UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Specialty of Information Technology

Oliver Leisalu

Comparative Evaluation of Two PHP Persistence Frameworks

Bachelor's Thesis (4AP)

Supervisor: prof. Marlon Dumas

Author: ..... “....” June 2009

Supervisor: ..... “....” June 2009

Allow to Defence

Professor: ..... “....” June 2009

TARTU 2009

Table of Contents

Introduction

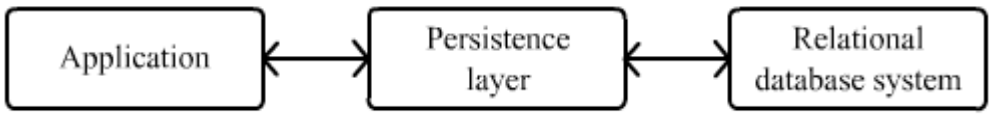
Object-Relational Mapping (ORM) frameworks allow software developers to manipulate objects that persist in relational tables. In the context of the PHP language, a number of ORM frameworks exist. These frameworks strike various trade-offs between richness of functionality, usability and performance. However, little work has been done previously to compare existing PHP persistence frameworks. This thesis compares two such frameworks, namely Doctrine and DomAR in terms of their functionality, usability and performance. Usability and functionality are assessed by means of an action research study in which two sample applications from different domains were created separately using the two frameworks, and the two implementation efforts were compared in terms of several factors, including programming effort and number of lines of code. Meanwhile, the performance of the two frameworks was compared using a small benchmark covering three common types of operations: traversals, queries and modifications.

Persistence Frameworks

Most enterprise applications use relational databases for data persistence, but very often they are programmed using an object-oriented programming language in which data is structured as objects. This difference in representation approaches leads to a phenomenon known as *impedance mismatch*. Impedance mismatch refers to the differences between Object-oriented technology and relational technology. Object-oriented technology relies on objects that have data and behavior. The objects have identity and are traversed using direct access whereas relational databases systems (RDBS) store data into tables. That data is manipulated and accessed using Data Manipulation Language (DML) such as SQL. Furthermore in RDBS tables have primary key and data is related with foreign keys and they do not have equivalent inheritance logic as object's have.

A representative example of impedance mismatch is when storing a string into varchar type into database. String in many programming languages can be as long as wanted, but data type varchar in database can be only 256 characters. In application everything may work when using longer values, but when storing that value into database an error will occur.

One approach to solve the impedance mismatch is to introduce an abstract layer, called a Persistence layer. The persistence layer lies between RDBS and object model of the application. This way the database is fully encapsulated from application and most requests to database are done through a persistence layer. That layer validates all data that will be inserted into database.



A popular approach used by persistence framework nowadays is called object-relational mapping (ORM). An ORM framework converts and transparently moves data from the form it is managed in main memory (objects) to a persistent relational data store and vice-versa. In simpler terms, the ORM tool helps to manage the database access and the mapping between the database and objects. There are three patterns that can be used

when creating an ORM tool - Table data gateway, Row data gateway and Active record. All these design patterns were named by Martin Fowler in his book *Patterns of Enterprise Application Architecture* [3].

Table Data Gateway

"Table data gateway holds all the SQL for accessing a single table or view: selects, inserts, updates and deletes. Other code calls its methods for all interaction with the database." as explained by Martin Fowler [3]. The Table data gateway pattern has a very simple interface which has several methods for create, read, update, delete (CRUD) actions. Each of these methods map input into an SQL call and executes it. Usually one class is made for each table.

Row Data Gateway

Row data gateway is an object that acts as a single row in a database. "It gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language. All details of data source access are hidden behind this interface." as mentioner by Fowler [3].

Active Record

"Active record is an approach to access data in a database. A database table or view is wrapped into a class, thus an object instance is tied to a single row in the table. After creation of an object, a new row is added to the table upon save. Any object loaded gets its information from the database; when an object is updated, the corresponding row in the table is also updated. The wrapper class implements "accessor" methods or properties for each column in the table or view." [3]. Active record is very similar to Row data gateway as they both contain database access. The main difference with respect to row data gateway is that active record also contains domain logic.

A simple example of how the active records work is given in the code snippet below. This code snippet shows how to create an object, assign values to it and insert it into the database without writing any SQL.

```
$author = new Author();
$author->name = 'John Smith';
$author->phone = +37512345678;

$book = new Book();
$book->title = 'Persistence frameworks';
$author->books[] = $book;

$author->save();
```

In this example, we create an author object, assign name John Smith and phone number +37512345678 to it. Then we create a book object titled “Persistence frameworks”. We bind book and author together simulating a relationship where author has written that book. By calling save method we insert a book and an author row into database.

```
$author = Author::load( (int)ID );
$author = Author::find( 'John Smith' );
$books = $author->books;
```

To load the author we can use static method called “load” that takes ID as an argument. This method returns an Author object or NULL, if row with that ID does not exist. Then we use a static method called “find”. This method finds author by name. It also returns Author object or NULL. When we have author loaded we can go through his books by using field “books”.

```
$author->delete();
```

Deleting can be done calling method called “delete” to Author object. This will invoke DELETE query to database for that Author. Depending on relationship type books may be deleted (this is called composition) or left with out author (called aggregation).

