When defining a class attribute you can associate a relation with it using the `RELATIONSHIP` column. For example, a `Developer` class might contain an attribute called `MangledApp` coupled with the `mangles` relation. Suppose this attribute was of type `MO`. Given an instance of this class, D, then you could add a row to the `MORELATE` table, associating a managed object M with D using relation `mangles` by assigning M to the `MangledApp` attribute. Furthermore, if this class attribute also had the `RELATEDCLASS` column set to `Application` then this would restrict M to be an instance of this class. Note that the name of the class attribute plays a relatively unimportant part in this process. IDEAS uses the relation name, not the attribute name, to perform queries on the association. The main purpose of the attribute is to indicate that the class plays a particular role in the specified association. The type of the attribute also gives some indication of the intended multiplicity of the opposite role. For example, if the type of `MangledApp` is `MO` then this suggests that the multiplicity of `mangles` is 0..1.

Although not essential, the association can also be populated from the reverse direction given a suitable attribute definition. For example, the `Application` class could have a `Manglers` attribute, of type `SequenceOfMO`, associated with relation `mangledBy` and with a related class of `Developer`. This would allow

a sequence of mangling developers to be associated with an application by assigning them to this attribute. Furthermore, this attribute gives a hint that the multiplicity of the `mangledBy` role is *.

Given a `Developer` object D and an `Application` object A, they can be related by the `mangles-mangledBy` association in one of two equivalent ways. We could state that D mangles A or that A is mangled by D. These would be represented by one of two different rows in the `MORELATE` table.

MO1	MO2	KIND	...
D	A	mangles	
A	D	mangledBy	

Which row is used to represent this association depends on the attribute used to create it. This implies that when determining which developers are responsible for mangling an application we must check rows corresponding to both directions of the association.

At first glance, it would appear that the attributes defined for the `Developer` and `Application` classes capture the spirit of our earlier diagram. But this impression is perhaps misleading. A developer can only mangle one application at a time when using the `MangledApp` attribute. But we cannot enforce this multiplicity as there is nothing stopping multiple applications blaming the same developer using the `Manglers` attribute. What about the type of the related objects? Consider a third class, `Router`, that also contains a `Manglers` attribute with the same properties as the `Application` version. If R is an instance of this class then it can use the `Manglers` attribute to create an association with D. From the perspective of the `Developer` class it is tempting to think that developers can only be accused of mangling applications. However, when querying the association developers may discover they have mangled routers as well.

The use of special attributes to populate the associations raises some additional issues. Should such attributes be viewed as write-only? If they can be read then what value should be returned, bearing in mind that the association can also be populated from the other end? In particular, what value should be returned for an attribute of `MO` type if it is related to multiple objects? The current NETeXPERT GUI appears to make no use of the converse relation when displaying such attributes. For example, if application A adds developer D to the `Manglers` attribute then the `MangledApp` attribute of D will not record this fact. In the light of this they should probably be considered as write-only attributes. Other interesting scenarios occur when a class has two attributes connected to the same relation, perhaps with different types.

The `Manages` and `Contains` relations, and their inverses, are treated specially by the system. The `PARENT` and `MOM` columns of the `MO` table encode these relations *in addition to* the usual entries in the `MORELATE` table. Any changes to these relations must therefore be made in two separate places. The NETeXPERT GUI does not allow attributes to be associated with these relations, simplifying this task. Other non-standard features include relational attributes of type SequenceOfMO where there are entries in the `MORELATE` and `MOATT` tables.

3.9 Object creation

Objects are created in the NETeXPERT system via the `netx_createmo` stored procedure.

```
PROCEDURE netx_createmo (
    mo_in IN INTEGER,         name_in IN VARCHAR2,
    xclass_in IN INTEGER,     parent_in IN INTEGER,
    mom_in    IN INTEGER,     repas_in  IN VARCHAR2,
    namingatt_in IN INTEGER, isamom_in IN INTEGER,
    retkey_out OUT INTEGER,  retval_out OUT INTEGER )
```

The parameter `mo_in` contains the key for the new object. The value `0` indicates the procedure itself should generate a new key. The chosen key is returned via the `retkey_out` parameter. Every managed object must have a unique name and this is specified using the `name_in` parameter. The class of managed object is specified using `xclass_in`, where the value is used as a key into the `CLASS` table. If `parent_in` $\neq 0$ then this value indicates the parent of the new node in the `ContainedIn` relationship. Similarly a non-zero value for the `mom_in` parameter is used to populate the `ManagedBy` relationship. The `repas_in` parameter initializes the `REPORTEDAS` column in the new row. If `isamom_in` is non-zero then a row for the new object is added to the `MOM` table. The success, or otherwise, of the call is returned in the `retval_out` parameter, with 0 indicating success.

The `netx_deletemo` procedure can be used to delete managed objects.

```
PROCEDURE netx_deletemo (
    mo_in IN INTEGER, retval_out OUT INTEGER )
```

4. THE INTELLIBO JDO IMPLEMENTATION

There are numerous JDO implementations available, LiDO from LIBeLIS, Kodo from TechTrader, OpenFusion from PrismTech, and intelliBO from SignSoft being just some of them. Although the JDO specification defines how user code interacts with JDO objects, the details of how such objects are mapped to a relational database are left deliberately vague. This allows plenty of scope for different JDO vendors to differentiate their products. Some vendors concentrate on providing support for applications that place no constraints on the underlying database schemas. Such applications use the database to store persistent Java objects but have no need to access legacy data. A JDO implementation targeted at such a market is free to choose whatever database schema best suits the Java classes being persisted. The details of the mapping from Java class to database table may be of no concern to such an application developer.

Another class of application requires access to legacy data, with JDO objects providing a convenient wrapper around this data. In such cases the database schema will have been fixed in advance, and the developer needs to explicitly provide mapping information from database tables to JDO classes. The expressiveness of the mapping "language" differentiates JDO implementations.

One area in which JDO implementations can significantly differ is in their support for inheritance. Indeed some JDO implementations don't support inheritance at all. Amongst those that do there is still plenty of scope for different representation choices:

— An implementation could create one table for each class, with a column for each attribute. This is conceptually simple, but is not efficient when the scope of a query includes subclasses.

— Another strategy is to create one table for each class hierarchy, rolling each subclass and its attributes into one table. This is efficient for subclass queries, but can potentially result in very wide tables.

— An implementation could also create one table for each class, with a column for each attribute directly defined on this class. Inherited attributes are retrieved from the tables corresponding to the classes in which the attributes are defined.

— Another option is possible when a particular pattern of use can be identified and this is to map a set of classes into one table. This enables queries to be optimized for particular combinations of objects.

The eventual aim of most JDO implementations is to support a variety of inheritance mapping schemes, but at the time of writing every JDO implementation investigated only supports at most one such scheme.

Given our need to interface to legacy data, we clearly need to choose a JDO implementation that allows the developer to specify explicit mapping information. Furthermore, given the particular schema we have to use, we would like the mapping facilities to be as expressive as possible. Every managed object class is derived from the top class, and so using an encoding of persistence that requires a table for each class hierarchy will lead to a single table with as many columns as there are attributes in the system. We could trim off this "top" type, losing the ability to iterate over all managed objects. But a better alternative would be to use an implementation that handles inheritance via a table for each class. SignSoft's **intelliBO** product is an example of such an implementation. Although the prototype makes use of some features specific to intelliBO, as the JDO market matures it seems likely that an interface to NETeXPERT could be based on other implementations with little difficulty.

In the sections that follow we describe some of the distinctive features of the intelliBO implementation. However, the reader is directed to [Signsoft 2001] for full details of this product.

4.1 SJDO Files

The JDO files describe which object attributes should be treated persistently. They also define which of these attributes should be fetched when the object is initially loaded, and which should be loaded lazily. Attributes can contain (references to) other objects. Without such control over the loading strategy we may end up fetching the whole world as a result of loading a single object. The mechanism for specifying how to map these objects to database tables is vendor-specific. The JDO specification defines an element that can be used for vendor attributes. However, Signsoft has chosen to specify the mapping details in separate SJDO files to avoid polluting the JDO files. The SJDO file corresponding to the JDO file from Section 1 would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<sjdo>
  <package name="...">
    <class name="Person" ...>
      <field name="name">
        <field-mapping table="TPERSON" field="FNAME">
      </field>
      ...
    </class>
  </package>
</sjdo>
```

This configuration file states that the value of the `name` attribute of the class `Person` can be found in column `FNAME` of the `TPERSON` table. Each persistent non-relational attribute of the class would have a similar `<field>` entry in this file.

4.2 Relations

One of the more powerful features of the JDO specification is its support for relations. Many objects have attributes that refer to other objects, forming a complex mesh of relationships. A persistence mechanism should support such relationships in a transparent fashion, silently and lazily loading objects as relationship links are traversed.

Sometimes there is a one-to-one relation between objects. A person, for example, can have an address. Instead of adding the address attributes into the person they can be separated into an address class. The class `Person` then contains exactly one reference to an object of class `Address`. The `addr` attribute of the `Person` class would then have an SJDO entry that specified which column in the `TPERSON` table contained the foreign key of the address, and which table to use to find the address object corresponding to this key.

