F.B. Botelho
Xaraya Development Group
January 2003

Query Abstraction

Status of this Memo

This document specifies an Xaraya standards track protocol for the Xaraya community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Xaraya Official Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright © The Digital Development Foundation (2003). All Rights Reserved.

Abstract

This RFC presents a proposal for Query Abstraction inside Xaraya Architeture.

Table of Contents

1 Introduction **3**

2 Pros **4**

3 Requirements List **5**

4 Sugestions List **6**

5 Possible Implementations Reasearch **7**

6 Implementation Idea **8**

7 Concept of how the query abstraction would be used **9**

8 Concept sketch of the proposed code **11**

9 Example Implementation Idea **18**

9.1 Description 18

9.2 API 18

9.3 Examples 19

10 Relevant References **23**

Author's Address **24**

A Example appendix **25**

Intellectual Property and Copyright Statements **26**

1. Introduction

Query Abastraction has been made necessary to give us the hability to generate efficient SQL queries which are portable across all our Database Pool. It will provide us a way to create queries without any preocupations about the underlying storage system.

2. Pros

"Allows for the creation of queries in a persistent storage neutral manner. Expressed as objects, queries remain consistent across multiple persistent storage solutions. Should the database change from RDB to an OODB " or even to XML Based Query Languages (A lot of research is being done in the field - References [3,4]), "no part of the application will need to be modified." - Reference [2]

3. Requirements List

The following is list of requirements about the implementation of the Query Abstraction. Please add your own.

1. Turn Xaraya completely database agnostic. [Main reason to be implementing this]
2. As we already have a DB Abstraction Library (ADODB), this should be an extension of it. [Gary]
3. Devs should be able to keep using their known SQL for simple queries, where portability is not a problem. [LadyofDragons]
4. The query structure should be able to be passed around the program and receive input from many different modules. [Nuncanada]

4. Sugestions List

The following is list of sugestions made for the RFC. Please add your own.

1. Make the methods name closer to their meaning in english than as the names being close to their sql equivalent [???]
2. Make a interface thru a meta-language, that will be easier to read and understand than the nested methods [Jason Judge]
3. Abstract INSERT/UPDATE queries too [Marcinmilan]

5. Possible Implementations Reasearch

There just a few query abstraction packages for PHP, 2 found in the PEAR archives. Both have serious shortcomings to our needs (DB_DataObjects and DB_QueryTool). Both were made without taking in account the necessity to handle multiple DBs.

Research done in other languages for inspiration has shown that most query abstractor out there have too simplistic approaches, mainly:

1. Simplistic - They simply support the most basic features of the SELECT query, where the own SQL language is already the basic abstraction.
2. Placeholders - They simply store 'common' queries with wildcards showing where the input values should be added. Clearly this doesnt produce DB independent SQL.
3. Stored queries - They store the DB-specific query in files to be called, would be a similar system to that of using our MLS system to translate SQL queries to a specific DB dialect. This requires you to know how to implement the query in every DB, and although it is able to support more complex queries than the simplitic approach. It is not able to translate dynamic queries.

So there are two possible implementaions:

1. Stored queries - It's not able to translate dynamic queries. And it will probably create bigger problems than translations themselves (as i think it is easier to find a translator than a DB specialist). Requiring that the Mod Dev himself creates the translation to all possible DBs doesnt seem like a good idea.
2. OO Query Abstraction - The good part is that it can implement all our requirements and could be a link towards future storage systems. Still it will obligate Mod Devs to learn it (for more complex queries) instead of using the well known SQL.

6. Implementation Idea

The following is where the proposed implementation itself is shown

There should be two separate concepts, that of the query structure which preferebly would be a class used only to hold the devs wishes, and drivers to take this annotated wish list and transform it into a proper query.

