Rapid Application Development toolkit for building Administrative Web Applications

The Model-View-Controller (MVC) Design Pattern for PHP

By Tony Marston

2nd May 2004
Amended 15th April 2014

As of 10th April 2006 the software discussed in this article can be downloaded from www.radicore.org

Introduction

I am no stranger to software development having been a software engineer for over 25 years. I have developed in a variety of 2nd, 3rd and 4th generation languages on a mixture of mainframes, mini- and micro-computers. I have worked with flat files, indexed files, hierarchical databases, network databases and relational databases. The user interfaces have included punched card, paper tape, teletype, block mode, CHUI, GUI and web. I have written code which has been procedural, model-driven, event-driven, component-based and object oriented. I have built software using the 1-tier, 2-tier and the 3-tier architecture. I have created development infrastructures in 3 different languages. My latest achievement is to create an environment for building web applications using PHP that encompasses a mixture of 3 tier architecture, OOP, and where all HTML output is generated using XML and XSL transformations. This is documented in A Development Infrastructure for PHP.

Before teaching myself PHP the only occasion where I came in contact with the concept of the model-view-controller design pattern was when I joined a team that was replacing a legacy system with a more up-to-date version that had web capabilities. I was recruited because of my experience with the language being used, but I quickly realised that their design was too clumsy and the time taken to develop individual components was far to long (would you believe two man-weeks for a SEARCH screen and a LIST screen?). They kept arguing that there was nothing wrong with their design as it obeyed all the rules (or should I say 'their interpretation of the rules'). I was not the only one who thought their implementation was grossly inefficient - the client did not like their projected timescales and costs so he cancelled the whole project. How's that for a vote of confidence for their design!

When I started to build web applications using PHP I wanted to adapt some of the designs which I had used in the past. Having successfully implemented the 3-tier architecture in my previous language I wanted to attempt the same thing using PHP. My development infrastructure ended up with the following sets of components:

- Presentation layer - a set of component scripts, screen structure scripts, and a series of reusable XSL stylesheets and controller scripts.
- Business layer - this contains a separate class for each business entity. As each business entity also has a database table, and as the code to communicate with a database table is virtually the same regardless of the business entity, I put all the sharable code into an abstract class and defined each business entity as a subclass. This enables me to share all the standard properties and methods of the superclass through inheritance. This I have documented in Using PHP Objects to access your Database Tables (Part 1) and Using PHP Objects to access your Database Tables (Part 2).
- Data Access layer - a single component which contains all the DML (Data Manipulation Language) statements for a particular RDBMS engine. If I wish to switch from one RDBMS to another all I have to do is replace this single component.

Although I have never been trained in OO techniques I read up on the theory and used the OO capabilities of PHP 4 to produce a simple class hierarchy that had as much reusable code as possible in a superclass and where each business entity was catered for with a subclass. I published an article Using PHP Objects to access your Database Tables (Part 1) and (Part 2) which described what I had done, and I was immediately attacked by self-styled OO purists who described my approach as "totally wrong" and "unacceptable to REAL object-oriented programmers". I put the question to the PHP newsgroup and asked the opinion of the greater PHP community. This generated a huge response that seemed to be split between the "it is bad" and "it is good" camps, so I summarised all the criticisms, and my responses to those criticisms, in a follow-up article entitled What is/is not considered to be good OO programming.

I have absolutely no doubt that there will be some self-styled MVC purists out there in Internet land who will heavily criticise the contents of this document, but let me give you my response in advance:

I DO NOT CARE!

I have read the principles of MVC and built software which follows those principles, just as I read the principles of OOP and built software which followed those principles. The fact that my implementation is different from your implementation is totally irrelevant - it works therefore it cannot be wrong. I have seen implementations of several design patterns which were technical disasters, and I have seen other implementations (mostly my own) of the same design patterns which were technical successes. Following a set of principles will not guarantee success, it is how you implement those principles that separates the men from the boys. The principles of the various design patterns are high-level ideas which are language-independent, therefore they need to be translated into workable code for each individual language. This is where I have succeeded and others have failed. I have spent the last 25+ years designing and writing code which works, and I don't waste my time with ideas that don't work. Other people seem to think that following a set of rules with blind obedience is the prime consideration and have no idea about writing usable or even efficient code. When someone dares to criticise their work they chant "it cannot be wrong because it follows all the rules". The difference between successful and unsuccessful software is not in the underlying rules or principles, it is how those rules or principles are implemented that counts. Where implementation 'A' makes it is easier and quicker to develop components than implementation 'B', it does not mean that 'A' is right and 'B' is wrong, it just means that 'A' is better than 'B'. Where an implementation works it cannot be wrong - it can only be wrong when it does not work.

The Principles of the MVC Design Pattern

After researching various articles on the internet I came up with the following descriptions of the principles of the Model-View-Controller design pattern:

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:

Input --> Processing --> Output
Controller --> Model --> View

Model

- A model is an object representing data or even activity, e.g. a database table or even some plant-floor production-machine process.
- The model manages the behavior and data of the application domain, responds to requests for information about its state

and responds to instructions to change state.
- The model represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.
- The model is the piece that represents the state and low-level behavior of the component. It manages the state and conducts all transformations on that state. The model has no specific knowledge of either its controllers or its views. The view is the piece that manages the visual display of the state represented by the model. A model can have more than one view.

Note that the model may not necessarily have a persistent data store (database), but if it does it may access it through a separate Data Access Object (DAO).

View

- A view is some form of visualisation of the state of the model.
- The view manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. Instead of a bitmapped display the view may generate HTML or PDF output.
- The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented.
- The view is responsible for mapping graphics onto a device. A view typically has a one to one correspondence with a display surface and knows how to render to it. A view attaches to a model and renders its contents to the display surface.

Controller

- A controller offers facilities to change the state of the model. The controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- A controller is the means by which the user interacts with the application. A controller accepts input from the user and instructs the model and view to perform actions based on that input. In effect, the controller is responsible for mapping end-user action to application response.
- The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application they appear as HTTP GET and POST requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.
- The controller is the piece that manages user interaction with the model. It provides the mechanism by which changes are made to the state of the model.

In the Java language the MVC Design Pattern is described as having the following components:

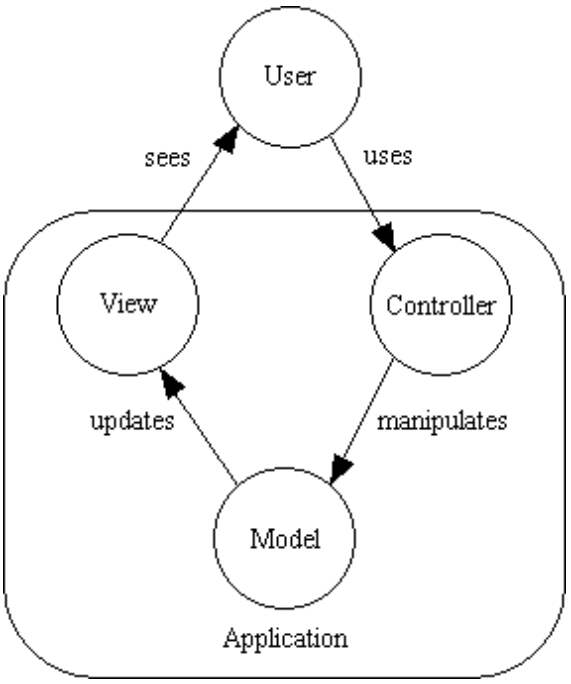
- An application model with its data representation and business logic.
- Views that provide data presentation and user input.
- A controller to dispatch requests and control flow.

The purpose of the MVC pattern is to separate the model from the view so that changes to the view can be implemented, or even additional views created, without having to refactor the model.

How they fit together

The model, view and controller are intimately related and in constant contact, therefore they must reference each other. The picture below illustrates the basic Model-View-Controller relationship:

Figure 1 - The basic MVC relationship



Even though the above diagram is incredibly simple, there are some people who try to read more into it than is really there, and they attempt to enforce their interpretations as "rules" which define what is "proper" MVC. To put it as simply as possible the MVC pattern requires the following:

- It must have a minimum of three components, each of which performs the responsibilities of either the Model, the View or the Controller, as identified above. You may include as many additional components as you like, such as a Data Access Object (DAO) to communicate with a relational database.
- There is a flow of data between each of those components so that each may carry out its designated responsibilities on that data. What is not specified in MVC is how the mechanics of that flow are to be implemented. The data may be pushed by the Model into the View, it may be pulled by the View from the Model, or pulled into an intermediary (such as the Controller or an Observer) before it is pushed into the View. It does not matter how the data flows, just that it does flow. The actual mechanics are an implementation detail and can therefore be left to the implementor.
- The MVC pattern does not identify from where each of these components is instantiated and called. Is there a mystical 4th component which oversees the 3 others, or can one of them oversee and direct the other two? As this is not directly specified in MVC the actual mechanics can be treated as a separate implementation detail.

