

# **Object Oriented Programming for Lasso Professional**

The following is a series of e-mail articles originally posted to CorralTalk ([www.corralmethod.org](http://www.corralmethod.org)) to discuss the basics of OOP and Lasso as I discovered them. There were discussions that brought out finer points, and the CorralTalk archives should be reviewed for those.

I can say that after rereading these, there are some things that need rewritten. There's some things explained a little awkwardly, or not entirely accurately. I tried to fix these with update notes. I did delete some sections that originally posed a bunch of rhetorical questions because I didn't know the answers then. These posts are largely my original submissions with some fine tuning to correct some errors, but I don't have the time right now to fix them all.

Having worked more with Lasso's custom types since writing these, I expect to someday compile a more practical set of examples, or you can refer to the FWPro framework at [www.fwpro.com](http://www.fwpro.com) once it is released. Until then, if you have any questions, bring them up CorralTalk or LassoTalk so the discussion is archived. There's several people now that have used custom types to leverage some OOP advantages, so there's more depth about the topic in the community now.

-- Greg Willits  
-- July 21, 2003

This document can be found at:  
[http://ldml.gregwillits.ws/downloads/oop\\_and\\_lasso\\_gwonct.pdf](http://ldml.gregwillits.ws/downloads/oop_and_lasso_gwonct.pdf)

# Article #1

**From:** "Greg Willits" <gw@gregwillits.ws>

**Date:** Sat Dec 14, 2002 2:05:21 AM America/Los\_Angeles

**To:** Multiple recipients of corraltalk <corraltalk@corralmethod.org>

**Subject:** OOP and Lasso

On CorralTalk, the question of just how do we take advantage of Custom Types has come up. Mostly in relation to Corral-based programming, but also overall. I said I would write out what I think I know and how I see the concepts of OOP relating to Lasso's new capabilities, and how that would compare to the traditional methods. This is my first installment.

I'm just learning about OOP myself and how it relates to Lasso, so, let's start with some terms.

## Classes and Objects

First, a class and an object are not the same thing. A class is the definition of, or the template for, creating an object. An object is a manifestation or instance of that class. To use a database analogy, a class is somewhat like a field definition. It defines the field, but unto itself it isn't anything that we normally manipulate. The data we put in the field is the object. It's a tangible manifestation that adheres to the rules of field definition. Each time we add a record we get a new data object based on that field definition, but there is still only one definition. It's the same with classes and objects. We create multiple objects—instances of data adhering to the class definition—but there is only the one class.

So, when we talk about rewriting our code to be object oriented. The majority of the discussion is really about creating classes, not objects.

In practical terms, a class is the collective program code that defines what is in the object ("attributes") and what it can do ("methods"). An object is the unit that is defined, or organized, by the class. A class can have attributes (data), methods (functions and procedures), or both. If it has both, this is termed an encapsulated class.

## Lasso and Classes and Objects

In Lasso, a Custom Type is the means we use to create a class. By programming a custom type, we have created a class. The local variables defined within that custom type (instance variables the LP docs call them) are the attributes of the class. The member tags of the type, created by programming custom tags within the custom type, are the methods for that class.

OOP allows for attributes of a class to be public or private. A public attribute is a data value that can be altered by any code in the application. It does not require

using the class method. In Lasso this means if the custom type creates standard variables of the \$ kind, then that variable can easily be altered by any LDML code. This would be a public attribute. If the custom type creates local variables of the # kind, then these cannot be altered outside of the code in the custom type. To alter that variable requires using the code within the custom type, most likely via a member tag. This would be a private attribute. It is important in the class design process to know which attributes will be public and private. For anyone who has played with custom tags, you've also had to consider those issues.

[**Update note:** as it turns out, Lasso does not support true private attributes. Even # local variables can be manipulated through some Lasso tags. This means that there is no way to guarantee private variables; however, using conventions such as \$ for public vars and # private ones, or naming conventions of the local vars in a custom type can at least establish what vars the programmer should consider private. For example, I have adopted using *fw\_* prefixes for all local vars in custom types which are private, and no prefixes for ones which are public.]

Likewise, methods can be public or private. From what I gather, I think all member tags of a Lasso custom type are public methods. That is, the method (a function or procedure) can be invoked using the LDML syntax at any time outside of the custom type. A private method would be a routine inside the custom type that is only ever invoked from within the custom type itself. I'm not sure that I understand the coding for custom types well enough yet to know whether there's a practical construct for a private method in a custom type. I suppose we could still write custom tags (member tags) and not document them, thus being unknown they're private, but that's just a facade.

So, some of the practical applications of OOP theory as implemented in Lasso are these:

- a class is a Lasso custom type
- a private attribute is a Lasso local variable within a custom type that *should* only be modified by code within the custom type, but there is no native way to enforce this. These vars should *not* be altered using the syntax of `$object->'attribute'='newValue'` or other Lasso tags
- a public attribute is a Lasso variable originally set by a custom type but is available for the application to freely modify using normal LDML to alter that variable at any time such as `$object->'attribute'='newValue'`
- a method is Lasso custom type member tag. All member tags are public methods, and there appears to be no way to create private methods, short of not documenting them, and compiling the code to a LassoApp.