The basic idea is a simple one, you will have a class (our query abstraction class) which will *hold* (with pointers, creating a rigid structure of relations) what the developer is trying to achieve. Then when he thinks he already has everything he wants in the 'shopping list', this structured list will be passed over to a specific DB driver which will translate the given structure in the best possible query for the developer.

7. Concept of how the query abstraction would be used

How would it work:

```
    This would work if there was data present in the xartables.php informing us
what are the tables,
    their fields, their foreign keys with the nature of the relationship (1-1,
1-Many, Many-Many,
    what to do with missing values on one side and/or the other?)

    $query =& new sql_select;
    $query->select('name','description','left','right');
    $result =& $query->run();

    Assuming there is a 1-1 relationship between users and their groups, in two
separated tables
    and that this is properly set in the xartables.php
    $query =& new sql_select;
    $query->select('username','groupname');
    $result =& $query->run();

    {Now on omitting $query =& new sql_select; }

    You would be able to do any query including joins that easily unless there
were conflicting field names
    then you would have to set their respective tables:

    Suppose there is an alias field in the user table and a alias in the groups
table
    This way the bulder wont know which alias to choose

    $query->select('username','groupname','alias');


    Right now i am thinking about this:

    $field =& $query->create('field','alias');
    $field->table('groups');
    $query->select('username','groupname', $field);

    But can be changed to other syntaxes:  (After writing, i think this one seems
better)

    $field =& $query->field('alias', 'groups');
    $query->select('username','groupname', $field);

    Or:
    $field =& $query->field('alias');
    $field->table('groups');
    $query->select('username','groupname', $field);

    Which one is the best? Please comment!!!!

    Now the part where i havent found a better way of doing the syntax:
    It is simple, but a lot of nested code...

    $query->where($query->condition('userid','>','1000'));

    So, something like this:
    "SELECT
            COUNT(P2.".$categoriescolumn['cid'].") AS indent,
            P1.".$categoriescolumn['cid'].",
            P1.".$categoriescolumn['name'].",
            P1.".$categoriescolumn['description'].",
            P1.".$categoriescolumn['image'].",
            P1.".$categoriescolumn['parent'].",
            P1.".$categoriescolumn['left'].",
            P1.".$categoriescolumn['right']."
       FROM $categoriestable AS P1,
            $categoriestable AS P2
      WHERE P1.".$categoriescolumn['left']."
            >= P2.".$categoriescolumn['left']."
```

```
        AND P1.".$categoriescolumn['left']."
            <= P2.".$categoriescolumn['right'];"

    Would be something like:

    $P1 = $query->table('categories','P1');
    $P2 = $query->table('categories','P2');
    $query->select(
        $query->function('COUNT',$query->field('cid',$P2),'indent'),
        $query->field('cid',$P1),
        $query->field('name',$P1),
        $query->field('description',$P1),
        $query->field('image',$P1),
        $query->field('parent',$P1),
        $query->field('left',$P1),
        $query->field('right',$P1));

    $query->where(
        $query->condition(
$query->condition($query->field('left',$P1),'>=',$query->field('left',$P2)),
            'AND',
$query->condition($query->field('right',$P1),'<=',$query->field('right',$P2))));


    Comment!
```