Note the following:

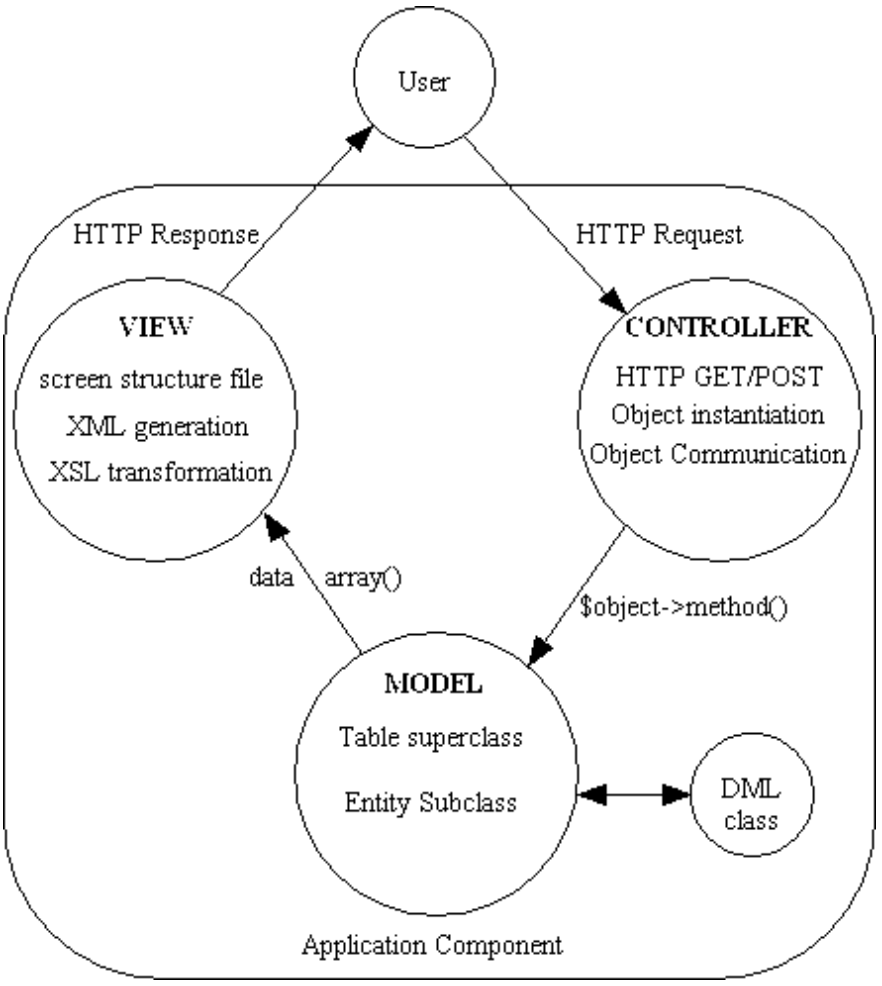
- HTML or PDF output can only be generated and output in one place - the View.
- Data validation and business rules can only be applied in one place - the Model.
- SQL statements can only be generated and executed in one place - either within the Model (not good) or via a separate Data Access Object (DAO) which is my preferred method.

There are some people who criticise my interpretation of the "rules" of MVC as it is different from theirs, and therefore wrong (such as how data moves from the Model to the View), but who is to say that their interpretation is the only one that should be allowed to exist? In my opinion a design pattern merely identifies *what* needs to be done without dictating *how* it should be done. Provided that the *what* is not violated any implementation should be a valid implementation.

My Implementation

In my implementation, as shown in Figure 2 and Figure 3, the whole application is broken down into a series of top-level components which are sometimes referred to as tasks, actions, functions, operations or transactions (that's *user* transactions, not *database* transactions), each of which is may be related to a Use Case. Each transaction component references a single controller, one or more models, and usually a single view. Some components do not have a view as they are called from other components in order to perform a service, and once this service has been completed they return control to the calling component. Each component is self-executing in that it deals with both the HTTP GET and POST requests.

Figure 2 - My implementation of the MVC pattern (1)



- The **Model** is implemented as a series of business entity classes each of which contains all the properties and methods required by a single business entity. It is actually a subclass to an abstract superclass which contains properties and methods which are common to all database tables. The model is responsible for all data validation, business rules and task-specific behaviour, while the actual generation of DML (Data Manipulation Language) statements is performed in a separate DML class.

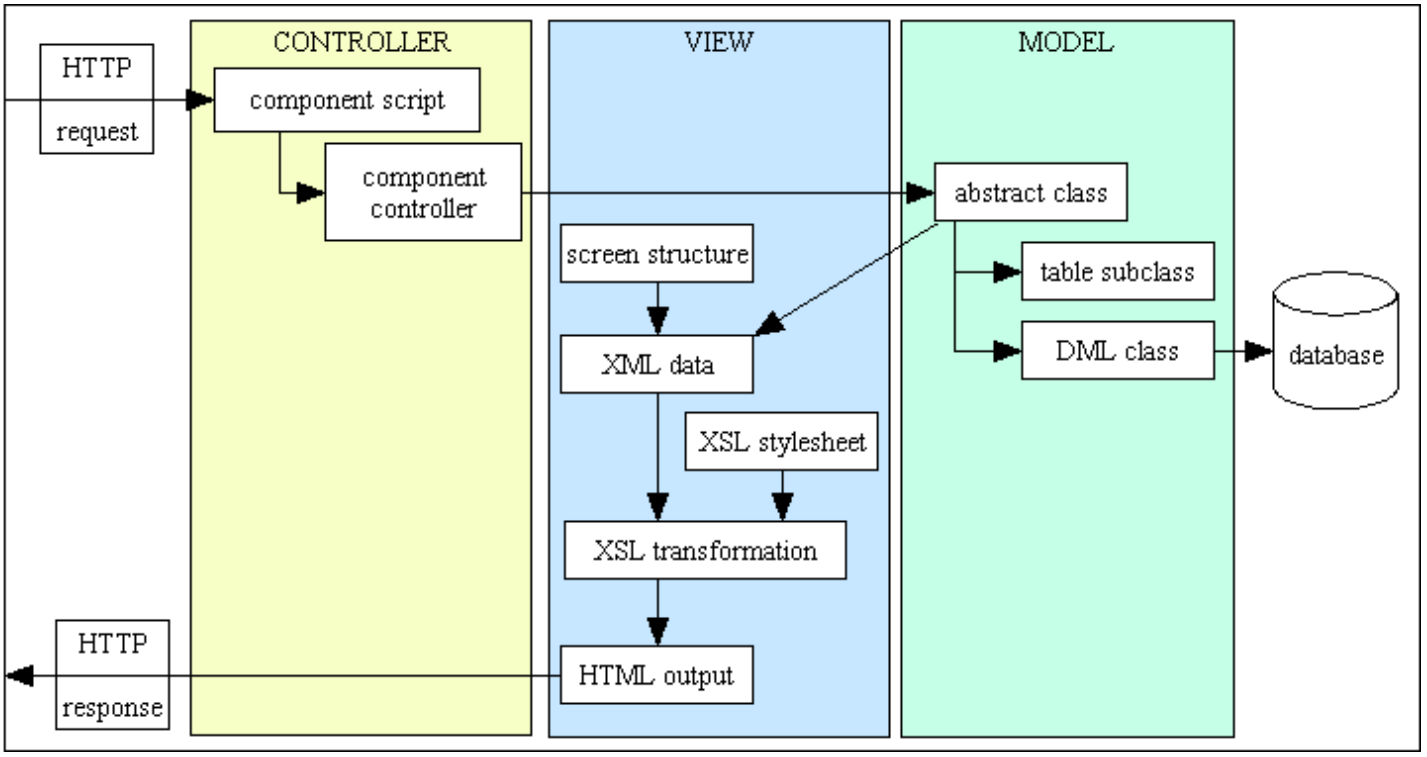
- The **DML** class or Data Access Object (DAO) is the only object in the entire framework which is allowed to communicate with the database, and it can only be called by a **model** component. This isolates the model from the underlying database and allows the application to be switched from one RDBMS to another simply by switching to another DML class. Only the DML class deals with database resources - all communication with the entity subclass is by standard database-agnostic arrays.

- The **View** is implemented as a series of screen structure scripts which are combined with the output from each database table class to produce an XML document. This file will also include the data for the menu bar, navigation bar, action bar, pagination and scrolling areas. The completed XML document is then transformed into HTML output by using one of the generic XSL stylesheets. For each database table there is typically a list view (containing multiple occurrences with data arranged horizontally) and a detail view (containing a single occurrence with data arranged vertically).