Hibernate

Hibernate is a very popular open source ORM library for the Java language. Hibernate's main feature is mapping from Java classes and data types to database tables and types. It provides data query (Hibernate language query, HQL) and retrieval facilities. Developer is relieved from manual data to object conversion and it keeps application portable to all supported SQL databases. A sample of hibernate XML mapping language is given below.

```
<hibernate-mapping>
  <class name="example.model.Author" table="authors">
    <id name="id" column="authorId">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <property name="phone"/>
    <set name="books" inverse="true" table="books">
      <key column="authorId"/>
        <many-to-one column="author"
          class=" example.model.Book"/>
    </set>
  </class>
</hibernate-mapping>
```

In this example, we have Author described in XML. Author has properties named id, name, phone and books. Author to books is a one-to-many relationship meaning that each author has many books, but single book can have only one author.

A code snippet for simple Create-read-update-delete (CRUD) operations in Hibernate is given below:

```
SessionFactory sessFact =
HibernateUril.getSessionFactory();
Session sess = sessFact.getCurrentSession();
org.hibernate.Transaction tr = sess.beginTransaction();

Author author = new Author();
author = new Author();
author->setName('John Smith');
author->setPhone('37512345678');

Book book = new Book();
book->setTitle('Persistence frameworks');
author->books[] = $book;

sess.save( author );
tr.commit();
sessFact.close();
```

Hibernate uses a session factory that creates a session object and transaction object that will be used to commit queries. The remaining code is similar to the example introduced in section 'Active record' with the only difference that the Author object does not have save method, but Session object has.

In Hibernate, data loading is done by means of a session object created in previous

example. Session class has method named “load” which takes ID as an argument and returns Author object from database.

```
Author author = (Author)sess.load(Author.class, (int)ID
);
```

Deleting also relies on a session object. The Session class has a method named “delete” which takes object as argument. This object is deleted from database.

```
sess.delete(author);
```

PHP Frameworks

While Java has had persistence tools from 1999, PHP is quite new to the world of object-relation mapping. PHP introduced a powerful object oriented programming (OOP) structure in PHP 5.0 at the end of 2003, which created possibility to start using ORM tools. One of the first well-known ORM tool for PHP was Propel. The project begun in August 2003, but it was difficult to use and did not become very popular. In 2006, the Doctrine project was started and it is currently the most popular ORM tool in PHP world.

Below we provide an overview of some PHP persistence frameworks.

EzPDO

EzPDO is written for PHP 5.2 and its main goal was to design a lightweight and easy-to-use persistence solution for PHP. EzPDO follows active record pattern. It is a first project that introduced annotations in PHP ORM tools.

EzPDO should only be used with smaller projects as it doesn't give any functionality for more complex applications – e.g. data validation. EzPDO does not require any SQL knowledge to be used. In other words, the framework hides a lot to the programmer and makes many assumptions about the underlying database structure. Hence, the database design cannot be optimized much when using this framework.

A known problem with database design which has a strong impact on object-relation mappings is that of handling N:M relations. EzPDO stores two rows into database for binding two objects, but it should insert only one. It should be noted that EzPDO is only suitable for greenfield projects since the framework forces a particular database schema, so it is not easy to retrofit existing database to work with this framework. Another

disadvantage is that the development of EzPDO has slowed down since the last patch was released in March 2007 so the viability of the project in the long-term is questionable.

Propel

Propel is the first ORM tool for PHP 5 and is based on Apache Torque which is an ORM solution for the Java language. It is an open source library. It allows accessing database using a set of objects and provides an API for storing and retrieving data. Propel allows developer to work with database in the same way as working with other classes and objects.

It should be noticed that Propel is not an active record implementation but a row data gateway and table data gateway implementation.

Propel is a set of two tools: a generator and a runtime framework. The generator first builds PHP classes based on XML schema describing database. These classes can be used to interact with data model. Runtime framework allows using generated classes in PHP scripts to transparently handle writing and reading from database.

Propel depends on three libraries that are required to use it: Phing, Creole and Pear::Log. This is a disadvantage for Propel, as most of the competing frameworks do not need any external libraries.

The following XML code snippet shows the generator directions to create classes:

```
<database name="sample" defaultIdMethod="native">
  <table name="book">
    <column name="id" type="integer" required="true"
            primaryKey="true" autoIncrement="true"/>
    <column name="title" type="varchar" size="255"
            required="true" />
  </table>
<table name="author">
    <column name="id" type="integer" required="true"
            primaryKey="true" autoIncrement="true"/>
    <column name="name" type="varchar" size="128"
                required="true"/>
    <column name="phone" type="varchar" size="128"
            required="true"/>
```

```
</table>
</database>
```

The following code snippet shows how to code simple CRUD operations in Propel.

```
$author = new Author();
$author->setName('John Smith');
$author->setPhone(37512345678);

$book = new Book();
$book->setTitle('Persistence frameworks');
$book->setAuthor( $author );

$author->save();


// LOAD
$author = AuthorBeer::retrieveByPK( (int)ID );

// loading author book
$c = new Criteria();
$c->add( BookBeer::author, array((int)ID),
Criteria::EQUAL );
$books= BookBeer::doSelectJoinAuthor( $c );

// DELETE
AuthorBeer::doDelete( $author );
```

Simple CRUD operations are mostly the same as in the active record sample. The main difference is its query language for retrieving data by different criteria. For that a Criteria object is needed and to it different restrictions are added. In the above example, we want to find author books. This is done by creating a join query using a BookBeer class.

Propel's main disadvantage is that its documentation is not well maintained. There are basic samples but a developer cannot get help if he starts working with more complex scenarios. Another related problem is that the Propel user community is small as evidenced by the low number of updates in the Propel subversion code repository and the low level of activity in the related forums.

