Java Data Objects
A White Paper

February 2002

Java Data Objects
A White Paper

Introduction

Java Data Objects (JDO), the emerging Java standard for persisting objects, is creating great interest in the developer community. But will it live up to its promises and, in particular, how will it fit in to the J2EE architecture? PrismTech has many years of experience developing persistent object systems and has made significant contributions to the emerging standard. PrismTech is therefore strongly supporting this standard and has developed a commercial implementation that was released in November 2001.

This paper provides a background to the problems that JDO overcomes and provides a brief introduction to the specification currently being developed. A simple example is provided using PrismTech's object relational implementation, OpenFusion JDO, and the relationship between JDO persistence and existing J2EE persistence mechanisms such as container and bean managed persistence is discussed. The paper concludes with a short description of the key features of OpenFusion JDO.

Developing Persistent Java Applications Today

In this section, we review the techniques that can be used to persist Java objects using current technologies. We discuss the problems that are inherent in these technologies, particularly, in common with most large scale enterprises, when using relational databases. Finally, we present a vision of development in a world where these problems are resolved.

The Current Situation

The Java programming language is fast emerging as the programming language of choice, particularly when developing new applications. These applications cover a wide range from embedded applications in mobile phones to fully distributed enterprise scale applications for thousands of users.

In many cases, the objects that the application requires in order to run are created by the application, used by it and destroyed when it completes. Their lifetime is scoped to that of the application, they exist in its memory space, and are lost when it terminates.

But for many applications this simple scenario is not sufficient. The information contained in the objects has a lifetime longer than that of an application that uses them, or longer than the user session in which they were accessed. It may be necessary for many users to have access to this information over potentially long periods. There are many applications with this requirement: CRM, on-line purchasing applications, accounting systems. To meet this requirement, it is necessary to save the state of an object from the memory of an application to some form of persistent storage that is outside the scope of the application.

Applications such as these will use one of the persistence mechanisms available today.

- Flat files: usually restricted to simple applications or simple data models, storing a relatively small amount of information, the system being developed not having any concurrency or query requirements.
- Object databases: generally reserved for use in specialized applications where performance is a primary concern; often embedded in the application and not visible to the end-user,
- Relational databases: the most commonly used persistence mechanism for enterprise level applications; characterized by scalability and reliability.

Several solutions based on these persistence mechanisms have been developed specifically for the Java world and are being used quite successfully.

Object Serialization

In simple cases, it may be sufficient to save the state of an object in a file, and for this, the `java.io.Serializable` interface has been provided. This allows a developer to explicitly save an object to an output stream, for example by writing its fields to a file which is stored on a local disk. Serialization is achieved with a very simple interface, but it is powerful, despite this.

Objects are serialized with the `ObjectOutputStream` class and deserialized with the `ObjectInputStream` class, both of which are part of the `java.io` package. An object is serialized by passing it to the `writeObject()` method of `ObjectOutputStream`. This causes the values of all the object's fields, including its private fields and those that it inherits from its superclasses, to be written to an output stream. Where the value of a field refers to another object, an array or a string, `writeObject()` is invoked recursively to write out that object. In this way, an entire graph of objects can be serialized with one call to `writeObject()`.

The process is reversed in order to deserialize or read in an object. The `readObject()` method of `ObjectInputStream` is invoked and this causes the object to be recreated with the same state that it had when it was serialized. Objects that are referenced are deserialized recursively, so an entire object graph can be rebuilt with a single call.

JDBC

JDBC is an application programming interface that provides Java programs with access to almost any kind of tabular data, but most commonly, relational databases. It enables developers to create database applications entirely in Java. It simplifies the task of sending SQL statements to relational database systems and, by supporting all dialects of SQL, shields developers from the differences between databases. So, in keeping with the Java philosophy of "write once, run anywhere", a developer can write one application that will run unchanged with Oracle, Sybase, or IBM DB2 databases.

The JDBC API is used to invoke SQL calls directly, so applications can send queries or update statements to the database. In addition to this, JDBC was also designed to be a base that other interfaces or tools could be built upon. For example, SQLJ, which is an embedded SQL for Java specified by IBM, Oracle, Sun and others, is built on JDBC. While JDBC requires SQL statements to be passed as strings to Java methods, SQLJ enables the developer to embed SQL statements in Java code or to use Java variables in SQL statements. SQLJ provides a preprocessor that converts this combination of Java and SQL into Java with JDBC calls. JDBC supports dynamic SQL requiring some runtime processing with the overhead that entails, whereas SQLJ provides a static SQL binding that allows some of the processing to be performed at development time instead of runtime.

Object/Relational Mapping Tools

Creating a relational database that can hold a Java object model in an efficient way is not a trivial exercise except for the most simplistic of cases. There are many ways that the object model can be mapped onto the tables and choosing the wrong approach in a given situation can have a dramatic impact on the performance of the application. In some circumstances, the relational database may pre-exist the Java application, so that application model will have to built to fit the relational model.

Recognizing this, a number of vendor have produced so-called 'Object/Relational Mapping Tools'. These are tools that simplify the process of creating relational models from object models, in many cases by providing graphical interfaces that can display a visual representation of the mapping. The code that performs the translation is generated automatically.

In general, Object/Relational Mapping Tools have proprietary interfaces and may use patented mapping technology. Applications generated using these tools are not generally portable. The tools may well use JDBC as the interface to the underlying RDB, thus providing a degree of database independence.

Enterprise Java Beans

The Enterprise Java Beans (EJB) architecture is a server-side technology for developing and deploying distributed components that contain the business logic on an enterprise application. There are two types

of EJBs: Session Beans, which carry out operations such as a calculation or database access on behalf of a client, and Entity Beans which are persistent objects representing data held in a database. Entity beans can either manage their own persistence or can delegate this to their container.

EJBs are hosted in an EJB container which provides access to transaction and persistence services and to services provided by the Java 2 Platform Enterprise Edition (J2EE Platform). The EJB architecture primarily provides the capability of distributing components across multiple systems. The infrastructure necessary to support such a capability brings with it a model, interface, and infrastructure that does introduce overhead and consequently impacts on performance. However, many developers use EJB not so much for these distribution capabilities, but to gain access to the transaction and persistence services that come with the EJB environment.

