Object-to-Relational Mapping: The Crossing Chasms Pattern and Implementation Considerations

Use of Meta Data in the Java Persistence Layer

Presented by Toby Sarver
Agenda

• What is a Persistence Layer and why do I want one?
• Crossing Chasms: storing objects in a relational database.
• Process for applying the Crossing Chasms pattern.
• Overview of two implementations.
What Is a Persistence Layer?

Retrieve

Change

Store
Why Do I Need a Persistence Layer?

• Objects need to live longer than the program.
• Change boundary
  – Protects the application developer from changes in the structure of the database.
• Encapsulates the business logic rather than simply manipulating the data.
Crossing Chasms
Recap of “Crossing Chasms”

- Do initial object-oriented design, then consider persistence.
- Implement a persistence framework as early as possible.
- Use the “Broker” pattern rather than placing in the domain objects or subclassing.

<table>
<thead>
<tr>
<th>Object</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainObject</td>
<td>Table (Record)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column</td>
</tr>
<tr>
<td>Object Reference</td>
<td>Foreign Key</td>
</tr>
</tbody>
</table>
Additions to “Crossing Chasms”

• Consider using a class for each ObjectID because what makes an object unique can be different for each class.
  – Consider EJB’s “FooKey” interface
  – The ObjectID can interact with the persistence layer for reading and writing itself, as well as searches

• Differentiate between references to independent objects vs. “owned” objects (containment).
Responsibilities of a Persistence Framework

• Identify transaction boundary (a.k.a. “Unit of Work”), which objects go together
• Store and retrieve single objects, object references, and collections of objects
• “Speak” to the persistence mechanism (e.g., call JDBC API)
• Search for objects based on attributes
Why Do I Need a Persistence Framework?

• Hide complexity
  – not everyone has to understand the pattern

• Reduce amount of code

• Centralize “Best Practices”
  – Refine the code as you learn more or require more functionality

• Encapsulate the persistence mechanism
  – Implements a well-defined interface for persistence
  – ADO, JDBC, native Oracle, stored procedures, etc.
Applying the “Crossing Chasms” Pattern

- Identify the objects that need to be persisted, and the transaction boundaries.
- Deal with inheritance, if any.
- Write mechanism for storing and retrieving individual objects.
- Overlay a tree structure on the instance graph
  - tree, parent, and sibling pointers
Applying the “Crossing Chasms” Pattern (2)

• Examine each object relationship and model it in the database and the persistence layer appropriately.
  – cardinality
  – ownership
Modeling Relationships

- Owned child (single or multiple)
  - use standard foreign key of parent in child record
- Heterogeneous collection of owned children
  - same as above, but search multiple tables
- Sibling Pointer
  - “swizzle” the ObjectID into an object reference one all siblings are retrieved.
- Many to many
  - Use a mapping table, possibly a mapping object
Modeling Relationships (2)

• Non-Owned Relationship
  – Keep a registry of such objects
  – “Swizzle” the ObjectID to object reference
  – For example, keep a registry of all publicly-traded stocks. A customer owns the stock, but only stores the reference (e.g., the ticker).
Implementation Considerations

• Implementation considerations:
  – Use of meta-data, if any
  – Options in distributing the responsibilities of the persistence layer

• Storing individual objects
  – Naïve: delete row and insert new value
  – Evolved: detect new vs. existing, do Insert or Update
  – Nirvana: keep dirty bit for each attribute, only Update the changed attributes, skip if no change
Implementation Considerations (2)

• Storing Collections of Objects (same parent)
  – Naïve: Delete all child rows and re-insert
  – Evolved:
    • Detect an Add: Insert it, otherwise Update it
    • Detect a Delete: Delete the corresponding row

• Storing Network of Objects
  – Naïve: Delete all objects from the root and re-insert
  – Nirvana: Detect all changes and only make required DB accesses (a.k.a., Partial Persistence)
Overview of Implementations

• Example of JDBC Persistence
• Example of ADO Persistence
• Discussion of Partial Persistence
• Some Similarities Between the Two
  – e.g., use of meta data
    – **Ref:** PLOP 1: “Bridging the Gap Between Objects and Relational Databases” by Kyle Brown et al.
    – **Ref:** PLOP 2: “Reflection” by Buschmann et al.
JDBC Implementation

– Written for a large financial services firm in Boston.
– Needed to support a highly functional, Web-deployed application.
– Used the Visitor pattern extensively.
– Used the meta data to generate some Java classes
O2R Mapping

• Identity
  – get / set DbId()
  – Assigned by the persistence layer (DB)

• Pointers
  – Tree Pointers
    • Explicit during read
    • Implicit during write
  – Parent Pointers
    • Implicit (Visitor)
  – Non-containing Pointers

• Sibling Pointers
  • Explicit (Visitor)

• Subclassing
  – Data-oriented
    • Used separate tables with overlapping column names
  – Named
    • Used a Java class name in the table
  – Visitor handled creating the right subclass
How the Responsibilities Are Delegated (JDBC)

- **Visitor**: Identify transaction boundary (a.k.a. “Unit of Work”), which objects go together
- **Persistent Row** (generated): Store and retrieve single objects, object references, and collections of objects
- **Persistent Row**: “Speak” to the persistence mechanism
- **Persistent**: Search for objects based on attributes
Partial Persistence