```
<field name="addr">
  <one-to-one-mapping>
    <simple-ref
      from-table="TPERSON"
      from-field="FADDRKEY"
      to-table="TADDRESS"
      to-field="FKEY"/>
    <element-type>
      Address
    </element-type>
  </one-to-one-mapping>
</field>
```

Suppose we wish to associate many addresses with a single person. This is an example of a one-to-many relationship. In this case the `addr` attribute would represent a collection of addresses, and have a type such as `List`. The JDO file would specify the type of the elements within this collection.

```
<field name="addr">
  <collection element-type="Address"/>
</field>
```

The corresponding SJDO file defines what collection type to use to hold the addresses, and the mapping between the `TPERSON` and `TADDRESS` tables as in the one-to-one case.

```
<field name="addr">
  <one-to-many-mapping>
    <simple-ref
      from-table="TPERSON"
      from-field="FADDRKEY"
      to-table="TADDRESS"
      to-field="FKEY"/>
    <result-type>
      java.util.ArrayList
```

```
    </result-type>
  </one-to-many-mapping>
</field>
```

Finally, consider the situation where we wish to assign any number of addresses to a person, and any number of persons to an address. Such a relation typically involves the use of an auxiliary table that relates rows in the `TPERSON` and `TADDRESS` tables. Such many-to-many relations can also be specified in the SJDO file. The reader is referred to [Signsoft 2001] for further details of the relationship mapping process.

4.3 Inheritance

The JDO specification allows a persistent-capable class to specify the name of its persistent-capable superclass, if any, in the JDO file. How the fields of such classes are mapped to table columns is left to the implementation. We have already mentioned how intelliBO stores the attributes for each class in a separate table. The relationships between the tables has to be defined in the SJDO file, using a `<superclass-mapping>` element, e.g.

```
<superclass-mapping>
  <simple-ref from-table="JDO_MO"
              from-field="MO"
              to-table="JDO_EQUIPMENT"
              to-field="MOID"/>
</superclass-mapping>
```

This is not sufficient, however. To support polymorphic references the system needs to be able to determine, efficiently, the Java class corresponding to each row in such a table. To enable this the system requires a `class-type` element in the least derived persistence capable class indicating which column in the corresponding table contains the class name for this object, e.g.

```
<class name="MO">
  <class-type table="JDO_MO" field="TYPE"/>
  ...
```

The `TYPE` column in this example should contain the fully qualified Java class name, excluding the file extension.

4.4 Exchange Operators

In some cases the natural representation of a class attribute is very different to the column type of the database table. A good example is the treatment of enumerated types in NETeXPERT. They are stored as strings within the database, but it may be more natural to map these to elements of enumerated type classes. To deal with such issues intelliBO introduces exchange operators. These are classes that implement the `ExchangeOperation` interface.

```
public interface com.signsoft.ibo.jdbc.ExchangeOperation {
  //  Initializes the exchange operation.
  void init(java.util.Properties p)
  //  Intercepts a database read field call.
  Object fromDatabase(Object o)
  //  Intercepts a database write field call.
  Object toDatabase(Object o)
}
```

An exchange operator can be set on a per-attribute basis. The `toDatabase` and `fromDatabase` methods are used to convert the data on the way to and from the database. The SJDO file indicates which fields have exchange operators associated with them, and the parameters to pass to the `init` method where applicable. For example, an `administrativeState` attribute in an `equipment` class might contain an SJDO entry similar to the following

```
<field name="administrativeState">
  <field-mapping table="JDO_EQUIPMENT"
                 field="ADMINISTRATIVESTATE"/>
```

```
  <exchange-operation
    class-name="administrativeStateEnum$ExchangeOperation">
  </exchange-operation>
</field>
```

The current implementation of exchange operators in intelliBO is flawed in one respect. The query language is not aware of their presence, and so queries involving attributes on which query operators are defined do not work properly at present.

4.5 Key Generation

Every managed object in the OSI database must have a unique key. Consider the scenario where the IDEAS server and a JDO implementation are both trying to create objects. The JDO code might look like the following:

```
tx.begin();
equipment e = new equipment(...);
pm.makePersistent(e);
tx.commit();
```

At what point is a new key allocated for this object, and how do we ensure it doesn't conflict with a key allocated by the IDEAS server or another JDO-based application? One approach involves the user simply assigning values to an object's primary key attributes prior to persisting the object. But this just pushes the problem onto the user. A JDO implementation can also be configured to automatically generate a primary key at the time a persistent object is first committed. This usually involves using a database table to record a "high-water" mark for allocated keys. The OSI system uses the `SURROGATE` table for this purpose. To allow the JDO and OSI implementations to coexist peacefully, it is important that they both use the same mechanism and database table. Fortunately the intelliBO system allows the details of the key generation mechanism to be parameterized by the user. The following element, when included in the JDO entry for a base class, tells the JDO system to use the `SURROGATE` table to allocate managed object keys.

```
<key-generator
    class-name="com.signsoft.ibo.jdbc.key.SequenceTable">
  <param key="table-name" value="SURROGATE"/>
  <param key="id-field" value="ID"/>
  <param key="name-field" value="TID"/>
  <param key="new-connection" value="false"/>
  <param key="pre-allocation-size" value="1"/>
  <param key="sequence-name" value="1"/>
  <param key="field-name" value="id"/>
</key-generator>
```

For applications where a lot of objects are created, for example a bulk importer, the `pre-allocation-size` entry should be increased to avoid accessing this table for every created instance.

4.6 The Connection Object

For reasons that will become clearer in Section 5.4, sometimes it is convenient to be able to escape from the confines of JDO and use raw JDBC calls. However, it is important that such calls take place within the *same* transaction as the JDO code. To achieve this requires access to the `java.sql.Connection` object being used by the JDO system for the current transaction. Unfortunately the JDO specification does not prescribe any standard mechanism for accessing this object. Indeed for some JDO implementations such an object may not even exist. Fortunately the intelliBO system does allow access to the underlying connection object via the following code sequence.[4]

```
com.signsoft.ibo.jdbc.rm.ResourceManager rm =
  ((com.signsoft.ibo.jdbc.JDBCPersistenceManager)pm).currentResourceManager();
java.sql.Connection con =
  rm.getConnection(rm.DEFAULT_DATA_SOURCE_NAME, pm);
```

[4]This feature is not publicly documented, and the details may alter in the future.

4.7 SQL Queries

The JDO standard defines JDOQL as a query language. This language allows queries to be expressed using Java field names and expression syntax. In some cases it can be convenient, and much more efficient, to express queries using native SQL syntax. IntelliBO supports such queries and Section 5.10 describes how this feature is used to implement relations in the OSI2JDO translator. The general idea is quite simple. A query is constructed using a SQL expression as filter, together with an additional parameter indicating that this filter is written in SQL, rather than JDOQL. However, there are various additional constraints on the form of this filter, primarily to ensure that the corresponding result set contains enough details to allow the persistent objects to be constructed. In particular, all fields of the default fetch group should be part of the result set, as otherwise these attributes will never be retrieved.

5. BRIDGING THE GAP

5.1 MO classes

We would like to model each managed object class by an equivalent JDO class. Generating a Java class definition for each MO class is fairly straightforward. But to make such a definition into a JDO class requires us to generate JDO and SJDO mapping files as well. These, in turn, require us to specify which table holds the data for each MO class. For example, suppose we constructed the `MO` and `equipment` classes from the managed object database:

