**Data Mapper**

**Motivation**

Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. Doing so leads to variant schemas; that is, the object schema and the relational schema don't match up.

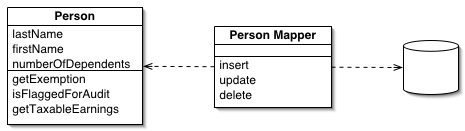
You still need to transfer data between the two schemas, and this data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, changes in one tend to ripple to the other.

The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.) Since it's a form of **Mapper**, **Data Mapper** itself is even unknown to the domain layer.

**Intent**

A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

**Implementation**



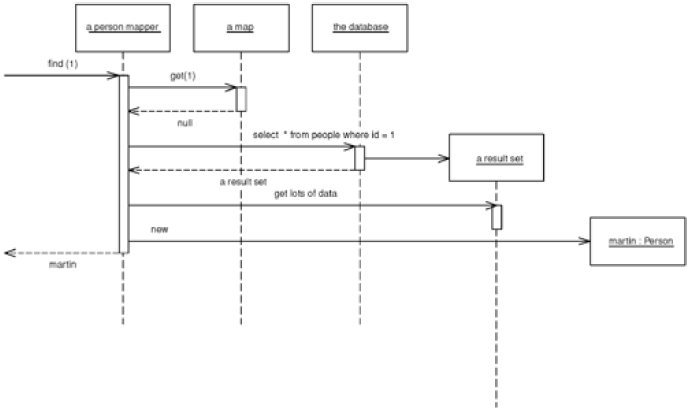
**How it works?**

The separation between domain and data source is the main function of a *Data Mapper,* but there are plenty of details that have to be addressed to make this happen.

Considering the simple Person and Person Mapper class example shown in the implementation.

**Retrieval**

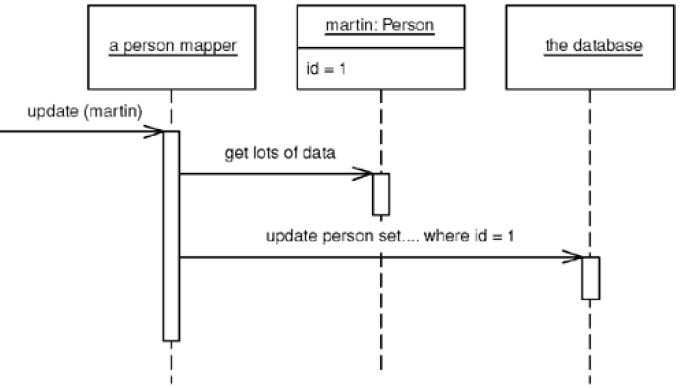
* To load a person from the database, a client would call a find method on the mapper
* The mapper uses an *Identity Map* to see if the person is already loaded; if not, it loads it.

****

**Handling Inserts and Updates**

Inserts and Updates, are handled as follows:

* A client asks the mapper to save a domain object.
* The mapper pulls the data out of the domain object and shuttles it to the database.

****

**Strategies to Implement Data Mapper**

The whole layer of *Data Mapper* can be substituted, either for testing purposes or to allow a single domain layer to work with different databases.

A simple *Data Mapper* would just map a database table to an equivalent in-memory class on a field-to-field basis.

Mappers need a variety of strategies to handle classes that turn into multiple fields, classes that have multiple tables, classes with inheritance, and the joys of connecting together objects once they've been sorted out.

**Unit of Work Pattern**

When it comes to inserts and updates, the database mapping layer needs to understand what objects have changed, which new ones have been created, and which ones have been destroyed.

It also has to fit the whole workload into a transactional framework.

The ***Unit of Work***pattern is a good way to organize this.

**Lazy Loading Pattern**

Loading a typical order with multiple order lines may involve loading the order lines as well.

The request from the client will usually lead to a graph of objects being loaded, with the mapper designer deciding exactly how much to pull back in one go.

The point of this is to minimize database queries, so the finders typically need to know a fair bit about how clients use the objects in order to make the best choices for pulling data back.

This example leads to cases where you load multiple classes of domain objects from a single query.

To load orders and order lines, it will usually be faster to do a single query that joins the orders and order line tables. You then use the result set to load both the order and the order line instances

Since objects are very interconnected, however it is necessary to stop pulling the data back at some point.

Otherwise it will be likely that the entire database will be pulled back with a request.

Again, mapping layers have techniques to deal with this while minimizing the impact on the in-memory objects.

Using Lazy Load, the in-memory objects can't be entirely ignorant of the mapping layer. They may need to know about the finders and a few other mechanisms.

**Metadata Mapping Pattern**

An application can have one Data Mapperor several. In case of hardcoding mappers, it's best to use one for each domain class or root of a domain hierarchy.

The Metadata Mapping patternenables the development of a single mapper class.