## Creating Objects

To create ("instantiate" in OOP-speak) an object means to create a variable using the custom type. For example if we have a custom type called `FW_AuthdUsr` (FW for my usual Framework prefix, and AuthdUsr to be read as Authorized User), then

we have a class called `FW_AuthdUsr`. To create an empty object from that class, we would use this code:

```
[var: 'crntUsrInfo'=(fw_authdUsr)]
```

To initialize an object with some data we would either have to supply the data as parameters, or supply some reference that the custom type would then use to acquire or generate data. In our example above, we might pass a user's login name and password as parameters:

```
[var: 'crntUsrInfo'=(fw_authdUsr: -nm=$logname, -pw=$logpass)]
```

The code in the custom type would then query a database to search for this user (let's ignore for now how it knows what database and table to search). Assuming it found a user, the code in the custom type would then load local variables with user profile data from the table fields. We now have an object named `crntUsrInfo` of the class `fw_authdUsr`.

Let's now say that the profile data we collected included whether the user was authorized, first name, last name, email address, and favorite color in local, or instance, variables:

```
local: 'usrAppvd'=true;  
local: 'usrNameFirst'=(field: 'first_name');  
local: 'usrNameLast'=(field: 'last_name');  
local: 'usrEmail'=(field: 'emailAddr');  
local: 'usrFavClr'=(field: 'fave_color');
```

The names of the local variables are now private attributes of the object. To extract that data from the object, we typically would refer to each local var with the following syntax:

```
[$crntUsrInfo->'usrAppvd']  
[$crntUsrInfo->'usrNameFirst']
```

The use of quotes around the variable name is not required in LP6 (it is in LP5), but I prefer to use them to distinguish between variables and member tags, the latter of which does not use quotes.

After creating our object by setting the `$crntUsrInfo` variable, we might now want to store some session data to that user's record. To do that we would write a custom tag within the custom type and call it `storeSession`. To execute that method of our object, we'd use this syntax in Lasso:

```
[$crntUsrInfo->storeSession]
```

This would invoke the custom tag `storeSession` within our custom type which would return perform whatever tasks we had defined. Parameters can also be used as with any custom tag:

```
[$scrntUsrInfo->storeSession: -vars='basketID']
```

## Why use Custom Types / OOP

So, assuming I have everything right so far [**update note**: I've edited several details for correctness for this PDF], what does this all mean to the Lasso programmer? Well, as to the larger issue of why use OOP, there's likely far more authoritative texts than this one to help you understand that. I'd recommend looking for some good books / web site for that one. Still, for a Lasso programmer, why use custom types instead of "normal" methods to accomplish the same thing?

Let's step back and look at a custom tag for a bit. Many of us have used them and programmed them. Custom tags are really just a more convenient syntax method of implementing an include. Instead of doing this:

```
[var:  
  'x'='whatever',  
  'x'='whatever',  
  'x'='whatever',  
  'x'='whatever']  
[include: $mylibspath + 'myspiffylib.inc']
```

You get to do this:

```
[MySpiffyLib:  
  -x='whatever',  
  -x='whatever',  
  -x='whatever',  
  -x='whatever']
```

I find the custom tag's parameter based syntax shorter, easier to read, and easier to explain in documentation (be it for source refc or training another programmer). In documenting the tag, it is a simple issue of defining the tag name, identifying the parameters, and explaining their purposes. There's no need to explain the supporting syntax, what has to be where, etc. In some cases this may not be vastly different than explaining a traditional variable/include approach, but for complex tags, the custom tag syntax prescribed by Lasso takes care of half of what you need to know. The advantage of custom types, at least at one level, is somewhat along that line.

In my current code, to authenticate a user I call an include:

```
library: ($c_gLibsPath + 'fw_glbl_getProfile.inc');
```

The include routine searches a database, and initializes a number of regular variables like our example of user profile attributes above. From that point on I use those variables. So far, the custom type method hasn't really saved me a lot of coding. The above include is simple enough to write, and actually

```
if: ($scrntUsrInfo->getAppvd) == true;
```

is longer to write than

```
if: $usrAppvd == true;
```

So why would I use custom types? Well, maybe one comparison of a simple object as this, it doesn't seem compelling. But understanding how all this works, I could now document and explain the use of all the code involved much more simply. For example:

"Class fw\_authdUsr is used to define an authenticated user's profile. The class has the following attributes: usrAppvd, usrNameFirst, usrNameLast, usrEmail, and usrFavClr. It has the following public methods: getAppvd, getNameFirst, getNameLast...."

Now from that simple description you already know how to implement using all the code. I haven't described the data types of each attribute (string, numeric, etc), nor specifically what each method returns, but hopefully you can see that documenting the use of an object, at least to me, seems to be a significantly more straightforward affair. The syntax is prescribed by Lasso. Once you know how to write syntax for one class, you can write for any class.