JDao

JDao is an active record implementation that is part of the Jelix framework. It is part of a bigger library and cannot be used as a stand-alone tool. This is also noted in the

documentation; the reason is that it has too many dependencies. JDao follows a table data gateway and a row data gateway pattern. Data mapping is done using XML. From that XML classes are automatically generated that can be used on runtime. JDao is mostly used in France and part of its source code and documentation is in French. JDao could become more appealing to the international community if its community of developers and users adopted English as their communication language.

Doctrine

Doctrine is an object relation mapper for PHP. It was written by Konsta Vesterinen in 2006 and has evolved a lot since early releases. Since Doctrine is an open source project - many people have joined and contributed to its development. Currently there is a team of about four active developers around Doctrine, and a few other less active developers. The first official version was released in September 1 2008.

Doctrine follows the active record pattern and sits on top of a database abstraction layer called PDO that allows easy access to all databases, such as very popular MySql and PostgreSql. It has been officially integrated into many PHP frameworks – Symfony is one of the most known.

Doctrine is very easy to use as it requires almost no configuration when a new project is started. It can generate classes from an existing database and the programmer can specify relations and add custom functionality to created classes. A project can also be started in the reverse direction as well, that is by first creating classes, specifying how to persist these classes, and then generating the database schema.

We have chosen to include Doctrine in our comparative study because it has very good technical manuals and tutorials and a very active community of users.

Below, we provide some code samples of Doctrine. First we define a couple of classes, and then we show code snippets to illustrate the use of the framework.

Author Class:

```
class Author extends Doctrine_Record {
  public function setTableDefinition() {
    $this->hasColumn('id', 'integer',11, array(
                           'notnull' => true,
                           'primary' => true,
                           'unsigned' > true,
                           'autoincrement' => true));
    $this->hasColumn('name', 'string',200, array(
                           'notnull' => true));
    $this->hasColumn('phone', 'int',11, array(
                          'notnull' => true));
    }
    public function setUp() {
        $this->hasMany('Book', array(
                           'local' => 'id',
                           'foreign' => 'author'));
    }
}
```

Book Class:

```
class Book extends Doctrine_Record {
  public function setTableDefinition() {
    // primary key column is added automatically, if not
set
    $this->hasColumn('name', 'string',200, array(
                           'notnull' => true));
    $this->hasColumn('author', 'int',11, array(
                          'notnull' => true));
    }
    public function setUp() {
        $this->hasOne('Book', array(
                           'local' => 'author',
                           'foreign' => 'id'));
  }
}
```

Usage Samples:
In the following code snippet, first an Author object is created. Name and phone number are set for the author. Then Book object is created and its title is set. Subsequently we relate author and book to each other and save them to database by calling "save" method for author. Both Author and Book are saved.

```
// create new author
$author = new Author();
$author->name = 'John Smith';
$author->phone = +37512345678;

// create one book for author
$book = new Book();
$book->title = 'Persistence frameworks';
$author->books[0] = $book;

// save author and book
$author->save(); // lets say our author ID is 1
```

Loading needs AuthorTable object. That object can create queries to database. Next, "find" method is called, which takes object ID as an argument and returns Author object. Then author name, number of books he has written and titles of these books are outputted.

```
$table = Doctrine::getTable('Author');
$author = $table->find( 1 );
echo 'Author '.$author->name.' has written
'.count($author->books).' book: ';
foreach($author->books as $book ) {
  echo '"'.$book->title . '"';
}
// would output:
// Author  John Smith has written 1 book: "Persistence
frameworks"
```

Deleting both author and book is done simply by calling the *delete* method on an Author object.

```
$author->delete();
```

DomAr

The DomAr project started in 2007 and is written by the author of this thesis, Oliver Leisalu. Currently it is not an open-source project so it is not available for public, but in near future it hopefully will be. The framework has been used to code a number of applications in commercial projects in which the author has been involved.

DomAR follows active record pattern. Its main purpose is to give a quick and easy solution for creating application model that can modify and validate complex data model. It has a query language called DomSql which supports object queries, data queries that return data as arrays for faster performance and writing your own SQL sentences for complex requests. Annotations are used for data mapping. Below we provide code snippets equivalent to those we provided above for the Doctrine framework.

Class Definitions

Author Class:

```
/**
* @orm tablename authors
*/
Class Author exnteds  DomArObject {
  /**
  * @orm char(50)
  */
  public $name;
  /**
  * @orm int(11)
  */
  public $phone;
  /**
  * @orm owns many Book inverse author
  */
  public $books;
}
```

Book Class:

```
/**
* @orm tablename books
*/
Class Book exnteds DomArObject {
  /**
  * @orm char(50)
  */
  public $title;
```

```
  /**
  * @orm has parent Author inverse books
  */
  public $author;
}
```

Usage Samples:

In the following code snipper, an Author object is created. Its name and phone number are set. Then Book object is created and its title set. After that Author and Book are binded together and “save” is called for Author object which saves both Author and Book into database.

```
// create new author
$author = new Author();
$author->name = 'John Smith';
$author->phone = +37512345678;

// create one book for author
$book = new Book();
$book->title = 'Persistence frameworks';
$author->books[] = $book;

// save author and book
$author->save(); // lets say our author ID is 1
```

An Author object is loaded from database by means of a static method called “load”. Then author name and number of books he has written is outputted. After that all his book titles are printed.

```
$author = Author::load( 1 );
echo 'Author '.$author->name.' has written '.$author-
>books->count().' books: ';
foreach($author->books as $book ) {
  echo '”'.$book->title . '”';
}

// would output:
// Author  John Smith has written 1 book: “Persistence
frameworks”
```

Deleting is done by calling “delete” method to Author object. Both author and his books are deleted.

```
$author->delete();
```

Comparative Evaluation

Functionality Comparison

Different classes of application require different sets of features from an ORM framework. In some applications, the data structures manipulated are fairly simple (e.g. simple one-way relationships between classes) . Other applications manipulate more complex data structures including bi-directional relations. Similarly, in some applications, full support for transactions is crucial, while smaller applications may not require it. Below, we list a number of features that one may expect to find in an ORM framework. The list includes features related to the data structures, support for integrity (transactions) and optimization (e.g. memory caching), and additional features such as event listeners or views. Finally, other features are non-functional (e.g. documentation and level of activity of the developer's community). The list is not intended to be comprehensive, but merely to provide a starting point covering common features.

