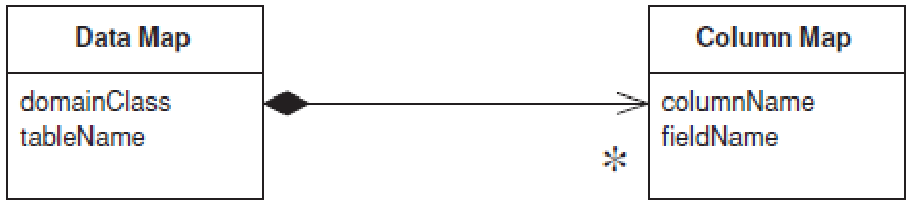
SOEN 387: Web-Based Enterprise Application Design

Chapter 13. Object-Relational Metadata Mapping Patterns

Metadata Mapping

Holds details of object-relational mapping in metadata.



Much of the code that deals with object-relational mapping describes how fields in the database correspond to fields in in-memory objects. The resulting code tends to be tedious and repetitive to write.

A *Metadata Mapping* allows developers to define the mappings in a simple tabular form, which can then be processed by generic code to carry out the details of reading, inserting, and updating the data.

Metadata Mapping is based on boiling down the mapping into a metadata file that details how columns in the database map to fields in objects.

Metadata means data about the data such as table names, column names, and column types.
See: https://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html

The biggest decision in using *Metadata Mapping* is how the information in the metadata manifests itself in terms of running code. There are two main routes to take: **code generation** and **reflective programming**.

With **code generation** you write a program whose input is the metadata and whose output is the source code of classes that do the mapping. These classes look as though they're hand-written, but they're entirely generated during the build process, usually just prior to compilation. The resulting mapper classes are deployed with the server code.

A **reflective program** By treating methods (and fields) as data the reflective program can read in field and method names from a metadata file and use them to carry out the mapping.

Code generation is a less dynamic approach since any changes to the mapping require recompiling and redeploying at least that part of the software.

With a **reflective approach**, you can just change the mapping data file and the existing classes will use the new metadata.

Reflective programming often suffers in speed, although the problem here depends very much on the actual environment you're using—in some a reflective call can be an order of magnitude slower.

On most occasions you keep the metadata in a separate file format. These days XML is a popular choice as it provides hierarchic structuring while freeing you from writing your own parsers and other tools. A loading step takes this metadata and turns it into programming language structure, which then drive either the code generation output or the reflective mapping.

In simpler cases you can skip the external file format and create the metadata representation directly in source code. This saves you from having to parse, but it makes editing the metadata somewhat harder.

Another alternative is to hold the mapping information in the database itself, which keeps it together with the data. If the database schema changes, the mapping information is right there.

Metadata Mapping can greatly reduce the amount of work needed to handle database mapping.

However, some setup work is required to prepare the *Metadata Mapping* framework. Also, while it's often easy to handle most cases with *Metadata Mapping*, you can find exceptions that really tangle the metadata.

Metadata Mapping can make refactoring the database easier, since the metadata represents a statement of the interface of your database schema. Thus, alterations to the database can be contained by changes in the *Metadata Mapping*.

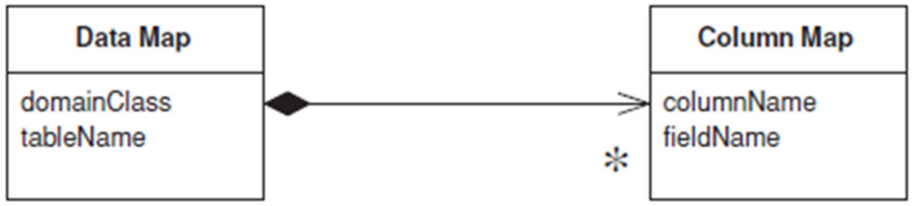
Example: Using Metadata and Reflection (Java)

Holding the Metadata The first question to ask about metadata is how it's going to be kept. Here I'm keeping it in two classes. The data map corresponds to the mapping of one class to one table. This is a simple mapping, but it will do for illustration.

You can write a routine to load the map from an XML file or from a metadata database.