8. Concept sketch of the proposed code

Sketch of the proposed code:

```
/* 5 basic SQL structure types: alias, table, field, function, condition
 * Special structures: CASE, USER_VARIABLE???
 */


// 3 very simple equal (if werent for the hierarchy) classes...
// Leaving like this for now, as it helps understand what is the idea.
// get_class($object) == 'sql_alias'


class sql_alias {
    var $alias;
    //Constructor
    function sql_alias ($alias) {$this->alias = $alias;}
    function get_alias () {return $this->alias;}
}

class sql_table extends sql_alias {
    var $table;
    //Constructor
    function sql_table ($table, $alias = NULL) {
        $this->table = $table;
        if ($alias !== null) {
            $this->sql_alias($alias);
        }
    }
}

class sql_field extends sql_table {
    var $field;
    //Constructor
    function sql_field ($field, $table = NULL, $alias = NULL) {
        $this->field = $field;
        if ($table !== null) {
            $this->sql_table($table);
        }
        if ($alias !== null) {
            $this->sql_alias($alias);
        }
    }
    function get_field () {return $this->field;}
}

class sql_function extends sql_alias{
    //Functions: SUM, AVG, COUNT, MIN, MAX, LIKE, NOT, any other??
    var $function;
    //Points to a sql structure (field, function or condition)
    var $pointer;
    function sql_function ($function, $pointer, $alias = NULL) {
        $this->function = $function;
        $this->pointer = $pointer;
        if ($alias !== null) {
            $this->sql_alias($alias);
        }

    }
    function get_function () {
        return Array('function' => $this->function, 'pointer' => $this->pointer);
    }
}

class sql_condition
{
    //Points to a sql structure (field, function or another condition))
    var $pointer1, $pointer2;
    //Connectives: +, -, /, *, =, >, <, <=, >=, any other??
    var $connective;
    function sql_condition ($pointer1, $connective, $pointer2) {
```

```
        $this->pointer1 = $pointer1;
        $this->pointer2 = $pointer2;
        $this->connective = $connective;
    }
    function get_condition () {
        return Array('pointer1' => $this->pointer1, 'pointer2' => $this->pointer2,
'connective'=>$this->connective);
    }
}

    // Should CASE (or DECODE or iff) be considered a function? No! It is a
special case
    // I think it maybe the case for having a special function (as i think ADODB
will have
    // such a function sooner or later)
    //Use Example : SELECT a.ManagerName, SUM(CASE WHEN b.Approval='Y' THEN 1 ELSE
0 END) as Y,
class sql_case
{
    var $case = Array ();
    function add_case ($when, $then, $else) {
        $case[] = Array ('when' => $when, 'then' => $then, 'else' => $else);
    }
}


// sql_connective or sql_operator?
// class sql_connective {}
//  Dont think it is necessary... They are considered strings...

/**
 * Helper Class to mount a SQL Select Query
 * Its a little more than a struct, keeping the different 'areas' from
 * the Select Query separated. So tables, fields and join statements can be
 * easily exchanged between different parts of the code.
 */
class SQL_SELECT
{
    var $areas = Array ('select'  => Array(),
                        'from'    => Array(),
                        'join'    => Array(),
                        'where'   => NULL,
                        'groupby' => Array(),
                        'having'  => NULL,
                        'orderby' => Array(),
                        'limit'   => Array('nrows'=>null,'offset'=>null),
                        'options' => Array());

    function checkNotTable (&$input) {
        // check('field') needs to be the first, as fields are the standard input
        if (!$this->check('field',$input)     &&
            !$this->check('function',$input)  &&
            !$this->check('condition',$input) &&
            !$this->check('case',$input)) {
            return false; //Error
        }
        return true;
    }

    function &condition ($pointer1, $connective, $pointer2) {
        if ($this->checkNotTable($pointer1)) {
            die ('Pointer1 is not a field, function or condition');
            return; //Error
        }

        if ($this->checkNotTable($pointer2)) {
            die ('Pointer2 is not a field, function or condition');
            return; //Error
        }

        $obj =& new sql_condition ($pointer1, $connective, $pointer2);
```

```
        return $obj;
    }

    function &function ($function, $pointer) {
        if ($this->checkNotTable($pointer)) {
            die ('Pointer is not a field, function or condition');
            return; //Error
        }

        $obj =& new sql_function ($function, $pointer);

        return $obj;

    }

    function &case ()
