Persistent Objects with O2DBI

Jörn Clausen[*]

Center for Genome Research

Bielefeld University, Germany

13th October 2000

[*]jc@Genetik.Uni-Bielefeld.DE

Contents

1 Introduction

This document describes O2DBI. This Perl module helps you to utilize the advantages of object-oriented programming and relational databases at the same time. You are able to define a set of objects, which are then transparently stored in a database. Thus objects are made persistent, and can be accessed from different locations.

I assume that the reader as some basic knowledge about object-oriented programming and relational databases. To keep the examples simple, all code fragments are shown without error handling. Don't try this at home! Always check return values.

2 The Beauty of Objects

2.1 Methods versus Procedures

Object-oriented languages, object-oriented programming, OO in short has become very popular during the last ten years. Some important languages were created by augmenting successful languages with objects, e.g. C++ and Perl5. Others were designed right from the start with OO technologies in mind, like Python.

OO-capable languages usually come with some more advanced methods, like overloading or exception handling (which are not directly OO-related), and inheritance and information hiding. One purely syntactical feature, which for itself already may advocate using an OO approach, is the enhanced readability of programs written in an object-oriented language.

Table 2.1 shows a comparison of two small code fragments. Their purpose is to store and access data on some persons, which are identified uniquely by their IDs. In the imperative version on the left side, the name of the person is extracted by the function `id2name`. In the OO version, the person is represented as an object, which is instantiated by the method `init`. The name can be queried by applying the method `name` to the object. Two more functions and methods are available to extract the address of the person and the appropriate zipcode. But the object-oriented version is far more elegant, as it introduces a new class for addresses. Note that twice an anonymous object is created, by calling the method `address` on `$person`. Then appropriate methods are used on this anonymous address object.

`$personid = 1234;`	`$person = person->init(1234);`
`$name = id2name($personid);`	`$name = $person->name;`
`$address = id2address($personid);`	`$address = $person->address->printable;`
`$zipcode = extract_zipcode($address);`	`$zipcode = $person->address->zipcode;`

Table 2.1: A comparison of imperative and OO programming

`$ctitle = cd_id2title($cdid);`	`$ctitle = $cd->title;`
`$btitle = book_id2title($bookid);`	`$btitle = $book->title;`
`sell_cd($personid, $cdid);`	`$book->sell($person);`
`sell_book($personid, $bookid);`	`$cd->sell($person);`

Table 2.2: Name clashes, and avoiding them

One common annoyance in imperative programming is, that you can easily create name clashes. Think of an application where you are dealing with books and CDs. Both are again identified by unique but unrelated IDs (i.e. you can have a book and a CD with the same ID). If you want to know the title of a book and the title of a CD, a simple function `id2title` wont do, as the function cannot know if it received the ID of a book or a CD. Usually, you end up with the solution shown in table 2.2. As a result, your program will be cluttered with long and complicated function names. In OO world, you don't even have this problem. You have two methods `title`, and depending on what object you apply this method, the Right Thing is done. Each object "knows" which method it has to use, as it is the method defined in its own class. Table 2.2 assumes that appropriate objects already have been instantiated. But if you want to get just the title of a CD, you can even do

```
cd->init($cdid)->title
```

and forget the object immediately afterwards.

Another benefit of OO notation is a better detection of common programming glitches. In table 2.3, some functions and methods are shown to sell books and CDs to persons. The functions deal only with numerical IDs, so you have to pay attention to pass a book ID as parameter to `sell_book`. You can easily mix things up, e.g. confusing the order of the parameters. Even in stronger typed languages than Perl such problems would go unnoticed. Using the OO version, such errors are harder to program and easier to detect. It's a matter of taste, if you define a method `sell` inside the book and CD classes, or a method `buy` inside the person class. In the latter case, this method can be called both with book and CD objects. Either the method can determine what is currently sold, or both classes define a set of common methods, so that the `buy` method doesn't have to know if it sells a book or a CD. Confusing the order of parameters is much harder. In the last example, a person is bought by a book. On the one hand, this will throw an error when executing the code, as the book class does not have a method `buy`. On the other hand, carefully reading this code should make the programmer suspicious.