The Challenge

Relational databases are the persistence technology of choice for most enterprise-level applications. This choice 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 a considerable mismatch between the application model and the database model. The application model will have ideally been developed using some form of object-oriented methodology (these days, often the Unified Modeling Language(UML)). The application model will be specified in terms of classes with attributes and methods and will almost certainly incorporate subclass-superclass relationships. The relational model, however, will consist of tables with relationships, but no methods and no subclass relationships. In a standard relational model, there is no direct way to support polymorphic references.

An application query will be expressed in terms of object attributes and/or methods and will return a collection of matching objects. The scope of a query may well include a class and its subclasses. Conversely, a relational query will be written in terms of tables and columns and will return a set of matching rows.

The application interface to the relational database for Java developers will most likely be JDBC. This allows the application to pass SQL statements to the database, but does nothing to hide the relational nature of the database from the application. At worst, in order to simplify the relationship between the database and the application, developers resort to using Java almost as a procedural language and lose the benefit of many object-oriented features.

However, it is more natural to create a mapping between the application model and the relational model. There are many ways that this can be done:

- 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.
- 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.

A third 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.

Once the relational model has been determined, queries must be written to retrieve objects. There are issues here too, particularly relating to mapping attribute types from Java to SQL.

Creating efficient mappings from application to relational models requires considerable skill and experience on the part of the developer. In particular, thorough knowledge of database design and the use of SQL is required. Some developers have these skills, usually acquired through years of experience, but most do not. It is also rare to find individuals knowledgeable in both object modeling and database modeling as these individuals often participate in different developer communities. Yet in doing this mapping between object and database models, such an individual is often essential.

In the past, many organizations that needed an object layer on a relational database have been persuaded to implement object/relational mapping software themselves. They are now finding themselves

constrained by the limitations of their implementations. They are forced to bear maintenance and development costs that may be significantly greater than first estimated and furthermore development and support of the layer distracts skilled resources from their primary business focus.

A further problem arises when third party components are incorporated into the application. The source code for these is usually not available, so they cannot be made persistent using traditional techniques.

Finally, as a result of the issues described, applications are often tightly coupled to specific databases. Consequently, they cannot be ported to different databases without considerable work.

A Vision

We can now envision a better world: where application programmers do not need to know what the database model looks like or how to program databases; where queries to the database are written in terms of the application model and not the database model; where we can make third party components persistent, even when we don't have access to their source; and where we can deploy with relative ease on different relational database systems without having to change our application code in any way. While there are proprietary tools that provide some steps towards realizing the vision, in short, what is required is a standard, scaleable, Java-centric persistence mechanism.

Java Data Objects (JDO) provide such a mechanism.

Java Data Objects

The Java Data Objects (JDO) specification 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. Mappings to the database, if required, are specified using an implementation-specific mechanism, but the application's Java interface is identical across all implementations. Furthermore, it is not necessary for programmers to explicitly fetch and store Java objects from a database: this is done automatically in JDO.

The JDO standard has been developed under the Java Community Process (JCP) as specification request JSR-000012. The JCP was established in 1995 as an open process to develop and revise the Java technology specifications, reference implementations and test suites. Over 300 companies and individuals are currently participating in the JCP. Development of the JDO Standard began in 1999 and has now reached the stage of 'Proposed Final Draft', the last stage of the process before final ballot and release. Ratification of the standard is expected in the first quarter of calendar year 2002.

PrismTech has been heavily involved in the development of the standard over the last year. Our JDO Technical Lead is a member of the expert group that has been specifying the standard and other members of our development team have made significant contributions to the work.

Rationale

The JDO Specification has been developed because, as we have seen, there is a need for a standard, Java-centric way to store Java objects in transactional data stores. Existing standards such as Java serialization and JDBC have limitations which JDO is intended to overcome.

JDO specifies a Java way of presenting a consistent view of data across a large number of applications and Enterprise Information Systems (EISs), with the result that component vendors don't need to customize their products for each type of data store and EIS vendors can provide a standard data access interface for their EISs.

Architectural Goals

The JDO architecture has been defined to meet the following goals:

1. To provide a transparent interface for application component developers so that they can store data without needing to learn a new access language for each type of datastore.
2. To enable developers to use the Java programming model to model the application domain and transparently store and retrieve data. For example, developers will no longer need to convert their application object models into relational models before they can make data persistent.
3. To be suitable for a wide range of uses, ranging from embedded systems to enterprise systems incorporating application servers.
4. To simplify the development of scaleable, secure, transactional JDO implementations for a wide range of EISs.
5. To be implementable for a wide range of persistence mechanisms, from local file systems to enterprise information systems.
6. To allow exploitation of critical performance features from EISs, for example, query evaluation and relationship management.
7. To use the J2EE Connector Architecture to provide standard connectivity to most EISs.
8. To enable plug-ability of JDO implementations across multiple applications servers.

JDO Overview

JDO implementations can be used either directly in two tier or embedded architectures (an *unmanaged* environment) or with an application server (a *managed* environment). This enables application components to access datastores using a consistent, Java-centric view of data. The mapping of the specific data types and relationships in the underlying datastore to and from Java objects is handled within the JDO implementation and is transparent to the application programmer.

In the unmanaged scenario, JDO hides the details of the persistence mechanism from the application. It hides issues such as data type mapping, relationship mapping, data retrieval and storage, so that the application sees all these expressed in native Java terms.

Using JDO in a managed environment has additional benefits as JDO provides transparency for system-level mechanisms such as distributed transactions, security and connection management.

What ever the environment in which JDO is deployed, the application developer is free to concentrate on business logic and presentation and no longer need to be concerned with the details of how objects are persisted.

Architecture

Figure 1 illustrates how JDO is deployed in a unmanaged environment.

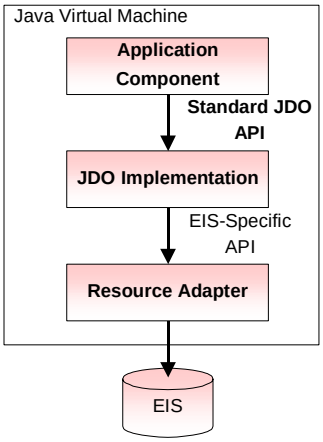

Figure 1: JDO in an Unmanaged Environment

The JDO implementation hides all of the details of the persistence mechanism from the application.