- Relationships: does the framework supports the persistence of one-way relationships (references) between classes?
- Bi-directional relationships: does the framework supports the persistence of two-way relationships (references) between classes, and maintaining the referential integrity between these two classes?
- Hierarchical data: Does the framework include tools for working with hierarchical data?
- Column aggregation inheritance – With column aggregation inheritance there can be objects from more than one class in a single table. Does the framework support handling this?
- Transactions – Does the framework support transactions?
- Dirty checking – Does the framework monitor and handle dirty objects? Object is called dirty when some of its property values are changed.
- Optimization – Does the framework have documentation about how to optimize data model for better performance?
- Memcache support – support for caching objects in memory so they don't have

to be loaded from database for each session.

- Support for hooks / event listeners – Does the framework support event listeners like before save, before update, after delete, etc.
- Support for views – Does the framework support handling database views?
- Automatic generation of schema – Can the database schema generated automatically from object model?
- Automatic generation of model – Can the model be generated automatically from database schema?
- Annotations – Can the data mapping be done using annotations?
- Data mapping language – Is there any data mapping languages besides default mapping? (Default for DomAr is annotations and PHP code for Doctrine)
- Supported databases – What databases are supported?
- Extendable - is there documentation about extending the framework? Writing custom property/column handlers etc...
- Query language – Is there any documented query language?
- Community – author's assessment about the community size and activeness.
- Model behaviours
    - Sluggable – automatic generation of nice search engine friendly ID used in URL-s. (Example: Article title is “Article about ORM” then its slug would be article-about-orm)
    - Timestampable – automatic timetracking of updates.
    - Versionable – stores and maintains versions of an object. Allows reverting back to older versions.
    - Searchable – automatically manages fulltext search index creation and management.

The following table provides a side-by-side comparison of Doctrine and DomAR in terms of the above set of features.

Criteria	**Doctrine**	**DomAr**
Relationships	Yes	Yes
Bi-directional relationships	Yes	Yes
Hierarchical data (tree-structured)	Yes	No. Loading from database cannot be done efficiently.
Transactions	Yes	Yes

Optimization	Yes	Yes
Column aggregation inheritance	Yes	Yes
Dirty checking	Yes	Yes
Support for hooks / event listeners	Yes	Yes
Support for views	No	No
Automatic generation of schema	Yes	Yes
Automatic generation of model	Yes	No
Annotations	No (But will be in Doctrine 2.0, which is not yet available.)	Yes
Data mapping language	Yes	No
Supported databases	All that PDO supports	Mysql, PostgreSql
Extendable, documented “how to”.	No	Yes
Query language	Yes (DQL)	Yes (DomSql)
Age	3 years from project start. Version 1.0 has been available for one year	2 years from project start. Final version 2.0 has been available for one year.
Version	1.0	2.0
Community	Large, Active	None
Memcache support	Yes	Yes
Useful model behaviours: Sluggable Timestampable Versionable Searchable	 Yes Yes Yes Yes	 Yes Yes Yes Yes
Package size (Only main libraries, no documentation.)	2.22MB	382KB
Files in package	343	72

Based on the above table, we can conclude that Doctrine has more features. The main advantage over DomAr is Doctrine's comprehensive documentation, support for PDO and a very active community. Support for tree-structured data is very useful for projects

that need it. Both frameworks can create database schema from model, but Doctrine can also create model from existing database. Advantages for DomAr are its small size and available documentation of how to extend it. Annotations are also a plus since this usually feels more natural for programmers. It should be noted that the next version of Doctrine also supports annotations.