  If output in an alternative format is required, such as PDF instead of HTML, there are separate transaction patterns which use a different view object to create PDF output in either LIST view or DETAIL view, each with its own report structure file. There is also another view object to create CSV output.

- The **Controller** is implemented as a series of component scripts which link to one of a series of transaction controller scripts. Unlike some implementations which require a separate controller for each transaction, each of my controllers is for a class (or type) of transaction, so the same controller script can be shared by all transactions of the same class. Each of these scripts deals with the following:
  - It handles the HTTP GET and POST requests.
  - It instantiates an object for each business entity identified by the component script.
  - It calls methods on those objects as appropriate. Most will perform some sort of communication with the database although some may not. Data is passed into and out of these objects as standard PHP arrays. In this way an object can deal with any number of database occurrences both as input or output.
  - It calls the relevant **view** object to create output in either HTML, PDF or CSV.

Figure 3 - My implementation of the MVC pattern (2)



Note that all the items in the above diagram are clickable links.

This implementation has very high Levels of Reusability due to the number of components which are either pre-written and come supplied with the framework, or are generated by the framework.

Controller Component

This component is comprised of the following:

- A separate and unique Component script for each user transaction. Each of these is generated from the Data Dictionary.
- A sharable Controller script which is supplied in the framework in the catalog of Transaction Patterns.

View Component

The Model is never supposed to handle the output of its data to any particular device or medium. Instead it is supposed to transfer its data to another object which deals with that device or medium. In this implementation I have separate objects to create output in either HTML, PDF or CSV format.

The HTML component operates as follows:

- It is given a Screen Structure file which is generated from the Data Dictionary initially, but which may be modified as required. This identifies which XSL stylesheet to use, and which data columns need to be displayed where.
- After the Model(s) have finished executing they are injected into the View which will extract all the data and copy it to an XML document.
- Information from the screen structure file is also copied into the XML document so that the XSL stylesheet will know which data elements to put where.
- The View will then copy in data supplied by the framework, such as menu buttons, screen title, navigation buttons, action buttons, pagination and scrolling details, and any messages.
- The View will then perform an XSL Transformation using the nominated XSL stylesheet and the newly constructed XML document to produce an HTML document.
- The HTML output is then sent back to the requesting client device.

The PDF component operates as follows:

- The Controller instantiates the Model and instructs it to issue its SQL select statement, but without actually retrieving any data.
- The Controller instantiates the PDF object and injects into it the MOdel and the name of the report structure file.
- The PDF object instructs the Model to fetch a record which it will format into a page for the DETAIL view or a line in the LIST view.
- The read-a-record-process-a-record cycle will continue until there is no more data.

The CSV component operates as follows:

- The Controller instantiates the Model and instructs it to issue its SQL select statement, but without actually retrieving any data.

- The Controller instantiates the CSV object injects the Model into it.
- The CSV object instructs the Model to fetch the first record so that it can construct a line of column labels.
- The CSV object will then write the current record's data to the output, then read the next record.
- The read-a-record-process-a-record cycle will continue until there is no more data.

Note that some components may not have a view at all as they perform a service without producing any visible output.

Model Component

This component is comprised of the following:

- A separate Business Entity class for each table within the application database. Each of these is generated by exporting the table's details from the Data Dictionary to produce a class file and a structure file. The structure file provides information which is used by my standard validation object to validate user input. This file can be regenerated whenever the table's structure changes.
- Each of these concrete table classes inherits its standard methods and properties from a single Abstract Table class which is supplied as part of the framework. This ensures that all the common code is defined in one place but can be shared by many objects.
- Each table class contains its own specific business rules and data validation rules, but all communication with the physical database is routed through a separate Data Access Object (DAO). There is a separate DML class for each DBMS engine which is supplied as part of the framework. This arrangement makes it possible to switch from one DBMS to another without having to make any changes to any table class. Currently supported DBMS engines are MySQL, PostgreSQL, Oracle and SQL Server.

Characteristics

My implementation also has the following characteristics:

Each of the components can be classified as one of the following:

- A **framework** component - the **Controller**, the **View** and the DAO.
- An **application** component - the **Model**, the component scripts and the screen structure files.

Some of these components are built into the framework while others you have to generate yourself:

- The **framework** components are pre-written and come supplied with the framework or are invoked automatically.
- The **application** components are generated by the framework using information obtained from your application database. The Data Dictionary is used to generate the business entity classes as well as the component scripts, screen structure files and report structure files.

Components are either application-agnostic or framework-agnostic:

- The **framework** components are application-agnostic as they have no knowledge of the internals of any application component. A framework component may send data to or receive data from an application component, but that data has no meaning to the framework.
- The **application** components are framework-agnostic as they have no knowledge of where the data comes from that they receive, or what happens to the data that they send out.

Component script

Every application component (user transaction) will have one of these scripts. It is a very small script which does nothing but identify which model (business entity), view (screen structure file) and controller to use, as shown in the following example:

```
<?php
//*****************************************************************************
// This will allow an occurrences of a database table to be displayed in detail.
// The identity of the selected occurrence is passed down from the previous screen.
//*****************************************************************************

$table_id = "mnu_user";                    // identifies the model
$screen   = 'mnu_user.detail.screen.inc';  // identifies the view

require 'std.enquire1.inc';                // activates the controller

?>
```

Where the controller requires access to more than one business entity, such as in a parent-child or one-to-many relationship in a LIST2 pattern, or a parent-child-grandchild relationship in a LIST3 pattern, then names such as `$parent_id`, `$child_id`, `$outer_id` or `$inner_id` are used.

The same screen structure file can used by components which access the same business entity but in different modes (insert, update, enquire, delete and search) as the XSL stylesheet is provided with a `$mode` parameter which enables it to determine whether fields should be read-only or amendable.

Note that no component script is allowed to access more than one view. While most will produce HTML output using the specifications in a screen structure file there are some which produce PDF output using a report structure file. Some components may have no view at all - they are called upon to take some particular action after which they return control to the calling component which refreshes its own view accordingly.

Each of these is generated from a component in the Data Dictionary.

There are more details available in the RADICORE Infrastructure guide.

Controller script

There is a separate pre-built controller script included in the framework for each of the patterns identified in Transaction Patterns for Web Applications. By simply changing the name of the controller script the whole character of the component will change.

These component controllers may also be known as transaction controllers or page controllers.

```
<?php
// name = std.enquire1.inc

// type = enquire1

// This will display a single selected database occurrence using $where
// (as supplied from the previous screen)

require 'include.inc';

// identify mode for xsl file
$mode = 'enquire';

// define action buttons
$act_buttons['quit'] = 'QUIT';

// initialise session
initSession();

// look for a button being pressed
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
   if (isset($_POST['quit']) or (isset($_POST['quit_x']))) {
      // quit this screen, return to previous screen
      scriptPrevious();
   } // if
} // if

// retrieve profile must have been set by previous screen
if (empty($where)) {
   scriptPrevious('Nothing has been selected yet.');
} // if

// create a class instance for the main database table
require "$table_id.class.inc";
$dbobject = new $table_id;

// check that primary key is complete
$dbobject->checkPrimaryKey = TRUE;

// get data from the database
$fieldarray = $dbobject->getData($where);

if ($dbobject->getErrors()) {
   // some sort of error - return to previous script
   scriptPrevious($dbobject->getErrors());
} // if

// check number of rows returned
if ($dbobject->getNumRows() < 1) {
   scriptPrevious('Nothing retrieved from the database.');
} // if
if ($dbobject->getNumRows() > 1) {
   $message[] = 'Multiple rows selected - only the 1st accepted';
} // if

// get any extra data and merge with $fieldarray
$fieldarray = $dbobject->getExtraData($fieldarray);

// build list of objects for output to XML data
$xml_objects[]['root'] = &$dbobject;

// build XML document and perform XSL transformation
$view = new radicore_view($screen_structure);
$view->buildXML($xml_objects, $errors, $message);
exit;

?>
```

The controller does not know the names of the business entities with which it is dealing - these are passed down as variables from the component script. It will append the standard `'.class.inc'` to each table name to identify the class file, then instantiate a separate object for each table. Note that I did say "entities" in the plural and not "entity" in the singular. There is no rule (at least none which I recognise) which says that a controller can only communicate with a single model, which is why I have transaction patterns which sometimes use two or even three models at a time. Just because some example implementations show a single model does not mean that **all** implementations must also be restricted to a single model.

The method names that the controller uses to communicate with each table object are the generic names which are defined within the abstract table class. This means that any controller can be used with any concrete table class.

The controller does not know the names of any fields with which it is dealing - what comes out of the business entity is a simple associative array of 'name=value' pairs which is passed untouched to the View object which will transfer the entire array to an XML document so that it can be transformed into HTML.

In any INPUT or UPDATE components the entire contents of the POST array is passed to the business entity as-is instead of one field at a time. Again this means that no field names have to be hard-coded into any controller script, thus increasing their re-usability.

