DESIGNING AN OBJECT RELATION MAPPING SYSTEM IN PHP

Dragos-Paul Pop[1]

Abstract

Object Relational Mapping is a programming technique used by software developers to convert data between incompatible systems. This technique is used in object-oriented programming languages, hence the "Object" in Object Relational Mapping. Often times we see ORM systems being used by developers when interacting with relational database management systems. ORM is about creating classes that mimic the tables in the database but in a more business oriented manner rather than the normalized form used by the RDBMS.

Keywords: ORM, object, relational, mapping, class, business model, normalization, primary key, foreign key, methods, properties

Introduction

The need for ORM systems arises from the way data is stored and manipulated in different DMS and programming languages. We can generalize things and group the two kinds of systems: the database management systems (the majority of which use the relational model) and the programming languages (the majority of them having object-oriented features or being completely object oriented). The way data is being manipulated by the two kinds of systems is completely different.

The relational model

Formulated by E.F. Codd in 1969, the relational model is the main storage model used by the majority of database management systems. The relational model's central idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values. The content of the database at any given time is a finite (logical) model of the database, i.e. a set of relations, one per predicate variable, such that all predicates are satisfied. A request for information from the database (a database query) is also a predicate.

The purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries.
Normalization is another property of relational databases. The goal of database normalization is to decompose relations with anomalies in order to produce smaller, well-structured relations. Normalization usually involves dividing large tables into smaller (and

[1] Ph.D Candidate at the Academy of Economic Studies of Bucharest and assistant teacher at the Romanian-American University of Bucharest, Romania; email: dragos_paul_pop@yahoo.com

less redundant) tables and defining relationships between them. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.

So, as we can see, the relational model, along with normalization is well suited for data storage because, if used well, it can provide a safe, fast, anomaly and redundancy free model for storing data. That's exactly what it was created for and it does its job well.
But what about retrieving and manipulating data? Well, that's what we have SQL for. It was created at IBM in 1970 to be used in System R, the first database system based on the relational model. SQL is based on relational algebra and math, as is the relational model, so we can see that and RDMS uses logic to store and manipulate data.

Object-oriented programming

Many people first learn to program using a language that is not object-oriented. Simple, non-OOP programs may be one long list of commands. More complex programs will group lists of commands into functions or subroutines each of which might perform a particular task. With designs of this sort, it is common for the program's data to be accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

By contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead the data is accessed by calling specially written functions, commonly called methods, which are either bundled in with the data or inherited from "class objects" and act as the intermediaries for retrieving or modifying those data. The programming construct that combines data with a set of methods for accessing and managing those data is called an object.

This paradigm is widley used because it uses structures to work with data, it groups variables and functions into properties and methods of objects keep track of data. This looks like the logical way for programming.

The relational model meets object-oriented programming

The problem is that, more often than not, the two models, paradigms, architectures or what have you meet, and when they meet it is not preety. That's because they are completely different. They are based on completely different views. One is used for storing and manipulating large amounts of data and the other for implementing algorithms for processing that data.

A programmer needs to use SQL to retrieve and send data back to the database management system. Moreover, he needs to know the exact structure of the database, the tables, the relations and the restrictions to work with data. This is a burden for every programmer, because it is something more to worry about. What if the database structure changes? The programmer needs to alter his code, his classes, his functions... And what about SQL? It is complex, especially when one needs to use all kinds of joins to query

tables and it is a burden for the programmer to filter query retrieved data and instantiate all his objects.

One thing is clear: using relational stored data in an object-oriented programming language can become very difficult at times. That's why the Object Relational Mapping technique was invented. It acts like a bridge between the two systems, so that the programmer can free himself from worrying about how data is stored and retrieve and focus on how it is processed in the application he is building. This is exactly the way it is supposed to be, because there must be a separation between data storage and data usage.

How the ORM system works

It works like a middle-man between the programmer and the database management system by doing the interacting with the database on behalf of the programmer and returning the data in an object-oriented programming language friendly way: objects...with methods too! Basically, the programmer doesn't query the database management system and doesn't need to worry about SQL anymore, all this is done by the ORM software. The programmer "queries" the ORM software instead. Actually, "query" is not the right word, because the programmer just uses the objects and the ORM software fetches the data as it is needed.

Let's get thing straight: by "ORM software" I don't mean some kind of strange, alien piece of software that has nothing to do with the programming language used by the programmer or the programmer himself. No, "ORM software" is actually a library (sometimes created by the programmer himself) that is used to work with one or more database management systems.

Designing an ORM library

As we will see, designing and implementing an ORM library is fairly easy. We just need to use some basic object-oriented programming features like inheritance, late static binding and dynamic properties (some very nice features implemented in PHP).