This to me is at least one compelling reason to use custom tags: the increased standardization of the syntax in dealing with my application's functions and data sets, and the relative simplicity of documenting the class vs all the raw inline code. If you can imagine a whole bunch of variable declarations and includes replaced by custom type syntax, I think you can see that the latter would likely be a lot cleaner to read.

Now, if you can imagine walking into a full application's source code, I can see it saving a massive amount of time if large chunks of the code were written as declarations of objects and execution of custom type member tags. The many conditional flag webs that we weave would be reorganized as attributes of classes. Think of how little custom inline code would actually have to be deciphered. You'd never need to look inside the class code unless you were going to modify the class. With a straightforward reference to the attributes and methods of each class, you'd have a self organized reference to most of what was going on in the application.

From what I see so far, a major advantage of OOP is the simplification of the code we actually deal with directly to manage complex operations and data sets.

[**update note:** I don't know how I missed it before, but the major reason to use custom types is the stability and reusability of the code. If you follow appropriate

techniques which require parameters for all inputs to the type, then the type becomes a black box. Feed it inputs, you get outputs and the code inside is protected from the rest of the application. This fosters significant application stability and code reuse which allows for faster application development.]

## **Where Should We Use Custom Types**

I suppose the extreme view is everywhere and for everything. I know nothing about Java and Objective C, but I get the impression they're completely object oriented. Everything is an object. Mac OS X is written that way, and from what I understand that's at the root of some of the downside to OS X as well as the upside. On the upside, writing an OS X app can be done very quickly (assuming you're not having to convert legacy code). I saw this watching NeXT for several years. The downside is that *everything* is an object. So there is a layer of overhead in the communications of code chunks to each other. This can cause performance degradation.

As with several other options in Lasso, we have to analyse the tradeoffs between rapid development and optimum performance (I talked about that quite a bit at Lasso Summit). We have the ability to follow several options to possible strike a balance.

Using OOP does not inherently result in good programming anymore than using PageMaker will inherently give you better page design than using Word. With OOP you are just as empowered to write crummy code. An interesting comment I read was that knowing a little OOP and implementing a little OOP can be worse than not using any. By using it a little bit here and there you end up with an application that's difficult to interpret by people that know OOP and people that don't know OOP. An interesting thought.

For Lasso / Corral web apps, I see there being three distinct arenas to apply OOP via custom types and tags.

First is our application data—customers, suppliers, users, catalogs, carts, etc. Each of these data sets can be defined as a class with specific attributes and methods to allow manipulating the data.

Second is presentation code—display items could be implemented as classes. The whole of Cocoa is done like this. Each GUI object such as windows, menus, buttons is a class. Let's say you have a new headline section. This could be implemented as a method as part of news application class. Or, it could be genericized as a presentation item that displays titles and dates. Whether the data comes from news records, articles, or uploaded files list doesn't matter, the class is a "headlines" class for which objects could be created from any data set. Therefore not only could we create a number application data classes, we could also create a number of presentation classes for lists, tables, popup menus, etc.

Third is our core framework code—the internal application management data and routines to display and manage menus, navigation, display preferences etc. This would include the Corral layer—stubs, templates, pageblocks, pages might gain advantages from being reconfigured in an object oriented approach.

I have spent most of my time thinking about the first two so far. With my latest Framework being structured in a modular approach, I have also taken the first step to treating a whole web site section such as News, Code, Articles as a single, rather complex, class. One class that might contain all methods of data management, presentation, and navigation using a separate configuration table (which would be classified as public attributes to the class).

### **Application Data as Objects**

I think if we substitute the `fw_authdUsr` example for any of our typical application data structures (suppliers, customers, catalogs, articles, news stories, libraries) we can see that defining a class wouldn't be too tricky.

However, implementing multiple database records as objects puzzles me a bit. Let's take a simple example of displaying a found list table of 10 records. Currently a simple inline is used, and we can use `[records]` to handle looping through each record and directly displaying the fields we want.

How do I do that with objects? Assuming I have a class defined, do I really go through the overhead of converting every record to an object, then calling methods to extract the field data? This sounds very inefficient compared to the traditional LDML way. Or, do I create a method within the class to generate the table which would in turn simply execute the normal code way of doing things? Generating the table itself would be bad design having embedded the presentation into the method.

One of the principles of good OOP, it seems to me, is a high degree of private attributes. That is the data in an object should be modifiable and available by communicating with the object. So, having Lasso code that uses objects to manipulate individual records where it is easy, but reverting to standard LDML to work with groups or records seems to defeat a lot of the purpose of using OOP.

[**update note:** I have since worked through this for a few objects, and this needs a whole new article to discuss. In short, I have created methods which merely create a Lasso named inline. The name of the named inline is stored as an attribute, but rather than duplicate the data into arrays, maps, or some other structure, I have been leaving the data buckets as named inlines. This requires less processing and less RAM. Certainly there are occasions where a series of maps or arrays is necessary to perform application-specific data manipulation. If you need that, then do it. Otherwise, just to show a list, just use a named inline.]