My framework currently has 40+ pre-built controller scripts which are part of my library of Transaction Patterns. The same pattern can be used with virtually any business object in the application, thus making them loosely coupled and highly reusable.

There are more details available in the RADICORE Infrastructure guide.

Abstract Table class

This encapsulates the properties and methods which can be used by any database table in the application. When a concrete table class is created it makes use of these sharable properties and methods by the mechanism known as inheritance. The generic methods which are used by the Controller script when it calls a Business entity object are all defined with this abstract class, and as they are all then available to every concrete table class this enables the concept known as polymorphism.

There are more details available in the RADICORE Infrastructure guide.

DML class

This is responsible for all communication the database by constructing queries using the standard SQL Data Manipulation Language (DML) using instructions passed down from the calling Business Entity class. There is a separate DML class for each different DBMS engine, such as MySQL, PostgreSQL, Oracle and SQL Server, which means that the entire application can be switched from one DBMS to another simply by changing a single entry in the configuration file. This means that it is also possible to develop an application using one DBMS engine but deploy it to use another.

Note that this class does not have any inbuilt knowledge of any database or database table, so a single class can be used to access any table in any database.

There are more details available in the RADICORE Infrastructure guide.

Business Entity class

Each business entity is implemented as a separate class so that it can encapsulate all the properties and methods of that entity in a single object. As each of these business entities also exists as a database table a great deal of common code can be shared by inheriting from the abstract table class.

In its initial form a concrete table class requires very little information to differentiate it from another database table - just the database name, the table name, and the table structure. Rather than having to create each class file manually, as of June 2005 these can be generated for you as described in A Data Dictionary for PHP Applications. This provides the following facilities:

- IMPORT - Populate the dictionary database using details extracted from the physical database schema.
- EDIT - Modify the details for use by the application.
- EXPORT - Make the details available to the application in the form of disk files which can be referenced by means of the include() function. Two files will be created for each database table:
  - <tablename>.class.inc - this is the initial class file for the database table. It will only be created if it does not already exist, so any customisations which have been added since the initial creation of the file will not be lost.
  - <tablename>.dict.inc - this contains the column details, key details and relationship details. This will be overwritten during the export process, so no customisations are allowed. All customisation should be performed in the dictionary and then exported to this file. Alternatively it is possible to make temporary customisations at runtime by placing code in the _cm_changeConfig method of the relevant table subclass.

The variables in the class constructor do not contain application data (that which passes through the object between the user and the database) but instead contain information *about* the application data. Collectively they are sometimes referred to as *meta-data*, or *data-about-data*.

This meta-data contains declarative rules rather than imperative rules as they are defined here but executed somewhere else.

When the time comes (i.e. when the user presses a SUBMIT button on an HTML form) the user data passes from the Controller into the Model where it is passed to a validation object along with the meta-data, and it is the responsibility of the validation object to ensure that the user data conforms to those rules. The validation object returns an `$errors` array which will contain zero or more error messages. If there are any error messages then the operation is aborted.

Screen Structure file

The structure file is a simple PHP script which identifies the XSL stylesheet which is to be used and the application data which is to be displayed by that stylesheet. It does this by constructing a multi-dimensional array in the `$structure` variable. The following formats are available:

- A simple 2-column DETAIL view (vertical) - one record per page.
- A more complex multi-column DETAIL view (vertical) - one record per page.
- A standard LIST view (horizontal) - multiple records per page.

Each structure file has the following components:

xsl_file	This identifies which XSL stylesheet to use. Note that the same stylesheet can be referenced in many different structure files.
tables	This associates the name of a zone within the XSL stylesheet with the name of some user data within the XML document. A stylesheet may contain several zones, and the XML document may contain data from several database tables, so it is important to identify which table data goes into which zone. Among the different zone names which you may see are `main`, `inner`, `outer`, `middle` and `link`.

zone columns	This allows you to specify attribute values for each table column in this zone.
	
	The following keywords for column attributes are supported in the list/horizontal view:
	
	• class - see HTML specification.
	• width - see HTML specification.
	• align (left|center|right|justify) - see HTML specification.
	• valign (top|middle|bottom) - see HTML specification.
	• char - see HTML specification.
	• style - see HTML specification.
	• size - see additional attributes.
	• imagewidth - see additional attributes.
	• imageheight - see additional attributes.
	• nosort - stops the column heading from being a hyperlink for sorting.
	
	Note that the 'align', 'valign' and 'class' attributes when added to a <COL> element in the HTML output are poorly supported in most browsers, so will be dealt with as follows:
	
	• The 'align' and 'valign' attributes will be converted to 'class' when written to the XML output.
	• The 'class' attribute will no longer be added to the <COL> element, but instead will be added to the <TD> element for every cell within the relevant column.
	
	NOTE: if your screen structure file contains two or more of the 'align|valign|class' attributes for the same field then they will be combined automatically into a single 'class' value, so code such as:
	
	`$structure['main']['columns'][] = array('width' => '100', 'align' => 'right', 'valign' => 'middle', 'class' => 'foobar');`
	
	will be treated as if it were:
	
	`$structure['main']['columns'][] = array('width' => '100', 'class' => 'right middle foobar');`

zone fields	This identifies which fields from the database table are to be displayed, and in which order. For a horizontal view (as in Figure 7) this identifies the columns going across the page. For a vertical view (see Figure 4) this identifies the rows going down the page. Note that each element within this array is indexed by a column number (horizontal view) or a row number (vertical view). The following keywords for additional field attributes are supported in the detail/vertical view: • colspan - allow the field to span more than 1 column. • rowspan - allow the field to span more than 1 row. • cols - will override the value for **multiline** controls. • rows - will override the value for **multiline** controls. • class - specifies one or more CSS classes. • align - specifies horizontal alignment (left/center/right). • valign - specifies vertical alignment (top/middle/bottom).

	• display-empty - will force a line of labels to be displayed even if all values are missing. • imagewidth - will override an image's default width. • imageheight - will override an image's default height. • javascript - includes optional javascript.

Screen Structure file for a simple 2-column DETAIL view (vertical)

This is an example of a structure file to produce a standard 2-column DETAIL view as shown in Figure 4.

```
<?php
$structure['xsl_file'] = 'std.detail1.xsl';

$structure['tables']['main'] = 'person';

$structure['main']['columns'][] = array('width' => 150);
$structure['main']['columns'][] = array('width' => '*');

$structure['main']['fields'][] = array('person_id' => 'ID');
$structure['main']['fields'][] = array('first_name' => 'First Name');
$structure['main']['fields'][] = array('last_name' => 'Last Name');
$structure['main']['fields'][] = array('initials' => 'Initials');
$structure['main']['fields'][] = array('nat_ins_no' => 'Nat. Ins. No.');
$structure['main']['fields'][] = array('pers_type_id' => 'Person Type');
$structure['main']['fields'][] = array('star_sign' => 'Star Sign');
$structure['main']['fields'][] = array('email_addr' => 'E-mail');
$structure['main']['fields'][] = array('value1' => 'Value 1');
$structure['main']['fields'][] = array('value2' => 'Value 2');
$structure['main']['fields'][] = array('start_date' => 'Start Date');
$structure['main']['fields'][] = array('end_date' => 'End Date');
$structure['main']['fields'][] = array('selected' => 'Selected');
?>
```

Where each row contains only a single label followed by a single field the format of each entry is `'field' => 'label'`.

Note that under some circumstances neither the field nor the label will be output, in which case the entire row will be dropped rather than showing an empty row. These circumstances are:

- The fieldname does not exist in the XML document.
- The field has the `nodisplay` attribute set in the $fieldspec array.
- This behaviour can be overridden by adding `'display-empty' => 'y'` to the item in the `['zone']['fields']` array in the screen structure file. This will cause every cell in that row to be output, even if it is empty.