The main interface for persistent-aware applications is the `PersistenceManager` interface. A `PersistenceManager` 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 are managed through the `PersistenceCapable` interface which provides services such as life cycle state management. However, applications never access the `PersistenceCapable` interface which is used only by the JDO implementation. We describe exactly how objects are able to support the `PersistenceCapable` interface later in this document.

One example of the use of JDO in a managed environment is shown in Figure 2.

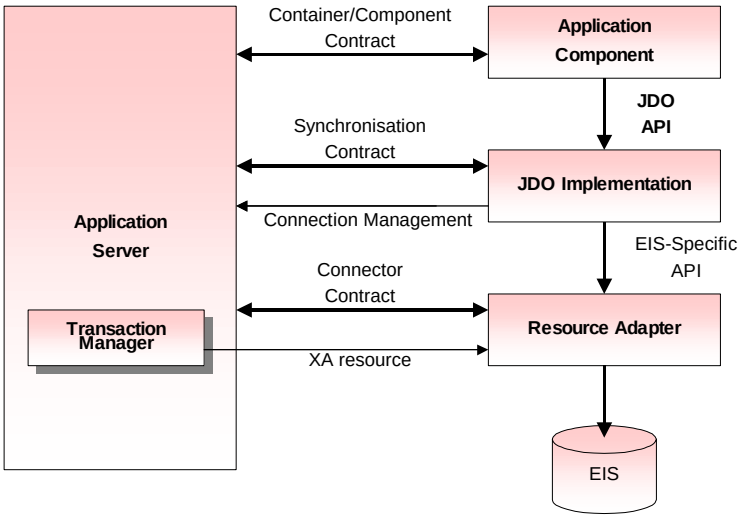

Figure 2: JDO in a Managed Environment

When a JDO implementation is used with an application server, the JDO Architecture uses the J2EE Connector Architecture to standardize the use of system level features such as connection, transaction and security management. The Connector Architecture defines a standard set of system level contracts between an application server and an EIS; these contracts are implemented by a resource adapter in the EIS.

Figure 2 shows JDO used in a layered architecture with an application server and an EIS. In this scenario, the JDO implementation will interact with the application server through the Connector API to obtain connections to the EIS and will use the transaction management contracts to ensure transaction integrity.

Identity of PersistenceCapable Instances

The identity of a PersistenceCapable instance is a vital characteristic and it is here that the JDO specification defines some extensions to standard Java behavior.

There are two issues to be considered: *identity* and *equality*. Identity is used to determine if two instances are the same instance. In Java, identity is a function of the Java Virtual Machine and two instances are identical only if they occupy the same physical location in the JVM. Equality is used to determine if two instances represent the same data, i.e. have the same value (based on the abstraction being modeled). In Java, equality is determined on a class basis: instances are equal if they represent the same data.

The interaction between identity and equality is an equally important issue for JDO. Java object equality is an application issue and JDO implementations must not change the application's definition of it. JDO requires that there is only one JDO instance for each PersistenceManager representing the persistent state of the corresponding data store object. To achieve this, JDO defines object identity differently from both Java identity or application equality. JDO provides identity that transcends a single JVM, providing a unique reference to an object that can be stored in the database and used by all JVMs that access the object.

JDO defines three types of object identity:

1. Application Identity, often called primary key or natural identity. In this form of identity, values in the instance determine the identity of the object in the data store, so JDO identity is managed by the application and is enforced by the data store.
2. Data Store Identity. In this form of identity, the identity of the object in the data store does not depend on any values in the instance and the JDO implementation must guarantee uniqueness for all instances in the data store. The JDO identity is not tied to any particular JDO instance values. It is managed by the JDO implementation and is not guaranteed to be portable.
3. Non-data Store Identity. In this case, uniqueness is guaranteed in the JVM but is not supported in the data store. This form of identity, which is intended for use by log files or other similar storage mechanisms that do not support unique identifiers for data or where performance is a primary concern.

An implementation must provide at least one of these forms of identity and a `PersistenceCapable` class can only support one form of identity.

Lifecycle of PersistenceCapable Instances

In JDO, a PersistenceCapable instance can be transient or persistent. If the instance is transient, it behaves exactly the same as if it were a normal Java instance. If it is a persistent instance, it represents the state of data held in a data store and its behavior is linked to the data store with which it is associated. For example, the JDO implementation automatically tracks changes to the values in the instance and automatically refreshes values from or to the data store to preserve the transactional integrity of the instance. The fact that this is done automatically is what achieves the 'transparency' of the persistence to the application.

During its lifetime, the PersistenceCapable instance transitions through various lifecycle states under the control of the JDO implementation and governed by the state model defined in the JDO Specification.

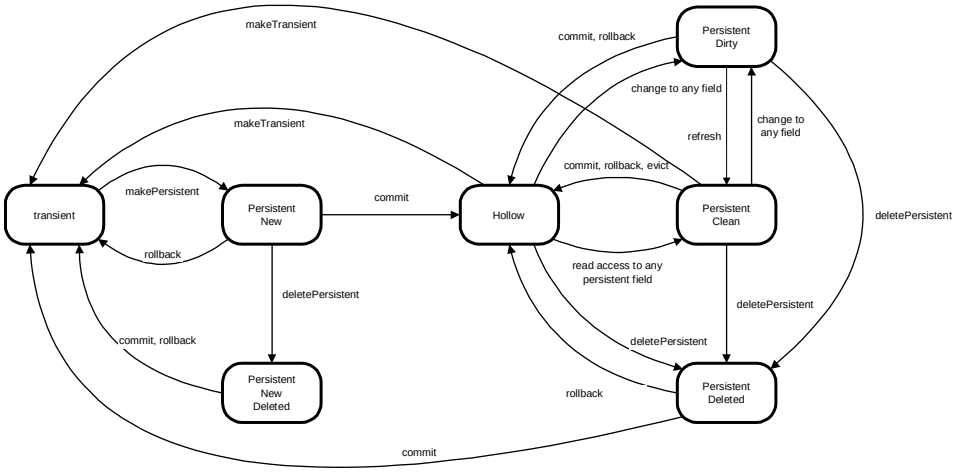

Figure 3: Lifecycle of a JDO Object

Figure 3 illustrates the states that a JDO instance can have, the transitions that are allowed between them, with the events that cause the transitions.

