Object Oriented Persistence Middleware

System Architectures
Principles and techniques
Persistence Managers: OBJ, Hibernate

References
- S. Ambler: The Design of a Robust Persistence Layer
  http://www.ambysoft/persistenceLayer.pdf
- S. Ambler: Encapsulating Database Access
- Java Data Objects
  http://access1.sun.com/jdo/
- The Object Relational Bridge (OBJ)
  http://db.apache.org/ojb/index.html
- Relational Persistence For Idiomatic Java
  http://www.hibernate.org/
- Gopalan Suresh Raj: Java Data Objects (JDO)
  http://my.execpc.com/~gopalan/java/jdo/jdo.html
  http://citeseer.nj.nec.com/yoder98connecting.html

Architecture for Persistent objects

Design principles
- Keep language clean from DB access code
- “persistence manager” between client code and DB
- DB specific code generated for use by the persistent mgr

API for Application Program ?
(“Business Logic”)
Persistence Manager
Mapping to Database?
ODMG OQL
EJB-QL
Java Data Objects JDO
All hbc

Persistance Layer

Benefits
- Reduction of database and object schema coupling
- Implementation of DB related code in one place: less effort for adapting to DB schema changes (“evolution”)
- Makes application development easier: concentrate on application logic but DB access code
- Allows for data oriented business rules e.g. “if an employee earns more than 100K he/she does not get overtime payment” But integrity constraints?

Risks
- Direct DB access is easier for small applications -> investment required
- Loose of control over DB access, e.g. when to write objects to DB example: EJB Container Managed Persistence
- Replication of DB functionality
- Programming language dependent

Principle Persistence Architectures

Traditional: Direct database access

Business objects
Separate Data Access objects
Persistence Layer
- Transaction facilities
- Query language mapping
- ... and much more

Direct DB access

Customer

Construct select
select

Work on data
Result set

Violates most of the design principles
- No “separation of concerns”
- ... which means here: application logic and DB access should be separated
  - Schema evolution
  - DB-vendor independence (but language independence is missing from most frameworks)
Direct data access

- SQL / OQL... statements for implementing the "customer" operations
- JDBC standard when using Java
- ... but code is written quickly
- viable for small applications

Data Access Objects (DAO)

- Encapsulation of DB access logic in classes independent from business objects
- Typically one access object for each business object
- Access Objects may or may not follow standards (ActiveX ADO, JDO).

Persistence Framework

- Mapping between DB data and objects defined in a Meta Data Repository (XML document, database, ...)
- Basic functionality: create, read, update, write, transaction support
- Enhanced: fault tolerance, mapping generation, caching etc.

Persistence Framework

- Features
  - Implicit persistence:
    - framework automatically makes business objects persistent
    - no action taken by application program
  - Examples: Enterprise Java Beans, Java Data Objects (container managed)
  - Explicit persistence
    - application program indicates when to save
    - Most common in Persistence frameworks
    - DB access generated:
      - dynamically: flexible
      - at compile time: simpler, better performance
  - during system start up
    - (example: Hibernate framework)

Persistence Framework

- Steps during Data access in Meta Data driven

Persistence Framework

- Reading a business object via persistence framework
Example(1) Meta data Query for
"Return all orders placed Jan. 27, 2003"

<Query>
  <Search For>Order</Search For>
  <Clause>
    <Attribute>"dateOrdered"</Attribute>
    <Comparison>"="</Comparison>
    <Value>"27-Jan-2003"</Value>
  </Clause>
  <Clause>
    <Attribute>"subtotalBeforeTax"</Attribute>
    <Comparison>">="</Comparison>
    <Value>"1000.00"</Value>
  </Clause>
</Query>

Example(2): Mapping for Order class / table

<table>
<thead>
<tr>
<th>Property</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>orderID</td>
<td>Order/orderID</td>
</tr>
<tr>
<td>dateOrdered</td>
<td>Order/dateOrdered</td>
</tr>
<tr>
<td>dateFulfilled</td>
<td>Order/dateFulfilled</td>
</tr>
<tr>
<td>getTotalTax</td>
<td>Order/getTotalTax</td>
</tr>
<tr>
<td>subtotalBeforeTax</td>
<td>Order/subtotalBeforeTax</td>
</tr>
<tr>
<td>shipToContactID</td>
<td>Order/shipToContactID</td>
</tr>
<tr>
<td>billToContactID</td>
<td>Order/billToContactID</td>
</tr>
<tr>
<td>lastUpdate</td>
<td>Order/lastUpdate</td>
</tr>
</tbody>
</table>

Mapping typically very simple

Example (3)

```
SELECT *
FROM Order
WHERE Order.DateOrdered = '2003-01-27'
AND Order.SubtotalBeforeTax > 1000.00
```

Result presentation generated as well

More sophisticated mappings needed ... but not provided in most frameworks

How to employ efficient data base query processing when loading objects?

Why complex mappings are needed

- Customer object with two orders and two / one items
- Data base representation using three tables
- Wanted: All Customers located in Berlin with orders of july 30 with items not yet delivered (items may be delivered individually)

Complex mapping (cont.)

```
Select c.name, c.something, o.id, o.etc, i.no, i.etc
from customer c, order o, item i
where o.cid = c.cid
and c.city='Berlin' and o.date='7-30-2003'
and i.id in (select j.id from item j
where j.oid = o.oid
and j.delivered = false)
```

Map result tuples to objects customer, order, item

Inefficient alternative:

- Access tables independently
- Many independent database accesses: customer, for each customer his order, for each order its items: expensive

Implementation issue: Lazy loading, otherwise the object construction will take some time...