`sell_book($personid, $bookid);`	`$book->sell($person);`
`sell_cd($personid, $cdid);`	`$person->buy($cd);`
`sell_book($bookid, $personid);`	`$book->buy($person);`

Table 2.3: Passing parameters, type checking

2.2 Doing it in Perl

Defining classes in Perl is quite easy. A class is just a Perl module, and an object is a reference to a variable which ”knows“ from which class it was instantiated. Inside a class, you define constructors, destructors and methods, which are just ordinary subroutines defined in the module. Data encapsulation usually forbids direct access to the attributes of an object. Instead, methods have to be used to store, read or modify data inside an object.

As an example, consider another person class. The module has a constructor `new` and two methods `name` and `dob` to get or set the name of the person and the date of birth, respectively. Probably there are other methods. Part of the module is shown in table 2.4. The `package` statement tells Perl that the following code defines a class `person`. The method `new` can be used to create new person objects. It takes as arguments the name of the person and the date of birth:

```
$person = person->new('Joe User', '1.1.1970');
```

Inside the method, the data is stored in a hash. A reference to this hash is *blessed*, i.e. turned into an object. The reference is then returned to the caller of the constructor. Now each method applied to this object is looked up in the same module as the constructor.

```
package person;

sub new {
   my ($class, $name, $dob) = @_;

   my $person = { name => $name,
                  dob  => $dob,
                  ... };
   bless($person, $class);
   return($person);
}
```

Table 2.4: The constructor of the person class

```
sub name {
   my ($self, $name) = @_;
   if (defined($name)) {
      $self->{name} = $name;
   }
   return($self->{name});
}
```

Table 2.5: Getting or setting an attribute

One such method is `name`, which can be used to get or set the name of the person. When used without an argument, the name of the person is returned:

```
$name = $person->name;
```

When called with an argument, it is stored as the new name:

```
$newname = $person->name('Joe N. User');
```

In addition, and as check if an error occured, the newly set name is returned. Table 2.5 shows the implementation of this method. Other methods accessing the basic attributes of an object are very similar.

3 Persistent Objects and Relational Databases

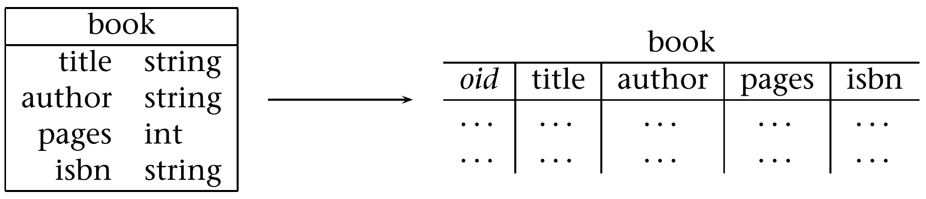
3.1 Two Worlds

Objects are short-lived things. When the application terminates, all currently instantiated objects are discarded, and their contents are lost. In larger applications, where you deal with massive amounts of data, this is usually not desired. When you store data inside an object, this data has to be ”remembered“, so that you can use the same object again later. Objects need some kind of persistence.

Databases are used to store large amounts of data. Although there are different types of databases, the most widespread are relational databases. The basic idea of an RDBMS (Relational Database Management System) is to store data inside tables. The virtue of creating a database schema is to come up with appropriate tables for your application. You try to model a view of the real world, which has to be correct, sufficient and non-redundant.

So, how do objects and databases fit together? At first glance, these are two different worlds, with very different concepts. In OO-land, an object represents an entitity, which is clearly seperated from all other objects. In an RDBMS, data concerning a single ”thing“ can be spread across several tables. A single table contains data of all the other ”things“ as well. So the same information can have two very different representations, when stored inside objects or inside a database.

Surprisingly, this dilemma can be solved rather easily. Objects can be mapped to tables, so a relational database can be used to store the data contained in objects. The interesting fact is, that this mapping process can be fully automated. When designing an application, the developer can think in terms of objects, and some tool derives the database schema to store these objects in an RDBMS.



book	
title	string
author	string
pages	int
isbn	string

book

oid	title	author	pages	isbn
...	...	...	...	...
...	...	...	...	...

