In the summer of 2006 working for Java/Struts author Chuck Cavaness, I incrementally refactored dozens of Struts actions and in so doing implemented a "template method" without any prior knowledge of this pattern. Chuck had implemented a library for paginating through records, but about 15-20 lines of code for utilizing pagination were copied/pasted into dozens of Struts actions (in an existing application I was using as a reference), when only one datum was ever different, the column on which to sort. So, while figuring out how to utilize the pagination library in my new application, on my first refactoring I developed 3 or 4 methods to acheive the same effect as the duplicated code.

While convinced this was better, I was still dis-satisfied with duplicating 3 or 4 method calls in dozens of actions. So on my next occasion to work with pagination code, I developed a new method that called each of the previous methods in turn. This reduced what had been 15-20 lines of code in each action, to just 2: one for the method for specifying the column on which to sort, the other for the new method which in turn would call all the others.

My design was affirmed when I had occasion to read the source code for Struts' ActionServlet and noticed that it's "init()" method worked much the same way, calling specially designed methods one after another:

```
318     public void init() throws ServletException {
319
320         // Wraps the entire initialization in a try/catch to better handle
321         // unexpected exceptions and errors to provide better feedback
322         // to the developer
323         try {
324             initInternal();
325             initOther();
326             initServlet();
```

This inspired me to pick up my copy of Design Patterns for the first time in about 5 years. When I first tried reading it I only had 2 or 3 years experience, and the only methods I understood were the relatively simple ones: Singleton and the various Factory methods.

Upon reading Design Patterns again, I realized that I had stumbled upon the template method in my own design. After this the conceptual connections that I made occurred by leaps and bounds. For instance, the template method is where the notion of "inverted control" originated, *not* with IOC (Inversion of Control) containers. I also went on to use the State pattern, so that orders in different states could all be summarized on one and the same page, but different action/URL's would be made available to the user depending on the state the order was in. Not that this in itself is so impressive, but rather, an idea hit me along the way: doesn't the State pattern merely "decorate" objects with state-specific behavior?

Probably most significantly though, I developed a thread-safe way of programming to an interface using Struts (thread-safe through the notion of "thread containment" as described in Java Concurrency in Practice):

get/setMapping
get/setForm
get/setRequest
get/setResponse
actionForward
applyInput
exposeModel

The first four correspond to the four arguments required by Struts' Action's "execute()" method; these were all implemented in a default, abstract implementation. The next one, actionForward(), returns an ActionForward, just like execute(). So, under this paradigm, execute() delegates to actionForward(), and actionForward() in turn is a template method (again, implemented in a default, abstract implementation) comprised of two steps, the next two methods in the interface, applyInput() and exposeModel().

So in the end, only applyInput() and exposeModel() were to be concretely implemented. I developed these methods because I was unhappy with Struts' insufficient separation of concerns, applyInput() being strictly for gathering data which would then be used by exposeModel(), which was used strictly to set attributes on the request and/or session.

Chuck validated my design by comparing it to Smalltalk, the language in which MVC was first implemented, and the language with which Chuck programmed before Java. I have since refined this design, using counterparts to the 3 non-Struts methods above, to implement MVC frameworks in approx. 50 lines in both PHP and Python:

Struts	***PHP/Python***
actionForward	sendResponse
applyInput	applyInputToModel
exposeModel	applyModelToView

For PHP and Python I split the library into 2 interfaces, one for the controller, one for the "ModelTransferObject". The latter I believe to be a term I've invented, the concept however not necessarily being new: Spring WebMVC's "ModelAndView" abstraction actually qualifies as a "ModelTransferObject", but the name "ModelAndView" does not properly convey its purpose (neither do Struts' "Action" abstraction, nor it's "execute()" method: execute what? An action? That hardly clarifies matters).

Needless to say, successfully implementing my concepts in multiple languages only further validates my design and I am confident I would have no problem porting these concepts to other OO web platforms, such as Ruby and/or .NET. Then I believe I will have the basis for a publication I intend to be entitled "The Web and the Metamorphosis of MVC."