These are the mandatory states, i.e. the states that a JDO implementation is required to implement.

Transient: an instance in the transient state behaves exactly like a normal Java object. It has no JDO identity associated with it.

Persistent New: this is the state of an instance that is newly persistent in the current transaction. The persistence manager is now responsible for state transitions and state interrogation. The instance has a JDO Identity.

Hollow: this state represents a specific persistent object in the data store, but one whose values are not yet in the instance. Hollow instances have a JDO Identity associated with them, so uniqueness is guaranteed.

Persistent Clean: this state represents a specific transactional instance in the data store, but one whose values have not been changed in the current transaction.

Persistent Dirty: this state represents persistent data that has been changed in the current transaction.

Persistent Deleted: this state represents persistent data in the data store and which has been deleted in the current transaction.

Persistent New Deleted: this state represents an instance that has been newly made persistent and then deleted.

Principal Interfaces

We have already mentioned two of the main interfaces specified by JDO, `PersistenceCapable` and `PersistenceManager.` These are described in more detail in this section, together with some of the other interfaces defined in the specification.

PersistenceCapable

The `PersistenceCapable` interface defines methods that allow an instance to be managed by the implementation. It follows that every instance to be managed must be an instance of a class that implements the `PersistenceCapable` interface. The interface defines methods that enable JDO aware applications to discover the runtime state of an instance and to discover the `PersistenceManager` associated with an instance. The interface also provides methods that manage the identity of an instance.

The recommended method for interrogating the state of an instance is to use a helper class, `JDOHelper`. `JDOHelper` provides static methods that delegate to the class if it implements PersistenceCapable or if it does not, returns the values that would be returned by a transient instance.

In order to avoid name conflicts in user-defined classes, all methods defined by the `PersistenceCapable` interface are prefixed with '`jdo`'.

The method by which a class may acquire the `PersistenceCapable` interface is not mandated by the specification. Thus a class developer may choose to explicitly declare that the class implements the interface. In this case, the developer is responsible for ensuring that the class correctly implements the `PersistenceCapable` contract. An alternative approach is to make use of a process called 'byte code enhancement'. In this process, byte codes produced by a Java compiler are modified by a tool called an enhancer with the result that after enhancement, the class supports the `PersistenceCapable` interface. The behavior of an enhancer is defined in the specification and an example of how one is used is given later in this paper. It would equally be possible to support the interface through a process of source code enhancement.

PersistenceManager

`PersistenceManager` is the primary interface for JDO-aware applications. It is the factory for the `Query` interface and the `Transaction` interface and provides methods for managing the lifecycle of persistent instances.

The architecture of the `PersistenceManager` has been designed to support a variety of environments and data sources. These can range from embedded systems with small footprints using simple local persistent mechanisms to large enterprise application servers. The `PersistenceManager` may be layered on top of a standard Connector implementation such as JDBC or it may include connection management and distributed transaction support itself.

Portability of application code is an important goal of the `PersistenceManager` architecture. As a result of this, no changes to application code should be necessary to change from one vendor implementation to another. This feature could be used to provide portability of applications across different persistence mechanisms, for example across object and relational databases, or across flat file and relational databases. Further, applications can be moved from non-managed to managed environments with minimal code changes. In this way, an application could be developed in a simple two-tier environment, but deployed on an application server.

A `PersistenceManager` supports any number of `PersistenceCapable` instances at a time and manages their identities. However, a `PersistenceManager` will support only one transaction at a time and only one database connection at a time, although it may support multiple transactions or database connections serially. Therefore, if an application needs to manage multiple database connections in parallel, it can create multiple `PersistenceManagers`.

A `PersistenceManager` acts as a cache manager for its application. Simple cache management normally takes place automatically and transparently: instance states are evicted from the cache automatically on completion of a transaction. However, if an application needs more control over that behavior of the cache, several additional methods are provided by the interface.

PersistenceManagerFactory

The `PersistenceManagerFactory` interface is provided to allow applications to construct `PersistenceManagers`.

In a managed environment, an application will create a `PersistenceManager` in two stages. First it will use JNDI to locate a `PersistenceManagerFactory`. Then it will call a method provided by the `PersistenceManagerFactory` to create the `PersistenceManager`. The same approach is possible in a non-managed environment: an alternative it to use a constructor to create the `PersistenceManagerFactory`, then configure it and create a `PersistenceManager` by calling an appropriate method. The constructor for a `PersistenceManagerFactory` is not part of the JDO specification, therefore portable applications should use the JNDI approach.

The `PersistenceManagerFactory` contains the default settings of properties for all `PersistenceManagers` created by it. These include properties that control the behavior of the cache and queries, as well as the properties for database connections (for example, the connection factory to use, user name and password).

Query

Applications will require access to JDO instances so that they can invoke methods on those instances: for example, they might navigate from one instance to another, repeatedly through a graph of objects. The transparent persistence capability of JDO will ensure that the objects are retrieved from the datastore when required, but how does the application get to the first object?

There are three ways in which an application might locate an object. Firstly, the identification of the object may be known. In this case the application can recover the object by calling the `PersistenceManager's getObjectById` method. Secondly, the application can use the extent interface to iterate through the entire set of instances for a class. And finally, the application can use the query interface to retrieve an object (or objects) based on specific search criteria. This section describes some of the features of the JDO Query facility.

The JDO Query facility consists of two parts: the query API and the query language, which is called 'JDOQL'. A `Query` instance is created by a `PersistenceManager.` The execution of a query might, however, be delegated by the `PersistenceManager` to its datastore in order to take advantage of optimizations that the datastore offers. In this case, the JDO implementation is required to translate the query from JDOQL into another, database-specific language.

The JDO query interface has been designed to achieve a number of goals:

1. The JDO Query Language should be neutral, since the query may have to be translated into any of a number of different underlying languages.
2. It shall be possible to optimize the query for a specific underlying language. Alternatively, the query must contain enough information to enable the JDO implementation to optimize the query for a particular underlying language (for example, SQL).
3. Multi-tiered architectures must be accommodated. Queries might be implemented in memory, or delegated to a datastore, or a combination of both, and the Query interface must support these.
4. Large result sets must be supported: a query may potentially return a huge number of instances and it must be possible to process these within the constraints of the execution environment.
5. In order to reduce processing at run time, it should be possible to compile queries in advance of executing them.

