More About Models (optional)

lesson #basic03-more

James L. Parry
B.C. Institute of Technology

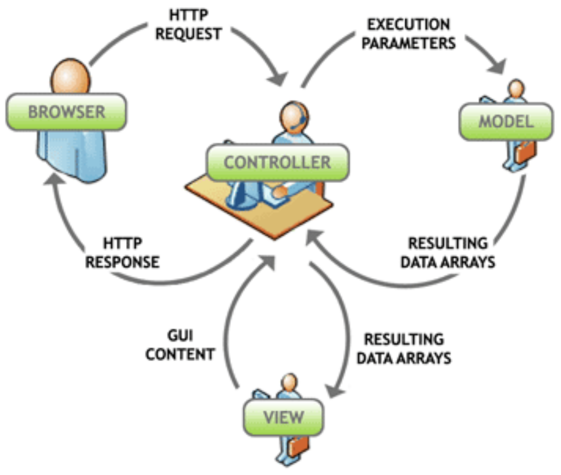
Agenda

MODEL PATTERNS

Models are the "model" component of the MVC design pattern.

The figure to the right shows this from the perspective of a webapp.

Fundamentally, models encapsulate data sources.

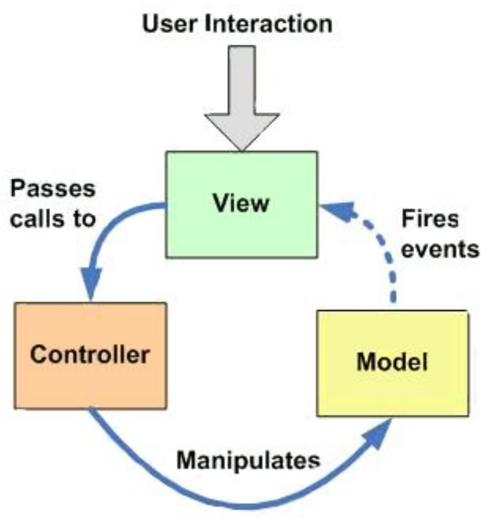


Classic Model-View-Controller

The figure to the right presents a "classic" view of the MVC pattern, from a programming perspective.

A user interacts with a **view**, using view components such as links, buttons, and form fields to initiate a request to a controller.

The "tricky" part? The view is supposed to be bound to a model as an event handler; when model state changes, an event is fired, and the view would respond to the event, updating its presentation as warranted.



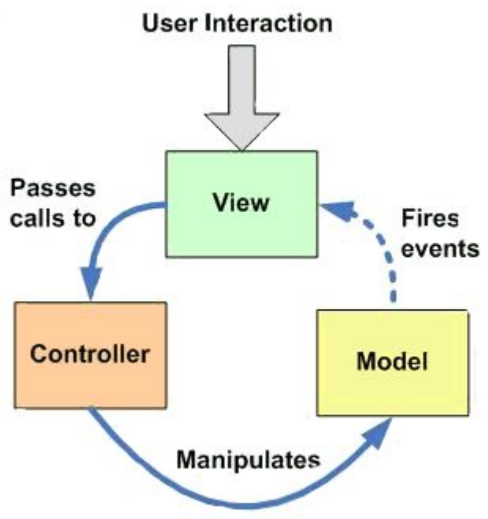
Model-View-Controller

Classic MVC and the Web

The "classic" MVC has two issues from a webapp's perspective:

- Accessing a model inside a view is legal, but considered a poor practice
- Propagating an event in a distributed system is awkward

The next two slides present solutions to these.



Model-View-Controller

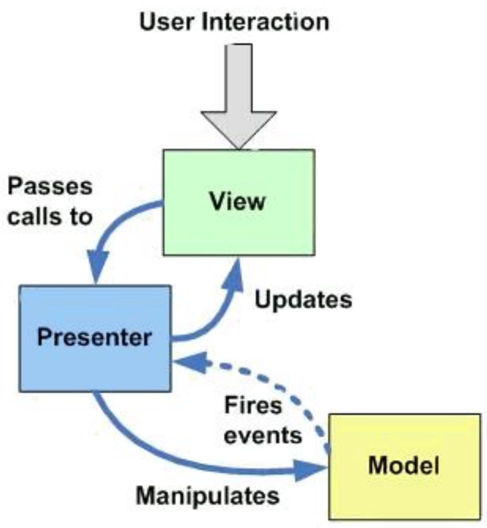
Model-View-Adapter

The model-view-adapter design pattern has the controller handle all interaction with a model, and pass data to a view through parameters.

The view is unaware of the source of the data it presents.

True event handling is still awkward with PHP, as our objects are not memory-resident.

This pattern is also called model-view-presenter.