In the latter case the limiting problem is the find methods.

With a large application it can be too much to have a single mapper with lots of find methods, so it makes sense to split these methods up by each domain class or head of the domain hierarchy.

Therefore there are a lot of small finder classes, but it's easy for a developer to locate the finder needed.

As with any database find behavior, the finders need to use an *Identity Map* in order to maintain the identity of the objects read from the database. Either you can have a *Registry* of *Identity Maps* or you can have each finder hold an ***Identity Map***(providing there is only one finder per class per session).

**Handling Finders**

In order to work with an object, it needs to be loaded it from the database.

Usually the presentation layer will initiate things by loading some initial objects. Then control moves into the domain layer, at which point the code will mainly move from object to object using associations between them.

This will work effectively providing that the domain layer has all the objects it needs loaded into memory or *Lazy Load* to load in additional objects when needed.

In some cases the domain objects may need to invoke find methods on the *Data Mapper.*

With a good *Lazy Load,* this can be completely avoided. For simpler applications, though, it may not be worth trying to manage everything with associations and *Lazy Load.*

However there is still the need to avoid a dependency from domain objects to *Data Mapper.*

This dilemma can be solved by using ***Separated Interface Pattern****.*

The separated interface pattern implies putting any find methods needed by the domain code into an interface class that can be placed in the domain package.

**Applicability**

The primary occasion for using *Data Mapper* is when there is a need to let the database schema and the object model to evolve independently.

*Data Mapper's* primary benefit is that when working on the domain model, it is possible to ignore the database, both in design and in the build and testing process. The domain objects have no idea what the database structure is, because all the correspondence is done by the mappers.

This helps in the code because developer can understand and work with the domain objects without having to understand how they're stored in the database.

It also enables modification of the *Domain Model* or the database without having to alter either. With complicated mappings, particularly those involving existing databases, this is very valuable.

The price, of course, is the extra layer which is not required with *Active Record* so the test for using these patterns is the complexity of the business logic.

In case of a fairly simple business logic, there is no need to use a *Domain Model* or a *Data Mapper.* More complicated logic leads to *Domain Model* and therefore to *Data Mapper.*

It is not recommended to use *Data Mapper* without *Domain Model ,* but can is it recommended to use *Domain Model* without *Data Mapper?*

If the domain model is pretty simple, and the database is under the domain model developers' control, then it's reasonable for the domain objects to access the database directly with *Active Record.* Effectively this puts the mapper behavior discussed here into the domain objects themselves. As things become more complicated, it's better to refactor the database behavior out into a separate layer.

Remember that developers don't have to build a full-featured database-mapping layer. It's a complicated beast to build, and there are products available that do this. For most cases, It is recommended to use a database-mapping layer rather than building one from scratch.

Examples of PHP ORM:

* Doctrine
* Propel

**Example**

<?php

**require\_once** 'domainObjectException.php';

**abstract class** DomainObjectAbstract

{

**protected** $\_data = **array**();

**public function** \_\_construct(**array** $data = **NULL**)

{

**if** ($data !== **NULL**)

{

// populate domain object with an array of data

**foreach** ($data **as** $property => $value)

{

**if** (!**empty**($property))

{

$this->$property = $value;

}

}

}

}

// set domain object property

**public function** \_\_set($property, $value)

{

**if** (!array\_key\_exists($property, $this->\_data))

{

**throw new** ModelObjectException('The specified property is not valid for this domain object.');

}

**if** (strtolower($property) === 'id' AND $this->\_data['id'] !== **NULL**)

{

**throw new** DomainObjectException('ID for this domain object is immutable.');

}

$this->\_data[$property] = $value;

}

// get domain object property

**public function** \_\_get($property)

{

**if** (!array\_key\_exists($property, $this->\_data))

{

**throw new** DomainObjectException('The property requested is not valid for this domain object.');

}

**return** $this->\_data[$property];

}

// check if given domain object property has been set

**public function** \_\_isset($property)

{

**return isset**($this->\_data[$property]);

}

// unset domain object property

**public function** \_\_unset($property)

{

**if** (**isset**($this->\_data[$property]))

{

**unset**($this->\_data[$property]);

}

}

}

<?php

**class** DomainObjectException **extends** Exception{}

<?php

//Concrete Domain Object

**require\_once** 'domainObjectAbstract.php';

**class** User **extends** DomainObjectAbstract

{

**protected** $\_data = **array**('id' => **NULL**, 'firstname' => '', 'lastname' => '', 'email' => '');

}

**require\_once** 'mysqlAdapterException.php';

**class** MySQLAdapter

{

**private** $\_config = **array**();

**private static** *$\_instance* = **NULL**;

**private static** *$\_connected* = **FALSE**;

**private** $\_link = **NULL**;

**private** $\_result = **NULL**;

// return Singleton instance of MySQLAdapter class

**public static function** *getInstance*(**array** $config = **array**())

{

**if** (**self**::*$\_instance* === **NULL**)

{

**self**::*$\_instance* = **new self**($config);

}

**return self**::*$\_instance*;

}

// private constructor

**private function** \_\_construct(**array** $config)

{

**if** (count($config) < 4)

{

**throw new** MySQLAdapterException('Invalid number of connection parameters');

}

$this->\_config = $config;