        if ((func_num_args()%3)!=0) {die('Case can only be assigned in triples :
WHEN THEN ELSE');}
        $args = func_get_args();

        $obj =& new sql_case ();

        for ($i=0;$i+=3;$i<func_num_args()) {
            $obj->add_case($args[$i], $args[$i+1], $args[$i+2]);
        }

        return $obj;
    }

    /**
    * Adds to the SELECT structure
    *
    * @param string Where in the structure you want to add it? (select, from,
join, where, groupby, having, orderby)
    *                SELECT ... FROM ... JOIN ... WHERE ... GROUPBY ... HABING ...
ORDERBY
    *                This was made this way to make it easier to understand how to
use it
    * @params mixed The rest of the parameters depend on the chosen place of
structure youre adding to:
    *                SELECT  -> any number of fields
    *                FROM    -> any number of tables
    *                JOIN    -> $table, $on, $jointype
    *                WHERE   -> 1 condition
    *                GROUPBY -> any number of fields
    *                HAVING  -> 1 condition
    *                ORDERBY -> any number of fields
    *
    * @author Nuncanada
    * @access public
    */
/*
 *
 * Can be done both ways :
    As $query->add('select'....)
    Or $query->select(...);
    The 2nd seems more natural for us SQL zombies
    function add () {
        $args = func_get_args();
        if (func_num_args()>0) {

            $case = strtolower(array_shift($args));

            switch($case) {
                case 'select':
                    foreach ($args as $arg) {
                        if ($this->checkNotTable($arg)) {
                            return; //Error
                        }
                    }

                    $this->areas[$case] = array_merge($this->areas[$case], $args);
```

```
                    break;


                case 'groupby':
                case 'orderby':
                    foreach ($args as $arg) {
                        if (!$this->check('field',$arg)) {
                            return; //Error
                        }
                    }

                    $this->areas[$case] = array_merge($this->areas[$case], $args);
                    break;

                case 'from':
                    foreach ($args as $arg) {
                        if ($this->check ('table',$arg)) {return;};
                    }

                    $this->areas[$case] = array_merge($this->areas[$case], $args);
                    break;

                case 'join':
                    if (count($args)!= 3) {die('Wrong parameter count. Where and
Having can only have 1 condition parameter');}
                    if (!$this->check ('table', $args[0])) {return;}
                    if (!$this->check ('condition', $args[1]) {return;}

                    $this->areas['join'][] = Array('table' => $args[0],
                                                   'condition' => $args[1],
                                                   'joinType' => $args[2]);

                    break;

                case 'where':
                case 'having':
                    if (count($args)!=1) {die('Wrong parameter count. Where and
Having can only have 1 condition parameter');}
                    if (!$this->check('condition', $args[0]) {return;} //Error

                    $this->areas[$case] = $args[0];
                    break;

                default:
                    die('Unknown Area in the query'); //Error
                    break;
            }
        } else {
            die('Nothing to add');
        }
    }
  */
    var $select = Array();
    function select () {
        $args = func_get_args();
        foreach ($args as $arg) {
            if ($this->checkNotTable($arg)) {
                return; //Error
            }
        }
        $this->select = array_merge($this->select, $args);
    }

    var $groupby;
    function groupby () {
        $args = func_get_args();
        foreach ($args as $arg) {
            if (!$this->check('field',$arg)) {
                return; //Error
            }
        }
        $this->groupby = array_merge($this->groupby, $args);
```

```
    }

    var $orderby = Array();
    function orderby () {
        $args = func_get_args();
        foreach ($args as $arg) {
            if (!$this->check('field',$arg)) {
                return; //Error
            }
        }
        $this->orderby = array_merge($this->orderby, $args);
    }

    var $from = Array();
    function from () {
        $args = func_get_args();
        foreach ($args as $arg) {
            if (!$this->check('table',$arg)) {
                return; //Error
            }
        }
        $this->from = array_merge($this->from, $args);
    }