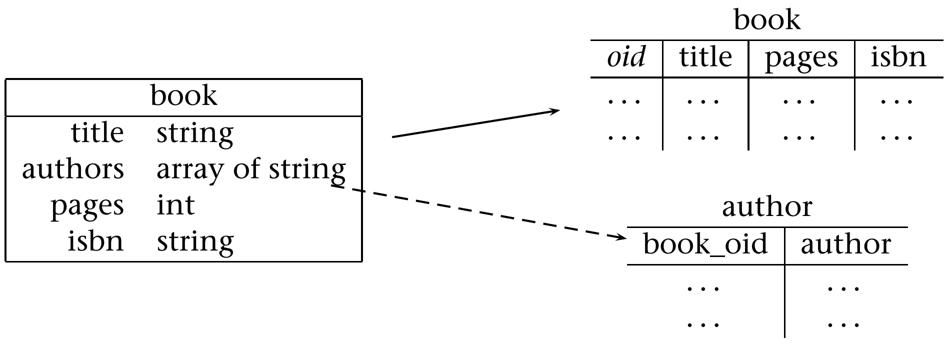
Table 3.1: Mapping a class to a table

3.2 Mapping Objects to Tables

The process of transforming a class description into a database schema is straight forward. For simple objects, that contain only atomic member variables, the class is represented by a table with one column for each attribute. Each object is then stored as one row in the table. In addition, each row, i.e. each object, gets an additional field called *object identifier*. This is a numerical value, which identifies each object uniquely in the table. In other words: It is the primary key of this table. This *oid* will be used later to reference objects. Table 3.1 shows a class description and the corresponding table.

As one book may have several authors, the object shown in table 3.2 is more realistic. Instead of one author, an array of authors is associated with each book. The corresponding description of such a *one-to-many relation* within a database is a seperate author table, which is connected to the book table via keys. The column *book_oid* contains the object identifier of the corresponding book. To reconstruct a book object from the database, one row from the book table and all rows from the author table containing the appropriate book identifier have to be read.

As one author may have written several books, an even more realistic description



book	
title	string
authors	array of string
pages	int
isbn	string

book

oid	title	pages	isbn
...	...	...	...
...	...	...	...

author

book_oid	author
...	...
...	...

Table 3.2: Mapping arrays to external tables

book	
title	string
authors	array of author
pages	int
isbn	string

author	
name	string
country	string
DOB	date
DOD	date

book

oid	title	pages	isbn
...	...	...	...
...	...	...	...

book_author

book_oid	author_oid
...	...
...	...

author

oid	name	country	DOB	DOD
...	...	...	...	...
...	...	...	...	...

Table 3.3: Describing many-to-many relations

is the one shown in table 3.3. The authors are modeled as a seperate class, with some additional information. The book object now contains an array of author objects instead of strings. This allows one object to be used in several books, which also reduces the redundancy of the stored data. To describe such a *many-to-many relation* in a database, three tables are necessary. Again, the objects (books and authors) and their atomic attributes are transformed to tables. An additional table connects books and authors by matching book oids with author oids.

3.3 Doing it again in Perl

The methods from section 2.2 are easily extended to communicate with a relational database. Instead of a simple constructor `new`, two new methods are introduced to instantiate objects. Two cases have to be covered: Creating a new object from scratch and creating a new object based on data already stored in the database.

To create a brand-new object, a method `create` is used. It is similar to the `new` constructor shown above, but as a side effect the data is written to the database. To uniquely distinguish the object, a new object identifier has to be assigned. The `create` method from the person class is shown in table 3.4. This and all following examples assume, that a global DBI handle `$dbh` is available, i.e. the connection to the database has already been established.

```
sub create {
   my ($class, $name, $dob) = @_;

   # fetch a new object identifier
   my $id = newid('person');

   # store the data inside the database
   $dbh->do(qq {
      INSERT INTO person (id, name, dob) VALUES ($id, $name, $dob)
    });

   # create the object
   $person = { id   => $id,
               name => $name,
               dob  => $dob };
   bless($person, $class);
   return($person);
}
```

Table 3.4: Creating and storing a new object

To reconstruct an object from data already in the database, the method `init` is used. Its usage was shown very briefly in section 2.1. This constructor takes as argument the object identifier that was assigned to the object when it was created by the `create` method. Table 3.5 shows the implementation. More constructors are possible and usually necessary. Sometimes it is handy to instantiate a person object based on the name of the person instead of its oid.