Service: an operation offered by a computing entity that can be invoked by other computing entities (Ambler)

Examples

- Corba
- Web services
- Stored procedures
**Comparison: DB access from business object**

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Disadvantage</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very simple approach. Can develop code very quickly. Can support access to very bad data designs (although performance may suffer).</td>
<td>Directly couples your object schema to your data schema. Application developers need to learn database access language (e.g. SQL). Database refactoring impeded due to high coupling. Difficult to reuse database access code.</td>
<td>At beginning of a project when your persistence approach is still in flux. For small applications (less than 20 business classes) and/or prototypes.</td>
</tr>
</tbody>
</table>

**Comparison: DB access via Data objects**

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Disadvantage</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database access code encapsulated into its own set of classes. Business classes no longer coupled to database. Database refactoring easier due to lowered coupling. Can support access to very bad data designs (although performance may suffer). Possible to reuse data access objects.</td>
<td>Object schema still coupled to your data schema, via the data access objects. Application developers need to learn SQL. Often platform specific.</td>
<td>Medium-sized application (20-100 business classes).</td>
</tr>
</tbody>
</table>

**Comparison: DB access via persistence framework**

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Disadvantage</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application programmers do not need to know the data schema. Application programmers don't even need to know where the data is stored. Frameworks reflect performance expertise of its builders. Administration facility can ease database refactoring because it simplifies impact analysis by tracing columns to object attributes. Administration facility aids performance tuning because it makes it easy to change mappings. Possible to reuse framework and mapping meta data between applications.</td>
<td>Perceived performance impact to your applications (if the framework is poorly built). Requires reasonably clean data designs because the framework may not support the overly complex mappings. Often platform specific.</td>
<td>Medium and large sized applications. When it is common practice within your organization to use a persistence framework.</td>
</tr>
</tbody>
</table>

**Comparison: Service access to data**

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Disadvantage</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potential to create platform independent services. Web services quickly becoming an industry standard. Supports reuse between applications.</td>
<td>Web services standards and tools still evolving. Performance becomes a problem when combining several services in serial or simply when services are invoked across a network.</td>
<td>Medium to large sized applications. Whenever an appropriate service already exists that you can reuse.</td>
</tr>
</tbody>
</table>

**Persistence Frameworks**

- **ObjectRelationalBridge (OBJ)**
  - O/R Mapping tool and persistence framework by Apache (open source)
  - Supports various interfaces to application objects
    - ODMG 3.0
    - JDO
    - CustomLanguage OBJ
  - Sophisticated mappings (1:n, n:n)
  - Cache
  - Lazy loading
  - Distributed lock management (pessimistic)
  - In addition optimistic synchronisation
  - Prefetching relations
  - Interface to many relational DBS
Persistence Frameworks: Hibernate

- Hibernate (free software, LGPL)
  - mapping at startup time
  - High performance
  - Mapping using reflection facilities
  - Custom language
  - ODMG 3.0 as beta implementation
  - Lazy loading
  - Cache and query cache (optional)
  - Outer join, loading object graph with one select
  - Optimistic synchronisation
  - Extensive documentation

Hibernate example

The database

CREATE TABLE `users` (
  `LogonID` varchar(20) NOT NULL default '0',
  `Name` varchar(40) default NULL,
  `Password` varchar(20) default NULL,
  `EmailAddress` varchar(40) default NULL,
  `LastLogon` datetime default NULL,
  PRIMARY KEY (`LogonID`));

The application classes

```java
public class User {
    private String userID;
    private String userName;
    private String password;
    private String emailAddress;
    private Date lastLogon;

    public String getID() { return userID; }
    public void setID(String newUserID) { userID = newUserID; }
    // ... a bunch of other properties // using getXXX() and setXXX()
}
```

Defining the mapping

```xml
<?xml version="1.0"?
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping.dtd">
<hibernate-mapping>
  <class name="dbdemo.User" table="users">
    <id name="ID" column="LogonId" type="string">
      <generator class="assigned"/>
    </id>
    <property name="userName" column="Name" type="string"/>
    <property name="password" type="string"/>
    <property name="emailAddress" type="string"/>
    <property name="lastLogon" type="date"/>
  </class>
</hibernate-mapping>
```

Define Database (property file)

Create Datastore object

```java
Datastore ds = Hibernate.createDatastore();
```

Build session object session

```java
session = session.openSession();
```

Business logic, custom query language

```java
List myUsers = session.find("from User in class dbdemo.User where User.ID = :id", "joe_cool", Hibernate.STRING);
if (myUsers != null) {
    for (User nextUser : myUsers) {
        System.out.println("Resetting password for User: " + nextUser.getUserName());
        nextUser.setPassword("secret");
    }
} else {
    System.out.println("Didn't find any matching users..");
}
```

Close session

```java
session.close();
```

Define Database (property file)

Create Datastore object

```java
Datastore ds = Hibernate.createDatastore()
```