## **LDML Inlines and Includes as Methods**

We talked about classes having the built-in code for performing actions on the data of the class. But, what about when multiple classes could all use the same code? If one of the purposes of any good modern code base is to eliminate redundant code, do we write the same code over & over as tags within our custom types?

Each of us has written general purpose LDML routines. I have some that abstract the process of acquiring and inserting data into LassoMySQL. The same code is used for any table using small configuration data sets to make the routines adapt to unique tables on the fly. Should I replicate this code in every custom type? It seems that I would want to keep only one instance of that code for all the same reasons I do it now (lower maintenance, debugging, etc).

Some cursory views of various sources lead me to believe that OOP allows for standard general purpose routines to be compiled into common libraries that are shared by multiple classes. Therefore, rather than write those routines internal to each custom type, I would simply reference those routines with a normal LDML include statement or custom tag. It's simply the case that the library containing that code must be present in order for the classes to be complete. This appears to be normal procedure, and certainly makes a lot of sense. We get all the benefits of standard general routines, but we would now indirectly reference them through classes.

### **Update Note:**

Here's how I solved that one. I rewrote each of the original include files as standard custom tags. One for addPrep, add, deletePrep, delete, updatePrep, update, and others. I then wrote a custom type with a number of predefined attributes, and member tags correlating to the actions of the custom tags. Within the member tag definition I used the local attributes to feed to the custom tags. It sounds redundant, but isn't. I use the custom type format to contain several attributes about each object instance, but I defer much of the manipulation code to a shared library of custom tags. I can create any number of custom types with different sets of attributes and methods, but for those common database access actions, I maintain only a single set of shared library tags.

### **What This Means to FrameWork**

If FrameWork existed entirely as suite of classes, using that suite as an API toolkit would be vastly simpler than using it today where you feel compelled to understand every line of the source code. Additionally, to add a particular feature to a routine you could use the OOP approach of inheritance to create a new class based on one of my existing classes and add your routine as method unique to that new class. This would be significantly easier and more reliable than starting with my source code and rewriting it.

I have a lot of work to do, but I think I see the majority of exactly how to implement this now. Initially I see starting with some core routines and data structures, and application modules. Then I can take on looking at the whole Corral structure to see if it should be restructured as custom types. Does the "page" become the fundamental class? Not sure.

[**update note:** several months have passed and I have indeed rewritten the FrameWork Pro I released in early 2003 as several API toolboxes of custom tags and types. Have a look at [www.fwpro.com](http://www.fwpro.com). I have also decided that core page objects are not worth defining as objects. It adds unnecessary layers to the overhead of creating the page. See my comments at the beginning of the Article #7.]

### **Feature Needs from BlueWorld**

I haven't used all this stuff yet, but I see one major feature that would go a long way to encouraging the use of these advanced features, and the building of truly plug and play reusable code amongst developers (we tend to share, but then rewrite each other's code all the time!). That feature is the ability to load libraries into RAM and have them stay there without restarting LassoService. Sticking a library in LassoStartup is great. But people on shared servers need a way to load libraries and have them remain in RAM without reloading the library on every page.

[**update note:** Doh! LassoApps do that already! However, LassoApps are server wide regardless of where they are loaded from. So what we really need is for Lasso to load LassoApps from a virtual host and have that code scoped to just that domain. This way different sites can run different versions of LassoApps. I can run my API as a LassoApp, but when I want to update the API, I might have to update all my sites on that server.]

Well, I think that covers the current status of what I think I know, and what I think I'm going to do to explore using Lasso in an OOP approach. The real question is how extreme to take it.

Questions, corrections, rebuttals, and extensions to this discussion are encouraged.

## Article #2

**From:** "Greg Willits" <gw@gregwillits.ws>

**Date:** Fri Dec 20, 2002 1:27:16 AM America/Los\_Angeles

**To:** Multiple recipients of corraltalk <corraltalk@corralmethod.org>

**Subject:** Re: OOP and Lasso

To repeat my disclaimer: I'm figuring this all out myself and reporting as I go. Feel free to correct, rebutt, and extend...

I've been doing a fair bit of reading (and still have a lot to go), but I've got a few more basics covered, and some rudimentary code to show several OOP principles and how Lasso implements them.

I've been researching to find more examples of practical implementations, and philosophies of design specific to the web etc. Pretty difficult so far. Everything is very fundamental tutorial oriented. Most OOP commentary is oriented to standard desktop apps. So it is hard to see how others have put it to work in a larger scope for a web application. But I'm getting some good ideas.

### **Abstract, Super, and Sub Classes**

First, a little more on classes before we get to the new stuff. There are different kinds of classes, more or less based on what functionality they contain. A basic class (I haven't found any particular adjectives) is as we discussed before, a template of attributes and methods for creating objects. However, there is also an abstract class. This kind of class is not for creating objects, but for creating other classes. It's a generic class template for creating more detailed classes.