The methods for accessing the attributes are extended, too. In table 3.6, the method `name` now writes a new name directly back to the database. In this approach, the data is read once from the database and written back immediately, every time an attribute is updated. Other methods are possible:

- The data can be read every time, when it is accessed. Instead of storing the attributes inside the object, each call of a method like `name` queries the database. This will decrease the performance, but can ensure correct data, when concurrent applications access the database.
- Writing back the data can be defered until the object is destroyed. The destructor of the object is responsible for storing all attributes inside the RDBMS. This may increase the performance, depending on how often data is changed within one single object. But changes may get lost, e.g. if the application crashes and the destructor is not executed.

```
sub init {
   my ($class, $id) = @_;

   # fetch all data associated with the given ID
   $sth = $dbh->prepare(qq {
      SELECT name, dob FROM person WHERE id=$id
   });
   $sth->execute;
   ($name, $dob) = $sth->fetchrow_array;
   $sth->finish;

   # create the object
   $person = { id   => $id,
               name => $name,
               dob  => $dob };
   bless($person, $class);
   return($person);
}
```

Table 3.5: Reconstructing an object from the database

```
sub name {
   my ($self, $name) = @_;
   if (defined($name)) {
      $id = $self->id;
      $sth = $dbh->prepare(qq {
         UPDATE person SET name=? WHERE id=$id
      });
      $sth->execute($name);
      $sth->finish;
      $self->{name} = $name;
   }
   return($self->{name});
}
```

Table 3.6: Database aware access to an attribute

4 Code generation with O2DBI

4.1 A simple Example

Using the methods shown in the two previous chapters, it is easy to create a set of persistent objects. You get a collection of modules that share a lot of common or similar code. All the modules contain the same types of constructors and methods to access their member variables. Obviously, generating this code automatically would be a great benefit. This would save the programmer from a lot of typing, and the number of errors would be reduced.

O2DBI is such a tool. The object schema is given as a Perl program, that generates all modules and the necessary SQL commands to initialize the database. Table 4.1 shows the description of the author class from table 3.3.

When executing this script, O2DBI generates several files:

- For each class, a module with the shown standard methods is generated. In the example above, one file called `author.pm` is created. It is placed inside a directory `simpleDB`, which is the name of the project, as given in the last line of the script. You can include this class in your own applications with

  ```
  use simpleDB::author;
  ```

- For each class, a second file is generated. In this example it is called `author_add.pm`. Initially, this file is empty, except for a few comment lines. The developer can use this file to add more methods to the class. These methods cannot be described in an abstract way, so they have to be "hand coded". `author.pm` reads `author_add.pm`, so all methods defined in this file are members of the class.

  When the O2DBI script is executed again, all the main class modules are overwritten, to reflect changes in the object schema. But none of the additional modules are regenerated, if they already exist. The developer may only change code inside the `_add.pm` modules, not the base modules.

```
use O2DBI;

%schema = ( 'author' => {
                members => {
                    'name'    => 'CHAR(40)',
                    'country' => 'CHAR(40)',
                    'dob'     => 'DATE',
                    'dod'     => 'DATE'
                    },
                creator => [ 'name', 'country', 'dob' ],
                constructors => [
                    [ 'name' ]
                    ]
            }
          );

O2DBI->deploy(\%schema, 'simpleDB', 'postgres');
```

Table 4.1: The author class in O2DBI

- A file `DBMS.pm` is placed inside the project directory. This file opens a connection to the database and provides a DBI database handle. It is used by the class modules internally, it must not be used inside any application directly.
- A file with the necessary SQL statements to initialize the database is generated. In this example, it is called `simpleDB.sql`. It creates all the necessary tables and defines indexes where appropriate. The database administrator has to create a database `simpleDB` and execute these statements.

4.2 Using O2DBI

To describe the object schema for O2DBI, you collect all class definitions inside a hash. The keys of this hash are the names of the classes. Table 4.2 shows this part of the code to describe a more complex schema.

The value for each key is a hash reference, denoted by the curly braces. Each of these hashes contains three keys: `members`, `creator` and `constructors`. The contents of these keys define the member variables, the parameters of the `create` method, and the available constructors to instantiate objects based on data inside the database.

```
%schema = ( 'author' => {
                ...
            },
            'book'   => {
                ...
            },
            'cd'     => {
                ...
            }
          );
```

Table 4.2: Defining several classes

The value of the `members` key is another hash reference. Each key in this hash is a member variable of the class. The value is the type of the variable. It has to be a valid SQL expression, that will be used later for the appropriate `CREATE TABLE` statement. Two special entries for keys and values are allowed:

- The member name may be prefixed with an `@`. Instead of an atomic variable, this denotes a list (or array, hence the perlish "at" sign) of the given type. O2DBI then creates an additional table, as shown in table 3.2.
- Instead of an SQL type, the value may be "`ref on`" followed by the name of an object. This describes a reference to another object.