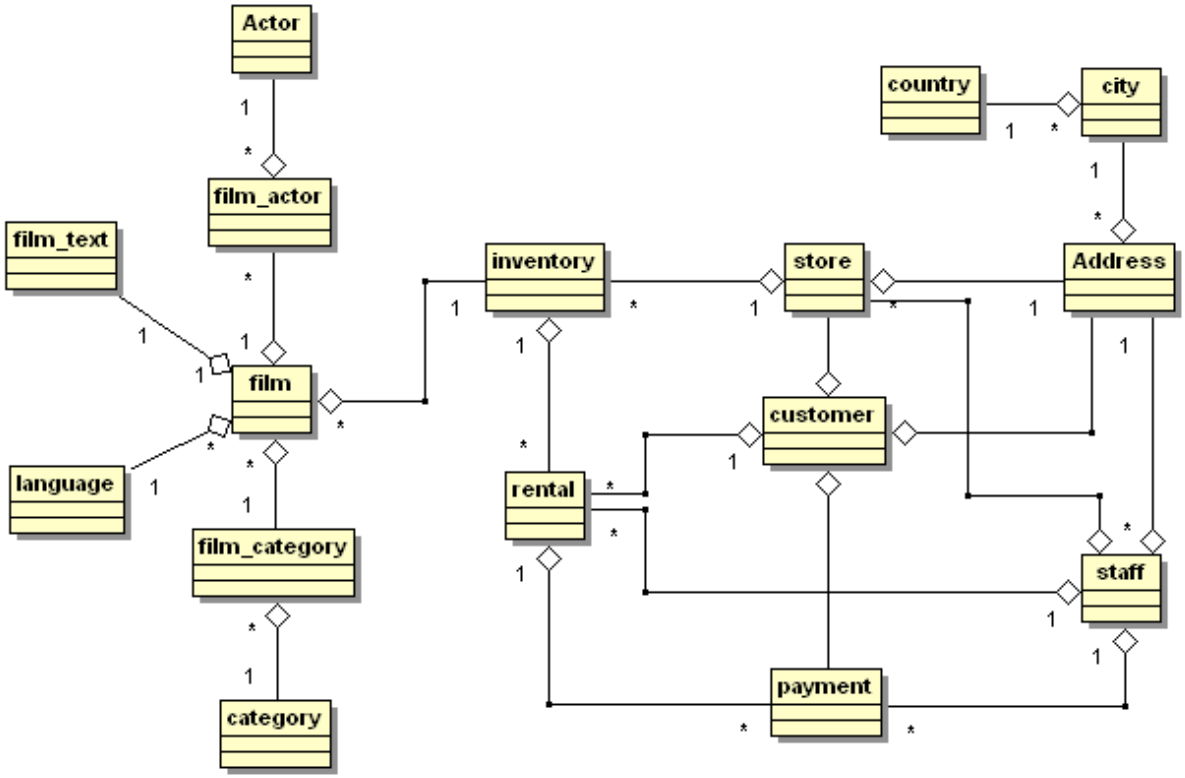
Application Complexity

To test application complexity the author chose a pre-existing database and built a working object model on top of it. The database chosen is called *Sakila*. It is a sample database that was originally build to highlight the features of MySql relational database system. The Sakila database provides a good testbed for an ORM tool as it uses many of the features that mainstream database systems have.

Sakila Database

The Sakila database was developed by a former Mysql AB documentation team member Mike Hillyer. It is intended to provide a standard schema that could be used in books, articles, samples and so forth. It also highlights the latest features of MySql. Sakila database is designed to represent a DVD rental store.

The below class diagram illustrates the model structure that was created for DomAr and Doctrine. The model structure for both implementations was the same.



Implementation Notes

In the beginning of the implementation, there were some problems with the DomAr solution. DomAr does not allow defining custom primary key property. It only accepts primary key property named as “id”, but Sakila database had primary key fields named as film_id, actor_id, etc. To solve this issue for DomAr, the database schema had to be modified.

There were also problems with some data types. There was no support for Year and Timestamp data types. In model definition these were replaced with similar types.

DomAr has a very easy way to handle many-to-many relationships. Unfortunately this “easy” approach imposes the relational table to have a strict structure with two fields named childId and parentId. The Sakila database field names were different. An alternative solution could be to create an association class that captures an N:M relationship by means of N:1 and 1:N relationships. This solution is also promoted by Doctrine. The author decided to use the second option since Doctrine also needed the same approach.

Both of the ORM solutions did not support views, so these were not created. Of course, the views can always be manually added to the relational database, but they remain outside the scope of the ORM framework.

Usability Comparison

Per the author's assessment, the complexity of coding the sample application in the Doctrine and in DomAR was comparable. In the subjective assessment of the author, DomAr annotations are more intuitive to use than the Doctrine mapping language. The annotations syntax is very simple and easy to remember. On the other hand DomAr had problems with database design (id columns and some data types were not supported) requiring modifications to the database. So for databases that already exist, it might be better to use Doctrine, especially since it can generate a model automatically from an existing database.

DomAr had a bit more lines of code (mostly because of three line annotations), but the difference was not significant. For both implementations there were no seemingly unnecessary lines.

It also should be noted that DomAr followed the active record pattern more closely, as it

had find and query methods within model class. Doctrine on the other hand had them in a Table class, which matches Table data gateway pattern. Doctrine solution is said to be better for testing.

One of the problems that was found during Doctrine testing was that if the same row was loaded from the database twice, Doctrine gave different instances of that object. This is an unexpected occurrence and might cause problems in some applications. DomAr did not have that problem and properly returned same instance. (It actually did not make the second query to database, instead used object cache.)

Performance Comparison

The problem of evaluating the performance of ORM frameworks is related to that of evaluating the performance of Object Database Management Systems (ODBMS). ODBMS arose during the late 80's and early 90's when object-oriented programming started to gain in popularity and the need to deal with the impedance mismatch between object and relational models became a problem of practical significance. In ODBMS, objects are stored and manipulated as first-class citizens and association traversal was the basic querying operation (as opposed to joins as in relational databases). One of the problems of ODBMS however was their poor performance relative to established relational database technology. Accordingly, the need to evaluate and optimize the performance of ODBMS became a pressing problem.