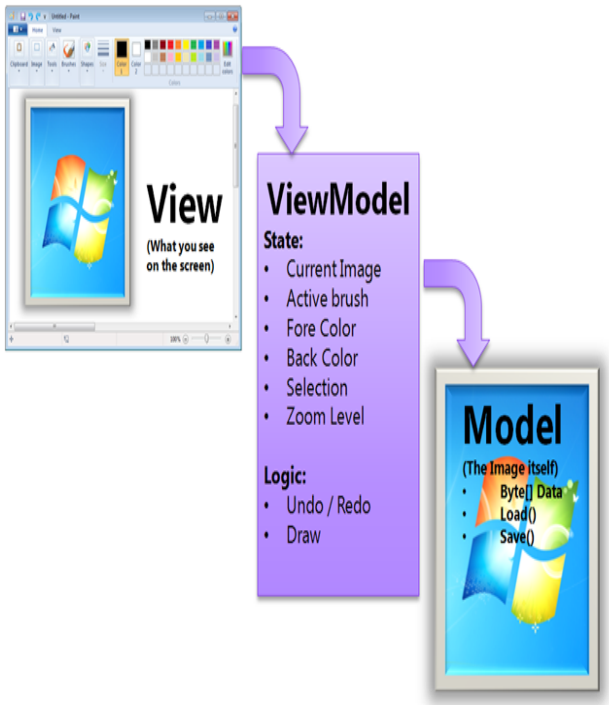
Model-View-Presenter

Model-View-ViewModel

The model-view-viewmodel design pattern has an explicit component to handle the model and related events, while keeping the view itself unaware of the source of its data.

In a webapp, this could be implemented by data-aware "widgets", typically programmed using Javascript.

This pattern is common in enterprise applications (eg. ASP.net and JavaServerFaces), and supported by AngularJS.



KINDS OF MODELS

What is a model?

1) Models represent data sources <data access layer>, each a "collection" of entities

2) Models represent entities <domain model>, each an object in a "collection"

You've seen models corresponding to RDB tables as data sources. Let's dive deeper!

Domain Models

Domain models are awkward to enforce in PHP, without strong typing. They are normal in Java & C#.

For instance, Java supports JavaBeans (serializable, standard accessor and mutator methods) for generic domain models.

Java also supports Enterprise JavaBeans, following a managed, server-side component architecture. These can be session based, message-based, or persistent. The management here is at the entity level.

Active Record Models

The active record design pattern, common in webapps, suggests that domain models have collection-centric methods in addition to entity property accessor/mutators. This leads to model methods like...

- `add($record)` – add a new record to the collection
- `get($key)` – retrieve a keyed record
- `update($record)` – replace a record
- `delete($key)` – delete a record

Some additional methods common to active record models:

- `create()` – return a suitable empty object
- `all()` – return all records
- `some(...)` – return some of the records
- `size()` – return the number of records

Active Record in Practice

Given how easy it is to inject or eliminate properties in PHP objects, many developers use entity properties directly, rather than attempting to follow the rigor of other language domain models

Given the way that PHP has evolved, many developers will also provide methods that return or work with associative arrays, using them to store entity properties.

This means doing something like `$customer->name` rather than something like `$customer->getName()`

Active Record in Practice /2

If adding collection centric methods to a "conventional" domain model, it is a good idea for your model to have collection-specific properties too, for instance the name of a database table and the fieldname used as a primary key.

Regardless of any accessors or CRUD methods you might have, make sure your model provides suitable methods for extracting whatever data your controllers need to pass on to a view!

This means having properties for the table name that a model is bound to, or for the XML document used to record state.

Data extraction methods,not related to a domain model, could be something like `findNewestBlogEntry()` or `processOrder($items)`

Note that these suggestions run counter to many "purist" perspectives ... PHP tends to be more relaxed!

Models Can Be Used For... ?

Models can be used to encapsulate many different sources of data, as you will see in the next few slides :)

Beyond relational database tables, we will take a quick look at encapsulating XML documents, file system folders, and even services.

These different kinds of models will be revisited later in the course, when we talk about XML and services.

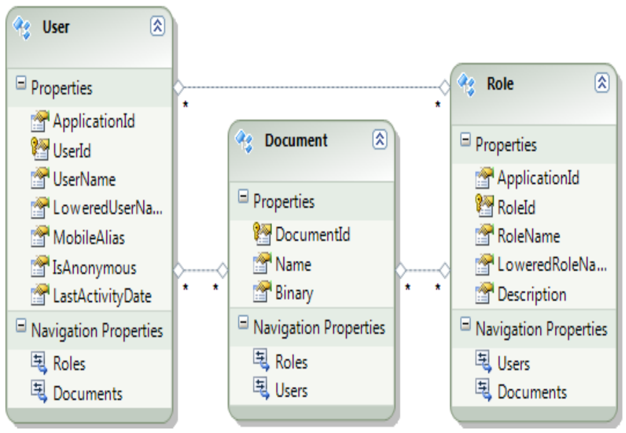
Models for RDB Tables

You would normally have one model per table in the relational database you are using.

CRUD methods in your model would deal with rows in the RDB table.

Reasonable model management properties in this kind of model would usually include the table name and the field name to use as the primary key.

Clearly 3 models...



