Persistency Patterns

Repository and DAO
Repository pattern

• Basically, the Repository pattern just means putting a façade over your persistence system so that you can shield the rest of your application code from having to know how persistence works.

• Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects

• Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes.
Definition from: Martin Fowler. *Patterns of Enterprise Architecture.*

- "A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection.
- Client objects construct query specifications declaratively and submit them to Repository for satisfaction.
- Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes.
- Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer.
- Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers."
Repository solution

• For each type of object that needs global access, create an object that can provide the illusion of an in-memory collection of all objects of that type.

• Set up access through a well-known global interface. Provide *methods to add and remove* objects, which will encapsulate the actual insertion or removal of data in the data store.

• Provide *methods that select* objects based on some criteria and return fully instantiated objects or collections of objects whose attribute values meet the criteria, thereby encapsulating the actual storage and query technology.

• Provide REPOSITORIES only for AGGREGATE roots that actually need direct access.

• Keep the client focused on the model, delegating all object storage and access to the REPOSITORIES.
Client asks for what it wants in model terms

Client

selection criteria

matching objects

repository

REPOSITORY encapsulates database access technology and strategy

delegate

database interface
METADATA MAPPING
FACTORIES
other REPOSITORIES
QUERY OBJECTS
etc.
Repository advantages

- Encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer.

- Supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

- Decouples application and domain design from persistence technology, multiple database strategies, or even multiple data sources.

- Allows easy substitution of a dummy implementation, for use in testing (typically using an in-memory collection -mock).
Example

SaleWindow (from View)

SaleController (from Controller)

Product (from Domain)
- ID : Integer
- name : String

ProductRepository (from Persistancy)
+ add(p : Product)
+ delete(p : Product)
+ findById(id) : Product
+ save(p : Person)
+ findByName(name : String) : Collection<Person>

ProductRepositoryMock (from Persistancy)
ProductRepositoryFile (from Persistancy)
ProductRepositoryDB (from Persistancy)

View
Controller
Domain
Persistancy
Repository and Bridge
Abstract type

- A REPOSITORY "contains" all instances of a specific type, but this does not mean that you need one REPOSITORY for each class. The type could be an abstract superclass of a hierarchy.

- The type could be an interface whose implementers are not even hierarchically related. Or it could be a specific concrete class.

- This face with constraints imposed by the lack of such polymorphism in the database technology.
General:
findbyID

Specific:
findBySpecificCriteria
Take advantage of the decoupling from the client

- You have more freedom to change the implementation of a REPOSITORY than you would if the client were calling the mechanisms directly.

- You can take advantage of this to optimize for performance, by varying the query technique or by caching objects in memory, freely switching persistence strategies at any time.

- You can facilitate testing of the client code and the domain objects by providing an easily manipulated, dummy in-memory strategy.
Leave transaction control to the client

- Although the REPOSITORY will insert into and delete from the database, it will ordinarily not commit anything.

- It is tempting to commit after saving, for example, but the client presumably has the context to correctly initiate and commit units of work.

- Transaction management will be simpler if the REPOSITORY doesn’t have initiative.
Relation between Repositories and Factories

- The client of a REPOSITORY should be given the illusion that the objects are in memory. The object may have to be reconstituted, but it is the same conceptual object, still in the middle of its life cycle.

- The reconstitution of a stored object is not the creation of a new conceptual object.
- In this DD design, FACTORIES and REPOSITORIES have distinct responsibilities.
  - The FACTORY makes new objects;
  - the REPOSITORY finds old objects.

A FACTORY handles the beginning of an object's life; a REPOSITORY helps manage the middle and the end.
Data Access Object

• Problem
You want to encapsulate data access and manipulation in a separate layer.

• Forces
  – You want to implement data access mechanisms to access and manipulate data in a persistent storage.
  – You want to decouple the persistent storage implementation from the rest of your application.
  – You want to provide a uniform data access API for a persistent mechanism to various types of data sources, such as RDBMS, LDAP, OODB, XML repositories, flat files, and so on.
  – You want to organize data access logic and encapsulate proprietary features to facilitate maintainability and portability.