When this information is written out to the XML document it will look something like the following:

```
<structure>
  <main id="person">
    <columns>
      <column width="25%"/>
      <column width="*"/>
    </columns>
    <row>
      <cell label="ID"/>
      <cell field="person_id" />
    </row>
    <row>
      <cell label="First Name"/>
      <cell field="first_name"/>
    </row>
    <row>
      <cell label="Last Name"/>
      <cell field="last_name"/>
    </row>
    <row>
      <cell label="Initials"/>
      <cell field="initials"/>
    </row>

    ....

    <row>
      <cell label="Start Date"/>
      <cell field="start_date"/>
    </row>
    <row>
      <cell label="End Date"/>
      <cell field="end_date"/>
    </row>
  </main>
</structure>
```

Screen Structure file for a more complex multi-column DETAIL view (vertical)

Here is an example of a screen structure file that will produce a multi-column DETAIL view as shown in Figure 5:

```
$structure['main']['fields'][1] = array('person_id' => 'ID',
                                        'colspan' => 5);

$structure['main']['fields'][2][] = array('label' => 'First Name');
$structure['main']['fields'][2][] = array('field' => 'first_name',
```

```
                                        'size' => 15);
$structure['main']['fields'][2][] = array('label' => 'Last Name');
$structure['main']['fields'][2][] = array('field' => 'last_name',
                                        'size' => 15);
$structure['main']['fields'][2][] = array('label' => 'Initials');
$structure['main']['fields'][2][] = array('field' => 'initials');

$structure['main']['fields'][4] = array('picture' => 'Picture',
                                      'colspan' => 5,
                                      'imagewidth' => 32,
                                      'imageheight' => 32);
....
$structure['main']['fields'][11] = array('value2' => 'Value 2',
                                       'colspan' => 5);

$structure['main']['fields'][12][] = array('label' => 'Start Date');
$structure['main']['fields'][12][] = array('field' => 'start_date');
$structure['main']['fields'][12][] = array('label' => 'End Date');
$structure['main']['fields'][12][] = array('field' => 'end_date',
                                        'colspan' => 3);
```

Notice the following differences between this and the standard 2-column view:

- Each row is numbered explicitly so that multiple entries can be specified for rows 2 and 12.
- Rows which contain a single label and field (rows 1, 4 and 11 in this example) can be expressed as `'field' => 'label'`
- Rows which require more than one field require a separate entry for each label and each field. Please note the following:
  - The `label` entry must be defined before the `field` entry.
  - It is possible to have a `field` without a `label`.
  - It is not possible to have a `label` without a `field`.

Field Options

Each entry may include any of the following options:

- `colspan` - will allow that field to span more than one column in the HTML table. Without it the remaining columns in that row would be blank.
- `rowspan` - similar to `colspan`, but allows the field to extend vertically for more than 1 row.
- `cols` - will override the value for **multiline** controls supplied in the $fieldspec array.
- `rows` - will override the value for **multiline** controls supplied in the $fieldspec array.
- `align` - specify horizontal alignment within this cell. See the HTML specification for details.
- `valign` - specify vertical alignment within this cell. See the HTML specification for details.
- `size` - will set the size of the textbox control for that field when data can be input or amended. This overrides the **size** value specified in the $fieldspec array. This option is available in both the detail/vertical and list/horizontal views.
- `display-empty` - if the field is not present in the XML file then by default neither the label nor the field will be displayed. This option will cause the empty row to be displayed instead of being dropped.
- `class` - will allow that label or field to be enclosed in a class attribute, where the class name should be defined in the CSS file.
- `imagewidth` - will override the value in the $fieldspec array when an image is displayed. This option is available in both the detail/vertical and list/horizontal views.
- `imageheight` - will override the value in the $fieldspec array when an image is displayed. This option is available in both the detail/vertical and list/horizontal views.
- `javascript` - see Inserting optional JavaScript for details.

Here is an example of a screen structure file that will produce a multi-column DETAIL view as shown in Figure 6:

```
$structure['main']['fields'][1] = array('prod_feature_id' => 'ID',
                                      'colspan' => 3);
$structure['main']['fields'][2] = array('prod_feature_cat_id' => 'Category',
                                      'colspan' => 3);

$structure['main']['fields'][3][] = array('label' => 'Measurement Required?',
                                        'rowspan' => 2);
$structure['main']['fields'][3][] = array('field' => 'measurement_reqd',
                                        'rowspan' => 2);
$structure['main']['fields'][3][] = array('label' => 'Measurement');
$structure['main']['fields'][3][] = array('field' => 'measurement');
$structure['main']['fields'][4] = array('uom_id' => 'Unit of Measure',
                                      'display-empty' => 'y');

$structure['main']['fields'][5] = array('prod_feature_desc' => 'Description',
                                      'colspan' => 3);
```

Please note the following:

- The first two entries in row 3 use the `rowspan` option to extend down into row 4.
- As the first two cells in row 4 are now occupied the entries for row 4 will now start from cell 3.
- If there is the possibility that row 4 could be empty thus causing it to be dropped from the display, the `'display-empty' => 'y'` entry will cause empty cells to be output instead. This avoids row 5 being moved up to take the position of the missing row 4, which is to the right of the first two cells and immediately below the last 2 cells in row 3.

Please note that additional options may be specified for use with javascript, as shown in RADICORE for PHP - Inserting optional JavaScript - Hide and Seek.

Screen Structure file for a LIST view (horizontal)

This is an example of a structure file to produce a standard LIST view as shown in Figure 7.

```
<?php
$structure['xsl_file'] = 'std.list1.xsl';

$structure['tables']['main'] = 'person';

$structure['main']['columns'][] = array('width' => 5);
$structure['main']['columns'][] = array('width' => 70);
$structure['main']['columns'][] = array('width' => 100);
$structure['main']['columns'][] = array('width' => 100);
$structure['main']['columns'][] = array('width' => 100, 'align' => 'center', 'nosort' => 'y');
$structure['main']['columns'][] = array('width' => '*', 'align' => 'right');

$structure['main']['fields'][] = array('selectbox' => 'Select');
$structure['main']['fields'][] = array('person_id' => 'ID');
$structure['main']['fields'][] = array('first_name' => 'First Name');
$structure['main']['fields'][] = array('last_name' => 'Last Name');
$structure['main']['fields'][] = array('star_sign' => 'Star Sign', 'nosort' => 'y');
$structure['main']['fields'][] = array('pers_type_desc' => 'Person Type');
?>
```

The `selectbox` entry does not relate to any field from the database. It is a reserved word which causes a checkbox to be defined in that column with the label 'Select'.

The `nosort` entry (which can be specified in either of the **columns** and **fields** arrays) is optional and will prevent the column label from being displayed as a hyperlink which, when pressed, will cause the data to be sorted on that column.

In some cases it might be useful to display an empty column (no heading, no data), and this can be achieved by inserting a line in any of the following formats:

```
$structure['main']['fields'][] = array('blank' => '');
$structure['main']['fields'][] = array('null' => '');
$structure['main']['fields'][] = array('' => '');
```

It is also possible to use some of the field options which have been described previously:

- Options which are added to the `['zone']['columns']` array will appear in the `<colgroup>` element of the HTML document.
- Options which are added to the `['zone']['fields']` array will appear in the `<td>` element of the HTML document.

When this information is written out to the XML document it will look something like the following:

```
<structure>
  <main id="person">
    <columns>
      <column width="5"/>
      <column width="70"/>
      <column width="100"/>
      <column width="100">/>
      <column width="100" align="center"/>
      <column width="*" align="right"/>
    </columns>
    <row>
      <cell label="Select"/>
      <cell field="selectbox"/>
    </row>
    <row>
      <cell label="ID"/>
      <cell field="person_id"/>
    </row>
    <row>
      <cell label="First Name"/>
      <cell field="first_name"/>
    </row>
    <row>
      <cell label="Last Name"/>
      <cell field="last_name"/>
    </row>
    <row>
      <cell label="Star Sign"/>
      <cell field="star_sign" nosort='y'/>
    </row>
    <row>
      <cell label="Person Type"/>
      <cell field="pers_type_desc"/>
    </row>
  </main>
</structure>
```

Although the contents of the screen structure file is fixed it is possible to make temporary amendments before it is used to build the HTML output. Please refer to the following:

- How can I make a single row in a multi-row area non-editable?
- How can I remove the select box from a single row?
- How can I hide/remove columns in a multi-row display?
- How can I modify screen labels at runtime?
- How can I replace a column and its label at runtime?

There are more details available in the RADICORE Infrastructure guide.

XML data

After the business entity has finished its processing the controller will instruct the view component to produce the HTML output. It does this by putting all the relevant data into an XML document. This data comes from the following sources:

- The screen structure file in order to identify what application data goes where on the screen. This will appear in the <structure> element of the XML document.
- Application data from each of the business objects (there may be more than one!) which were specified in the component script. This data will appear in the data area of the HTML output. This area is broken down into one or more zones (e.g. 'main', or 'outer' and 'inner', or 'outer', 'middle' and 'inner'), and each object's data is allocated to one of those zones.
- Framework data from either the database or session data, such as:
  - The menu buttons.
  - The navigation buttons.
  - The title bar.
  - The column headings for LIST screens.
  - The action buttons.
  - The message area for errors and warnings.
  - The pagination and scrolling areas.

This document will then be transformed into HTML.

There are more details available in the RADICORE Infrastructure guide.

XSL transformation

This is a standard process by which an XML document is transformed into HTML output using instructions contained within an XSL Stylesheet. This process is performed at the web server end by default, but it is also possible for it to be performed within the client browser, thus reducing the load on the server.

There are more details available in the RADICORE Infrastructure guide.