}

// prevent cloning class instance

**private function** \_\_clone(){

}

// connect to MySQL

**private function** connect()

{

// connect only once

**if** (**self**::*$\_connected* === **FALSE**)

{

**list**($host, $user, $password, $database) = $this->\_config;

**if** ((!$this->\_link = mysqli\_connect($host, $user, $password, $database)))

{

**throw new** MySQLAdapterException('Error connecting to MySQL : ' . mysqli\_connect\_error());

}

**self**::*$\_connected* = **TRUE**;

**unset**($host, $user, $password, $database);

}

}

// perform query

**public function** query($query)

{

**if** (is\_string($query) and !**empty**($query))

{

// lazy connect to MySQL

$this->connect();

**if** ((!$this->\_result = mysqli\_query($this->\_link, $query)))

{

**throw new** MySQLAdapterException('Error performing query ' . $query . ' Error : ' . mysqli\_error($this->\_link));

}

}

}

// fetch row from result set

**public function** fetch()

{

**if** ((!$row = mysqli\_fetch\_object($this->\_result)))

{

mysqli\_free\_result($this->\_result);

**return FALSE**;

}

**return** $row;

}

// get insertion ID

**public function** getInsertID()

{

**if** ($this->\_link !== **NUlL**)

{

**return** mysqli\_insert\_id($this->\_link);

}

**return NULL**;

}

// count rows in result set

**public function** countRows()

{

**if** ($this->\_result !== **NULL**)

{

**return** mysqli\_num\_rows($this->\_result);

}

**return** 0;

}

// close the database connection

**function** \_\_destruct()

{

is\_resource($this->\_link) AND mysqli\_close($this->\_link);

}

}

//Testing adapter class class\_name

$db = MySQLAdapter::*getInstance*(**array**('localhost', 'root', '', 'patterns'));

$db->query('SELECT \* FROM users');

**while** ($user = $db->fetch())

{

**echo** 'First Name: ' . $user->firstname . ' Last Name: ' . $user->lastname . ' Email: ' . $user->email . '<br />';

}

<?php

**class** MySQLAdapterException **extends** Exception{}

<?php

**abstract class** DataMapperAbstract

{

**protected** $db = **NULL**;

**protected** $table = '';

**protected** $identityMap = **array**();

**public function** \_\_construct(MySQLAdapter $db)

{

$this->db = $db;

}

// get domain object by ID (implemented by concrete domain object subclasses)

**abstract public function** find($id);

// insert/update domain object (implemented by concrete domain object subclasses)

**abstract public function** save(DomainObjectAbstract $domainObject);

// delete domain object (implemented by concrete domain object subclasses)

**abstract public function** delete(DomainObjectAbstract $domainObject);

}

<?php

**require\_once** 'dataMapperAbstract.php';

**class** UserMapper **extends** DataMapperAbstract

{

**protected** $table = 'users';

// fetch domain object by ID

**public function** find($id)

{

// if the requested domain object exists in the identity map, get it from the map

**if** (array\_key\_exists($id, $this->identityMap))

{

**return** $this->identityMap[$id];

}

// if not, get domain object from the database

$this->db->query("SELECT \* FROM $this->table WHERE id = $id");

**if** ($row = $this->db->fetch())

{

$user = **new** User;

$user->id = $row->id;

$user->firstname = $row->firstname;

$user->lastname = $row->lastname;

$user->email = $row->email;

// save domain object to the identity map

$this->identityMap[$id] = $user;

**return** $user;

}

}

// save domain object

**public function** save(DomainObjectAbstract $user)

{

// update domain object

**if** ($user->id !== **NULL**)

{

$this->db->query("UPDATE $this->table SET firstname = '$user->firstname', lastname = '$user->lastname', email = '$user->email' WHERE id = $user->id");

}

// insert domain object

**else**

{

$this->db->query("INSERT INTO $this->table (id, firstname, lastname, email) VALUES (NULL, '$user->firstname', '$user->lastname', '$user->email')");

}

}

// delete domain object

**public function** delete(DomainObjectAbstract $user)

{

**if** ($user->id !== **NULL**)

{

$this->db->query("DELETE FROM $this->table WHERE id = $user->id");

}

}

}

**Research Work**

Investigate on the motivation, intent and simple implementation of the following patterns:

* Lazy Loading
* Identity Map
* Unit of Work
* Separated Interface Pattern