


Friday, April 27, 2007


Complex Views with the Zend Framework - Part 3: Composite View Pattern

In the previous two parts of this series of blog posts, I've been looking at the task of implementing complex views with the Zend Framework. Part 1 [looked at what complex views are, what support for complex views the Zend Framework offers out of the box, and a reference to two design patterns](#) useful in adding further support: View Helper and Composite View. In Part 2, [I tackled the View Helper design pattern](#).

As a brief recap, the View Helper pattern promotes the idea of creating helper classes which allow the View layer of a Model-View-Controller application to directly query the Model layer in a read only fashion - effectively bypassing the Controller layer altogether. In such a way, the pattern offers the option to programmers using the Zend Framework to avoid using the current method of nesting Controller Actions with the `Zend_Controller_Action::_forward()` method which can be an overly complex approach for parts of a View we know are common to many pages.

In this post, I offer a brief explanation of the Composite View pattern. It's beyond its scope to show an implementation using the Zend Framework though that's what I'm building up to accomplish in a later blog entry 

If you haven't already guessed, the Composite View pattern is related to the Composite Pattern as defined by the Group of Four (GoF). If you're not familiar with the Composite Pattern, an [excellent PHP article for it was published on Zend Devzone](#) and is worth reading.

Here's a UML diagram of the class relationship although we do things a little different below 



In brief the related Composite View pattern organises View objects into a nested tree structure, in which both parents and children implement some common interface (let's say `render()` for now). You can think of each View representing a single element of the overall page, with parents representing sections of the overall page which may contain other component elements. The essential characteristic, inherited from the Composite Pattern, is that the common `render()` method can be called from the root View, and this method call propagates across all nested objects in the tree until the output from all child Views is finally aggregated into the overall page we intend sending to the browser.

With this description we can make a few assumptions regarding the Zend Framework. The main one being we will need multiple instances of `Zend_View` to pull it off (as distinct from multiple Controllers). Also all this talk of objects hasn't explained how the nesting is controlled. We'll handle the second now, and come back to the `Zend_View` multiplicity after.

There are two methods of handling nesting of View objects. The first follows a purely object based scheme, where objects are nested and combined at runtime. The second passes control over nesting to our friends - the templates. The template method basically involves adding a function (think of the Composite Pattern's interface for "composites") to include another template. Here we'll call it "attach", though it could be anything you prefer. We'll look at this approach below since it's the simplest one to follow if you're familiar with `Zend_View` templates. In addition, the template approach is probably the easiest for programmers and

designers to handle since it allows the View layout to be put in place around all these composite View includes.

Now, we may appear to have come full circle here - this sounds suspiciously like a glorified "include" statement but there are differences. The main one being each composite "attach" creates a new View object which can be sourced from other Modules, and with Module specific helpers and filters (and of course Model accessing View Helpers!). If we assumed the composite method was "attach" (common Composite Pattern method), a composite template could look like:


articlelist.phtml (part of the imaginary Articles application Module):

```
<?php $this->attach('stdheader.phtml', 'default'); ?>
<div class="body">
  <?php echo $this->attach('overbody.phtml') ?>
  <div class="content">
    <?php foreach($this->articles as $article): ?>
      <div class="article-abstract">
        <span class="title"><?php echo $article->title ?></span><br />
        <?php echo $article->summary ?>
      </div>
    <?php endforeach; ?>
  </div>
<?php echo $this->attach('underbody.phtml') ?>
</div>
<?php echo $this->attach('stdfooter.phtml', 'default') ?>
```

overbody.phtml:


```
<div class="overbody">
  <div class="headlines">
    <?php echo $this->attach('last5headlines.phtml', 'Blog', array('feedUri'=>
'http://www.planet-php.net/rss/')) ?>
  </div>
</div>
```

Here we have two templates, both utilise an imaginary attach() method from the View object to attach new sub Views to the template at specific locations within the layout (you're right in thinking attach() will handle rendering of the child View). It also allows for a template to attach Views from other Modules of the application if required (and would transparently manage the different View settings for that Module). I've assumed the lack of a "module" parameter means the attached View should come from the current Module only. The last overbody.phtml template allows for more dynamic parameters targeting the RSS View Helper utilised by the Blog Module's last5headlines.phtml template where we're using the View Helper pattern to grab headlines from the given RSS feed. We visited very similar functionality in my last post.


Of course in all this we haven't ruled out Controller nesting completely. That too is still possible, even alongside View nesting and in a very similar fashion. Since each element attached is an independent entity you can mix and match what you want to use, even from other Modules. Of course with all this floating around we still haven't seen usable code yet! Well, that one's for another blog entry 

Back to the multiplicity issue for a moment, i.e. the presumed use of multiple Zend_View instances. You can guess that Zend_View will need to be subclassed and/or a standard helper class containing the attach()

implementation added. There are other issues also. For example, each View object needs to be independently configured before use - i.e. for paths, encoding, etc. In Part 4 of this series I'll take a stab at adding finer control over creating View objects in a simple reusable fashion.

I'm sure many of you will already suspect the design pattern that will involve 

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 15:58

Another great post! Can't wait for the next one! 

Anonymous on Apr 27 2007, 16:39

Very good, as usual. I'm following this and the DevNetwork discussion closely. I'm relatively new to actually using Patterns, but have studied them for a while. I see the forest, I recognize trees, even climb them part time. But I still fall out of them too often to want to live there full time.