```
public class MO {
  public int id;
  public String name;
  ...
}

public class equipment extends MO {
  public administrativeStateEnum administrativeState;
  public location locationName;
  public operationalStateEnum operationalState;
  ...
}
```

To add JDO support to such classes we would need tables of the form

JDO_MO "Table"						
MO	CLASS	NAME	PARENT	MOM	...	TYPE
87	41	XC1000-1	0	0		OsiDemo.XC1000
89	41	DFW_XC1000	0	0		OsiDemo.XC1000

and

JDO_EQUIPMENT "Table"				
MO	ADMINISTRATIVE STATE	EQUIPMENTID	LOCATION NAME	OPERATIONAL STATE
87	unlocked	XC1000-1	394	enabled
89	unlocked	DFW_XC1000	392	enabled

Unfortunately the data for every class is distributed amongst the `MO` and `MOATT` classes. However, a view behaves in many respects like a table, and so we can satisfy the JDO requirements by defining a suitable database view for each MO class. Inheritance complicates the situation slightly. For example, in the `JDO_EQUIPMENT` view we must include a row for every object of the `equipment` class. But we must also include rows for managed objects whose classes are derived from this class. Fortunately, Oracle's **connect by** clause provides a simple mechanism to express this requirement. The `JDO_EQUIPMENT` view could be defined by a statement similar to the following:

```
create or replace view JDO_EQUIPMENT as
select MO.MO MOID,
       StrValue(MO.MO, 1000) ADMINISTRATIVESTATE,
       LowIntValue(MO.MO, 1022) LOCATIONNAME,
       StrValue(MO.MO, 1026) OPERATIONALSTATE
```

```
 from MO
where MO.CLASS in (select CLASS from CLASS
                   start with CLASS = 6 // The equipment class
                   connect by PARENT = PRIOR CLASS
);
```

Note that the view only contains columns for the non-structured attributes declared in this class; the other attributes are handled outside the JDO world for reasons that will be explained shortly. The `StrValue` and `LowIntValue` are helper functions. The expression `StrValue(MO.MO, 1000)` retrieves a row from the `MOATT` table where the `MOID` column has the value `MO.MO`, the `ATTRID` column has the value `1000`, and the `COMPID` column is `NULL`. If such a row exists then the value of the `STRVALUE` is returned; otherwise the function returns `NULL`. The other helper functions have a similar semantics, each returning a different column in the row.

The JDO implementation is quite happy to treat a view as if it were a table. For row retrieval this is just what we want. However, the JDO code will also try to insert, modify and remove rows from the view as if it was a table as well, and this will fail unless we supply the appropriate triggers to redirect these operations to the underlying tables. We return to this issue in Section 5.9.

The table, or view, used for the base of an inheritance hierarchy must contain a column indicating the Java class to use for each instance. The `MO` table does not contain such a column and so we must create a view that does. To do this we make use of an auxiliary table, `JDO_CLASSTYPE` that is constructed during the translation process. This table contains a map from class keys to fully qualified Java class names. The `JDO_MO` view can then be defined by

```
create or replace view JDO_MO as
select MO.MO MO,
       MO.CLASS CLASS,
       MO.NAME NAME,
       DECODE(MO.PARENT, 0, NULL, MO.PARENT) PARENT,
       DECODE(MO.MOM, 0, NULL, MO.MOM) MOM,
       MO.REPORTEDAS REPORTEDAS,
       JDO_CLASSTYPE.TYPE TYPE
  from MO, JDO_CLASSTYPE
 where MO.CLASS = JDO_CLASSTYPE.CLASS;
```

One slight drawback of this approach is that the package name used in the translation process is now embedded in the database, preventing two separate uses of the tool targeting different package hierarchies.

5.2 Simple attributes

There is a direct mapping between most simple attributes and their representation in the corresponding Java classes. For example, if an MO class has an attribute of type `String` then you would expect the constructed Java class to have a field with the same name, also of type `String`. In some cases it is more natural and convenient to rename some of the types. For example, an attribute of type `Integer32` is mapped to a field with type `Integer`. This avoids creating unnecessary classes and lots of tedious conversions. Java does not have any unsigned types, and so the conversion process creates `UnsignedInteger32` and `UnsignedInteger64` classes to represent attributes with these types.

The enhancement process affects how attributes are manipulated in the Java classes. Consider a simple attribute `attr` of type `Integer`. Prior to enhancement the attribute can be manipulated just like any other non-persistent attribute. However, clearly some magic must take place whenever such a field is accessed, for example to retrieve the value from the database, or mark the field when its value is changed so the new value is saved on a commit. The enhancement process defines "hidden" getter and setter methods for each such attribute, and it is these methods that are responsible for the magic. To ensure that no code can bypass these methods, which would be disastrous, the attributes themselves are changed to `private` scope. Any direct field accesses in the code being enhanced are replaced by calls to the corresponding getter and setter methods.

Any user code directly accessing the `attr` field would fail to work when using the enhanced version of the class as this field would now be private. The user code could also be enhanced, and this would fix the problem by replacing the direct field accesses by calls to the mediating methods. But this is a relatively tedious and time-consuming process. An alternative strategy defines public getter and setter methods in

the class, making the attributes private. Such encapsulation is often encouraged anyway, but in this case it has the additional advantage of isolating the user's code from the enhancement process. The prototype translator adopts this strategy.

The `CLASSATTR` table contains a `MANDATORY` column which indicates which attributes must be specified for an instance of the class, and which are optional. This property affects the mapping in two ways. First, we must ensure that any change to a mandatory attribute does not replace it by a "null" value. The setter methods perform this check for mandatory arguments. The constructor for the class also takes the mandatory arguments as parameters to ensure that they are initialized correctly. A mandatory attribute can also be represented by a simpler type in some cases. Consider an attribute of type `Integer32`. As mentioned above, this would normally be mapped to a field of type `Integer` to allow for null values. But if this attribute is mandatory for a class then we can map the attribute to an `int` which is more efficient and convenient to use.

5.3 Enumerations

Consider a type such as

```
AdministrativeState =
  Enumerated{"locked"(0),"unlocked"(1),"shuttingDown"(2)}
```

As described in Section 3.2, values of this type are stored in the database as strings. For each such type we need to define a corresponding Java class. For efficiency reasons we would like to ensure there is exactly one instance of this type for each enumeration value. A combination of constructor hiding and the provision of a `readResolve` method ensures this. For each enumeration class we also provide an ExchangeOperation class which is used to convert to and from the underlying database representation. So the `AdministrativeState` type would be converted into a Java class similar to the following.

```
public final class AdministrativeState
    implements java.io.Externalizable {
  private int value;

  private AdministrativeState(int value) { this.value = value; }
  private AdministrativeState() {} //  Required for externalization

  public static final AdministrativeState locked
    = new AdministrativeState(0);
  public static final AdministrativeState unlocked
    = new AdministrativeState(1);
  public static final AdministrativeState shuttingDown
    = new AdministrativeState(2);

  public static AdministrativeState get(int value) {
    switch(value) {
      case 0: return locked;
      case 1: return unlocked;
      case 2: return shuttingDown;
      default:
        throw new IllegalArgumentException();
    }
  }