Let's say we wanted to create classes for various geometric shapes. We'd create a class for a square, a circle, a triangle, etc. Each of these shapes might have attributes like a fill color, line color, line weight, transparency. The circle would need a diameter, the square a set of corner coordinates. Plus each shape would need methods to return vales, set values, and draw the shapes.

Well, I think you'll readily see that the shapes have a lot in common, yet also some unique features. To minimize the redundant code, and maximize code reuse, we want to gather up the common attributes and methods into something that can be shared. That something is an abstract class. Let's call that class *shape*, and in it we'll put the fill color, line color, line weight, and transparency attributes, and the methods to get and set those attributes, and a draw method.

Here's what that would look like (in part) as a Lasso custom type:

```
define_type:'shape';

// our attributes

  local:
    'fillcolor'=(named_param:'-fillcolor'),
    'linecolor'=(named_param:'-linecolor');

// our methods

  define_tag:'fillcolor';
    return: self->'fillcolor';
  /define_tag;

  define_tag:'linecolor';
    return: self->'linecolor';
  /define_tag;

  define_tag:'draw';
    return: null;
  /define_tag;

/define_type;
```

Now, we can't make an object from `shape`, it is too generic. There's no size, no location, no description of the actual shape. This is not a class that can create a concrete object as there are not enough details. In fact, while we have created a draw method, it can't do much because we don't know what to do based on the attributes. But we add it anyway, because all our shapes need a draw method. That is common to all of them. This is an abstract class. So what purpose does it serve?

## Inheritance and Sub-Classes

This is where inheritance plays a vital role in OOP. Inheritance is the ability of a class to acquire / use / inherit all the features of another class. We make a first class with some features. Then we make a second class and tell it to use/inherit all the features of the first class, plus we usually add a few more so it becomes a new unique class.

I said an abstract class was a template for more detailed classes. The next step in our shapes example is to make sub-classes from the `shape` class. We want to create a new class called `circle`, and have it inherit all the features of the `shape` class. We don't need to rewrite the code for all the `shape` features because the `shape` class already has that code. For `circle`, all we have to do is add a radius attribute and a specific method to draw a circle.

In Lasso, we'd define another custom type like shown below. Note that in the first line we identify the current custom type (class), but we also reference the `shape` class. In OOP-speak this is a superclass. In Lasso-speak it is a parent custom type. The custom type `circle` is a child type in Lasso, or a subclass in OOP. Notice also that none of the code from `shape` is repeated exactly, but we have some new code. The `circle` class will actually use the `shape` code as though it were written within the `circle` custom type. Yes, we do have `draw` which was in the `shape` class, but it is not exactly the same code. We'll come back to this in a bit. So, even though we did not write a `linecolor` tag for `circle` we can still use the `linecolor` method because `circle` inherited everything that `shape` has.

```
define_type:'circle','shape';

    local:'radius'=(named_param:'-radius');

    define_tag:'radius';
        return: self->'radius';
    /define_tag;

    define_tag:'draw';
        return: 'My linecolor is ' + self->'linecolor';
    /define_tag;

/define_type;
```

A parent / superclass does not have to be an abstract class. It can be full fledged class. I think the experienced would say that all top level classes should be abstract classes as a matter of good design, because that inherently drives the creation of generalized reusable code.

In OOP languages, it appears there is specific syntax for abstract and regular classes. The reasoning is that if an abstract class isn't specific enough to make objects from, then the language syntax is such that one *can't* make an object from an abstract class. Lasso does not differentiate them, and our ability to create abstract classes is merely through how we use a custom type. Lasso will not prevent me from making an object from the `shape` class.

## Polymorphism

This is essentially the ability of a class to have unique behaviors for each subclass. We want to use the same method name and syntax, but have it generate different results. In our example, all `shape` classes need a `draw` method. However, the subclasses of `circle` and `square` need a unique result when the `draw` method is invoked. This is accomplished by having the subclass override the method of the superclass. It sounds complicated, but is very simple in practice.

Look above again at the circle class. You can see that there is now some specific code for the draw method (member tag in Lasso-speak). Now let's look at a square subclass example:

```
define_type:'square','shape';

    local:
        'topleft'=(named_param:'-origin'),
        'btmright'=(named_param:'-drawto');

    define_tag:'origin';
        return: self->'topleft';
    /define_tag;

    define_tag:'drawto';
        return: self->'btmright';
    /define_tag;

    define_tag:'draw';
        return: 'My fillcolor is ' + self->'fillcolor';
    /define_tag;

/define_type;
```

Look again at the `draw` tag. It has different code yet again. In OOP, a method in a subclass that is defined with the same name as one in the superclass will supercede the actions of the superclass. In Lasso, the member tag definition in the child type square will supercede, or override, the one in the parent type shape. So, we have created a common syntax by using the draw method for all shapes, but each unique shape subclass (child type) generates a unique result. That, in a nutshell, is polymorphism.