Sample Model for an RDB Table

Referring to the contacts example from week 2, `application/core/MY_Model.php` is a sample base model you might use. `models/Contacts.php` is a model using the RDB base model. The following provides access to all the CRUD methods defined in MY_Model, making it easy to follow the active record pattern.

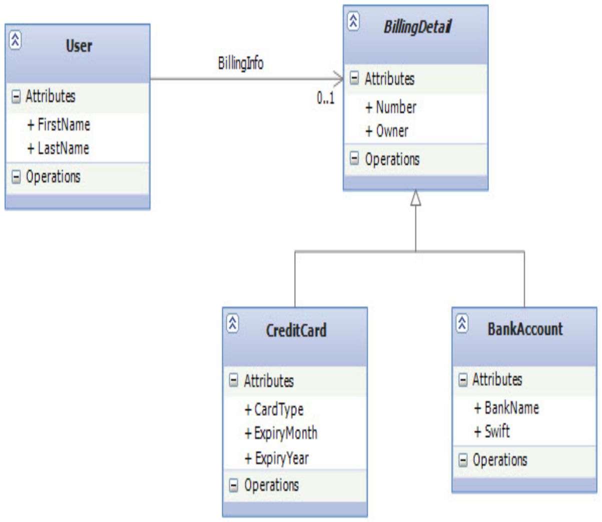
```
class Contacts extends MY_Model {
  function __construct() {
    parent::__construct();
    $this->setTable('contacts', 'ID');
  }
}
```

Models for XML

You could use a model for a suitably structured XML document, for instance one where each child of the root element stored the state of an entity. You would need to choose a convention for the "primary key" in such a case, for instance the "id" attribute of an entity's element.

CRUD methods in your model would deal with children of the root element, and would traverse each child's attributes and elements to build a record object (or associative) array that looked like it came from an RDB table.

Clearly 3 models, or maybe 4, or just 2? ...



Models for XML /2

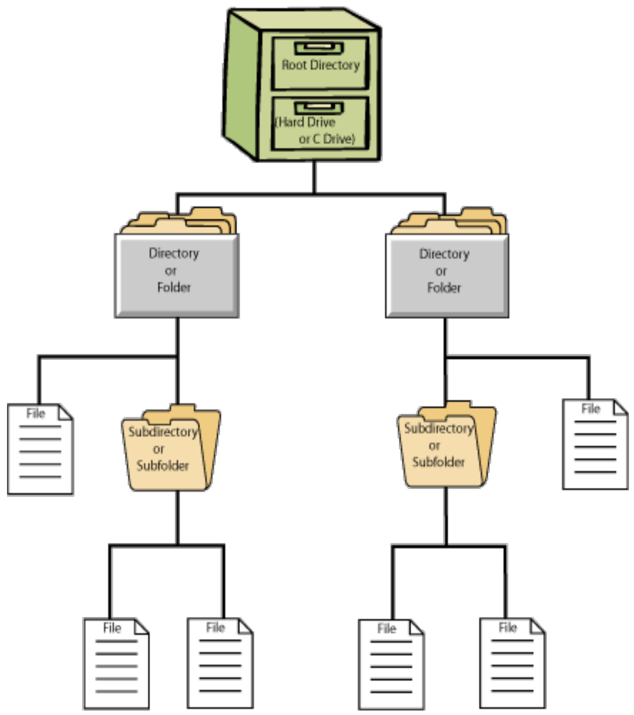
Management properties needed to build/rebuild the XML document for saving might include the document name, The element name of the root element, and the name of the attribute or nested child element to use as a "primary key".

Note that this approach could lend itself to using a DTD or schema for validation of "record" structure and content.

Models for Folders

You could use a model to encapsulate the contents of folders in your filesystem. The folder and subfolders would not have to be inside your webapp - they could be anywhere!

Such a model could be used for the files themselves, or to mimic RDB table rows through flat file representations of records.



Models for Folders /2

Management properties suited to this use might include the base folder name for your "collection", and perhaps the default extension/filetype. A filename itself might be considered a "primary key".

An example of this kind of model would be to provide access to a folder of images that support an image gallery. The `all()` method might return a list of image descriptor objects. You might use subfolder names as categories.

Models for Services

You could use a model to encapsulate a service, mapping the conventional CRUD methods to something more appropriate for a service. This isn't the only way to access a service, or even the best way - just one approach.

Your model would need some properties to handle the service configuration, for instance the endpoint of a remote service.

A sample method mapping is shown to the right. This is a foreshadowing of RESTful services :)

- `add($record)` sends a POST(x) to the service
- `create()` sends a POST() to the service
- `get($key)` sends a GET(x) to the service
- `all()` sends a GET() to the service
- `update($record)` sends a PUT(x) to the service
- `delete($key)` sends a DELETE(x) to the service

Congratulations!

You have completed lesson #basic03-more: More About Models (optional)

If you would take a minute to provide some feedback, we would appreciate it!

The next activity in sequence is: basic04 Views (DRAFT)

You can use your browser's back button to return to the page you were on before starting this activity, or you can jump directly to the course homepage, organizer, or reference page.