  private Object readResolve() { return get(value); }
  ...
  public static class ExchangeOperation
    implements com.signsoft.ibo.jdbc.ExchangeOperation {
    ...
  }
}
```

Any attribute with this type would have an `<exchange-operation>` element in the SJDO file containing this attribute.

5.4 Record types

The cases considered so far both have the property that an attribute can be modeled by a single row in the `MOATT` table. A Java developer will expect the type `RT`, from Section 3.3, to be modeled as a class. The JDO Object Model distinguishes between first and second class objects. A *first class* object is a persistence-capable class that has JDO identity, i.e. it has some form of key to identify each instance. A *second class* object does not have its own JDO identifier and therefore cannot be referenced by multiple objects in the data store; it is always associated with a first class object. Second class objects are stored as values along with the first class object that refers to them. Second class objects must track changes to themselves and notify their first class object that they have been changed so that their new state gets propagated back to the database. This is done by calling the method `jdoMakeDirty` on the first class object.

This raises the question of whether types such as `RT` should be modeled by first or second class objects. As the attribute values cannot be shared between MO instances, this suggests they could be modeled most efficiently by second class objects. The main drawback of this decision would be the requirement to call `jdoMakeDirty` whenever they were altered.[5]

Irrespective of whether a type such as `RT` is implemented using first or second class JDO objects, the mapping function will require us to specify a table, or view, containing instances of this type. But how many `RT` instances are there in a database? Note that the `MOATT` table does not have a `TYPE` column. So to find all the rows containing a string representing field `a` of an instance of the `RT` type involves a non-trivial query. One strategy involves constructing an auxiliary table recording which attributes each structured type occurs in, and the `COMPID` prefix of each occurrence. In the case of simple record types such a table is straightforward to construct, containing entries like

ATTRID	TYPEID	COMPID_PREFIX
1008	104	
2916	389	3.
2996	781	0.0.

Unfortunately, such types can also occur inside arrays, where the COMPID prefix cannot be determined statically. In such cases we could resort to pattern matching, with entries such as

ATTRID	TYPEID	COMPID_PREFIX
...	...	...
2999	785	_____.
2999	101	_____.0.

where "_" represents any digit. The situation becomes even more complex, and inefficient, when `Any` types are considered. An initial version of the prototype explored this route, with views using stored procedures to perform the pattern matching. However, the resulting views were very inefficient and complex, and it was difficult to see how such an approach would lead to a viable implementation.

What other options are open to use? There are clearly ways in which the database tables could be altered to simplify our task. The `MOATT` table could include a `TYPEID` column for example. Or this table could be split into multiple tables in various ways. But all these require changes to the NETeXPERT software, in some cases of a non-trivial nature. Although perhaps worth considering in the longer term, a less obtrusive approach was required for the prototype.

One possibility is to build such attributes ourselves from the raw data in the MOATT table. Obviously such attributes would no longer be directly under the control of JDO, and so couldn't be used in JDO queries for example. But we could still tie them into the JDO transaction mechanism, so for the most part users would be unaware of their different status, viewing them just like second-class JDO objects.

There are many ways of building attribute values from the `MOATT` rows.

(1) For each structured attribute we could define a persistent field containing the collection of `MOATT` rows representing this attribute. If this field was in the default fetch group them the `jdoPostLoad` method could construct the attribute value. Otherwise the code to assemble the value would be encapsulated inside the getter method for the attribute. This strategy is likely to require making separate database

[5]Unfortunately many JDO implementations only support first class objects at the present time.

queries for each attribute. Furthermore, we will need to sort the `MOATT` collection as there is no standard mechanism to order a 1-N relation in JDO, unlike the situation for JDO queries.

(2) We could have a single persistent field holding the `MOATT` rows for all the structured attributes of this object. This would avoid repeated database queries, and we could still assemble the individual attribute values lazily. However the initial fetch may be quite expensive and require fetching many rows that are never used.

(3) We could perform a separate JDO query in the `jdoPostLoad` method to load in all the `MOATT` rows and assemble the structured attribute values. In this approach JDO is completely unaware of the structured attribute fields.

Each of these strategies implies a corresponding action to be performed when committing an object containing structured attributes. The `jdoPreStore` method can be used to make the required alterations. Unfortunately the JDO specification does not specify clearly what operations are allowed to take place during a `jdoPreStore` method. For example, to what extent is the method allowed to alter the database? This makes it difficult to write code that doesn't accidentally exploit the implementation behaviour of a particular JDO implementation. A further disadvantage of each of these approaches is that they treat the `MOATT` rows as first-class JDO objects. Maintaining the identity of these objects, with the caching this entails, is likely to be quite expensive. As we are not really exploiting the "first-class" properties for these objects, a lighter weight alternative seems preferable. A more primitive interface such as JDBC would seem ideal for this purpose as long as it could be used within the same transactional context as the JDO session.

As described in Section 4.6, the intelliBO implementation exposes the current database connection via a private interface. We can use this to load the `MOATT` rows for an object without incurring the overhead of making each row a first-class JDO object. We still have to decide on when such rows are loaded and saved. The strategy adopted in the prototype is to load each attribute lazily. The first time a structured attribute is referenced, via its getter method, the system uses JDBC to load the corresponding rows from the `MOATT` table. These are then assembled into a structured value, cached, and the value returned to the caller. When an object is being saved to the database, the `jdoPreStore` method uses JDBC to save the rows corresponding to any altered structured attributes. The `jdoMakeDirty` method cannot be used to mark such a "second-class" object as being dirty as the JDO implementation is unaware of these attributes. However, the setter methods set the dirty bit for such attributes, and a "no-op" setter call will be sufficient to mark the attribute as dirty.

Each class representing a structured attribute type implements `jdoLoadType` and `jdoStoreType` methods. The `jdoLoadType` static method can be used to create an instance of the class from a JDBC result set containing the `MOATT` rows representing this value. The `jdoStoreType` uses a prepared statement to batch up and add the appropriate rows to the `MOATT` table. The `jdoPreStore` callback is responsible for calling `jdoStoreType` on every attribute that has been marked as dirty.

In some cases a caller may know, in advance, that the values for a collection of structured attributes may all be required. It may be more efficient to load all the rows for these attributes in a single database query, rather than loading them incrementally as each attribute is referenced. The prototype implementation provides a simple mechanism for pre-fetching a set of attributes. This acts as a "no-op" semantically, but may improve the efficiency of the application in some cases.

The asymmetry between the treatment of structured and unstructured attributes is regrettable, but perhaps unavoidable. Even if we could represent the structured attributes by second class objects then we would still have to inform a first class object whenever an embedded second-class object changed. In most situations the fact that some objects are under the direct control of JDO whereas other objects are only indirectly controlled by it should not be noticeable.

5.5 Arrays

Once we have made the choice about how to process structured attributes then arrays are fairly straightforward. We just need to decide on the Java representation for them. As they are a) sparse and b) of varying size, this suggests that a representation based on a map may be appropriate. We construct a separate class for each sequence type rather than just using something like TreeMap. This preserves more type information at the expense of additional classes.

The prototype implementation treats arrays atomically. Any change to an array element requires the whole array to be written back to the database. For many arrays this is probably adequate. However, for large arrays a more incremental approach could be developed. Similarly for large dense arrays we might be

better off using a vector. There are clearly many trade-offs here, depending on the type of the array and its particular usage pattern. Ultimately such implementation choices will need to be guided by the user, perhaps by using XML configuration files to guide the translation process.

Sometimes arrays are used to represent sets. For example, an `equipment` class might contain an attribute `contactNames` of type `SequenceOfMO`. Suppose we could be sure that the order of the contacts in this sequence was of no importance, i.e. the user just wanted to associate a set of contacts with each item of equipment. In such cases we could model the attribute by a JDO relation. This would be more powerful and much more convenient to use. However, JDO relations are unordered, and so this mapping would not be appropriate if we wanted to rank the contacts in order of importance, or if the contact index in the sequence conveyed some additional information. Without additional hints from the user we cannot exploit such modelling choices.

5.6 Type aliases

Unlike C++, Java does not support a "typedef" mechanism. This leaves us with two choices. We could create new types that extend, or encapsulate, the existing types. Unfortunately, subclassing isn't always possible as some classes are declared as "final", e.g. `String`. Encapsulation also has its drawbacks. The alternative is to replace all occurrences of the derived type by the implementation type. This approach throws away some type information, but seems the best option in this situation.

5.7 Choice types

Consider a type such as

```
ObservedValue = Choice{integer Integer, real Real}
```

A natural encoding of such a type translates `ObservedValue` into an abstract base class, with each choice mapping to a derived class. Java's **instanceof** operator can then be used to determine which branch of the choice is represented by a particular `ObservedValue` instance. The prototype translator uses this approach, nesting the choice classes within the abstract base class to minimize namespace pollution. The `ObservedValue` type would therefore be mapped to code similar to the following.