With that in mind, I have a question. You mentioned early in the thread that one reason you started this was that you haven't seen many complex projects with the Zend Framework yet. I haven't either. From what I have seen, many PHP developers work very hard to ignore the "M" component of the MVC method. They use it solely as a place to group their database interfaces. Nothing else.

Now the simplest description I've found of MVC was to think of it as a manufacturing plant. The Controller is the plant manager, making sure needed instructions and parts are delivered to the right workers, the Views are strictly sales reps, explaining only what is needed to customers based on their own specific order, and The Models do the actual work, guided by instructions from the Controller and their own job pre-defined job descriptions.

In most Zend implementations I've seen (including your project), Controllers do the thinking and working, Views do organizing, largely through pre-processing and post-processing hooks. Rendering is almost an afterthought. Models are data gateways and little else.

What I'm wondering: Why are you not using a dynamic Model object to compose the page's info instead of extending the View to do that? Wouldn't it make more sense to accumulate the page structure in a Composite Pattern based Model stored in the Registry? I know of nothing illegal with having a View draw info from a Model, rather than storing that data as a property of itself. Controllers talking to Models is normal. Storing the entire Composite Pattern process as part of the Models seems to fall in line with keeping the Models as the place for the Business Logic and Structure, might help keep Views more reusable, and keeps from having to build special mechanisms into custom Views to handle page composition. It also would seem to simplify radical page modifications late in the process and collection of summary data from the Model before any rendering has even been initialized.

I think you're on the right track, but I also think you may be hanging cabinetry from the sheetrock instead of from the house's frame. By all means, let me know what you think about this, because, basically, I'm just wandering, here.


Thanks again for all your hard work.

This project is helping me a lot. Anonymous on Apr 29 2007, 22:53

There are any number of ways of mixing Views and Models - the only non-negotiable point is that Views and Models exist in separate layers of the applications, with the View only capable of "read" access, i.e. it cannot write to the Model.

The Model itself is unaware of any presentation details - all it cares about is the data and datasource. The View is the only layer capable of transforming the Model into a presentable format (whether HTML, XML, JSON, Files, etc).


The danger in all this is breaking the layers. If the Model suddenly becomes aware of presentation concerns then it's loosing its focus and doing too much - which is a serious refactoring code smell. The Model will quickly get bogged down in a lot of presentation logic. If your idea pans out to a View class/library reading in a "raw" Model and transforming it then the separation is maintained - so long as that library cannot modify the Model.

So... it depends. 

If your idea of compositing Models has any specific relation to "presentation" than it's lost its focus. If the compositing is done by external "view" layer classes, then the separation of concerns is maintained. Anonymous on Apr 30 2007, 19:19

What is the difference with :

Anonymous on Apr 30 2007, 20:15

Might need to repost that? 

Anonymous on Apr 30 2007, 22:07

OK, Good to know I'm on reasonably firm ground with the MVC theory so far. I'm pretty sure my intentions have only one issue that blurs the lines of presentation vs composition.

Our current content system accumulates HTML sequentially from a number of modular subsystems (content, store, calendar, etc.) While the page is assembled, keywords are similarly assembled through concatenation, based on what content is being put together. In the next revision, I hope to make this a smarter process: Articles can be displayed in full, by page, as a summary or as an index link. Therefore, I want to limit the keyword list to full articles or pages, and, more importantly, I want to be able to sometimes suppress article and section keywords in favor of keywords lifted from embedded product lists and event listings. In short, I can't determine which keywords to include (or how to sequence them) until the page composition is fundamentally complete.

Under an extended View Composition method, this would require the View to store a good deal more information than it would need in to just render, and may put a lot of decision logic into the View's pre-rendering code. The determination of keywords would be HTML-specific, irrelevant to all other output forms, which would seem to make it more of a View issue, but to me, saving it all in an external Model Composition makes the process separation more successful, keeping the Views more focussed on rendering. Composition is in one place, rendering in another, instead of being tied together in a large render pile.

Sorry for the back and forth. I'm just working through the issues. Thank you for your help. Anonymous on Apr 30 2007, 22:13

nice article :),good work.

This is the right way to improve view layer of ZF. Reusable views can only be achieved by composition Anonymous on May 1 2007, 06:21

> Reusable views can only be achieved by
> composition

Yes 

A whole lot better than inheritance for example; The Composite pattern is pretty much the only solution for todays demands for the interactive internet.

But I'm asking myself why wasn't an implementation of Composite build into the ZF framework from the start... Anonymous on May 4 2007, 20:17

argh.. man what are you gonna do when you have a site with 1,000+ pages and you want to slightly change the position of the footer? you're gonna open all views and move the footer call? whatever happened to templating? Anonymous on May 9 2007, 03:24

@tencc. Why would I have 1000 different footer elements??? All you need is a single (or a handful) of base layout templates to set the tone for the overall website regardless of the number of pages. Even if each of the 1000 pages were unique one could just implement a Two-Step View with layouts to cut the duplication. Anonymous on May 9 2007, 04:58

Thanks for your interesting article. **Composite View** is very usable for views knowing how to get their data like helpers f.e.

But what about the "classical" philosophie of feeding data of the view by the correspondent controller via assign method? How get these views their data? Anonymous on Jul 20 2007, 23:33

I have developed a blogging website using PHP that is actually a Bangla Blog which covers lots of features of Web 2.0. But I cant figure it out, which key parts of Web 2.0 it have. Can you figure it out? Anonymous on Oct 17 2007, 14:40