– Original Strategy
  • load the whole Customer and all associated objects when the user logs in
  • save the whole Customer … when the session times out
– Problem was that if the server went down in this period, the user loses all the work
– Partial Persistence Strategy
  • load the whole Customer on session start, as before
  • save whatever objects changed during *each request*
    – after handling a request, the database is in sync with memory
Overview Partial Persistence

update 4 2
insert 7
delete 5

Delta

© Copyright 1999, 2001 Thomas W. Sarver
Delta Rules

- **Update**
  - If an object is being inserted or deleted, ignore update

- **Insert**
  - Always insert

- **Delete**
  - If the object is being inserted, remove it from the inserted list
  - If the object is being updated, move to deleted list

- **Dependencies**
  - Has rules for dealing with dependencies
  - Essentially does operations in order from root to leaf
Meta Data (JDBC)

– Stored as a script
– Object
  • Attribute : Type
    – implied naming conventions, get<attr>(), set<attr>()
    – exceptions were possible
      » for example, is<attr>, set<attr>(boolean)
– Generated to Java classes
  • <Object>PersistentRow extends PersistentRow
  • standard interface, fill(), empty()
  • method for each attribute
  • custom coding is possible at the Persistent level
ADO Example

• Architectural objectives
  – Build domain objects in memory from multiple sources
  – Encapsulate persistence from other parts of the system
  – Redeploy the app to another persistence mechanism (ADO, JDBC, ERP) with little effort

• Meta data present (and interpreted) at run time

• Introduces View and ViewItem
View and ViewItem

- Generic concept relating “parts to whole”
- Basis for translating between different architectural layers
- The concrete instances implement the “Base” layer (see Reflection pattern)
  - DomainViewItem knows how to read and write attributes of a domain object.
  - ADOViewItem knows how to read and write fields in a Recordset.

<table>
<thead>
<tr>
<th>View</th>
<th>ViewItems</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainObject</td>
<td>Attributes</td>
</tr>
<tr>
<td>Table (Record)</td>
<td>Columns</td>
</tr>
<tr>
<td>HTML Form</td>
<td>Form Item</td>
</tr>
</tbody>
</table>

© Copyright 1999, 2001 Thomas W. Sarver
How the Responsibilities Are Delegated (ADO)

• **DomainObject**: Identify transaction boundary (a.k.a. “Unit of Work”), which objects go together

• **ADOPersistence**: Store and retrieve single objects, object references, and collections of objects

• **ADOPersistence**: “Speak” to the persistence mechanism

• **None**: Search for objects based on attributes
Class Diagram (ADO 1)

**View**

+transfer(fromObj : Object, doView : View, toObj : Object) : void
+finishedWriting()
+finishedReading()
+getViewItems() : Vector
+transferMany(parentObj : Object, fromVector : Vector, toParent : Object, toView : View) : Vector
+prepareOneRead(fromObj : Object, toObj : Object)
+prepareMultiRead(parentObj : Object, fromObjs : Vector) : Enumeration
+prepareOneWrite(fromObj : Object)
+prepareMultiWrite(fromObj : Object, fromVector : Vector, toObj : Object)

**ViewItem**

+getName() : String
+getParent() : View
+read(srcObject : Object)
+write(srcObject : Object, newValue : Object)

**DomainView**

**DomainViewItem**

**DomainObject**

+getFullView() : DomainView
+retrieve(persistId : Object)
+retrieveMany() : Vector
+onRetrieve(persistComp : IPersistenceComponent) : Object
+store()
+storeMany(persistComp : IPersistenceComponent, parentObj : IPersistable, toStore : Vector)
+onDelete(persistComp : IPersistenceComponent)
+delete()
+deleteMany(persistComp : IPersistenceComponent, parentObj : IPersistable, toDelete : Vector)
+onDelete(persistComp : IPersistenceComponent)

**IPersistable**

+getPersistId() : Object
+setPersistId(persistId : Object)
Class Diagram (ADO 2)

**ADOPersistenceFactory**
- `#views : Vector`
- `+getInstance() : Object`
- `+getView(viewId : ViewID) : PersistenceView`
- `+assignNextIdFor(viewId : ViewID)`

**ADOPersistence**
- `+getConnection() : com.ms.data.ado.Connection`
- `+loadView(viewId : ViewID) : ADOView`
- `+assignNextIdFor(viewId : ViewID) : int`

**ADOView**
- `+getDescription() : String`
- `+getTableName() : String`
- `+getKeyColumn() : String`
- `+getParentColumn() : String`
- `+getPersistence() : ADOPersistence`
- `+getCurrentRecordset() : ADORecordset`
- `+assignNextId() : int`
- `+getLastAssignId() : int`

**IPersistenceComponent**
- `+getView(aViewID : ViewID) : PersistenceView`
- `+finishedReading()`
- `+finishedWriting()`
- `+abortTrans()`

**ADOViewItem**
- `viewItems`

**ViewItemTypeID**
- `hasType`
Instance Diagram

DomainObject

DomainView

DomainView Items

read

write

store

ADOView

ADOView Items

write

read

retrieve

ADOPersistence

Recordset

Connection
Wrap Up ADO

- Met the architectural objectives
  - Can have multiple instances of PersistenceFactory and implementations of IPersistenceComponent in a system
  - DomainObject has methods like retrieve(Object) and store() to encapsulate the persistence layer
  - Can redeploy to another persistence mechanism by implementing IPersistenceComponent, View, and ViewItem
Wrap Up

– Like all patterns, the “Crossing Chasms” pattern still leaves you with many design decisions.
– Use the Broker pattern
  • Avoid having the Domain Objects “know” how to persist themselves
  • This makes it very difficult to redeploy to another persistence mechanism
  • The use of DomainVisitor and DomainView make this possible
– Use meta data when possible to represent the object mapping