• Solution
Use a Data Access Object to abstract and encapsulate all access to the persistent store. The Data Access Object manages the connection with the data source to obtain and store data.
Class Diagram
Sequence Diagram
Related Patterns

- **Transfer Object**
  Data access objects in their most basic form use transfer objects to transport data to and from their clients. Transfer Object are used in other strategies of Data Access Objects. The RowSet Wrapper List strategy returns a list as a transfer object.

- **Factory Method** [GoF] and **Abstract Factory** [GoF]
  The Data Access Object Factory strategies use the Factory Method pattern to implement the concrete factories and its products (DAOs). For added flexibility, the Abstract Factory pattern is used as described in the Data Access Object Factory strategy.

- **Broker** [POSA1]
  The DAO pattern is related to the Broker pattern, which describes approaches for decoupling clients and servers in distributed systems. The DAO pattern more specifically applies this pattern to decoupling the resource tier from clients in another tier, such as the business or presentation tier.

- **Transfer Object Assembler**
  The Transfer Object Assembler uses the Data Access Object to obtain data to build the composite transfer object it needs to assemble and send as the model data to the client.

- **Value List Handler**
  The value list handler needs a list of results to act upon. Such a list is often obtained using a Data Access Object, which can return a set of results. While the value list handler has an option of obtaining a RowSet from the Data Access Object, it might be better to obtain and use a RowSet Wrapper List instead.
Java Database Connectivity (JDBC) API contains a set of classes that allows the access to the data.

Any type of sources could be accessed: relational data bases, spreadsheets, or files.

JDBC provides a set of interfaces that could be used for building specialized tools.

Packages:
- `java.sql` contains classes and interfaces for data accessing and processing.
- `javax.sql` contains classes and interfaces for adding new functions for the server side.
Establishing a connection

Two possibilities:

- **Class **DriverManager**:  
  - This class imposes the loading of a driver specific to the data base and then the connection is created using an URL.

- **Interface DataSource**:  
  - It is recommended for complex applications since it is allowed to configure the data source in a transparent way.
Connection creation
Using the interface **DataSource**

```
InitialContext ic = new InitialContext()
   //a)
DataSource ds = ic.lookup("java:comp/env/jdbc/myDB");
   Connection con = ds.getConnection();
   //b)
DataSource ds = (DataSource)org.apache.derby.jdbc.ClientDataSource()
   ds.setPort(1527);
   ds.setHost("localhost");
   ds.setUser("APP")
   ds.setPassword("APP");
   Connection con = ds.getConnection();
```
Using the class **DriverManager**

- Two steps:
  
  - Load the corresponding driver into the memory:
    (variants…)

  \[
  \text{Class.forName(<DriverClassName>)};
  \]

  - **Class.forName** creates an instance of the driver and registers it to the **DriverManager**.
    
    - It is not necessary to create an instance of the class.
    
    - Ex: \text{Class.forName(“com.mysql.jdbc.Driver”)};

- Connection creation
Connection creation using the class `DriverManager`

- The interface `Driver` is used for managing the drivers for a JDBC client.

- When the client asks for a connection and gives an URL, the class `DriverManager` is responsible to find the driver that could recognize the URL and to use it for establishing the connection.

- URL Syntax for a connection:
  
  `jdbc:subprotocol:<DataBase_identifier>[PropertiesList]`

```java
Connection conn = DriverManager.getConnection("jdbc:derby:COFFEEES");

String url = "jdbc:mysql:Fred";
Connection conn = DriverManager.getConnection(url, <user>, <passwd>);
```
Class Connection

• Represents a session with a particular data base.
• Any SQL statement is executed and the results are sent using the context of a connection.
• Methods:
  – close(), isClosed():boolean
  – createStatement(…):Statement //overloaded
  – prepareCall(…):CallableStatement //overloaded
  – prepareStatement(…):PreparedStatement //overloaded
  – rollback()
  – setAutoCommit(boolean) //tranzaction
  – getAutoCommit():boolean
  – commit()
Class Statement

• Is used for executing a static SQL statement and for returning the result.