So with the above custom types and member tags, what's the Lasso syntax to use them?

```
// create a new object mySquare based on the square type

[var:'mySquare'=(square: -fillcolor='green', -linecolor='red')]

Object: [$mySquare->type]<br>
[$mySquare->fillcolor]<br>
[$mySquare->linecolor]<br>
[$mySquare->draw]<br>

<br>
<br>

// create a new object myCircle based on the circle tag
// notice that we never create an object based directly on shape
```

```
[var:'myCircle'=(circle:  
  -fillcolor='blue',  
  -linecolor='yellow',  
  -radius=6)]
```

```
Object: [$myCircle->type]<br>  
[$myCircle->fillcolor]<br>  
[$myCircle->linecolor]<br>  
[$myCircle->radius]<br>  
[$myCircle->draw]<br>
```

```
[if: $myCircle->radius > 5]  
  My radius is bigger than 5.  
[/if]
```

You can see above that we use Lasso's predefined `->type` member tag on our custom types. There are other member tags that also work with custom tags. (see the docs).

You can see also that even though we never wrote `fillcolor` and `linecolor` tags for `square` and `circle`, we nevertheless can use those member tags because they are inherited from the parent type `shape` that each custom type was based upon.

Also, you see that the `->draw` member tag produces unique results for each custom type. Using the `->draw` name makes it easier to remember and document the methods for the types. We could use `->drawcircle` and `->drawsquare`, but that just makes more to remember and document. The concept of polymorphism helps to simplify the interface to our objects.

## Overloading

If you were watching the "`Array->(contains: 'substr')`" thread on LassoTalk you saw the topic of overloading. This is where a custom type (class) can customize the meaning and behavior of expression symbols like `+` `-` `++` `==` `+=` etc to suit the needs of the data structures within a class. I believe the example in the docs (or somewhere) was that if we had a class that was a list of contact info for people, then the `+` symbol could be used to add a new person to that list. It would not be a string concatenation, but rather the insertion of multiple criteria into an array (or whatever). The expression `$myContactList += $currentPerson` would take chunks of data in `$currentPerson` and append it to the data structure of `$myContactList`. That's pretty cool. Assuming these are both objects of custom types, imagine the simplicity of `$myContactList += $currentPerson` replacing lines upon lines of array manipulation. (Those lines exist in the custom type code of course, but you never have to see them!).

Speaking of overloading...it's late, and my brain is definately overloading.

This was a single pass written effort, so I hope it's coherent. I hope that helps someone see through the buzzwords (which I'll let you research on your own) into some practical Lasso syntax. Some day I'll collect all this and clean it up and expand on some details I know I'm leaving out.

This is getting cool, eh! I think I'm an OOP convert. I like the logic.

(corrections, additions, rebuttals encouraged).

## Article #3

**From:** "Greg Willits" <gw@gregwillits.ws>

**Date:** Sun Dec 22, 2002 12:29:58 AM America/Los\_Angeles

**To:** Multiple recipients of corraltalk <corraltalk@corralmethod.org>

**Subject:** Re: OOP and Lasso

### Installment #3: Inheritance and Composition

(notice I'm spelling inheritance right this time :-P)

Inheritance, as we've seen, allows a subclass to acquire all the attributes of its superclass(es). So, we know we can create an abstract or superclass called `automobile` with attributes common to all autos, then create subclasses which define unique species such as a `truck` subclass and a `station wagon` subclass, etc. Each of those can have subclasses to define models and manufacturer, etc.

One of the concepts of inheritance is that each subclass is a member of its superclass(es). If we were to take a Porsche 911 Turbo, that specific model "is a" sports car. And a sports car "is a" automobile. We can even make leaps like the 911 model "is a" automobile and the statement is still true (improper grammar intended here to keep our "is a" phrase constant).

This structure is useful when there are numerous objects with numerous shared attributes, yet are ultimately unique like our auto example.

Regardless of how many subclasses we design to minimize redundant code, it ultimately brings us to the complete detailed model we want to work with, which results in a single object with a number of inherited attributes and methods. Despite the number of attributes, there is still only one end object.

What about when we want to deal with objects that are made up of multiple objects? A car is not a single object, really. It is an aggregate of numerous objects: doors, tires, seats, windows, engine, etc. How do we represent that?

Now that we have inheritance figured out, and know that Lasso's custom type allows multi-class inheritance, we need to look at a complimentary concept called composition.

### Composition

As its name implies, composition is the concept of groups of objects. A composite of objects which work together as a whole, and can be addressed as a whole, yet retain the individuality of its component objects.

Now we want to look at automobiles in a different way, as an assembly which gets its definition from multiple objects. We could have a chassis object, an engine

object, a door object, a seat object, etc. Our automobile becomes a composite object made up of smaller component objects. Each of those objects, like our engine, could be composite objects too (crankshaft, pistons, valves, cams, etc).

Where inheritance uses "is a" to make its connections, composition uses "has a" to make its relationships. In our composite car above, our Porsche 911 Turbo "has a" door, and "has a" chassis, etc.