HTML output

This is the document which is sent back to the client's browser is response to the request. Its content should conform to the (X)HTML specification which is supervised by the World Wide Web Consortium.

There are more details available in the RADICORE Infrastructure guide.

XSL Stylesheet

The framework contains a series of reusable XSL stylesheets each of which will present the data in one of the structures described in Transaction Patterns for Web Applications. These stylesheets are reusable because of one simple fact - none of them contain any hard-coded table or field names. Precise details of which elements from the XML file are to be displayed where are provided within the XML file itself from data obtained from a screen structure script which is placed into the <structure> element. This is a great improvement over my original software which required a separate version of a stylesheet for each component.

There are two basic structures for displaying data, and while some stylesheets use one or the other there are some which contain the same basic elements in different variations and combinations.

- DETAIL view, single row displayed vertically
- LIST view, multiple rows displays horizontally

XSL Stylesheet for a DETAIL view (vertical)

This is a DETAIL view which displays the contents of a single database row in columns going vertically down the page with labels on the left and data on the right, as shown in Figure 4.

Figure 4 - Standard 2-column DETAIL view

label1	data1
label2	data2
label3	data3
label4	data4

In most cases the standard 2-column view will be sufficient, but there may be cases when it is desired to have more than one piece of data on the same horizontal line, as shown in Figure 5.

Figure 5 - Enhanced multi-column DETAIL view

label1	data1		
label2a	data2a	label2b	data2b
label3	data3		
label4	data4a	data4b	

It is important to note that it is not a simple case of adding in extra columns to a row. An HTML table expects each row to have the same number of columns, so it is necessary to make adjustments on all other rows otherwise they will contain empty columns. Notice the following points about Figure 5:

- 'data1' has been told to span 3 columns, otherwise it would be no wider than 'data2a'.
- Line 2 contains two labels and two fields. This dictates that each row must therefore contain 4 columns.
- 'data4b' does not have a label. This is allowed, but it is not possible to have a label without data.
- 'data4b' has been told to span 2 columns, otherwise it would be no wider than 'label2b'.

In an HTML table it is possible to have a single piece of data which spans more than one column, as shown in Figure 5. It is also possible to have a single piece of data which spans more than one row, as shown in Figure 6.

Figure 6 - Enhanced multi-row DETAIL view

label1	data1		
label2a	data2a	label2b	data2b
		label2c	data2c
label3	data3		

Notice here that the effect of spanning multiple rows requires data in the other columns in each of the rows that are being spanned.

Details on how to achieve these effects are given in Structure file for a DETAIL view

The same DETAIL view can be used by different components which access the same business entity in different modes (insert, update, enquire, delete and search) as the controller script will pass its particular mode to the XSL stylesheet during the transformation process. The XSL stylesheet will use the mode value to alter the way that it deals with individual fields by making them read-only or amendable as appropriate.

Here is a sample of the XSL stylesheet which provides a DETAIL view:

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">

<xsl:output method="xml"
            indent="yes"
            omit-xml-declaration="yes"
            doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"
            doctype-system = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
/>

<!-- include common templates -->
<xsl:include href="std.buttons.xsl"/>
<xsl:include href="std.column_hdg.xsl"/>
<xsl:include href="std.data_field.xsl"/>
<xsl:include href="std.head.xsl"/>
<xsl:include href="std.pagination.xsl"/>

<!-- get the name of the MAIN table -->
<xsl:variable name="main" select="//structure/main/@id"/>
<xsl:variable name="numrows">1</xsl:variable>

<xsl:template match="/">

  <html xml:lang="{/root/params/language}" lang="{/root/params/language}">
    <xsl:call-template name="head" />
  <body>

  <form method="post" action="{$script}">
```

```
    <div class="universe">

      <!-- create help button -->
      <xsl:call-template name="help" />

      <!-- create menu buttons -->
      <xsl:call-template name="menubar" />

      <div class="body">

        <h1><xsl:value-of select="$title"/></h1>

        <!-- create navigation buttons -->
        <xsl:call-template name="navbar_detail" />

        <div class="main">
          <!-- table contains a row for each database field -->
          <table>

            <!-- process the first row in the MAIN table of the XML file -->
            <xsl:for-each select="//*[name()=$main][1]">

              <!-- display all the fields in the current row -->
              <xsl:call-template name="display_vertical">
                <xsl:with-param name="zone" select="'main'"/>
              </xsl:call-template>

            </xsl:for-each>

          </table>
        </div>

        <!-- look for optional messages -->
        <xsl:call-template name="message"/>

        <!-- insert the scrolling links for MAIN table -->
        <xsl:call-template name="scrolling" >
          <xsl:with-param name="object" select="$main"/>
        </xsl:call-template>

        <!-- create standard action buttons -->
        <xsl:call-template name="actbar"/>

      </div>

    </div>

  </form>
  </body>
  </html>

</xsl:template>

</xsl:stylesheet>
```

Full a full breakdown of the different parts within the XSL stylesheet please refer to Reusable XSL Stylesheets and Templates.

XSL Stylesheet for a LIST view (horizontal)

This is a LIST view which displays the contents of several database rows in horizontal rows going across the page. The top row contains column headings (labels) while the subsequent rows contain data, each row from a different database occurrence.

Figure 7 - Standard multi-occurrence LIST view

select	*label1*	*label2*	*label3*	*label4*
x	data1	data2	data3	data4
x	data1	data2	data3	data4
x	data1	data2	data3	data4

Here is a sample of the XSL stylesheet which provides a LIST view:

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">

<xsl:output method="xml"
            indent="yes"
            omit-xml-declaration="yes"
            doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"
            doctype-system = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
/>

<!-- include common templates -->
<xsl:include href="std.buttons.xsl"/>
<xsl:include href="std.column_hdg.xsl"/>
<xsl:include href="std.data_field.xsl"/>
<xsl:include href="std.head.xsl"/>
<xsl:include href="std.pagination.xsl"/>

<!-- get the name of the MAIN table -->
```

```
<xsl:variable name="main" select="//structure/main/@id"/>
<xsl:variable name="numrows" select="//pagination/page[@id='main']/@numrows"/>

<xsl:template match="/"> <!-- standard match to include all child elements -->

  <html xml:lang="{/root/params/language}" lang="{/root/params/language}">
    <xsl:call-template name="head" />
  <body>

  <form method="post" action="{$script}">

    <div class="universe">

      <!-- create help button -->
      <xsl:call-template name="help" />

      <!-- create menu buttons -->
      <xsl:call-template name="menubar" />

      <div class="body">

        <h1><xsl:value-of select="$title"/></h1>

        <!-- create navigation buttons -->
        <xsl:call-template name="navbar">
          <xsl:with-param name="noshow"   select="//params/noshow"/>
          <xsl:with-param name="noselect" select="//params/noselect"/>
        </xsl:call-template>

        <div class="main">

          <!-- this is the actual data -->
          <table>

            <!-- set up column widths -->
            <xsl:call-template name="column_group">
              <xsl:with-param name="table" select="'main'"/>
            </xsl:call-template>

            <thead>
              <!-- set up column headings -->
              <xsl:call-template name="column_headings">
                <xsl:with-param name="table" select="'main'"/>
              </xsl:call-template>
            </thead>

            <tbody>
              <!-- process each non-empty row in the MAIN table of the XML file -->
              <xsl:for-each select="//*[name()=$main][count(*)>0]">

                <!-- display all the fields in the current row -->
                <xsl:call-template name="display_horizontal">
                  <xsl:with-param name="zone" select="'main'"/>
                </xsl:call-template>

              </xsl:for-each>
            </tbody>

          </table>

        </div>

        <!-- look for optional messages -->
        <xsl:call-template name="message"/>

        <!-- insert the page navigation links -->
        <xsl:call-template name="pagination">
          <xsl:with-param name="object" select="'main'"/>
        </xsl:call-template>

        <!-- create standard action buttons -->
        <xsl:call-template name="actbar"/>

      </div>

    </div>

  </form>
  </body>
  </html>

</xsl:template>

</xsl:stylesheet>
```

There are more details available in the RADICORE Infrastructure guide.

Levels of Reusability

Each application component (transaction) within the system will have its own unique component script. Although this is not sharable, it is a very small script which does nothing but identify which Model, View and Controller to use, most of which are sharable. This arrangement allows me to achieve the following levels of reusability:

- There is a separate Model class for each business entity (database table) within the application which encapsulates all the properties and methods necessary for that entity. This class can be shared by any number of components.
- Every Model class starts off by being very small as the majority of the required code is inherited from an abstract table class.

- Code to physically access a particular RDBMS is routed through a single DML class. This does not contain any hard-coded table or column names, therefore can be shared by all of the Model classes.