```
public abstract class ObservedValue {
  public static class IntegerChoice extends ObservedValue {
    public Integer integer;

    public IntegerChoice(Integer integer) {
      this.integer = integer; }
    ...
  }

  public static class RealChoice extends ObservedValue {
    public Real real;

    public RealChoice(Real real) {
      this.real = real; }
    ...
  }
}
```

The translator drops the "Choice" suffix when no naming confusion would arise from its omission. A frequent idiom in NETeXPERT uses the `Null` type as one of the choices to indicate an optional value. The system optimizes such cases to avoid generating multiple `Null` instances.

5.8 Any type

Section 3.7 describes the encoding of `Any` types. Note that the name that appears in the `MOATT STRVALUE` column may not be the same as the Java class used to represent this type. There are numerous reasons for why these names may differ, and these are discussed in Section 6.1. At this stage it is sufficient to note the need for a map from OSI names to Java class names.

A static type map initially maps strings representing OSI class names to their corresponding Java class name. When an Any type is encountered that uses this class then Java reflection is used to find the `jdoLoadType` method for the corresponding Java class. The map is updated by replacing the Java class name by the Method object. This object is then invoked with the appropriate arguments to construct an object from the `MOATT` rows in the result set. Here is a sketch of the `jdoLoadType` method for the `Any` class, excluding any error checking:

```
public class Any
  ...

  public static Object jdoLoadType(
      int attrid, String prefix, ResultSet rs, PersistenceManager pm) {
    if (attrid != rs.getLong(1)) return null;
    String compid = rs.getString(2);
    if (compid.equals(".")) compid = "";
    if (!compid.startsWith(prefix)) return null;
    String classNameString = rs.getString(3);
    String osiName = // Strip off 0< and > brackets
      classNameString.substring(2, classNameString.length()-1);
    Object o = typeMap.get(osiName);
    if (o instanceof String) { // Find jdoLoadType method for class
      String className = (String)o;
      Class clasz = Class.forName(className);
      Method m = clasz.getDeclaredMethod("jdoLoadType",
        new Class[] { int.class, String.class,
                      ResultSet.class, PersistenceManager.class });
      typeMap.put(osiName, m); // Cache result for next time
      o = m;
    }

    Method m = (Method)o;

    if (prefix.length() == 0)
      prefix = classNameString;
    else
      prefix = prefix + "." + classNameString;

    rs.next();
    return m.invoke(null, //  It is a static method
      new Object[] { new Integer(attrid), prefix, rs, pm });
  }
}
```

5.9 Triggers

Although views allow us to massage the OSI schema into a form that is more amenable to processing by a JDO implementation, there is a catch. Whenever an object is made persistent, is updated, or deleted, the JDO implementation will try to perform the corresponding database operation on the view. The database system needs to be told how such operations should be translated into equivalent operations on the supporting tables. We do this by defining INSERT, UPDATE and DELETE triggers on all the managed object views. Take the `equipment` class as an example. The view constructed for this class was described in Section 5.1. When the JDO system tries to insert a row into this view we must translate this operation into a sequence of inserts into the MOATT table, one for each attribute. The trigger for achieving this would have the following form.

```
create or replace trigger JDO_EQUIPMENT_I
instead of INSERT on JDO_EQUIPMENT
for each row
```

```
begin
  insert into MOATT
    ( MOID, ATTRID, STRVALUE, SYNTAX )
    values ( :NEW.MOID, 1000, :NEW.ADMINISTRATIVESTATE, 3 );
  insert into MOATT
    ( MOID, ATTRID, LOWINTVALUE, SYNTAX )
    values ( :NEW.MOID, 1022, :NEW.LOCATIONNAME, 6 );
  ...
end;
```

When the JDO implementation creates an instance of a derived class it has to insert a row into each view in the inheritance chain. So creating an instance of an `XC1000` class might result in rows being inserted in the `JDO_XC1000`, `JDO_EQUIPMENT` and `JDO_MO` views. The trigger for the `JDO_MO` view is responsible for calling the `netx_createmo` procedure.

```
create or replace trigger JDO_MO_INSERT
instead of INSERT on JDO_MO
for each row
declare ...
begin
  select CLASS into clasz from JDO_CLASSTYPE
   where JDO_CLASSTYPE.TYPE = :NEW.TYPE;
  select NAMINGATTRIBUTE, ISAMGR into namingatt, mom from CLASS
   where CLASS.CLASS = clasz;
  netx_createmo(:NEW.MO, :NEW.NAME, clasz, NVL(:NEW.PARENT,0),
                NVL(:NEW.MOM,0), :NEW.REPORTEDAS,
                namingatt, mom, retkey, retval);
  if (retval <> 0) then raise creation_failed; end if;
exception
  ...
end;
```

Note that the insertions must be performed in the natural order, starting at the base class and finishing at the most derived class, to ensure the triggers are fired in the correct order.

Updating an attribute is handled in a similar fashion.

```
create or replace trigger JDO_EQUIPMENT_U
instead of UPDATE on JDO_EQUIPMENT
for each row
begin
  if :NEW.ADMINISTRATIVESTATE <> :OLD.ADMINISTRATIVESTATE then
    update MOATT set STRVALUE = :NEW.ADMINISTRATIVESTATE
     where MOID = :NEW.MOID and ATTRID = 1000;
  end if;
  if :NEW.LOCATIONNAME <> :OLD.LOCATIONNAME then
    update MOATT set LOWINTVALUE = :NEW.LOCATIONNAME
     where MOID = :NEW.MOID and ATTRID = 1022;
  end if;
  ...
end;
```

Deletion is even easier. The `netx_deletemo` procedure tidies up all the `MOATT` entries for an object. So as long as the deletes for the object are performed in the reverse order to the inserts then all the work can be left to the delete trigger on `JDO_MO`. The delete trigger for the `equipment` class therefore does nothing.

```
create or replace trigger JDO_EQUIPMENT_D
instead of DELETE on JDO_EQUIPMENT
for each row
begin
  return;
end;
```

The delete trigger on the JDO_MO view actually deletes the object.

```
create or replace trigger JDO_MO_DELETE
instead of DELETE on JDO_MO
for each row
declare ...
begin
  netx_deletemo(:OLD.MO, retval);
  if (retval <> 0) then raise deletion_failed; end if;
exception
  ...
end;
```

5.10 Relationships

There are two aspects of relationships we must consider when generating classes for managed objects. Given an object we must be able to determine which other objects it is associated with by a specified relation. We must also be able to add and remove associations between objects. We consider these two cases separately.

The OSI2JDO translator generates a `Relation` class to represent relations. An instance of this class can be obtained via a static `find` method given the relation's name. The `MO` class defines a `get` method with the signature

```
java.util.Collection get(Relation r)
```

If at most one object is expected to be associated with this object by the specified relation then the `get1` method can be used. It raises an exception if two or more related objects are encountered.

```
MO get1(Relation r)
```

The implementation of these methods is slightly tricky. The normal JDO query mechanism can filter a set of objects based on the values of selected attributes. However, when querying for the targets of a managed object relation the information that determines which objects to select is contained in a separate table. One strategy might be to make the rows in the `MORELATE` table JDO objects in their own right. We could then add an additional attribute to each managed object that was mapped to the relevant subset of the `MORELATE` objects, using a JDO relation. The `get` methods could then filter such sets to select the objects for a particular relation. However, this all starts to sound fairly expensive. Given keys for a managed object and a relation it is easy to write a SQL query that returns rows containing a key for each matching managed object. We could use separate queries to map these keys into managed objects, but it would be more efficient, and neater, if the JDO implementation could do this for us. Section 4.7 described how SQL can be embedded within JDO queries, and this mechanism is used to implement the `get` methods efficiently.

There are a variety of mechanisms we could introduce to add and delete associations between objects. Ideally we would like a mechanism that could enforce typing and multiplicity constraints. However, there is a limit to how far one can go in this direction given the schema we have to work with, and the need to interoperate with NETeXPERT. The approach adopted by the prototype translator uses the presence of relational attributes to trigger the generation of methods to manipulate the corresponding relations. This is best illustrated by an example.

In Section 3.8 we introduced an `Application` class with a `Manglers` relational attribute associated with the `mangledBy` relation. On discovering this attribute the translator generates the code

```
private java.util.List mangledBy;
public  java.util.List getMangledBy() { return mangledBy; }
```

Note that the attribute is only used to trigger the generation of this code. It's name is not used. The corresponding SJDO file establishes this field as one end of a many-to-many relationship.

```
<field name="mangledBy">
  <many-to-many-mapping>
    <ext-ref
      from-table="JDO_APPLICATION"
      from-field="MOID"
      lookup-table="JDO_MANGLEDBY_REL"
      lookup-from-field="MO1"
      lookup-to-field="MO2"
```

```
        to-table="JDO_MO"
        to-field="MO"/>
      <result-type>
        java.util.ArrayList
      </result-type>
    </many-to-many-mapping>
  </field>
```

The JDO_MANGLEDBY_REL view takes care of the fact that both directions in an association must be considered. It is defined by

```
create or replace view JDO_MANGLEDBY_REL as
    select MO1, MO2 from MORELATE where KIND = mangledBy key
  union
    select MO2, MO1 from MORELATE where KIND = mangles key
```

One of the advantages of using JDO is that it hides all the messy details of manipulating such relationships. For example, given an application object A, the expression A.`getMangledBy()` will return the collection of objects, typically developers, that are currently mangling this application. Furthermore, any changes to this collection will be transparently written back to the database when the transaction is committed. The JDO implementation will generate INSERT and DELETE statements in such cases, and the triggers on the relation views make the appropriate modifications to the `MORELATE` table.

Relational attributes of type `MO` are treated slightly differently. For example, the presence of the `MangledApp` attribute in the `Developer` class would result in the following code being generated.