A `PersistenceManager` is the factory for queries. The `Query` interface itself provides methods for binding parameters, setting options (such as whether to include the cache in the query scope), compiling and executing queries.

The filter is specified using JDOQL and is very similar to a Java Boolean expression where some of the Java expressions are not permitted. A simple example is given later.

Transaction

The `Transaction` interface provides for management of transaction options and for transaction boundary demarcation in the non-managed environment.

In a managed environment, transaction completion is handled by the XAResource which is enlisted by the `ConnectionManager` of the Java Connector Architecture.

In both cases the `PersistenceManager` is responsible for setting up the appropriate interface to the connection infrastructure.

Extent

The `Extent` interface is provided to enable applications to define a candidate set of instances over which a query is to be executed. In the case of the `Extent` interface, the set includes all instances of a class, both in the data store and in the cache.

The goal of the `Extent` interface is to efficiently manage large result sets and application resources.

The principal method provided by the interface is the ability to iterate over the extent.

Using JDO

In this section we illustrate the use of JDO by means of a simple application implemented using PrismTech's relational implementation, OpenFusion JDO.

Figure 4 illustrates the logical model of the example application..

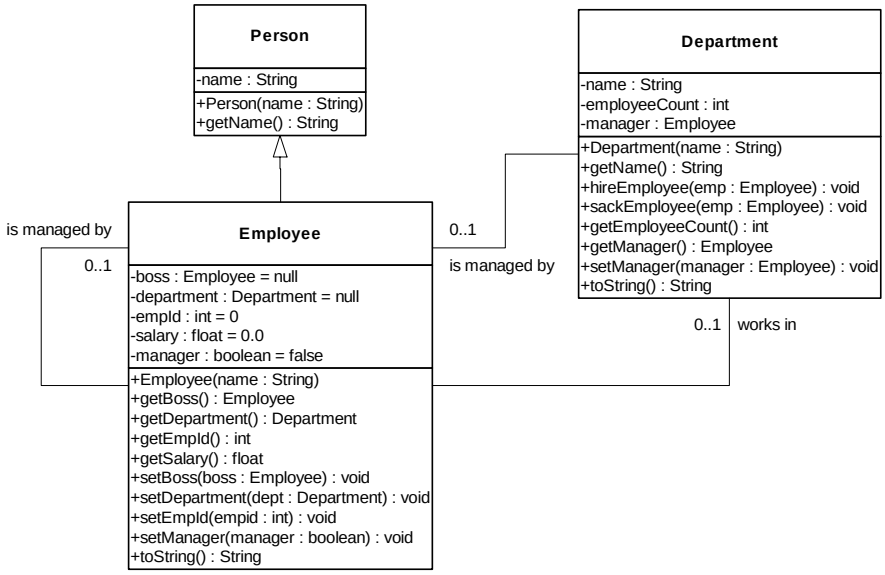

Figure 4: Logical Application Model

The example has three classes:

- `Person`, which has a field "`name`"
- `Employee`, which is a subclass of "`Person`" and has fields "`boss`" (another employee and shown by the relationship 'is managed by'), `department`, which is a `Department` and is shown by the relationship 'works in', employee ID, salary and a flag to indicate if the employee is a manager.
- `Department`, which has a name, a count of the number of employees in the department and a manager, shown by the relationship 'is managed by'.

Part of the source for one of the classes, `Employee`, is listed in Figure 5.

```
package UserDefinedClasses;

public class Employee extends Person
{
   private Employee boss = null;
   private Department department = null;
   private int empId = 0;
   private float salary = 0.0f;
   private boolean manager = false;

   public Employee( String name )
   {
      super( name );
   }

    // more methods . . .
}
```

Figure 5: Example Class

The other classes would be similar. The important point to note from Figure 5 is that it contains standard Java. We have made no changes to the source in order to make this class persistence-capable.

Persistence Descriptor

While we don't have to make any changes to the source code to make the classes persistent, we do need to write a 'persistence descriptor'. This is an XML document, written against a DTD that is defined in the JDO specification, that describes the persistence characteristics of the classes in the application.

Figure 6 shows the persistence descriptor for the example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/C:\PrismTechJDO/xml/schema/jdo.dtd">
<jdo>
   <package name="UserDefinedClasses">
      <class name="Employee" identity-type="datastore">
         <field name="empId" default-fetch-group="true"/>
         <field name="department"/>
         <field name="boss"/>
         <field name="manager"/>
         <field name="salary"/>
      </class>
      <class name="Department" identity-type="datastore">
         <field name="name"/>
         <field name="employeeCount"/>
         <field name="manager"/>
      </class>
      <class name="Person" identity-type="datastore">
         <field name="name"/>
      </class>
   </package>
</jdo>
```

Figure 6: XML Persistence Descriptor

The persistence descriptor contains information about each class in the application that is required to be made persistent, together with each field in the class that is to be made persistent.

In fact, the example in Figure 6 contains more information than is strictly necessary according to the JDO specification. There is no requirement to specify individual fields provided that the default

attribute values are appropriate. Therefore, in the example, Employee.department, boss, manager, salary; Department.name, employeeCount, manager; Employee.name could be omitted.

Each class also has an attribute that specifies the JDO identity type to be applied to that class. The default type is 'datastore' so this could also be omitted from the example.

Fields have an attribute called the 'default fetch group'. When set to 'true', this indicates that this field is to be managed as a group with other fields, and more specifically, that these are the fields that are initially populated in the object when it is read from the database. If one field in the group is retrieved, then the others will also be retrieved, thus allowing database access to be optimized.

Enhancement

Now we have the source classes and a description of their persistence characteristics, we can make them persistent. In OpenFusion JDO, this is achieved through a process called ‘byte code enhancement’. The OpenFusion JDO persistence Tool incorporates an enhancer (which is specified in the standard) that modifies the standard output of a Java compiler to add persistence capabilities to the specified classes. The process is shown in Figure 7.

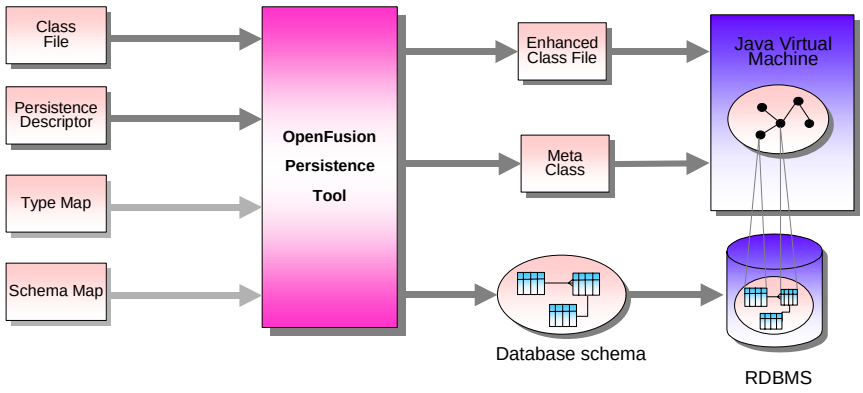

Figure 7: Byte Code Enhancement

In addition to the class file and persistence descriptor described above, the OpenFusion Persistence Tool may optionally be provided with a further input file. This is the Type Map, an XML file that can be used to override the default mapping of Java types to SQL types for a specific database. OpenFusion JDO includes type maps for common relational databases and users may modify these to add additional databases or to optimize the mapping for their specific requirements.

An optional input file may also be provided. This is the Schema Map, an XML file created by the user to override the default table and column names generated by the Persistence Tool.

The OpenFusion Persistence Tool generates three outputs. The first is a set of enhanced class files for all the classes that are specified as persistent in the descriptor. The enhanced class files include the `PersistenceCapable` interface, so enabling a `PersistenceManager` to manage instances of the class. The OpenFusion Persistence Tool also creates meta classes for each of the enhanced classes; the meta classes contain the information describing the mapping of the Java class and attributes to relational tables and columns. Finally, the Persistence Tool creates a set of SQL files specifying the database schema required to support the persistent classes. The SQL created for the example is shown in Figure 8.