    var $join = Array();
    function join ($table, $condition, $joinType = '') {
        if (!$this->check ('table', $table)) {return;}
        if (!$this->check ('condition', $condition) {return;}
        $this->join[] = Array('table' => $table, 'condition' => $condition,
'joinType' => $joinType);
    }

    var $where = NULL;
    function where ($condition) {
        if ($this->where != NULL) {
            die('Where condition already set');
        }
        if (!$this->check('condition', $condition) {return;} //Error
        $this->where = $condition
    }

    var $having = NULL;
    function having ($condition) {
        if ($this->having != NULL) {
            die('Where condition already set');
        }
        if (!$this->check('condition', $condition) {return;} //Error
        $this->having = $condition
    }

    /**
    *   Adds one option to the SELECT syntax. ie DISTINCT
    *
    * @param string Select Option
    * @author Nuncanada
    * @access public
    */
    function addOption($option)
    {
        // Examples: DISTINCT, etc etc
        $this->areas['options'] = $option;
    }

    /**
    * Sets a Limit to the SELECT
    *
    * @param integer Number of rows
    * @param integer Starting Number
    * @author Nuncanada
    * @access public
    */
    function limit($nrows, $offset)
```

```
    {
        $this->areas['limit'] = array('nrows'=>$nrows,'offset'=>$offset);
    }

    /**
    * Check if an input is of the desired structure
    * If it is a string/int, then turn it into one object that represents the
desired structure.
    *
    * @param string SQL structure
    * @param mixed User input - String or object
    * @author Nuncanada
    * @access private
    */
    function check ($structure, &$input) {
        if (!is_object($input)) { //String or int

            //As this is a private function, this is just a reminder for myself
while prototyping/testing
            if ($structure == 'condition' || $structure == 'function') {
                die ('Conditional or Functional statements cannot be input as
strings.
                      They need their proper encapsulation'); // Error - Call
Exceptions
                return;
            } else {
                $input = &$this->create($structure, $input);
            }
        } elseif (get_class($structure) != 'sql_'.$structure) {
            return false;
        }

        return true;
    }

    /**
    * Creates the desired structure, return it as an object
    *
    * @param string $structure sql structure: 'alias', 'table', 'field',
'function'
    * @return mixed a newly created SQL structure object, or NULL
    * @author Chuck Hagenbuch
    * @access private
    */
    function &create ($structure, $input) {
        $classname = "sql_${structure}";

        if (!class_exists($classname)) {
            return;             //Call Exception
        }

        $obj =& new $classname ($input);

        return $obj;
    }

    /**
    * Get the strings present in a given place in the SQL structure
    * Will be used only by the DB drivers
    *
    * @param string Place in the SQL structure (from, join, where, groupby,
having, orderby)
    * @author Nuncanada
    * $access private
    */
    function get ($area)
    {
        return $this->areas[$area];
    }
}
```

```
class SQL_SELECT_MYSQL extends SQL_SELECT
{
    var $intro = Array ('select'  => 'SELECT ',
                        'from'    => ' FROM ',
                        'join'    => ' ',
                        'where'   => ' WHERE ',
                        'groupby' => ' GROUP BY ',
                        'having'  => ' HAVING ',
                        'orderby' => ' ORDER BY ',
                        'limit'   => ' LIMIT ');


    function buildQuery ()
    {
        $query = '';
/*  To be done
        foreach ($this->areas as $area => $array)
        {
            if (count($array)>0)
            {
                $string = '';

                switch ($area)
                {
                    case 'select':
                    case 'from':
                    case 'groupby':
                    case 'orderby':
                        $string = implode(', ', $array);
                        break;
                    case 'join':
                        $string = implode(' ', $array);
                        break;
                    case 'where':
                    case 'having':
                        $string = implode(' AND ', $array);
                        break;
                    default:
                        $string = 'Unknown Area in the query';
                        break;
                }
                $query .= $this->intro[$area] . $string;
            }
        }
*/
        return $query;
    }
}
```