First we need to build the database connection class. The example below is used to connect with the MySQL RDMS, but it can be extended to work with other database systems as well. I have omitted the function codes because it can take up quite some space.

```
<?php
class MySQLDatabase {

	private $connection;
	public $last_query;
	private $magic_quotes_active;
	private $real_escape_string_exists;

	function __construct() {...}
	public function open_connection() {...}
	public function close_connection() {...}
	public function query($sql) {...}
```

```
        public function escape_value( $value ) {...}
        public function fetch_array($result_set) {...}
        public function num_rows($result_set) {...}
        public function insert_id() {...}
        public function affected_rows() {...}
        private function confirm_query($result) {...}
}
?>
```

As we can see, the class takes care of usual stuff, like establishing a connection, querying the database with a given SQL string, escaping values for security reasons, fetching data as an array and so on.

Next, we build the main piece of the ORM library, the base class for future objects, the parent class. This is the DatabaseObject class. It will have special methods and properties that will enable programmers to use objects to interact with data.

```
<?php
class DatabaseObject {
protected static $table_name;
protected static $related = array();
public $related_objects = array();
public $parent = NULL;
public function save() {...}
public function create() {...}
public function update() {...}
public function delete() {...}
private static function instantiate($record) {...}
private function has_attribute($attribute) {...}
public function attributes() {...}
public static function fields() {...}
protected function sanitized_attributes() {...}
public static function tbl_name() {...}
public static function find_all($order=NULL) {...}
public static function find_by_id($id=0) {...}
public static function find_by_prop($prop, $value, $parent=NULL) {...}
public static function find_all_by_prop($prop, $value, $order=NULL,
$parent=NULL) {...}
public static function find_by_props($props_vals) {...}
public static function find_by_prop_page($page, $prop='1', $value=1,
$page_size=10, $order=NULL) {...}
public static function find_by_sql($sql="", $order=NULL, $parent=NULL)
{...}
public static function count_all($where = NULL) {...}
public function find_related() {...}
private static function find_set_parent($object, $prop, $val, $single,
$parent) {...}
public function __set($name, $value) {...}
public function &__get($name) {...}
public function __isset($name) {...}
}
```

The Database object class is the most important because it is the link between an object and its corresponding table in the database. The *table_name* property is speaking for itself, but the others need some explanations.

The static property *related* is an array that contains all of the relations the object has with other objects. It is a reflection for the foreign key constraints in the database. It can be used like this:

```
0 => array (
       "type" => "m:n",
       "object" => "project",
       "table" => "projects",
       "keys" =>
              array (
              "this" => "id_user",
              "relation" => "id_project"
              ),
       "relation_object" => "project_user"
),
1 => array (
       "type" => "1:n",
       "object" => "group",
       "table" => "groups",
       "key" => "id_group",
       "key_object " => "user",
       "label" => array("nr", "series", "year")
),
2 => array (
       "type" => "1:1",
       "object" => "type",
       "table" => "typeuri",
       "key" => "id_type",
       "key_object" => "project"
)
```

This property will later be used by the *find_related* method to instantiate all of the current object's relations (related data). These objects will be stored in the *related_objects* array.
The *parent* property stores the information about the object's parent, if any.
There are, of course, methods for creating, updating, deleting, saving, instantiating and finding (a lot of) data.
The class is never instantiated by itself, but it is used as a base class for other classes. These classes will each mirror tables in the database and are called *models* in the MVC (model-view-controller) architecture.
Here is a typical usage:

```
//the User class inherits from the DatabaseObject class
//the Project class inherits from the DatabaseObject class
//the relationship between User and Project is many-to-many

$user = User::find_by_prop("username", "John"); //this retrieves the user
called John and all his related projects so that we can have a call like
the following (display the name of the first project of the user)

echo $user->projects[0]->name;

//we can also have calls to assign a new project to the user

$project = new Project();
$project->name = "last project";
```

```
$project->date = date(“d-m-Y”);
$project->save();

$user->projects[] = $project;
$user->save();

//the examples can go on and on
```

As we can see there was no need for SQL after we created the ORM library. All the programmer has to do is use the objects naturally, the interaction with the database system is being done by the ORM library.

Conclusion

ORM software can cut down a significant amount of work and time, letting the programmer focus on what is important, the business logic. If the database schema changes, the programmer just needs to reflect these changes in the ORM software, but not in his programs, so the applications don’t break down. There are a lot more advantages of using ORM, but there are drawbacks as well. Just to mention one: I had a lot of trouble trying to figure how to fix the infinite loop problem in retrieving an object’s related entities, because in the relational model, relations are mutual (one-to-many means many-to-one read backwards, for example). The problem was solved by using the *parent* property and setting a certain level of nesting depth.

Acknowledgements

This work was cofinaced from the European Social Fund through Sectoral Operational Programme Human Resources Development 2007-2013, project number POSDRU/107/1.5/S/77213 „Ph.D. for a career in interdisciplinary economic research at the European standards”

Bibliography

1. http://www.kevinskoglund.com/
2. Julia Lerman – Programming Entity Framework, August 2010, O’Reilley, ISBN: 978-0-596-80726-9
3. http://en.wikipedia.org/wiki/Object-oriented_programming
4. http://en.wikipedia.org/wiki/SQL
5. http://en.wikipedia.org/wiki/Object-Relational_Mapping