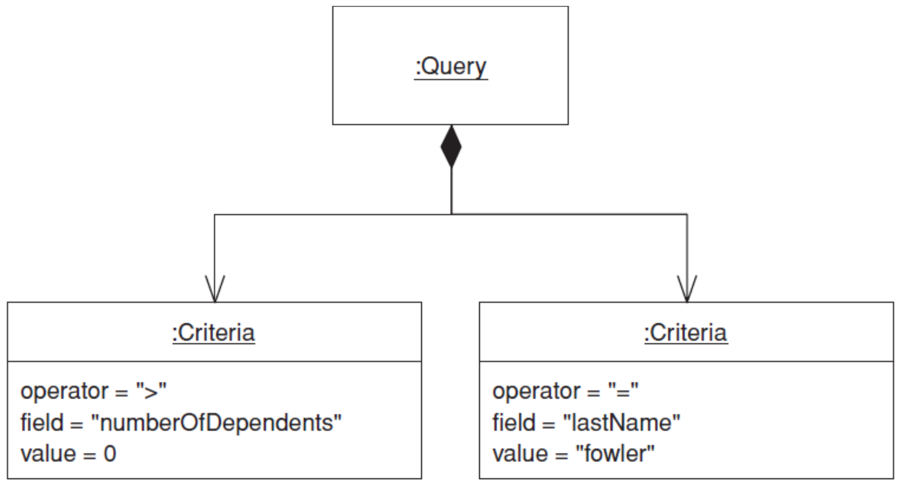
```
class DataMap...
        private Class domainClass;
        private String tableName;
        private List columnMaps = new ArrayList();


class ColumnMap...
        private String columnName;
        private String fieldName;
        private Field field;
        private DataMap dataMap;
```



Query Object

An object that represents a database query.



A *Query Object* is an interpreter , that is, a structure of objects that can form itself into a SQL query.

You can create this query by referring to classes and fields rather than tables and columns. In this way those who write the queries can do so independently of the database schema and changes to the schema can be localized in a single place.

Its primary roles are to allow a client to form queries of various kinds and to turn those object structures into the appropriate SQL string.

A common feature of *Query Object* is that it can represent queries in the language of the in-memory objects rather than the database schema. That means that, instead of using table and column names, you can use object and field names.

For multiple databases you can design your *Query Object* so that it produces different SQL depending on which database the query is running against

A particularly sophisticated use of *Query Object* is to eliminate redundant queries against a database. If you see that you've run the same query earlier in a session, you can use it to select objects from the *Identity Map* and avoid a trip to the database.

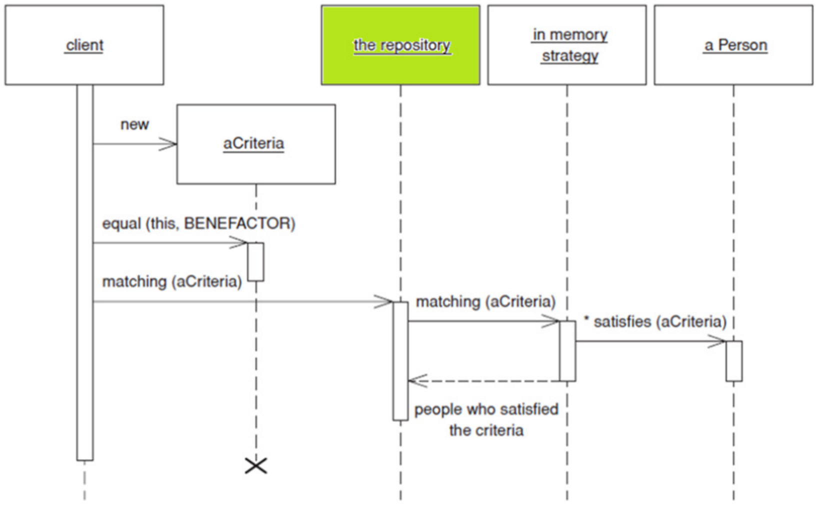
A more sophisticated approach can detect whether one query is a particular case of an earlier query, such as a query that is the same as an earlier one but with an additional clause linked with an AND.

Even then *Query Objects* aren’t always necessary, as many developers are comfortable with SQL. You can hide many of the details of the database schema behind specific finder methods.

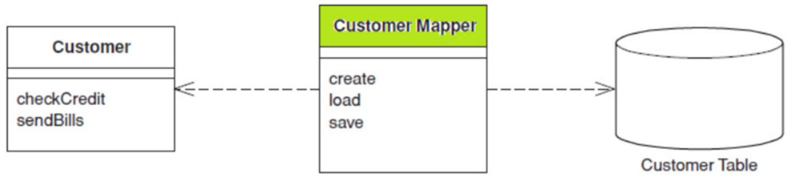
The advantages of *Query Object* come with more sophisticated needs: keeping database schemas encapsulated, supporting multiple databases, supporting multiple schemas, and optimizing to avoid multiple queries.

Repository

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



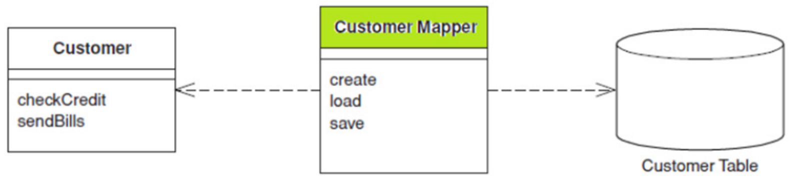
A system with a complex domain model often benefits from a layer, such as the one provided by *Data Mapper* , that isolates domain objects from details of the database access code.