• Methods:
  – `execute(sql: String, ...): boolean // for any statement`
    • `getResultSet(): ResultSet // SQL`
    • `getUpdateCount(): int`
  – `executeQuery(sql: String, ...): ResultSet // pentru SELECT`
  – `executeUpdate(sql: String, ...): int // INSERT, UPDATE, DELETE`
  – `cancel()`
  – `close()`
Example

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>AutoNumber</td>
</tr>
<tr>
<td>title</td>
<td>Text</td>
</tr>
<tr>
<td>authors</td>
<td>Text</td>
</tr>
<tr>
<td>isbn</td>
<td>Text</td>
</tr>
<tr>
<td>year</td>
<td>Number</td>
</tr>
</tbody>
</table>
Example

//insert
String insert="insert into books (title, authors, isbn, year) values ('Nuvele', 'Mihai Eminescu', '4567567', 2008)";
Statement stmt=null;
try {
    stmt=conn.createStatement();
    stmt.executeUpdate(insert);
} catch (SQLException e) {
    System.out.println("Insert error "+e);
}finally{
    if (stmt!=null)
        stmt.close();
}

//delete
String delString=“delete from books where isbn=‘tj234’”
Statement stmt=conn.createStatement();
stmt.executeUpdate(delString);
stmt.close();
ResultSet

- Contains a table that represents a result of a SELECT statement.
- An object of ResultSet type contains a cursor that indicates the current line in the table.
- Initially the cursor is set before the first line.
- The method next moves the cursor on the next line. When there are no other lines the result is equal to false.

Some configurations are possible: the table could be modified, the iteration way, etc.)
- These could be done using the method createStatement(...):

```java
Statement stmt = con.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT name, address FROM users");

// rs could be iterated, can be modified,
// and it is not notified about the updates made by other users.
```
ResultSet

- Methods:
  - absolute(row:int)
  - relative(n:int)
  - afterLast(), beforeFirst(), first(), last(), next():boolean
  - getRow():int
  - getInt(columnIndex|columnLabel):int
  - getFloat(...), getString(...), getObject(...), etc
  - updateInt(columnIndex,columnLabel, newValue)
  - updateFloat(...), updateString(...), etc
  - updateRow()
  - refreshRow()
  - rowDeleted(), rowInserted(), rowUpdated()

- Implicitly an object of type ResultSet is unidirectional, with forward iteration, and unmutable.
ResultSet

Statement stmt=conn.createStatement();
ResultSet rs=stmt.executeQuery("select * from books");

while(rs.next()){
    System.out.println("Book "+
        rs.getString("title")+
    
        ' +rs.getString("author")+
        ' +rs.getInt("year")
    );
}
rs.close();
stmt.close();
Class **PreparedStatement**

- An instance of type **PreparedStatement** receives a SQL statement when it is created.
- The SQL statement is sent to the SGBD of the database, where it is compiled.
- When a statement associated to a **PreparedStatement** is sent, SGBD directly executes the SQL statement without any other verification.
- It is more efficient than **Statement**.

- The parameters marked using the character ‘?’.  
  ```java
  PreparedStatement preStmt = 
    con.prepareStatement("select * from books WHERE year=?");
  ```

- The value of a parameter is sent using setter methods (setXYZ, XYZ represents the parameter type).
  ```java
  preStmt.setInt(1, 2008);
  ResultSet rs=preStmt.executeQuery();
  ```
Transactions

- Implicitly each SQL statement is treated as being a transaction and it is registered/operated immediately after the execution.

- The implicit behavior could be modified using the method: `setAutoCommit(false)` of the class `Connection`.

Methods:
- `Commit`
- `Rollback`
- `setSavePoint`
Example

c.on.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?"");
updateSales.setInt(1, 50);
updateSales.setString(2, "Black");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?"");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Black");
updateTotal.executeUpdate();
c.on.commit();
c.on.setAutoCommit(true);
Identity map pattern

• Ensures that each object gets loaded only once by keeping every loaded object in a map.
• Looks up objects using the map when referring to them.