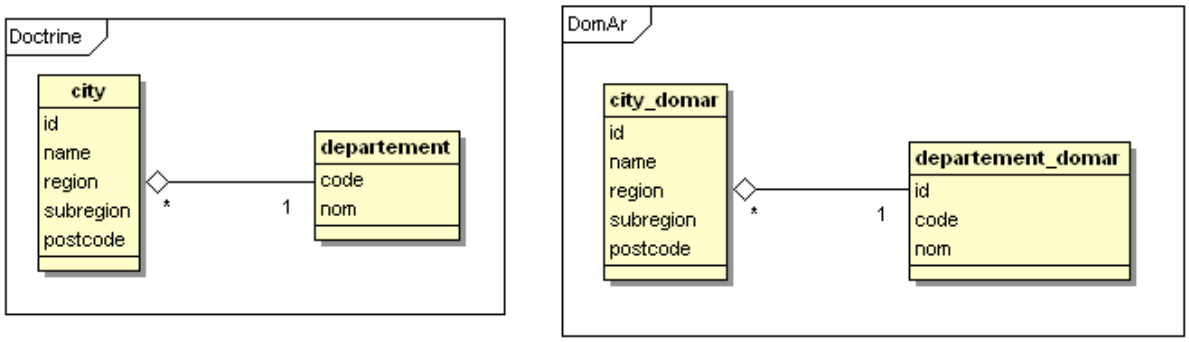
In order to evaluate the performance of ODBMS, a number of benchmarks were created: HyperModel, OO1 and OO7. The OO1 was intended to study the performance of engineering applications. The HyperModel approach was based on earlier versions of the OO1 benchmark. It used a more complex model with more complex relationships and wider variety of operations. The OO7 benchmark was based on both of these benchmarking efforts. It includes queries to evaluate the performance of associative operations, traversals, updates on simple and complex object types. While the OO7 benchmark was originally designed to assess the performance of ODBMS, but it has also been adapted to evaluate the performance of Java ORM frameworks, particularly Hibenate [10].

Unfortunately none of these benchmarks have been realized for PHP. Hopefully this will change soon as there are already many ORM tools available for PHP, but no standardized benchmarks currently exist that would help to choose between them. Since

creating that kind of benchmark is very complex undertaking it will not be done as part of this project but is a possible extension to it. Accordingly, in this work we use a small benchmark introduced by L. Jounneau in an online blog post [11].

The Jounneau Benchmark

The Jounneau benchmark is a small-sized benchmark originally designed to compare four PHP ORM tools: JDao from Jelix framework, Doctrine, Propel and phpMyObject. This benchmark tests the speed of selecting records from database. For analysis, the author added DomAr to that benchmark and ran these tests on our computer.



Model structure class diagram for Jounneau benchmark tests.

The database had to be modified for DomAr since it does not support relationships with custom keys. Original tests table “departement” had field called “code” as primary key, but DomAr did not support that and new column was added called “id”. For testing the author also updated Doctrine version to the latest 1.1. Previously a beta version was used.

The Jounneau benchmark consists of 12 tests listed below:

Nr	**Test name**	**Description**
1	Little select	Simple select query is used to retrieve 20 city rows
2	Little complex select	20 cities and their regions are loaded
3	Medium select	All 96 departments are retrieved.
4	Big select	1000 cities are retrieved.
5	Big complex select	1000 cities are retrieved with their regions.
6	Huge select	10000 cities are retrieved.
7	Stress little select	Same query is used as in little select but is ran for 100 times.
8	Stress little complex select	Same as previous but is ran for 100 times.

9	Stress medium select	Same as previous but is ran for 100 times.
10	Stress many individual record	A single record is loaded from database for 1000 times.
11	Stress many individual complex record	A single record is loaded from database for 100 times with its region.
12	Stress select with criteria	28 cities are loaded from database that match criteria id<200 and postcode LIKE %50

Per author's assessment, the Jounneau benchmark has some limitations. It focuses on select queries and does not have insertions, updates nor deletes. There is only one relationship type (1:N) used and the data types are very simple. Some of the tests does not show much useful information. If tests are run on single machine there is no difference if the query is called once or 100 times. Stress tests should be done by running all queries are run coincidently. Memory usage should also be tested.

Benchmark Results

Tests where run on a 2.2Ghz computer with 2GB RAM. Apache version was 2.2.8, PHP version 5.2.8 and MySql version 5.0.51. Operating system was Windows XP with service pack 3. Each test was run 10 times and average results were calculated.

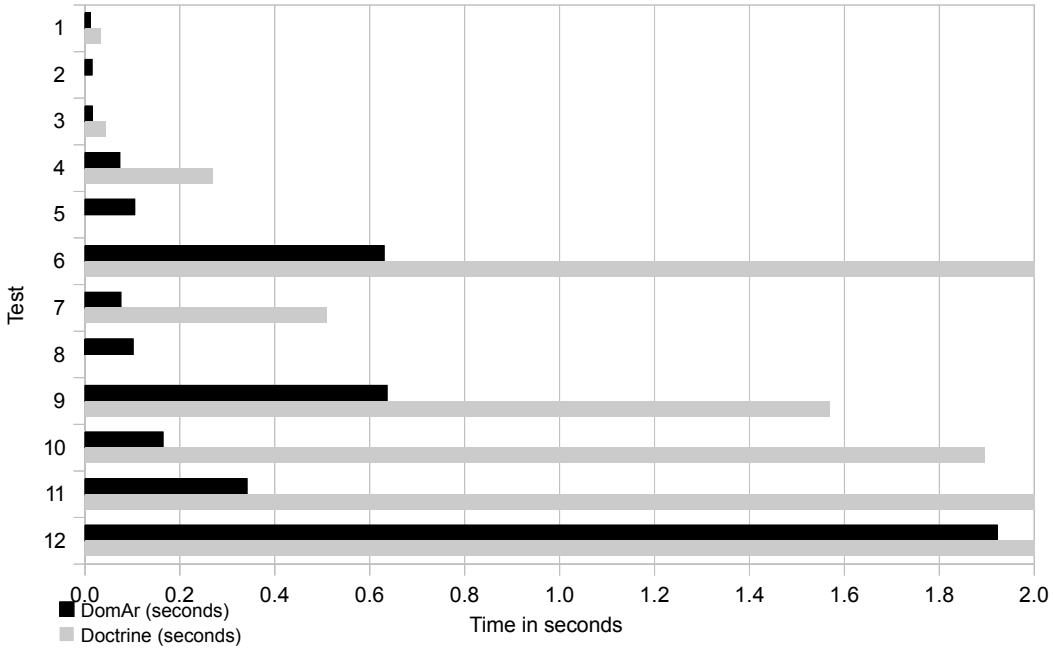
Test with numbers 2, 5 and 8 were not working with Doctrine because of a SQL syntax error. This seemed to be a bug in Doctrine, as the code used was the same as introduced in the manual. This bug was also reported in [11].