- Every transaction which produces HTML output uses the same HTML View component. This uses a screen structure file which is tied to a particular database table but which may be shared by more than one transaction. Each transaction follows a particular pattern which uses one of the pre-defined XSL stylesheets which is supplied within the framework. There are a dozen or so of these XSL stylesheets which can be used by any number of different transactions, and as they do not contain any hard-coded table or column names they can be used with any number of Model classes.

- Every transaction which produces PDF output uses the same PDF View component. This uses a report structure file which is unique for each report.

- Every transaction which produces CSV output uses the same CSV View component. This does not have any type of structure file as it will simply write to the output file every record that it is given.

- Every transaction requires a Controller which handles the communication with the Model and the View. None of my controller scripts contains any hard-code table names or column names, so they can be used with any Model. Each Controller script will be tied to a particular type of View - HTML, PDF, CSV or no view at all in those cases where no visible output is required - and is catalogued in my library of Transaction Patterns.

Using this design I have built framework which contains the following:

- Each transaction accesses one of the pre-defined Controller scripts, one for each of the 43 Transaction Patterns.
- Each table (Model) class is generated from my Data Dictionary and inherits a large volume of code from a single abstract table class.
- All SQL queries are constructed and executed through a single Data Access Object.
- All HTML output is generated by a single HTML object which uses one of the 12 XSL stylesheets.
- All PDF output is generated by a single PDF object.
- All CSV output is generated by a single CSV object.

Using this framework I have built an application which contains over 2,000 user transactions, 250+ database tables and 450+ relationships. Each database table (Model) class was generated from my Data Dictionary and automatically shared code from my abstract table class and Data Access Object. Each user transaction was generated from one of my Transaction Patterns and automatically shared one of my Controller scripts and one of my View objects.

In *my* implementation I can add a new table to my database, import the details into my Data Dictionary, create the class file for the Model then create the initial transactions for the LIST1, SEARCH1, ADD1, ENQUIRE1, UPDATE1 and DELETE1 patterns, and be able to run them from my menu system, in under 5 minutes *without having to write a single line of code*. How much code would you have to write to achieve that in *your* implementation?

In *your* implementation how much code do you have to write for each Model in order to produce a working transaction? In mine it is none as the model classes are generated from my Data Dictionary and inherit a large amount of code from my abstract table class.

In *your* implementation how much code do you have to write to validate user input? In mine it is none as all user input is automatically validated using my standard validation object which compares each field's value with the field's specifications provided by the table structure file.

In *your* implementation how much code do you have to write for each View in order to produce a working transaction? In mine it is none as each HTML page is created from a pre-built View object which uses one of my pre-defined XSL stylesheets.

In *your* implementation how much code do you have to write for each Controller in order to produce a working transaction? In mine it is none as each transaction uses one of my pre-defined controller scripts.

In *your* implementation how many Controllers are tied to a particular Model and are therefore not sharable?. In mine I have 43 controllers which can operate on any Model, so they are infinitely sharable.

If *your* implementation cannot achieve the same levels of reusability/sharability as mine then please do not try to tell me that *my* implementation is inferior.

Criticisms of my implementation

Criticism #1

In this forum post someone asked advice on how the MVC pattern should be implemented, so I decided to offer some based on my experience of having successfully implemented the MVC pattern in a large web application. It seems that quite a few people took exception to my approach as it violated their personal interpretations of the rules of MVC.

In this post I stated the following:

> Should the Model transmit its changes directly into the View, should the View extract the changes directly from the Model, or should the Controller extract from the Model and inject into the View? Actually, there is no hard and fast

> rule here, so any of these options would be perfectly valid.

TomB responded with this:

> Well in MVC the view gets its own data from the model. Controller extracting from the model and passing to the view is wrong.

In this post he said:

> MVC states that views access the model directly (ie not using the controller as a mediator) and that models should not know of controllers and views.

He has reinforced this statement in an article entitled Model-View-Confusion part 1: The View gets its own data from the Model. What he fails to realise is that different examples of MVC implementations show different ways in which the model obtains its data, so he cites the ones that follow his personal opinion as being "right" and dismisses all the others as being "wrong".

I feel that his interpretation of the statement "models should not know of controllers and views" goes too far. A model may have any of its methods called by another object, or may call a method on an object of unknown origin which has been injected into it, but it should not have any hard-coded dependencies on specific view objects or any methods which are tied to specific objects as this would introduce tight coupling between those objects. By using generic methods which can be used by multiple objects you are introducing loose coupling which makes the code more maintainable and less prone to error. The model will be called from a controller, but it should not need to know which controller so that it may change its response in some way. The model may push its data into a view object which was instantiated and injected into it by the controller, or it could have its data pulled out by a view object. In either case it should not need to know which view or controller requested that data, it should simply return the data and let the calling object do what it wants with it, which may be to render that data it whatever form is appropriate, which could be HTML, PDF, CSV or whatever.

My point of view is reinforced by How to use Model-View-Controller (MVC) which lists the following in the **Basic Concepts** section:

> In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.
> - The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application.
> - The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
> - the **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller)
>
> The formal separation of these three tasks is an important notion that is particularly suited to Smalltalk-80 where the basic behavior can be embodied in abstract objects: View, Controller, Model and Object. The MVC behavior is then inherited, added to, and modified as necessary to provide a flexible and powerful system.

Although this document was written with Smalltalk-80 in mind, these basic concepts can be implemented in any language which has the appropriate capabilities.

Notice that it concentrates on the responsibilities of the three components, and does nothing but "suggest" how the data may flow between them. Note the use of the term "usually" in the above list, and the phrase "commanding the model **and/or** the view to change as appropriate." Later in the same document you may find references to particular implementations, but as far as I am concerned these are **suggested** implementations by virtue of the fact that the words used are **can**, **could** and **may**. These are not imperatives such as **must**, **should** or **will**, and as such should not be regarded as **required** in any implementations.

This Wikipedia article contains the following:

> In addition to dividing the application into three kinds of components, the MVC design defines the interactions between them.
> - A **controller** can send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document). It can also send commands to the model to update the model's state (e.g., editing a document).
> - A **model** notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands. A passive implementation of MVC omits these notifications, because the application does not require them or the software platform does not support them.
> - A **view** requests from the model the information that it needs to generate an output representation.

So even in this definition there are several possibilities:

- The **controller** can send commands to both the **model** and the **view**.
- The **model** can send commands to both the **view** and the **controller**.

- The **view** can receive commands from both the **model** and the **controller**.

Which variation of these options you choose to implement is therefore down to your personal preferences or what options are available in your design.

So even though the MVC pattern does not (or should not, IMHO) dictate how the flow of data should be implemented, whether it should be pulled or pushed, or who should do the pulling or the pushing, TomB seems to think that the only implementations which are allowed are those which have received his personal seal of approval. Isn't this a little arrogant on his part?

Criticism #2

In this post TomB stated:

> By state I meant does the model contain a 'current record set'. Selecting which records to show is clearly display logic (along with ordering and limiting).

I replied in this post with the following:

> I disagree (again). The Model contains whatever records it has been told to contain, either by issuing an sql SELECT statement, or by having data injected into it from the Controller. The View does not *request* any particular records from the Model, it deals with *all* records that are currently contained in the Model. Ordering and limiting are data variables which are passed from the Controller, through the Model and to the DAO so that they can be built into the sql SELECT statement. The View does *not* decide how to limit or sort the data - the data which it is given has already been sorted and limited.

TomB argued in this post with the following:

> Ordering, limiting and generally deciding which records to show is 100% display logic. By moving this to the model you've reduced reusability of the view.

I don't know about you, but if there are 1,000s of records to display, but the user wants to see them in pages of 10, it is much more efficient, both in time and memory, for the Model/DAO to use **limit** and **offset** values so that only the relevant records are actually read from the database. All of these records are then passed to the View, and the View displays every record which it has been given. It just does not sound practical to pass 1,000s of records to the View, then get the View to decide which 10 to display. Similarly it is far easier to sort the records by adding **order by** to the SQL query than it is to have code to sort them in the View.

This is no way reduces the reusability of the view as all HTML output is produced from a single pre-written View object which uses one of my pre-written XSL stylesheets. This single object can be used to create the HTML output for any transaction and any Model. What could be more reusable than that?

Criticism #3

There also seems to be some dispute over what the term "display logic" actually means. A software application can be broken down into the following types of logic:

- **Display logic** - code which generates the HTML output that is sent to the user. This exists in, and only in, the View.
- **Business logic** - code which implements the business rules. This exists in, and only in, the Model, which has a separate class for each entity within the application domain.
- **Data Access logic** - code which constructs and executes SQL queries to read or update data in a database. This exists in, and only in, the Data Access Object (DAO) which is accessed from the Model.
- **Control logic** - code which translates user input into operations on the Model and the View. This exists in, and only in, the Controller.