```
private java.util.List mangles;

public MO getMangles() {
  if (mangles.size() > 1) throw new IllegalStateException();
  return (mangles.size() == 0) ? null : (MO)mangles.get(0);
}

public void setMangles(MO mangles) {
  this.mangles.clear();
  this.mangles.add(mangles);
}
```

The `mangles` field is still treated as a many-to-many mapping. However, the type of the relational attribute is taken as a hint that the multiplicity of this relation is expected to be 0..1. If, at run-time, this turns out not to be the case then the `getMangles` method raises an exception.

The translator has to handle the `Contains` and `Manages` associations specially. One reason is that there are no relational attributes for these relations, and so the mechanism described earlier would not generate any code for them. The state for these relations is also duplicated in the `MO` table. The `MO` class seems a natural place to define getter and setter methods for manipulating the `ContainedIn` relation. Every class which isn't a managed object manager also has getter and setter methods for the `ManagedBy` relation. Unfortunately the ability to manage objects is orthogonal to the class hierarchy of managed objects, and so there is no convenient place in this hierarchy in which to define these methods just once. We may even need to define disabling versions of these methods if a superclass is a not a managed object manager, and a subclass is.

The attributes supporting these methods are declared in the `MO` class. JDO propagates any changes to these attributes to the `JDO_MO` view. Triggers on this view then ensure that both the `MO` and `MORELATE` tables are suitably updated.

```
create or replace trigger JDO_MO_UPDATE
instead of UPDATE on JDO_MO
for each row
begin
  if :NEW.PARENT <> :OLD.PARENT then
    update MO set PARENT = :NEW.PARENT where MO = :NEW.MO;
    delete from MORELATE where MO1 = :NEW.MO and KIND = 3;
    insert into MORELATE values (:NEW.MO, 0, :NEW.PARENT, 0, 3);
```

```
  end if;
  if :NEW.MOM <> :OLD.MOM then
    update MO set MOM = :NEW.PARENT where MO = :NEW.MO;
    delete from MORELATE where MO1 = :NEW.MO and KIND = 5;
    insert into MORELATE values (:NEW.MO, 0, :NEW.MOM, 0, 5);
  end if;
end;
```

6. A PROTOTYPE IMPLEMENTATION

In this section we describe some of the finer points of the prototype OSI2JDO translator.

6.1 Namespaces

Name translation is one of the more tedious aspects of mapping OSI classes to JDO classes. We would like to map OSI class names to Java class names, for example. However, a valid OSI class name may not satisfy the rules for Java identifiers and so the name must be altered in some cases. We must also treat class and attribute names differently. Although Java is case-sensitive, Java classes must be stored in files with matching names. As some file systems, e.g., Windows, are case-insensitive, class names may clash when they only differ in case. The translator must therefore maintain information about what names have currently been allocated in each namespace, and rename identifiers where necessary. This process should minimize any changes to make it easy to relate the Java names back to the OSI names. The translator also emits a special `osi.name` JavaDoc tag when the names differ to make the connection clearer. OSI allows type names to be the same as managed object class names. To avoid renaming in such cases the types are translated into a separate package. In the current implementation this has the secondary advantage that the MO classes are JDO-capable, whereas all the classes in the `Types` subpackage are not.

The system constructs various views and triggers for the MO classes it encounters, and we need to generate names for these elements as well. Oracle and Java have different sets of rules governing valid identifiers and reserved words. The translator therefore maintains separate namespaces for these.

6.2 JDO Use in the Translator

The translator needs to access various database tables during the translation process. The prototype uses JDO to simplify this task. Some tables are naturally mapped to a class hierarchy rather than a single class. The `ABSTRACTTYPE` table is an obvious example. The translator has separate classes for translating records, enumerations, primitive types, and so on. As described in Section 4.3, the intelliBO implementation requires a column in the base class table containing the class of the instance corresponding to this row. The `ABSTRACTTYPE` table has no such column, but we can create a view that does. The view can also hide the messy details associated with the use of segments in this table.

The first step is to define a stored function that can classify an `ABSTRACTTYPE` row into a Java class name.

```
create or replace function AbstractTypeClass
  (reference in varchar2, notation in varchar2)
return varchar2 is
begin
  if (INSTR(notation, 'Sequence{') = 1)
    then return 'com.agilent.osi.JDO.RecordType';
  elsif (INSTR(notation, 'Enumerated{') = 1)
    then return 'com.agilent.osi.JDO.EnumeratedType';
  elsif (INSTR(notation, 'Choice{') = 1)
    then return 'com.agilent.osi.JDO.ChoiceType';
  elsif (INSTR(notation, 'SequenceOf ') = 1)
    then return 'com.agilent.osi.JDO.ArrayType';
  elsif (reference = 'MO')
    then return 'com.agilent.osi.JDO.MOType';
  elsif (reference = 'Any')
    then return 'com.agilent.osi.JDO.AnyType';
  elsif (notation = reference)
    then return 'com.agilent.osi.JDO.PrimitiveType';
  else return 'com.agilent.osi.JDO.TypeDef';
  end if;
```

```
end;
```

A second function is used to concatenate all the notation segments for a particular type.

```
create or replace function Notation_Suffix(ty in number)
  return VARCHAR2
is
  cursor notation_cursor is
    select NOTATION from ABSTRACTTYPE
     where TYPE = ty and SEGMENT > 0 order by segment;
  notation_val notation_cursor%rowtype;
  result varchar2(32767);
begin
  for notation_val in notation_cursor loop
    result := result || notation_val.notation;
  end loop;
  return result;
end;
```

With these building blocks in place we can construct a view to use for the `AbstractType` base class.

```
create or replace view JDO_ABSTRACTTYPE as
select TYPE, REFERENCE,
  NOTATION || Notation_Suffix(TYPE) NOTATION,
  AbstractTypeClass(REFERENCE, NOTATION) CLASS
from ABSTRACTTYPE where SEGMENT = 0;
```

Each subclass also requires a view, but such views are trivial to construct as they only requires a single column, e.g.

```
create or replace view JDO_ENUMERATEDTYPE as
select TYPE from JDO_ABSTRACTTYPE
 where CLASS = 'com.agilent.osi.JDO.EnumeratedType';
```

Note that these views are only used at translation time, and therefore any potential inefficiencies arising from their use are of lesser importance. The tables are also never modified by the translator, and so such views require no triggers, unlike the managed object case.

6.3 Lazy File Writers

Ideally a translator should be incremental, processing just those classes and types that have changed since the last time the program was run. Unfortunately, it is not clear how to determine this without additional support from the NETeXPERT system. The prototype translator therefore processes all the classes and types in the database whenever it is run. Most of the files produced by this process will be identical to the previous versions. But if the translation process rewrites each file then the timestamp will change, forcing a lot of unnecessary recompilation and reenhancement. To avoid this situation the prototype makes use of a lazy file writer class. The idea is very simple. Instead of writing to a file, a lazy file writer writes to a buffer in memory. When the object is closed the contents of this buffer are compared to the old contents of the file on disk. If no file currently exists, or the contents differ, then the buffer is written to disk. This strategy prevents unnecessary timestamp changes, and significantly reduces the overall translation time in many cases.

6.4 Ant Tasks

The prototype uses Ant[ANT] to choreograph the various stages involved in the translation process. One advantage of Ant is the ease in which it can be extended using custom tasks. A task was developed to package up calls to the intelliBO enhancer. The enhancer can either process an entire folder, or can just enhance the files passed on the command line. The Enhancer task determines which files require enhancement, based on the time stamps of the class files and the configuration files, and then passes a list of selected files to the enhancer. Such an approach is not entirely safe, as changes to a base class may affect the enhancement of a subclass. The incremental enhancement process can therefore be disabled when required. Ideally the enhancer itself should be smarter as it has easier access to the information necessary for it to be both incremental and safe.

6.5 The `toString` Method

In addition to generating accessor methods for the class data, a translator can easily construct other utility methods as well. To illustrate this process, the translator defines a `toString` method on managed objects and attribute types. This has the additional advantage of aiding debugging. Managed object instances may refer to other managed objects, via the class attributes, and so any traversal operation must guard against cyclic references. A simple solution would be to pass a depth limit to the function, stopping the recursion when this depth is reached. However, Java's `toString` method does not take such a parameter. One technique for limiting the recursion, without requiring additional parameters, uses thread-local storage. The idea is simple. The "top-level" `toString` call initializes a thread-local variable with a depth-limit. Any recursive calls can then access the current value of this variable to determine the nesting depth, and the thread-local nature of this variable prevents multiple calls in separate threads from interfering with each other. The translator's implementation of `toString` uses this approach. The method also uses the pre-fetch mechanism to load structured attributes, avoiding unnecessary round-trips to the database.

6.6 User Code

In the simplest scenario the translator would generate packages containing the Java versions of the OSI classes, and types. These would then be compiled, enhanced, and stored in a JAR file. However, it is difficult to exploit the inheritance in these classes using such a technique, and user code would quickly become littered with **instanceof** tests. Users, in an attempt to avoid this situation, may be tempted to edit the Java source code generated by the translator, adding in class-specific methods to perform various tasks. But this approach creates its own problems as these changes will be overwritten when the translator is next invoked. Manually reapplying these changes each time is clearly unacceptable, and so a more automated strategy is required. The approach adopted by the prototype requires the user to write any additional code in separate "include" files. When the translator is generating code for a class it checks to see if any file with the same name exists in an include directory. If a match is found the contents are merged into the generated file. The included text is bracketed in comments reminding the user that such code should not be edited in-situ. The tags could also be used by a simple tool to merge in changes to user code without having to rerun the translator. However the enhancer would still need to be run again. The choice types require special treatment as in these cases a user may wish to add additional code to each of the choice classes as well as the outer abstract base class. The include approach is simple, avoiding the need for a "smart diff", and works well in practice. The same include path is used to locate some of the primitive classes, such as `MO` and `Any`, allowing these to be overridden by the user.

7. EXAMPLES

In this section we present a few code fragments to illustrate the general flavor of the JDO interface. Suppose we wished to display every managed object in the OSI database. This could be achieved by the following code:

```
Query query = pm.newQuery(MO.class);
results = (Collection)query.execute();
for (Iterator it = results.iterator(); it.hasNext();)
  System.out.println((MO)it.next());
```

If we wanted to restrict the output to just those objects derived from the `equipment` class, including its subclasses, and with an operational state of `active`, then the following code would suffice:

```
Query query = pm.newQuery(equipment.class,
      "operationalState == \"" + operationalStateEnum.active + "\"");
Collection results = (Collection)query.execute();
for (Iterator it = results.iterator(); it.hasNext();)
  System.out.println((equipment)it.next());
```

Given any managed object m, where `getContainedIn` returns the object m', you would expect m to be in the set of objects that m' contains. We can check that this is indeed the case for all objects in the database.

```
Relation contains = Relation.find("Contains", pm);
Query query = pm.newQuery(MO.class);
Collection results = (Collection)query.execute();
for (Iterator it = results.iterator(); it.hasNext();) {
```

```
  MO mo = (MO)it.next();
  MO cmo = mo.getContainedIn();
  if (cmo != null) {
    Collection c = cmo.get(contains);
    assert c.contains(mo);
  }
}
```

Creating new persistent objects is also straightforward. In the next example we create a new `XC1000` instance and add it to the database.

```
location DFW = (location)MO.find("DFW", location.class, pm);

XC1000 newXC = new XC1000("DFW_XC1000-1",
  administrativeStateEnum.unlocked,
  DFW,
  operationalStateEnum.enabled);

SequenceOfMO contacts = new SequenceOfMO();
contacts.constrainRangeType(contact.class);
contact DFW_Tech = (contact)MO.find("Fred_Jones", contact.class, pm);
contacts.put(0, DFW_Tech);
newXC.setContactNames(contacts);

Transaction tx = pm.currentTransaction();
tx.begin();
pm.makePersistent(newXC);
tx.commit();
```

Updating a structured attribute is slightly more cumbersome than manipulating a non-structured one as we have to remember to mark the attribute as "dirty".

```
tx.begin();
XC1000 testXC = (XC1000)MO.find("DFW_XC1000-1", XC1000.class, pm);
SequenceOfMO contacts = testXC.getContactNames();
contacts.put(1, MO.find("Bill_Smith", contact.class, pm));
textXC.putContactNames(contacts); // Marks the attribute as dirty
tx.commit();
```

Suppose we wish to manipulate the contacts and user labels for the equipment objects. Both of these are structured attributes and it may be more efficient to load these in advance of their use rather than doing this lazily. This is the purpose of the pre-fetch mechanism, as illustrated by the following fragment.

```
MO.FetchGroup fg = equipment.jdoFetchGroup(new String[] "contactNames", "userLabels");
Query query = pm.newQuery(equipment.class);
Collection results = (Collection)query.execute();
for (Iterator it = results.iterator(); it.hasNext();) {
  equipment e = (equipment)it.next();
  e.jdoPreFetch(fg);
  //  Do something using the contact names and user labels
  ...
}
```

It is simple to delete selected objects. Here is an example that also illustrates the use of query parameters to avoid rebuilding a query each time it is used.

```
Query query = pm.newQuery(XC1000.class);
query.setFilter("operationalState == state");
query.declareParameters("String state");
...
Collection results =
    (Collection)query.execute(operationalStateEnum.disabled.toString());
for (Iterator it = results.iterator(); it.hasNext();)
  pm.deletePersistent(it.next());
```

7.1 The IGP Detective

We now describe a slightly larger example built using the JDO interface. A router running an interior gateway protocol, such as OSPF, builds up a topology of the surrounding area by exchanging routing messages with its peers. The router uses this information to construct routing tables. But this data would be of more general use if it could be extracted from the network in a cost-effective fashion. There are two main strategies for obtaining details of an evolving topology. A polling interface returns the current state of the graph, including routers, interfaces, networks, and edges, on demand. An interface based on the *observer* pattern[Gamma et al. 1995] notifies a set of interested subscribers whenever the topology changes. The latter approach is usually the preferred solution. It is trivial to write an external process that uses this interface to provide a polling interface for other users. However, the opposite approach, building an observer interface on top of a polling interface, will always be very inefficient.

The topology of the network is unlikely to change very frequently, at least at the level of detail monitored by OSPF.[6] An SNMP-based observer interface will therefore place very little strain on high-end routers, where the packet-forwarding engines are typically decoupled from the management processor handling routing updates. However, low-end routers use the same processor for all their tasks. In such cases even small demands for extra work, such as the generation of additional SNMP traps, may be objectionable. This is the niche that the IGP Detective aims to fill. By eavesdropping on the OSPF signaling traffic, it can construct the same topology information as a router, and export this via polling or observer interfaces.

The IGP Detective uses Corba, rather than SNMP, to expose the topology information. For this particular task the choice is largely a matter of taste. For an observer interface the traffic volumes are sufficiently low that the encoding inefficiencies of SNMP can be safely ignored. Furthermore, using a MIB compiler to map the low-level SNMP calls into MIB-specific classes is analogous to using an IDL compiler to hide the details of IIOP. Fortunately, such differences can be easily hidden from users of the topology information, allowing Corba and SNMP-based solutions to peacefully coexist.

One potential use of the topology information is for auto-discovery, populating an OSS database with managed objects representing OSPF-aware routers and interfaces. Subsequent topology changes can then trigger corresponding changes to the state attributes of these objects. The OSI2JDO translator makes such applications almost trivial to develop. To evaluate this claim, a small program was developed to interface the IGP Detective to the OSI database. Only details of OSPF routers, interfaces, and their observed state, were to be included in the database. The edge information provided by the IGP Detective was discarded.

The first step was to define `OSPF_Router` and `OSPF_Interface` classes in NETeXPERT. They both have OSPF_Status attributes to record their current known status. On startup, all managed objects representing interfaces and routers are set to the disabled state. They are then re-enabled when the program is notified of their existence by the IGP Detective.

```
Query q = pm.newQuery(OSPF_Router.class);
Collection result = (Collection)q.execute();
for (Iterator it = result.iterator(); it.hasNext();) {
  OSPF_Router r = (OSPF_Router)it.next();
  r.setOSPF_Status(OSPF_Status.disabled);
  for (Iterator interfaces = r.contains().iterator();
       interfaces.hasNext();) {
    OSPF_Interface rif = (OSPF_Interface)interfaces.next();
    rif.setOSPF_Status(OSPF_Status.disabled);
  }
}
```

This approach is only adequate for monitoring a network with a single agent and area. What we should really do is disable just those routers in the OSPF area(s) monitored by the agent. Unfortunately the current API does not expose this information to the observer. However, if we knew the area was "`A`" then we would just replace the expression `pm.newQuery(OSPF_Router.class)` by a more refined filter such as `pm.newQuery(OSPF_Router.class, "area == A")`.

To create a new router you just need to create a new instance of the `OSPF_Router` class and make it persistent. All of the attributes of this class are declared as mandatory, and so they must all be provided as constructor arguments.

[6] Some of the traffic engineering extensions to OSPF may make topology changes much more frequent.