The results of running the Jounneau benchmark using DomAR and Doctribe are given in the following tables and graphs:

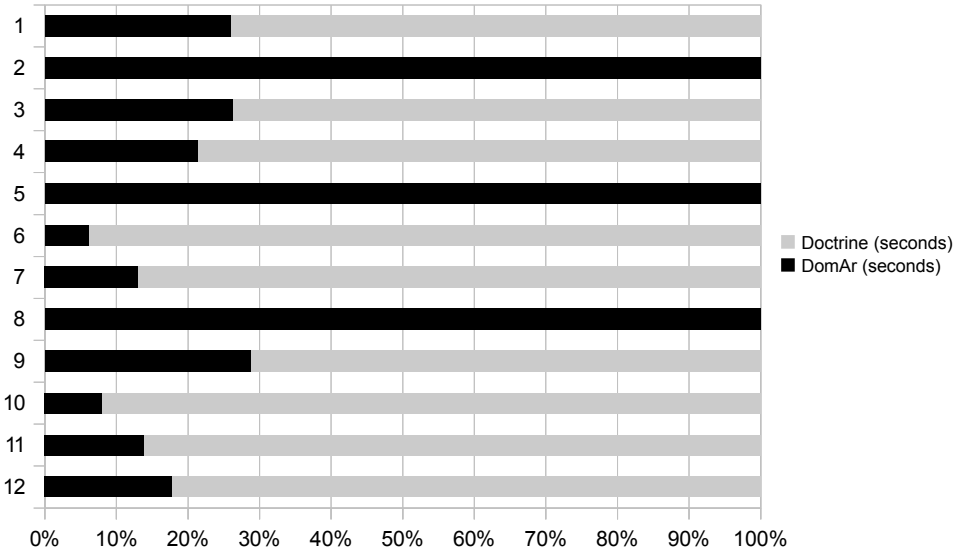
Nr	**Test name**	**DomAr (seconds)**	**Doctrine (seconds)**
1	Little select	0.01141	0.0323
2	Little complex select	0.01517	n/a
3	Medium select	0.01567	0.0438
4	Big select	0.07357	0.2691
5	Big complex select	0.10463	n/a
6	Huge select	0.63003	9.4332
7	Stress little select	0.07593	0.5081
8	Stress little complex select	0.10174	n/a
9	Stress medium select	0.63619	1.5679

10	Stress many individual record	0.16414	1.8962
11	Stress many individual complex record	0.34114	2.11713
12	Stress select with criteria	1.92202	8.86873

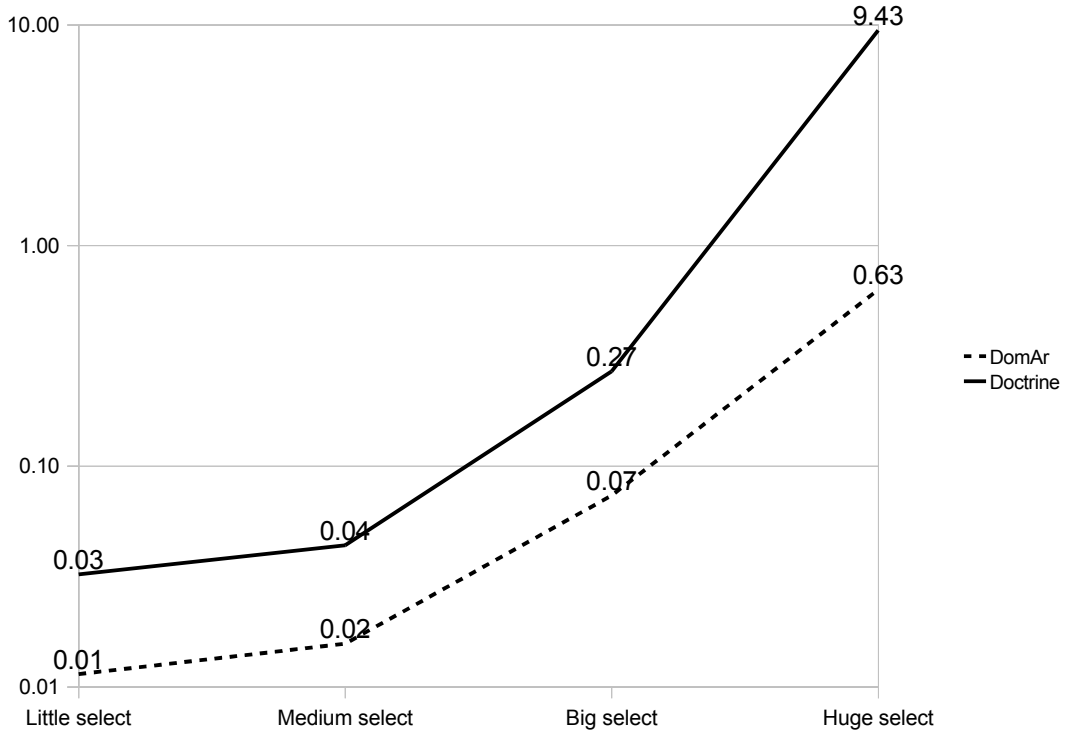
Test results table [a]



Test results bar chart. [b]



Percent stacked bar diagram of test results. [c] Note: tests #2, #5, and #8 did not return results for Doctrine, so the percent results for those tests are not relevant.