It is clear to me that **display logic** is that code which directly generates the HTML output. The data which is transformed into HTML may come from any source, but the code which obtains or generates that data is **not** part of the display logic. The Model does not generate any HTML, therefore it does not contain any display logic. Similarly the Model does not generate and execute any SQL queries, therefore it does not contain any data access logic. This can be summed up as follows:

- Code which transforms data into HTML **is** display logic.
- Code which creates or obtains the data which is subsequently transformed into HTML **is not** display logic.

I'm sorry if my interpretation of these minor details is different to yours, but just because my interpretation is different does not mean that it is wrong. I still have components which carry out the responsibilities of Model, View and Controller, so as far as I am concerned my version of MVC is perfectly valid. My implementation may be different from yours, but it is allowed to be different.

Criticism #4

According to some people there is supposed to be a rule which says that a Controller can only ever communicate with a single Model, but as I have never heard of this rule and have never been given any justification for its existence I see no

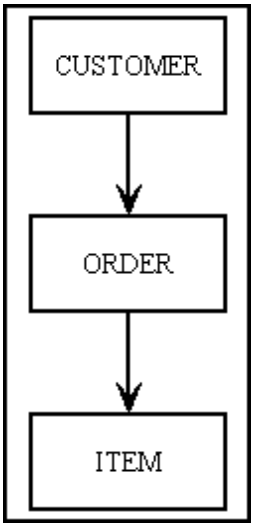
reason to be bound by it. In my implementation each HTML page is broken down into a series of zones or areas, and each zone is populated using the contents of a different object as follows:

- The menu bar is populated from the MNU_MENU table in the MENU database.
- The title bar is populated from the MNU_TASK table in the MENU database.
- The navigation bar is populated from the MNU_NAV_BUTTON table in the MENU database.
- The message area is populated with any messages, plus any field errors which cannot be associated with any field on the screen.
- The data area will display data from one or more application objects. How this data is displayed depends on how many occurrences (rows) can be shown at any one time:
  - For single rows there will be the following:
    - Application data will be displayed vertically with labels on the left and field contents on the right.
    - A scrolling area which will allow the user to move backwards or forwards through the available rows in the current selection.
  - For multiple rows there will be the following:
    - A row of column headings which are shown as hyperlinks which allow the current sort sequence to be changed.
    - Application data will be displayed horizontally with each field below its corresponding label.
    - A pagination area which will allow the user to move backwards or forwards through the available pages in the current selection.
    - The navigation bar will contain hyperlinks to allow the number of rows per page to be altered.
- The action bar is populated from a variable which is defined in the controller script.

In all my previous programming languages I was able to populate different areas on the screen from different tables in the database, so I saw no reason why I should not do exactly the same with PHP. Each of my page controllers will deal with each of those applications objects independently and separately, regardless of the number, and not through any single composite object.

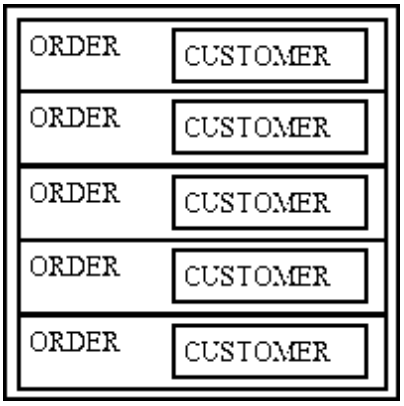
For example, take three database tables which are related as shown in figure 8:

Figure 8 - Three related tables



A typical screen will list multiple rows from the ORDER table in a single zone using the LIST1 pattern as shown in figure 9. Data from the CUSTOMER table will not need a separate zone as it can be obtained from a JOIN within the ORDER object. Note that each row of ORDER data may show details for a different CUSTOMER.

Figure 9 - a single zone (ORDER) with multiple rows



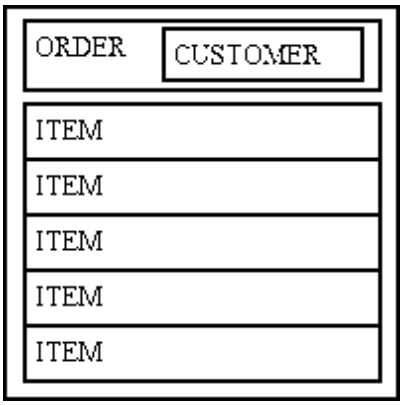
A variation of this may show only those ORDERs for a single CUSTOMER using the LIST2 pattern, as shown in figure 10. Here the CUSTOMER table has its own zone which displays one row at a time. This is separate from and in addition to the

ORDER zone which can display multiple rows. Only those ORDERS which are related to the current CUSTOMER will be shown.

Figure 10 - outer zone (CUSTOMER) and inner zone (ORDER)

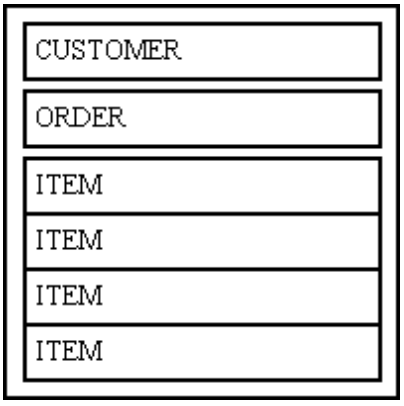
Another screen may only show those ITEMs for a single ORDER using the LIST2 pattern, as shown in figure 11. Here the ORDER table has its own zone which displays one row at a time. This is separate from and in addition to the ITEM zone which can display multiple rows. Only those ITEMs which are related to the current ORDER will be shown. Any CUSTOMER data is obtained using a JOIN within the ORDER object.

Figure 11 - outer zone (ORDER) and inner zone (ITEM)



Another screen may only show those ITEMs for a single ORDER for a single CUSTOMER using the LIST3 pattern, as shown in figure 12. Here the CUSTOMER table has its own zone which displays one row at a time. This is separate from and in addition to the ORDER zone which can also only display one row at a time, and the ITEM zone which can display multiple rows. Only those ITEMs which are related to the current ORDER which are related to the current CUSTOMER will be shown.

Figure 12 - outer zone (CUSTOMER), middle zone (ORDER) and inner zone (ITEM)



The idea of forcing the page controller to go through a single composite object in order to deal with the separate database objects never occurred to me. Doing so would require extra effort which would achieve absolutely nothing (except to conform to a rule which I did not know existed) so as a follower of the KISS principle it is my choice, if not my duty, to avoid unnecessary complexity and stick with the simplest thing that could possibly work.

References

- Wikipedia: Model-view-controller
- Using PHP Objects to access your Database Tables (Part 1)

- Using PHP Objects to access your Database Tables (Part 2)
- What is/is not considered to be good OO programming
- In the world of OOP am I Hero or Heretic?
- Generating dynamic web pages using XSL and XML
- Reusable XSL Stylesheets and Templates
- Using PHP 4's DOM XML functions to create XML files from SQL data
- Using PHP 4's Sablotron extension to perform XSL Transformations
- Using PHP 5's DOM functions to create XML files from SQL data
- Using PHP 5's XSL functions to perform XSL Transformations
- Transaction Patterns for Web Applications
- Customising the PHP error handler
- Component Design - Large and Complex vs. Small and Simple
- A Development Infrastructure for PHP
- FAQ on the Radicore Development Infrastructure
- UML diagrams for the Radicore Development Infrastructure
- A Sample PHP Application
- A Data Dictionary for PHP Applications

© ***Tony Marston***
2nd May 2004

http://www.tonymarston.net
http://www.radicore.org

Amendment history:

15 Apr 2014	Added Criticsm #4.
14 Jun 2013	Restructured the document into a more logical sequence, added new sub-sections for the Model, View and Controller objects, and turned the items in Figure 3 into clickable links.
30 Apr 2012	Added details about the View object.
01 Aug 2011	Modified XSL (screen) Structure file to change the way that the 'align' and 'valign' attributes are handled in LIST screens as they are poorly supported in most browsers, and not supported at all in HTML5.
26 Jun 2010	Modified How they fit together to include some additional comments. Added Criticisms of my implementation.
01 May 2010	Modified Screen Structure file for a DETAIL view to allow the `cols` and `rows` attributes.
01 Feb 2010	Modified Screen Structure file for a DETAIL view to allow the `align` and `valign` attributes.
01 May 2007	Modified Screen Structure file for a DETAIL view to allow for the addition of a `class` attribute on both the `label` and `field` specifications.
09 Aug 2006	Modified Screen Structure file for a DETAIL view to allow for additional options for use with javascript, as documented in RADICORE for PHP - Inserting optional JavaScript - Hide and Seek.
21 July 2006	Modified Screen Structure to allow for a wider range of attributes on the column specifications.
21 June 2005	Modified the contents of XSL Stylesheet and Screen Structure file.
17 June 2005	Moved all descriptions of the contents of the Business Entity Class to A Data Dictionary for PHP Applications.