9. Example Implementation Idea

9.1 Description

The following describes an implmentation of a subst of the query abstraction described in this document. This implementation does SELECTS, INSERTS, UPDATES and DELETES. Further extensions can bee added.

The exposee is divided into 2 parts: a short description of the API and a number of examples. Further xamples of how this code works can be found in the roles and ledger modules.

By way of a general description the query abstraction consists of a single class xarQuery, of which a given instance can contain and manage a single SQL statement. The class represents a standardized layer between the code and adodb. The different class methods can be used to:

- Accept whole SQL statements.or in parts in different imput syntax.
- Assemble and hold the SQL statement and its parts.
- Execute the statement.
- Produce the output.

Some of the benefits of this approach are:

- Assemble SQL statements dynamically, i.e. when needed.
- Reuse statements by modifying them.
- Reduce and standardize the code by incorporating some common features of queries into the class, e.g. number of rows sought for pager output, qstr() etc.

Input of tables and fields to the class can be as strings, lists of strings or arrays. This makes it possible to keep the syntax reasonably simple for simple queries.

The xarQuery class can coexist with the adodb syntax. For special cases or simple queries the adodb syntax may be a bit easier.

9.2 API

Example: Recalling a user's data from the database. The data of the user with the uname $this->uname is returned as an array.

```
$q = new xarQuery('SELECT',$this->rolestable);
$q->eq('xar_uname',$this->uname);
if (!$q->run()) return;
$foo = $q->output();
```

The syntax of the class call is

```
$q = new xarQuery('SELECT'|'INSERT'|'UPDATE'|'DELETE',
                  {array of query tables},
                  {array of fields}
);
```

The class methods (to be completed):

Tables and Fields:

- *addfields($fields):* add an array of fields to the query.
- *addtables($tables):* add an array of tables to thte query.
- *getfield($myfield):* return a field from the statement.
- *removefield($myfield):* remove a field from the statement.

- *join($field1,$field2):* join 2 tables via their respective fields..

Conditions:

- *getcondition($mycondition):* return a condition from the statement.
- *removecondition($mycondition):* remove a condition from the statement.
- *eq($field1,$field2):* add a EQ condition to the query.
- *gt($field1,$field2):* add a GT condition to the query.
- *ge($field1,$field2):* add a GE condition to the query.
- *le($field1,$field2):* add a LE condition to the query.
- *lt($field1,$field2):* add a LT condition to the query.
- *like($field1,$field2):* add a LIKE condition to the query.

Query output:

- *run($statement='', $flag=1):* assemble and execute an SQL statement. If a string with a statement is given it is directly executed. If $flag is 1 then the method output() contains the resultset. If $flag is 0 then the public variable $result contains the adodb resultset.
- *output():* return the query output as a 2 dimensional array where the keys of row elements corespond to the field names in the datatbase.
- *row($row=0):* return a single output row as a 1 dimensional array.

Database Connection:

- *open():* Open the default db connection.
- *close():* Close the default db connection.
- *openconnection($connction):* open a db connection.

Miscellaneous:

- *gettype():* return the typ of the query, e.g. SELECT.
- *getstartat():* return the record to start the query at.
- *getto():* return the records to end the query at.
- *getpagerows():* return the number of records to be displayed on a page.
- *getrows():* return the total number of reecords the query found
- *addorder($x = '', $y = 'ASC'):* add an ordering to the query results
- *setorder($x = '',$y = 'ASC'):* set the ordering of the query results
- *setstartat($x = 0):* set the record number to begin the query at.
- *setrowstodo($x = 0):* set the number of records to b returned.
- *uselimits():* turn on the limit filters, i.e. the query will start at a given record and return a certain number of records, depending on the setstartat() and setrowstodo() methods respectively.
- *nolimits():* turn off the limit filters, i.e the query will return all the records found.
- *setstatement($x = ''):* create a string representation of the SQL query. If $x is empty the query statement is assembled from the data in the query object.
- *getstatement():* return a string representation of the SQL query.