Main select queries line chart. [d]

Per the test results, DomAr appears to be faster than Doctrine by a factor of at least 2.5. Chart [d] shows that Doctrine's execution time grows more when selecting more rows from database. This means that creating record objects costs more in Doctrine than in DomAr in terms of time. In test #1 (Little select) Doctrine is 3 times slower than DomAr, but in test #6 (Huge select) it is 15 times slower.

The author was also interested in memory usage, so this line was added to the end of each test:

```
echo memory_get_usage(true)/1024;
```

This shows how much memory is used by a PHP script in KB. For test #6 DomAr used 14592KB and Doctrine used 28928KB. We can see that Doctrine uses about twice as much memory as DomAr.

Conclusion

Persistence frameworks are tools that can move data from their most natural form to a more permanent data store. They are meant to address the impedance mismatch between object-oriented programming languages and relational database systems. The most common patterns used by these frameworks are table data gateway, row data gateway and active record.

There are many ORM tools for PHP, but most of them are in early stages of development or development has stalled. Currently the most comprehensive publicly available ORM tool for PHP is Doctrine. DomAr is hopefully the next runner-up with its own pros and cons. Doctrine's main arguments against DomAr are its richness of functionality and the size of its community, but as we have seen here, it has a rather low performance when compared with DomAr. DomAr's main disadvantage is that it forces a stricter database design and its lack of functionality in some areas.

The benchmark test results of Doctrine and DomAr may be biased because of the characteristics of the Jounneau benchmark. As explained in the paper, this benchmark does not include updates and it focuses on select queries. A possible future extension of this work can be repeating the experiments with a more complex benchmark (OO7). It would also be desirable to include other PHP frameworks (e.g. JDao) in the benchmark in order to gain a deeper understanding of the tradeoffs between functionality, usability and performance in the field of PHP ORM frameworks.

References

[1] Yasser El-Manzalawy, *Accessing data through persistence frameworks,* 2005. Article at http://www.developer.com/db/article.php/3355151

[2] *Object-relational impedance mismatch*. Wikipedia article at

http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch

[3] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Staffort, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002

[4] Red Hat Inc. *Hibernate reference manual*. Http://hibernate.org

[5] *DomAr samples*. Tutorials and samples about DomAr http://inditel.ee/CODE/samples/

[6] *Doctrine project homepage* at http://www.doctrine-project.org/

[7] *Doctrine*. Wikipedia article about Doctrine at http://en.wikipedia.org/wiki/Doctrine_(PHP)

[8] *Propel project homepage* at http://propel.phpdb.org/trac/

[9] *EzPdo project homepage* at http://www.ezpdo.net/

[10] Pieter Van Zyl, Derrick G.Kourie, Andrew Boake. Comparing performance of Object databases and ORM tools. *In Proceedings of the SAICSIT Conference,* Cape Province, South Africa, October 2006. ACM Press. Available at

http://www.odbms.org/download/027.01%20Zyl%20Comparing%20the%20Performance%20of%20Object%20Databases%20and%20ORM%20Tools%20September%202006.PDF

[11] L. Jounneau. *Comparatif des performances des ORM PHP* (29.11.2007), Online blog post at http://ljouanneau.com/blog/post/2007/11/29/723-comparatif-des-performances-des-orm-php

[12] *Sakila database documentation.* Available at http://dev.mysql.com/doc/sakila/en/sakila.html

Kahe PHP püsivate andmete salvestamise raamistike võrdlus

Bakalaureusetöö

Oliver Leisalu

Resümee

Antud töö eesmärk on anda ülevaade püsivate andmesalvestus raamistike ideest. Tutvustada PHP jaoks valmistatud raamistikke ning näidata nende paremaid ja halvemaid omadusi. Oma omadustelt olid teistest üle kaks raamistiku: Doctrine ja DomAr. Edasistes võrdlustes kasutatigi neid kahte.

Võrdlemiseks koostati võrdlustabel, mis koosnes nii funktsionaalsetest omadustest kui ka mittefunktsionaalsetest. Võrdluse tulemusest selgus, et Doctrine eelis on põhjalik dokumentatsioon, palju lisavõimalusi ja aktiivne kasutajaskond.

Teise etapina testiti jõudlust kahe tähtsa omaduse jaoks: objektipuu läbimine ja andmebaasi päringud. Jõudlustestidest selgus, et DomAr on konkurendist oluliselt kiirem.

Lisaks analüüsiti nende raamistike kasutamise mugavust ja funktsionaalsust. Selleks loodi olemasoleva andmebaasi peale toimiv objekti mudel. Loodud rakendust võrreldi erinevatest aspektidest, milleks oli programmeerimiseks vajalik töö hulk ja koodi ridade arv. Analüüsist selgus, et nii töö hulk kui ka koodi ridade arv on mõlemas raamistikus enam-vähem võrdsed. Probleeme oli DomAr raamistikuga, mis nõudis rangemat andmebaasi struktuuri ja seetõttu oli olemasoleva andmebaasi peale mudeli loomine komplitseeritud. Lahenduseks muudeti andmebaasi struktuuri.

Antud töö võimalik edasiarendus on koostada uued võrdlustestid, kasutades mõnda üldtuntud püsivate andmete salvestamise raamistike testimissüsteemi. Lisaks oleks oodatud uute raamistike lisamine võrdlusesse.