So to represent composition, we need some way to have a single object (meaning a single variable in Lasso) contain multiple objects (custom types). In Lasso, the only way I can see to represent composition is with maps and arrays. At the moment, it seems likely that a map will be preferred most of the time, but I might be wrong. Thus:

```
[var:'myComposite'=(map)]  
  
[$myComposite->(insert: 'triangle'=$myTriangle)]  
[$myComposite->(insert: 'square'=$mySquare)]  
  
[$myComposite->(find:triangle)->fillcolor]
```

or, for our discussion:

```
[var:'myPorsche'=(map)] (yeah, I wish it was _my_ Porsche!)  
  
[$myPorsche->(insert: 'engine'=$myEngine)]  
[$myPorsche->(insert: 'turbo'=$myTurbo)]  
[$myPorsche->(insert: 'tires'=$myTires)]  
  
[$myPorsche->(find:engine)->horsepower]
```

## **Inheritance vs Composition**

So, if we needed an object for a Porsche 911 Turbo, would we represent it with classes and subclasses inheriting features as attributes, or with a composition of objects acquiring features through aggregate objects? Technically, it could be done either way.

From some reading it appears there's a pendulum going on with respect to inheritance vs composition. Inheritance has always been a founding concept of OOP, but it seems there's some whipper snappers that believe inheritance is bad design (I forget why, but they had a seemingly good reason), and composition is how things should be done.

Generally, it would appear that in the middle ground of the debate we'd want to use inheritance and subclass structures to define classification and set membership, and composition to put together objects which contain independent, manipulatable components. It seems that tests as simple as "is a" and "has a" are used to help

decide. If the end result object is described in terms of "is a" with respect to a class then the relationship is inheritance. If it is more natural to say "has a," then the relationship is composite.

For lots of examples and tips, this is where the many books by people smarter than me would be worth consulting :-)

As we design our object models and interfaces, we have to not only determine how to divide up our subclasses efficiently, but also decide what will be represented by inheritance vs composition.

Here's some basic code to fiddle with:

```
<?LassoScript
//[
define_type:'shape';

    local:
        'fillcolor'=(named_param:'-fillcolor'),
        'linecolor'=(named_param:'-linecolor'),
        'linewidth'='1.5pt';

    define_tag:'fillcolor';
        return: self->'fillcolor';
    /define_tag;

    define_tag:'linecolor';
        return: self->'linecolor';
    /define_tag;

    define_tag:'draw';
        return: null;
    /define_tag;

/define_type;

define_type:'square','shape';

    local:
        'topleft'=(named_param:'-origin'),
        'btmright'=(named_param:'-drawto');

    define_tag:'origin';
        return: self->'topleft';
    /define_tag;

    define_tag:'drawto';
        return: self->'btmright';
```

```

        /define_tag;

        define_tag:'draw';
            return: 'My fillcolor is ' + self->'fillcolor';
        /define_tag;

/define_type;

define_type:'circle','shape';

    local:'radius'=(named_param:'-radius');

    define_tag:'radius';
        return: self->'radius';
    /define_tag;

    define_tag:'draw';
        return: 'My linecolor is ' + self->'linecolor';
    /define_tag;

/define_type;

define_type:'triangle','shape';

    define_tag:'draw';
        local:'side'=(named_param:'-sidelen');
        return: 'My side length is ' + #side;
    /define_tag;

/define_type;

define_type:'energy';

    local:'power'=(named_param:'-power');

    define_tag:'power';
        return: self->'power';
    /define_tag;

/define_type;

define_type:'multiclass','circle','energy';

    define_tag:'draw';
        return: 'My linecolor is ' + self->'linecolor' + ' and energy is '
+ self->'power';
    /define_tag;

/define_type;

```

?>

```
[var:'mySquare'=(square: -fillcolor='green', -linecolor='red')]
```

```
Object: [$mySquare->type]<br>  
[$mySquare->fillcolor]<br>  
[$mySquare->linecolor]<br>  
[$mySquare->draw]<br>
```

<br>

<br>

```
[var:'myCircle'=(circle: -fillcolor='blue', -linecolor='yellow', -radius=6)]
```

```
Object: [$myCircle->type]<br>  
[$myCircle->fillcolor]<br>  
[$myCircle->linecolor]<br>  
[$myCircle->'linewidth']<br>s  
[$myCircle->radius]<br>  
[$myCircle->draw]<br>
```

```
[if: $myCircle->radius > 5]  
    My radius is bigger than 5.  
[/if]
```

<br>

<br>

```
[var:'myTriangle'=(triangle: -fillcolor='purple', -linecolor='black')]
```

```
Object: [$myTriangle->type]<br>  
[$myTriangle->fillcolor]<br>  
[$myTriangle->linecolor]<br>  
[$myTriangle->(draw: -sidelen=9)]<br>
```

<br>

<br>

```
[var:'myMulticlass'=(multiclass: -fillcolor='blue', -linecolor='yellow', -  
radius=6, -power=8)]
```

```
Object: [$myMulticlass->type]<br>  
[$myMulticlass->fillcolor]<br>  
[$myMulticlass->linecolor]<br>  
[$myMulticlass->radius]<br>  
[$myMulticlass->draw]<br>
```