9.3 Examples

The following are equivalent:

```
        $dbconn = xarDBGetConn();
        $query = "SELECT * FROM $this->rolestable WHERE xar_name = '$this->name'";
        if(!$dbconn->Execute($query)) return;

        AND

        $q = new xarQuery('SELECT',$this->rolestable);
        $q->eq('xar_name',$this->name);
        if (!$q->run()) return;

        AND

        $q = new xarQuery();
        if (!$q->run("SELECT * FROM $this->rolestable WHERE xar_name =
'$this->name'")) return;
```

The following are equivalent:

```
        if (!$q->run($query)) return;
        $invoices = array();
        foreach ($q->output() as $out)
            $invoices[] = array_pop($out);
        return $invoices;

        AND

        $result = $dbconn->Execute($query);
        if (!$result) return;
        $invoices = array();
        while(!$result->EOF) {
            list($this->id) = $result->fields;
            $invoices[] = $this->id;
            $result->MoveNext();
        }
        return $invoices;
```

Check if a role already exists:

```
        Old code:

        $this->dbconn = xarDBGetConn();
        // Confirm that this group or user does not already exist
        if ($this->type == 1) {
            $query = "SELECT COUNT(*) FROM $this->rolestable
                  WHERE xar_name = '$this->name'";
        } else {
            $query = "SELECT COUNT(*) FROM $this->rolestable
                  WHERE xar_uname = '$this->uname'";
        }

        $result = $this->dbconn->Execute($query);
        if (!$result) return;

        list($count) = $result->fields;

        if ($count == 1) {//error message}


        New code: Note only the WHERE clause changes dpending on the role type

        // Confirm that this group or user does not already exist
        $q = new xarQuery('SELECT',$this->rolestable);
        if ($this->type == 1) {
            $q->eq('xar_name',$this->name);
        } else {
            $q->eq('xar_uname',$this->uname);
        }

        if (!$q->run()) return;

        if ($q->getrows() == 1) {//error message}
```

Adding a group or user to the roles table

```
        Old code:

        $this->dbconn = xarDBGetConn();
        $nextId = $this->dbconn->genID($this->rolestable);
        $createdate = mktime();

        if ($this->type == 1) {
            $query = "INSERT INTO $this->rolestable
                        (xar_uid, xar_name, xar_type, xar_uname, xar_valcode,
xar_date_reg)
                      VALUES (?, ?, ?, ?, ?, ?)";
            $bindvars = array(
                $nextId,
                $this->name,
                $this->type,
                $this->uname,
                $this->val_code,
                $createdate,
            );
        } else {
            $query = "INSERT INTO $this->rolestable
                        (xar_uid, xar_name, xar_type, xar_uname, xar_email, xar_pass,
                        xar_date_reg, xar_state, xar_valcode, xar_auth_module)
                      VALUES (?,?,?,?,?,?,?,?,?,?)";
            $bindvars = array(
                $nextId,
                $this->name,
                $this->type,
                $this->uname,
                $this->email,
                md5($this->pass),
                $createdate,
                $this->state,
                $this->val_code,
                $this->auth_module,
            );
        }
        // Execute the query, bail if an exception was thrown
        if (!$this->dbconn->Execute($query, $bindvars)) return;

        New code: Note the query is assembled incrementally

        $nextId = $this->dbconn->genID($this->rolestable);

        $tablefields = array(
            array('name' => 'xar_uid',      'value' => $nextId),
            array('name' => 'xar_name',     'value' => $this->name),
            array('name' => 'xar_uname',    'value' => $this->uname),
            array('name' => 'xar_date_reg', 'value' => mktime()),
            array('name' => 'xar_valcode',  'value' => $this->val_code)
        );
        $q = new xarQuery('INSERT',$this->rolestable);
        $q->addfields($tablefields);
        if ($this->type == 1) {
            $groupfields = array(
                array('name' => 'xar_type', 'value' => 1)
            );
            $q->addfields($groupfields);
        } else {
            $userfields = array(
                array('name' => 'xar_type',       'value' => 0),
                array('name' => 'xar_email',      'value' => $this->email),
                array('name' => 'xar_pass',       'value' => md5($this->pass)),
                array('name' => 'xar_state',      'value' => $this->state),
                array('name' => 'xar_auth_module','value' => $this->auth_module)
            );
            $q->addfields($userfields);
        }
        // Execute the query, bail if an exception was thrown
        if (!$q->run()) return;
```