```
CREATE TABLE Department
(
    Class_ID INTEGER NOT NULL,
    Inst_ID LONG NOT NULL,
    name VARCHAR2(2000),
    employeeCount NUMBER(38,0),
    Department_manager_class_ID INTEGER,
    Department_manager_inst_ID LONG
);
CREATE TABLE Person
(
    Class_ID INTEGER NOT NULL,
    Inst_ID LONG NOT NULL,
    name VARCHAR2(2000),
    manager CHAR(1),
    salary REAL,
    empId NUMBER(38,0),
    Employee_boss_class_ID INTEGER,
    Employee_boss_inst_ID LONG,
    Employee_department_class_ID INTEGER,
    Employee_department_inst_ID LONG
);
```

Figure 8: Example Schema

You can see that there are only two tables: Department and Person. That's because the default mapping generated by the OpenFusion enhancer creates a single table for each class hierarchy and rolls all the subclasses of the hierarchy into that one table. This is a useful default mapping: it's very efficient for queries relating to extents of classes or queries that select instances of a class and its sub-classes.

The enhancer also generates primary keys, indexes and constraints.

The primary key in this case will be created for the combination of Class_ID and Inst_ID, because this example uses datastore identity.

Indexes are created by default for the primary key and attributes used as foreign keys.

The database is created by executing the SQL files generated by the Persistence Tool

It is important to realize that the details of schema generation are specific to OpenFusion JDO and are not specified in the JDO Standard. This is deliberate and is intended to allow implementations to innovate and compete based on their mapping capabilities and technologies, while not impacting on the portability of the JDO applications.

Persisting Objects

An application will naturally need to persist instances of its persistent classes. Figure 9 lists the Java required to do this.

```
public void persistObjectsToDataBase()
{
    Employee[] employees = new Employee[5];
    Employee manager = new Employee("Manager");
    Department department = new Department("Data Services");
    manager.setEmpId(1001);
    department.setManager(manager);
    for (int i = 0; i < employees.length; i++ )
    {
        employees[i] = new Employee("Employee" + i);
        employees[i].setEmpId(1100 + i);
        department.hireEmployee(employees[i]);
    }

    PersistenceManager pm = getPersistenceManager();
    Transaction transaction  = pm.currentTransaction();

    transaction.begin();
    pm.makePersistentAll (employees);
    transaction.commit();

    pm.close();
}
```

Figure 9: Code to persist objects

We can ignore the first block of code. It is there simply to set up the network of objects to persist.

The first action is to get the PersistenceManager with a call to `getPersistenceManager()`. In this example, `getPersistenceManager()` is a local method that encapsulates a call to the `persistenceManagerFactory`'s `getPersistenceManager()` method. We have not detailed how the `persistenceManagerFactory` is created or located in this example. However, it may be created may be achieved using a constructor (which is not part of the JDO standard) or it may be located using JNDI or it may be created using `JDOHelper` and a set of properties.

Next the transaction context of this persistence manager is established with a call to the PersistenceManager's `currentTransaction()` method. The object returned is used to demarcate transaction boundaries.

Now the object can be persisted: a transaction is started, the employee objects are made persistent (they move from the transient to the persistent-new state) and the transaction is committed.

This example illustrates an important aspect of JDO called 'persistence by reachability'. We didn't need to explicitly make the department and manager objects persistent, but because they were associated with the department object, they are made persistent implicitly.

When the transaction is committed, all objects transition to the 'hollow' state. Their fields are written to the database and flushed from the cache. The JDO implementation should no longer have references to the objects and as long as the application has no more references to them, the garbage collector can free their storage for reuse. However, JDO provides a flag, `'retainvalues'`, which can be set to keep the field values in cache after commit time if this is appropriate.

Querying Objects

Once we've stored the objects, we'll want to get some of them back. This is achieved using a JDO query.

Queries are created using the factory methods in the `PersistenceManager` interface. Before creating the query, we need to specify

- The class to be queried

- The scope that the query is to be executed over. This may either be a collection of instances that has been created previously, or the entire extent of the class, i.e. all the instances of the class that exist in the database and current JVM.
- The filter used to select the instances. Unless we want all the instances, we must specify a filter to select those we want. The filter is specified using JDOQL and is essentially a string containing a Java Boolean expression.

  For example, to retrieve the employee objects for employees whose names are “Employee3” or “Employee4”, the filter would be

  ```
  "name == \"Employee3\" || name == \"Employee4\""
  ```

  Importantly, the filter is specified in terms of the application’s Java Model and not the Database Model.
- Whether to include the cache in the scope of the query. The query interface provides a flag , `IngnoreCache`, to control this. If the cache is included in the query scope, then instances that have not yet been committed may be included in the result set.

We can now create a query instance, compile the query and execute it. Compiling the query validates any elements that are bound to the query instance and reports inconsistencies by throwing an exception.

The result of the query is returned as a collection of instances.

A code fragment illustrating the above is shown in Figure 10.