```
OSPF_Router or = new OSPF_Router(
  routerName(ra), area, address, type, OSPF_Status.enabled);
pm.makePersistent(or);
```

To add a new interface to this router you would construct a new instance of the `OSPF_Interface` class, declare that it is both contained in, and managed by, the specified router, and then make it persistent.

```
OSPF_Interface rif = new OSPF_Interface(
  interfaceName(ifa), address, OSPF_Status.enabled);
rif.setManagedBy(or);
rif.setContainedIn(or);
pm.makePersistent(oif);
```

Although not required in this example, deleting an interface or router is equally simple. You would use a `Query` to find all objects matching a given name, area, or status. The `pm.deletePersistent` method can then be used to delete individual objects.

One might question whether it is appropriate to store the current state of an external component as a persistent attribute in a database object. Some details of an object are more ephemeral than others. Although there are similar examples in the OSI literature, maintaining ephemeral state in the database raises the question of how such state is kept synchronized when the OSI system is restarted. It may also make some kinds of inference expensive unless extensive caching is used, but this complicates replication. The location of a device, and its current operation state, should perhaps be treated, and stored, very differently in an OSS system.

The example requires about ten lines of code to find an IGP Detective agent and subscribe to it. A similar amount of boilerplate code is required to establish a JDO connection to the OSI database. Finally, a page of code implements the call-backs required by the Observer interface, including the JDO code to create and manipulate the managed objects. Clearly a production version of such a utility would require more code to make it robust, but it gives an indication of how easy it is to write such applications given the right building blocks.

8. CONCLUSIONS

The current prototype has demonstrated some of the advantages of providing a JDO interface to the IDEAS database. Some types of application can be developed very quickly using such an interface, as illustrated in Section 7.1. The OSI2JDO translator provides a mechanism for incorporating OSI data in a J2EE environment[Bodoff et al. 2002], and could very easily be extended to support standards like JMX[Ebro et al. 2001].

8.1 Timings

Without access to multiple test databases and some larger applications it is hard to draw too many conclusions about the performance of the prototype translator and runtime environment. Furthermore, no attempt has been made to optimise the translator, and we are using a debug build of the intelliBO enhancer. However, with these caveats out of the way, here are some rough performance measurements. The translator took 75 seconds to convert the test database into 196 managed object classes and 584 attribute type classes. These files took 50 seconds to compile, and a further 110 seconds to enhance. All the timings were made on a mid-range laptop running Windows 2000 and JDK1.4. Clearly at the present time the enhancer is the main bottleneck in the translation process. After the initial build then subsequent runs are a lot faster using the incremental feature of the Enhancer task described earlier. If a new class has been added then the OSI2JDO translator has to be rerun as it is not incremental at present.

8.2 Schemas

The system makes extensive use of views and triggers to bridge the gap between database schema and JDO implementation. Such features obviously place some additional load on the database, and further work is required to gauge the extent of this overhead. However, much of the inefficiency of the JDO interface is likely to stem from the schema itself, where a single attribute change may potentially involve altering many rows in the `MOATT` table. In some cases it may even be more efficient to update an object via a view than though separate remote updates to the database for each `MOATT` entry.

The current database schema is presumably optimized for ease of interpretation. However, is this the best trade-off given the typical use of such a system? New classes, attributes, and types are defined fairly infrequently, at least relative to their use. Performing a "compilation" step whenever a new class is introduced, and then using a more optimized schema for the classes, may improve the efficiency of IDEAS and be easier to map to a JDO implementation. The JDO classes produced by the translator could also be viewed as a stable interface between the user and an evolving database schema.

8.3 Caching

It seems likely that an IDEAS server caches data in memory for efficiency reasons, exploiting the fact that it is usually the only application accessing the database tables. The OSI documentation says very little about this aspect of the system, or where the transactional boundaries are. Given this lack of information, for the prototype we decided to ignore such complications entirely. For some applications, such as bulk loading, or report generation, this is unlikely to cause problems. However, a more robust solution would require a greater degree of cooperation between the JDO classes and the IDEAS server, e.g. using events to force cache refreshes in the server.

8.4 Licensing

JDO implementations typically license the enhancer, with the run-time support code being freely distributable. This model works well when a company uses JDO to implement a product and where the details of JDO are hidden inside the application. Unfortunately the OSI2JDO translator doesn't quite fit this pattern. Running the translator on a user's database produces files that must be enhanced before use. To do this the user requires access to a JDO enhancer. Shipping an enhancer with the product is problematic as the user could use the tool to process files unrelated to this application. However, it should be possible to sign each generated file to indicate it has been generated by the OSI2JDO translator. A JDO vendor might then be persuaded to adapt an enhancer to only work on such signed files, and license such a tool on more favorable terms.

8.5 Future Work

One should not underestimate the amount of effort required to transform most prototypes into production quality applications. The OSI2JDO translator, although reasonably robust, is no different in this respect and was written without any internal knowledge of the OSI system. An assessment of the likely impact of the translated code on the IDEAS server, and the development of mechanisms to allow the two systems to interact smoothly, are obviously areas requiring future work.

The OSI2JDO translator currently translates all the managed object classes and types it finds in the database. Obviously a more incremental approach would be preferable. If a new class is added to the database then we should only have to translate this class, plus any additional types it may introduce, and update the database with a small number of new views and triggers. In some scenarios a user may only require a small subset of the available classes to have JDO equivalents. Whilst pruning a JAR file to remove unwanted classes is a simple enough process, this is not an ideal solution.

There is one obvious missing part to the jigsaw that we have constructed. There is no mechanism for sending or receiving events, or for mapping the event objects into their natural Java classes. Adding the ability to eavesdrop on the IDEAS event stream would allow some IDEAS rules to be replaced by Java code where this is appropriate. The ability to send events to the IDEAS server would allow Java applications to act like gateway processors in some cases, and may also provide a simple mechanism for informing the server of changes that might invalidate various caches.

Interacting with SNMP agents forms an important part of many OSS systems. The current translator allows us to manipulate managed objects in the database using a JDO interface. A natural question to ask is whether we can manipulate the SNMP agents using a similar approach. What would this mean? A JDO enhancer could be written that was parameterized on configuration files mapping fields to SNMP Oids rather than database tables and columns. The JDO features would help batch fetches and updates, and the agents could be queried using a natural syntax. Ideally you would like to be able to change the state of a managed object in the database, and make an equivalent change to an SNMP agent, and then be able to commit the changes atomically using a two-phase commit. Unfortunately, SNMP doesn't support transactions, although there have been some proposals in this direction.[] Some MIBs provide transaction-like facilities that could perhaps be exploited by a JDO interface. In some scenarios it may also be possible to order the commit operations to minimize the effects of the lack of transactions, e.g. updating the agent

last so that the database changes can be rolled back if necessary. Although very speculative at this stage, the idea may have sufficient potential to warrant further exploration.

REFERENCES

Apache ant. http://jakarta.apache.org/ant.
Bodoff, S., Green, D., Haase, K., Jendrock, E., Pawlan, M., and Stearns, B. 2002. *The J2EE Tutorial*. Addison-Wesley.
Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide*. Addison Wesley.
Ebro, C. et al. 2001. Java Management Extensions (JMX). Tech. Rep. JSR000003, Sun Microsystems.
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns*. Addison-Wesley.
JDBC. 2001. JDBC Data Access API. http://java.sun.com/products/jdbc.
OSI 2000a. *NETeXPERT Overview*. OSI, 101 Park Way, Folsom, California 95630.
OSI 2000b. *Rule Language Guide*. OSI, 101 Park Way, Folsom, California 95630.
Russell, C. et al. 2001. Java Data Objects. Tech. Rep. JSR000012, Sun Microsystems.
Signsoft 2001. *intelliBO User's Guide, Version 2.5*. Signsoft, Leipziger Str. 118, 01127 Dresden, Germany.
SQLJ. 1998. SQLJ: Embedded SQL for Java. http://otn.oracle.com/tech/java/sqlj_jdbc.