Both extensions may be mixed, i.e. you can have a list of references to objects. Table 4.3 shows all possible combinations. The fourth entry is used to describe the situation shown in table 3.3.

The value of the `creator` key is a list reference. This list consists of those attributes, that are passed to the `create` method. The attributes are usually those

```
%schema = ( 'book'   => {
                members => {
                    'title'     => 'CHAR(40)',
                    '@keyword'  => 'CHAR(40)',
                    'publisher' => 'ref on publisher',
                    '@author'   => 'ref on author'
                }
          );
```

Table 4.3: Types of members

known at create time of the object. Certain other attributes, e.g. "date of death", are not necessarily known when the object is created. Instead of passing an empty or null value, they are not passed to the constructor.

The value of the `constructors` key is a reference to a list of lists (i.e. a list of references on lists). Each list defines a constructor, similar to the `init`-method shown in table 3.5. But instead of the object identifier, the constructor expects the attributes given in the list. In the example in table 4.2, a method `initby_name` would be generated, that returns the appropriate author object to a given name. This implies that the name has to be unique. This is enforced by creating a *unique index* on the corresponding table.

To convert the object schema into real code, O2DBI's only method is used, `deploy`. It is called with a reference to the hash, a project name and the type of database. All modules are written to a directory with the same name as the project. If this directory does not exist, it is created. O2DBI currently supports "`postgres`" and "`mysql`" as types of database.

4.3 Modules generated by O2DBI

Each module generated by O2DBI has the same structure. In the following description, these symbolic names are used:

`BaseName` The base name of the project, i.e. the name of the database and the name of the directory, to which all modules are written.

`class` The name of the class.

`$obj` An instance of an object from that class.

`$arg` An argument passed to a method.

`attr` The name of an attribute.

Each module contains these methods:

`$object = BaseName::class->create($arg1, $arg2, $arg3)`
A new object is created and inserted into the database. The arguments are those specified in the `create` directive in the description file.

`$object = BaseName::class->init_id($id)`
A new object is instantiated, based on data read from the database. The object is identified by the oid `$id`.

`$hashref = BaseName::class->fetchallby_id`

All objects of the class are instantiated an returned as a hash reference. The key to the hash is the object identifier, the value is the corresponding object.

`$object = BaseName::class->init_attr($arg)`
`$object = BaseName::class->init_attr1_attr2($arg1, $arg2)`

An object is instantiated, based on the passed atributes. For each element of the `creator` list, an appropriate method is generated. If a list contains more than one attribute, their names are concatenated by underscores.

`$hashref = BaseName::class->fetchallby_attr`
`$hashref = BaseName::class->fetchallby_attr1_attr2`

Similar to `fetchallby_id`, parallel to each constructor a method is generated, that returns a hash reference with all objects. The keys to this hash are the values of the attributes, concatenated with commas.

`$listref = BaseName::class->fetchall`

All objects are instantiated and returned, but this time as a list reference. If the order of the objects is not relevant, this method may be used instead of one of the `fetchallby_` methods.

`$listref = BaseName::class->fetchbySQL($whereclause)`

With the methods above, either single objects can be instantiated, or all objects together. But in a lot of cases, a set of objects is needed, based on certain selection criteria. This can be done by using this method. As argument, a fragment of SQL code is expected, that can be used as a `WHERE` clause in a `SELECT` statement. For example

```
$listref = simpleDB::person->fetchbySQL('age>=20 AND age<=45');
```

would select all persons with an age between 20 and 45 years, assuming that the `person` class has an attribute `age`. The SQL code has to be valid, otherwise the database will return an error. Only values from the corresponding table (`person` in this case) can be tested in the clause.

`$object->delete`

The data associated with the object is deleted from the database and the object is destroyed.

`$id = $object->id`

The object identifier of the object is returned.

`$value = $object->attr`
`$retvalue = $object->attr($newvalue)`

For each attribute, a corresponding method is generated. The method can be used to read or write the value of the attribute. If the method is called without an argument, the current value is returned. If an argument is given, the attribute is set to this new value. The database is updated immediately. To check for possible errors, either the new value or an error code are returned.

`$retvalue = $object->mset({ attr1 => $value1, attr2 => $value2})`

Each time the value of an attribute is changed, the database is updated. If several attributes of the same object are changed at the same time, this results in a number of subsequent connections to the database. To reduce the number of connections, these method calls can be accumulated with this method. It expects a hash reference as argument, with the attributes as keys, pointing to the new values.