```
public void queryOnLocalAttributes(Class target, String filter)
{
  PersistenceManager pm = getPersistenceManager();
  Transaction transaction = pm.currentTransaction();
  transaction.begin();
  Collection extent = pm.getExtent(target, false);

  Query query = pm.newQuery(extent, filter);
  query.compile();
  Collection result = (Collection)query.execute();

  processResult (result);

  pm.close();
}
```

Figure 10: Querying

JDO and Enterprise Java Beans

There is a lot of discussion about how JDO and Enterprise Java Beans (EJBs) are related or fit together. This section attempts to provide some answers to those questions. EJBs are defined in the Java 2 Enterprise Edition (J2EE) Specification.

First, we provide some background on persisting EJBs.

There are two types of Enterprise Java Beans, Entity Beans and Session Beans.

Entity Beans

The characteristics of entity beans are that they are persistent objects with non-trivial data-centric business rules, they are sharable among multiple clients or sessions, they are manipulated as one logical record in the persistent store and they have a long lifespan. They can be accessed remotely.

Because of these features, an Entity Bean needs significant system resources and therefore entity beans should be coarse grained objects. Yet they are intended, and are often used, for typical application domain objects, considered fine-grained objects.

Persistence in Entity Beans

The J2EE specification define two mechanisms by which entity beans can be made persistent. These are called 'Bean Managed Persistence (BMP)' and 'Container Managed Persistence (CMP)'.

In BMP, the bean provider directly implements persistence in the bean class. This provides the developer with good control over the persistence behavior, but makes it difficult to port beans to different persistent stores. To mitigate these problems, good design practice encapsulates the persistence behavior in a 'Data Access Object (DAO)'.

With CMP, persistence is delegated to the beans container and the bean provider has only to specify the persistence characteristics of the bean. The container provider supplies tools that generate database access calls at deployment time. The advantage of this approach is that the bean is defined independently of how it will be stored and so can be deployed without changes on different application servers. The disadvantage is that application server providers must provide sophisticated tools to manage the storage process. Object/relational mapping tools are often used in this context.

JDO and Entity Beans

Bean providers that must implement bean managed persistence for their beans can encapsulate access to data in Data Access Objects. JDO can then be used to implement the DAOs. JDO can also be used as the implementation technology for container managed persistence, but how this is done is an issue for application server vendors and is not discussed further here.

Bean-managed persistence

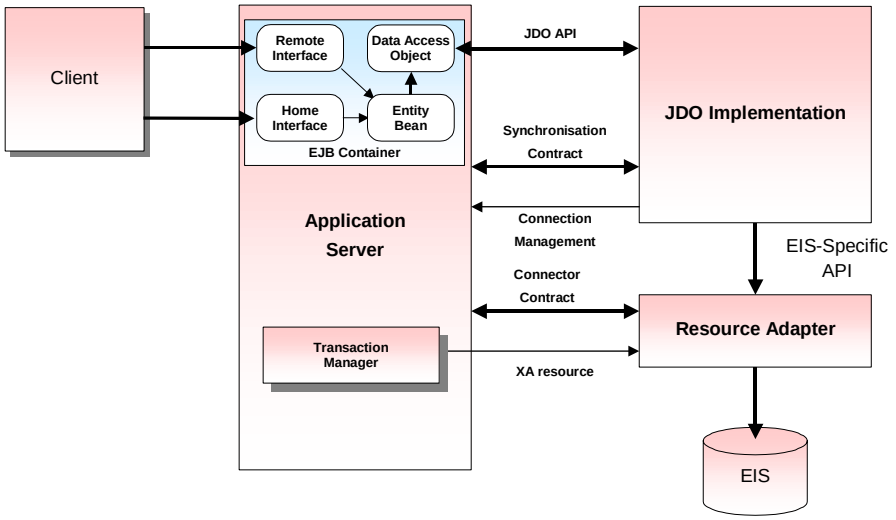

Figure 11: JDO and Bean Managed Persistence

Figure 11, developed from Figure 2, shows how JDO can be deployed to implement BMP for entity beans. A remote client accesses the entity bean through either the home or remote interfaces, using the home interface to create, delete or find a bean and the remote interface to invoke business methods. The remote interface, which extends the `EJBObject` class, locates the transactional instance of the entity bean associated with the user's transaction and delegates the business method to it. The entity bean in turn delegates methods relating to persistent state to the DAO which accesses the JDO implementation as described previously.

Figure 12 illustrates the lifecycle of an entity bean, showing the states and the events that cause transitions between the states. Overlaid on this diagram (shown in boxes) are the JDO events that must take place on the transitions.

For example, when setting the `EntityContext` for the bean, the JDO `PersistenceManagerFactory` should be located using JNDI and a reference to it stored in the bean; a `PersistenceManager` should be acquired from the factory when the bean is activated, and so on.

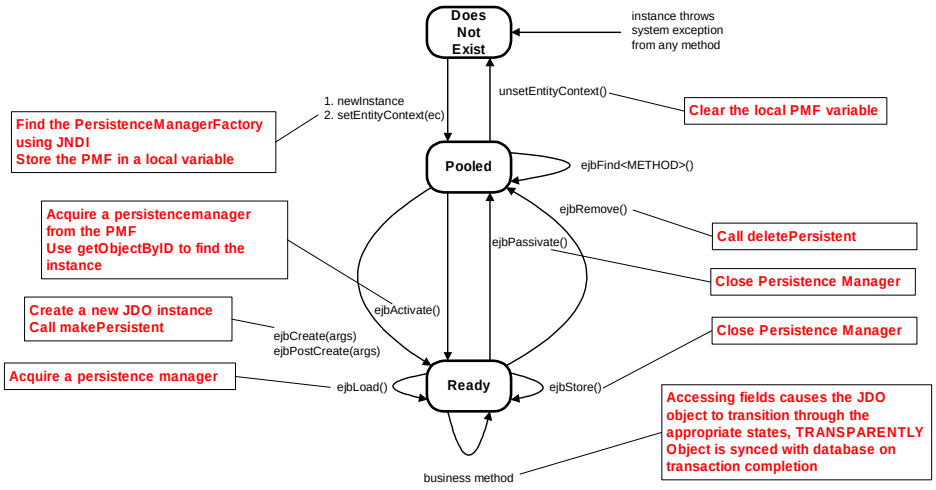

Figure 12: Entity Bean Lifecycle with JDO

Session Beans

Session beans are objects that hold client-specific business logic and are not intended for general access. They are not shared. They are objects that do not directly represent shared persistent data, their own state is non-persistent and they typically have a short lifespan. While session beans do not directly represent persistent data, they may be required to access it on behalf of a client.

There are two sub-types of session bean, *Stateful Session Beans* and *Stateless Session Beans*. A stateful session bean maintains state on behalf of a client. Such a bean might represent a shopping cart, for example. Stateless Session Beans don't hold client specific states, but they might hold non-client specific state. Since they do not retain client state between invocations, stateless session beans are often used to provide reusable services and can make very efficient of system resources.

JDO and Session Beans

JDO is a good choice of technology for session beans that are required to access persistent data on behalf of their clients. JDO can greatly simplify the process of making session beans portable. While the general technique for using JDO with session beans is similar regardless of the type of bean, there are some differences that are dependent on whether the bean is managing transactions directly, or whether it has delegated transaction management to its container. The distinction is the time at which the session bean acquires a `PersistenceManager`.