A Data Mapper *insulates the domain objects and the database from each other.*

In such systems it can be worth while to build another layer of abstraction over the mapping layer where query construction code is concentrated. This becomes more important when there are a large number of domain classes or heavy querying. In these cases particularly, adding this layer helps minimize duplicate query logic.

A *Repository* mediates between the domain and data mapping layers, acting like an in-memory domain object collection.



A Data Mapper *insulates the domain objects and the database from each other.*

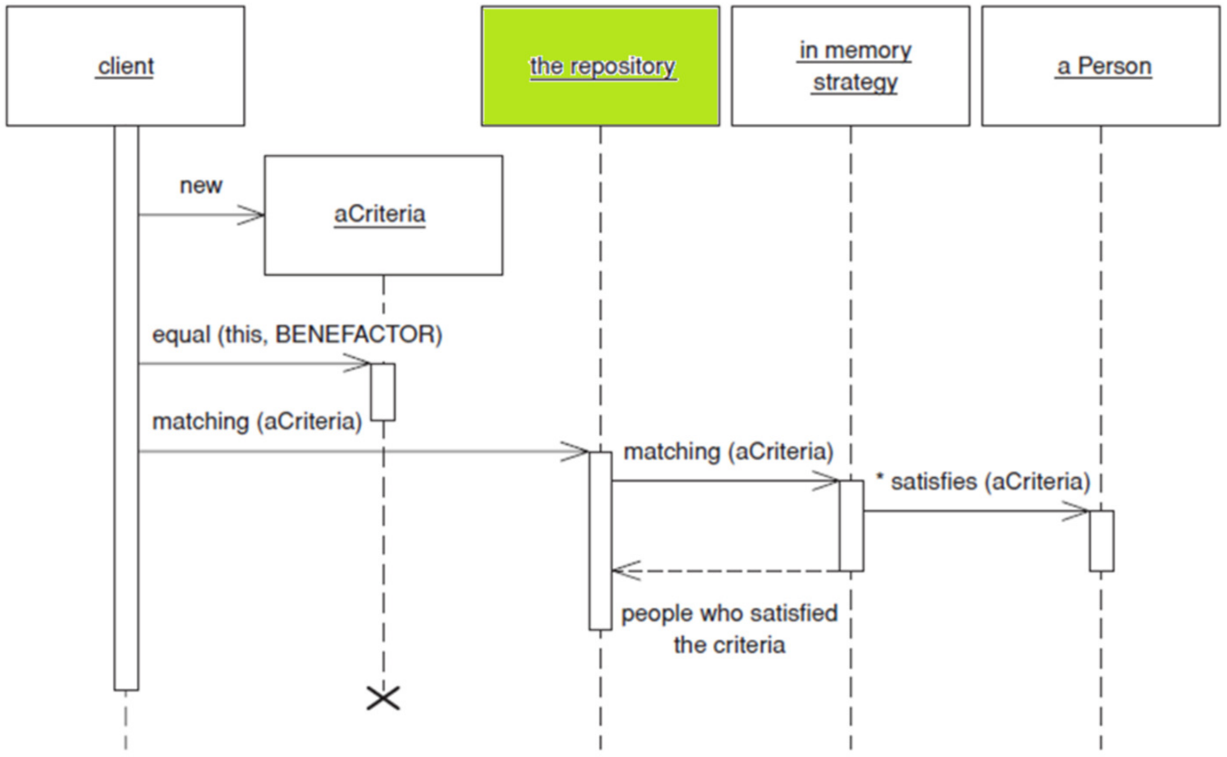
Client objects construct query specifications declaratively and submit them to *Repository* for satisfaction.

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 is a sophisticated pattern that makes use of a fair number of the other patterns. When used in conjunction with *Query Object* , *Repository* adds a large measure of usability to the object-relational mapping layer without a lot of effort.

Repository presents a simple interface. Clients create a criteria object specifying the characteristics of the objects they want returned from a query.



For example, to find person objects by name we first create a criteria object, setting each individual criterion like so: `criteria.equals(Person.LAST_NAME, "Fowler“)` and `criteria.like(Person.FIRST_NAME, "M").`
Then we invoke `repository.matching(criteria)` to return a list of domain objects representing people with the last name Fowler and a first name starting with M.

In a large system with many domain object types and many possible queries, *Repository* reduces the amount of code needed to deal with all the querying that goes on.

Repository promotes the Specification pattern (in the form of the criteria object in the examples here), which encapsulates the query to be performed in a pure object-oriented way.

Therefore, all the code for setting up a *query object* in specific cases can be removed. Clients need never think in SQL and can write code purely in terms of objects.