4.4 Using the modules generated by O2DBI

Table 4.4 shows a short Perl script that uses the author class from table 4.1. After including the module with the `use` statement, all methods from the class are available. First, a new object is created and written to the database. The object identifier that was assigned to this object is printed. Next, the object with the oid 1234 is instantiated back from the database and the name of that author is shown. Next, all authors are fetched from the database, and the stored data is printed.

Most of the methods return the value ”-1“ to indicate an error. Table 4.5 shows the recommended code to check if e.g. the `init` method succeeded. You should always check if the constructors returned an object or an error code.

```
use simpleDB::author;

$joe = simpleDB::author->create('Joe User',
                                'Germany',
                                '1.1.1970');
print "Joe User was assigned id ".$joe->id."\n";

$someone = simpleDB::author->init_id(1234);
print "fetched author 1234, which is ".someone->name."\n";

foreach $author (@{simpleDB::author->fetchall}) {
   $name = $author->name;
   $country = $author->country;
   $dob = $author->dob;
   $dod = $autor->dod;
   print "$name from $country, born $dob, deceased $dod\n";
}
```

Table 4.4: Usage of the author class

```
$joe = simpleDB::author->init_id(1234);
if ($joe < 0) {
   die "can't initialize author object for oid 1234\n";
}
$name = $joe->name;
```

Table 4.5: Checking the return value of a method

5 Current limitations and planned enhancements

In the current version of O2DBI, several features described in this paper are not present or not fully working. O2DBI is useful for creating prototypes of software very fast, but some aspects of the current design reduce its useabilty for large, real world applications. Of course the author intends to overcome these limitations and make O2DBI suitable for more complex problems. Feedback concerning this, and every other aspect of O2DBI is always welcome.

5.1 Things missing from the implementation

Although described in section 4.2, the data types available for the attributes are limited. Currently, only atomic types are fully supported. Neither lists of values nor references to other objects are supported. O2DBI can parse the shown notation, and the generated tables are correct (including necessary normalization steps), but the associated methods are not working yet. Part of this problem can be solved rather easily: Instead of using "`ref on object`", use "`INT`". Instead of an object, the associated method then expects the object identifier of the referenced object. So, instead of writing

```
    $publisher1 = $book1->publisher;
    $book2->publisher($publisher2);
```

you simply use

```
    $publisher1 = simpleDB::publisher->init_id($book1->publisher);
    $book2->publisher($publisher2->id);
```

```
'cd' => [
    members => {
        'title'  => 'CHAR(40)',
        '@songs' => {
                        'title'  => 'CHAR(40)',
                        'index'  => 'INT',
                        'length' => 'INT'
                    }
    }
```

Table 5.1: A struct-like data type

This is a little bit more verbose, and the error checking leaves a little bit to be desired, but it is working.

Lists of things are not working at all. Currently, O2DBI understands another notation, but again, the generated methods do not support it. You can define struct-like fields, as shown in table 5.1. These can even be nested. O2DBI performs all the required normalizations, but you cannot access these tables with the generated methods.

Access to the database should be nearly invisible to the application programmer and the user. The `DBMS.pm` module provides a database handle, that is used by all class modules. No additional login information or passwords may be used to create this handle. In certain situations, e.g. when dealing with sensible data, this may be impossible.

5.2 Plans for the future

Of course, all current limitations should be removed within the next releases of O2DBI. Other enhancements are possible and desirable:

- Caching of objects. Once an object is `init`ed from the database, further initializations should use it instead of rereading the data from the database.
- Better support for more databases. Currently, PostgreSQL and MySQL are supported. The generated SQL code can be made more portable among different RDBMSs.
- Supporting other languages. Although the object schema is currently defined

as a Perl script, this does not limit the generated code to be Perl. Modules or classes for other languages can be generated, e.g. Java or Python.

- Defining the object schema language independant. In the very early stages of O2DBI, the schema was defined by an XML document. This was dropped in order to concentrate on the real function of O2DBI, and not to be distracted by the parser. Once the syntax for the object schema has settled, describing it in XML again will be an easy transition.

Bibliography

[1] Scott W. Ambler. *Mapping Objects To Relational Databases*. White paper, AmbySoft Inc., 1999.
`http://www.AmbySoft.com/mappingObjects.pdf`

[2] Wolfgang Keller. *Mapping Objects to Tables - A Pattern Language*. Technical report, 1997.
`http://www.objectarchitects.de/arcus/publicat/mapo2t.ps.gz`