When it is created, the session bean should find a `PersistenceManagerFactory` using JNDI. It is important to use the same `PersistenceManagerFactory` instance for all beans that are sharing the same database resource. This ensures that the `PersistenceManagerFactory` can manage the association between `PersistenceManagers` and distributed transactions. When a session bean acquires a PersistenceManager, the `PersistenceManagerFactory` can look up the transaction association of the caller and either return the correct `PersistenceManager` or, if necessary, create a new one.

Container-managed Transactions

Container-managed transactions are the simple case. Considering stateless session beans first, each business method is an independent event, and can be dispatched to any bean in the ready pool. Therefore, each method must acquire its own `PersistenceManager`, with the `PersistenceManagerFactory` ensuring that it returns the `PersistenceManager` associated with the user's transactional context. Each method must close its `PersistenceManager` on completion.

In the case of a stateful session bean, methods are dispatched to a specific bean that is associated with a specific user. Otherwise, the behavior is the same as for stateless beans.

Bean-managed Transactions

Bean-Managed transactions offer the bean provider greater flexibility, but at the cost of extra complexity. With this approach, the bean provider establishes transaction boundaries and has the choice of using Java *UserTransactions* or using JDO transactions.

In the case of a stateless session bean, a `PersistenceManager` must be acquired and closed in each business method. However, if using `javax.transaction.UserTransaction,` the bean must establish its transaction context BEFORE acquiring the `PersistenceManager`. Because it is possible to have multiple transactions and multiple `PersistenceManagers,` beans must themselves track which transaction is associated with which `PersistenceManager`.

If `javax.jdo.Transaction` is used, acquiring a `PersistenceManager` without beginning a `UserTransaction` allows the `Persistencemanager` to manage the transaction boundaries using JDO transaction methods.

In the case of stateful session beans, the bean provider can manage transactional context as part of the state of the bean. It is therefore no longer necessary to acquire and close the `PersistenceManager` in each method. Apart from this, the behavior of stateful session beans is the same as for stateless beans.

OpenFusion JDO

OpenFusion JDO is PrismTech's commercial implementation of the JDO Standard. Version 1 of the product has been generally available since November 2001.

PrismTech, which has been involved in the development of the standard, is committed to releasing a conformant implementation of the product as soon as commercially practical after ratification.

The development of OpenFusion JDO has benefited from PrismTech's great experience in mapping complex object models to relational databases. This experience has been gained over many years of implementing models from the Oil and Gas industry (the POSC Epicentre model) and the process industry (the EPISTLE) model.

Product Features

OpenFusion JDO is an implementation of the mandatory requirements of the 0.98 version of the JDO specification for relational database systems. All major relational databases are supported, including Oracle, Sybase, Informix, IBM DB2 and Microsoft SQL Server. Relational schemas and application model to relational model mappings are generated automatically during the enhancement process. Version 1 of the product will support a single mapping convention of one relational table per class hierarchy. Access to relational databases requires a JDBC driver.

OpenFusion JDO offers the following features:

- Implementation of Application Identity and Datastore Identity
- Implementation of all mandatory lifecycle states
- Support for first and second class objects and collections

- Full implementation of the `PersistenceCapable` interface
- Instance Callbacks
- All mandatory methods of the `PersistenceManagerFactory` interface implemented
- All mandatory methods of the `PersistenceManager` interface implemented
- All mandatory methods of the `Transaction` interface implemented
- Full implementation of the `Query` interface and JDOQL, with the exception of bitwise complement and string concatenation operators.
- Full implementation of the `Extent` Interface

In addition, OpenFusion JDO includes PrismTech's implementation of the JDO byte code enhancer.

OpenFusion JDO has been carefully architected to enable re-implementation using alternative persistence mechanisms should this need arise.

Conclusion

In this paper, we have shown that problems exist with current methods of persisting data in Java applications. We have discussed JDO as a solution to those problems and have shown how it can be used with a simple example. We have shown how JDO can be used as a persistence mechanism for Enterprise Java Beans and finally, we have introduced a commercial implementation of the JDO specification in PrismTech's OpenFusion JDO.

References

The information in this paper has been drawn from a number of published sources and from PrismTech's experience in developing OpenFusion JDO. The principal published sources are:

1. Java Data Objects Specification, Version 0.96 (Proposed Final Draft), Sun Microsystems, 2001
2. Developing Enterprise Applications with the Java 2 Platform, Enterprise Edition, Version 1, Sun Microsystems, 1999
3. Enterprise Java Beans Specification, Version 2.0 (Proposed Final Draft), Sun Microsystems, 2000.