10 Relevant References

[1] Jurish, B, "relational query feature structures", <http://www.ling.uni-potsdam.de/~moocow/projects/diplom/html-plain/node1.html>, 2001.

[2] ObjectWave, "ObjectWave JGrinder", <http://www.objectwave.com/pdf/tools/JGrinder.pdf>, 2002.

[3] Chu, W., Chen, Q., and M. Merzbacher, "CoBase: A Cooperative Database System", <http://cobase-www.cs.ucla.edu/tech-docs/cqa.ps>, 2002.

[4] Cohen, S, Kanza, Y, Kogan, Y, Werner, W, Serebrenik, A, and Y Sagiv, "EquiX--A Search and Query Language for XML", <http://arxiv.org/abs/cs.db/0110044>, 2003.

[5] Wang, D and J Xie, "An Approach Towards Web Caching and Prefetching for Database Management Systems", <http://www.cs.duke.edu/~junyi/cps216/report.pdf>, 2003.

[6] National Lab. of Software Development Enviroment, Beihang University, "A Script Language for Data Integration in Database", <http://arxiv.org/ftp/cs/papers/0301/0301009.pdf>, 2003.

[7] PHP Devs, "Relevant discussion about query abstraction", <http://marc.theaimsgroup.com/?l=php-dev&m=101753543913458&w=2>, 2002.

[8] PHP Devs, "Relevant discussion about query abstraction", <http://marc.theaimsgroup.com/?l=php-dev&m=101752997709709&w=2>, 2002.

[9] PHP Devs, "Relevant discussion about query abstraction", <http://marc.theaimsgroup.com/?l=php-dev&m=101740330610221&w=2>, 2002.

[10] PHP Devs, "PHP dev stating that a SQL parser should be the solution", <http://marc.theaimsgroup.com/?l=php-db&m=100635358202596&w=2>, 2002.

[11] "PEAR Package Data Object (for query abstraction)", <http://pear.php.net/package-info.php?pacid=80&release=0.13&PHPSESSID=18e4b1d53db182aceeeb2c1e0b92e399>, 2002.

[12] "SubSelect Emulation for Mysql", <http://px.sklar.com/code.html?id=239>, 2002.

[13] "Article showing difference between db queries", <http://www.oreilly.com/news/sqlnut_1200.html>, 2002.

[14] "DAO Pattern implentation in Java", <http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-dao-p2.html>, 2002.

[15] "Discussion about Query Abstraction in another CMS", <http://ccm.redhat.com/bboard-archive/acs_design/000818.html>, 2002.

[16] "Ideas on how to abstract SQL queries from 1997", <http://www.joeyoder.com/papers/patterns/Reports/>, 2002.

[17] "Apache Object Relational Mapper", <http://db.apache.org/ojb/>, 2002.

[18] "Thesis on Object Relational Mapping", <http://www.lap.ttu.ee/erki/failid/konspekt/bakalaureusetoo/thesis.pdf>, 2002.

Author's Address

Flavio Borges Botelho
Xaraya Development Group
EMail: nuncanada@xaraya.com
URI: http://www.xaraya.com

A. Example appendix

Any section which is present after the references will become an appendix

Intellectual Property Statement

The DDF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the DDF's procedures with respect to rights in standards-track and standards-related documentation can be found in RFC-0.

The DDF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the DDF Board of Directors.

Acknowledgement

Funding for the RFC Editor function is provided by the DDF