<br>

<br>

```
[var:'myComposite'=(map)]
```

```
[$myComposite->(insert: 'triangle'=$myTriangle)]  
[$myComposite->(insert: 'square'=$mySquare)]
```

```
[$myComposite->(find:triangle)->fillcolor]<br>  
[$myComposite->(find:triangle)->'linewidth']
```

## Article #4

**From:** "Greg Willits" <gw@gregwillits.ws>

**Date:** Thu Dec 26, 2002 2:07:32 PM America/Los\_Angeles

**To:** Multiple recipients of corraltalk <corraltalk@corralmethod.org>

**Subject:** Re: OOP and Lasso

### Installment #4: Classes vs Prototypes

When I was in high school, we had a biology teacher decide to take over chemistry (or had to, not sure). I had him the second year he did this, but I had friends who had him the the first year. Apparently he only had part of the summer to prepare, and during the year he'd be doing his studying and preparation not too far in advance of teaching his class. Legends grew about how lesson B made no sense based on lesson A, and this teacher coming in and having to reteach lesson A because he'd gotten it all wrong. As a student, I felt for the students. As I grew older and took on the role of a teacher in several areas, I began to identify more with the teacher!

Well, folks, you're my first year object oriented programming students :-)

We're going to backtrack a little. Not about the basics, but about how Lasso is implementing object oriented programming. I've had a few short exchanges with Kyle, and he's helping to point me straight with respect to Lasso's take on OOP. I'm sure I don't have all the details yet, but a major interesting feature popped up that is worth straightening out.

So far, I have compared Lasso's custom types to classes. Easy to do. Much has been written about OOP and classes, so naturally we assume they're a fundamental part of OOP, and therefore want to find something in Lasso that looks like them. Truth be told, classes are not a required feature of object oriented programming. Classes are one means by which to implement the core attributes of OOP which are: encapsulation, inheritance, polymorphism, and overriding. All of which we have talked about. There are other ways to embody these principles.

An alternative and intentionally different technology for object implementation is a structure called a prototype. Notice this is a noun not a verb. We're not talking about the methodology of rapid interface construction and such. Rather, think in terms of a development prototype. An initial object design from which other objects will be designed—incorporating many of the same concepts of the original object, yet taking on new design features as well.

Explaining the difference between a class and a prototype in a few short sentences gets quite tricky. The differences are subtle, though quite meaningful from an engineering standpoint, but at first it will seem to be purely semantics. Also, at this stage it doesn't appear that Lasso implements all the features that prototypes offer (but I still need clarification on details). Additionally, prototypes can be implemented to look like classes (which we have actually already done). So, right

now it is difficult to give LDML examples to show the differences between prototypes and classes. However, Kyle said that as Lasso progresses, we will see more prototype features. The bottom line is that Lasso will evolve as a prototype based language and not a class based language. This is going to be very important for two reasons:

a) we don't want to get too hung up on how classes work in other languages and then compare Lasso to those standards and gripe about what it "lacks." That'd be like saying CART cars are racing cars, then griping NASCAR cars don't have wings, so they must not be "real" race cars. (Hence some of the prior complaints that lasso is not a "real" object language).

b) we want to study prototypes as much or more than classes so that we more fully understand how to take advantage of what they offer, and how to more appropriately address our feature requests to BW. (Otherwise we'll start asking for wings on NASCAR cars, and they just don't work that way).

Some highlights of prototypes that I found interesting include:

- they're designed to be more reflective of representing knowledge and categorizing that knowledge as we acquire it. Classes require that you define your world in advance, and break it apart into well established categories and differentiations through the abstract and subclasses. This must all be well defined prior to running a program. The guys behind prototypes say, "what if I don't know all those details yet. I need to be able to refine my objects as I acquire more knowledge." Prototypes are apparently able to be defined on the fly, where as classes can not (AFAIK).

- class based objects apparently are stored as the sum of all inherited classes. So, an object `myDobie` has all the variables defined by the abstract `dog` and subclass `guardDogs` and subclass `breed` etc needed to define a specific species, breed, color, etc all the way down to our instance. All those variables are stored within the `myDobie` object. Thus there is redundancy in the data.

For prototypes, each object only contains the unique data that separates it from the prototypes it inherits information from. So, all the info germane to all `dog` objects is kept in the `dog` object only. As we define specific breeds, the breed object contains the data that is unique for that breed. As we get to `myDobie`, that object only contains the name and birthday attributes (for example). Any request for the generic data such as breed is passed on to the breed object to respond to.

The advantage of the class structure is that all data is self contained, and data management performance can be more efficient. The downside is that it consumes a great deal more memory.

Prototypes end up sending more messages between objects (a performance overhead), but are very small, so more of them can fit into a given RAM space (which can mean more objects in RAM vs virtual RAM on disk). They can also be created and destroyed faster because they are smaller.

A generalization is that prototypes are better for systems that need to create many objects for a short period of time (less RAM footprint and less work to build them). This IMO means that prototypes are probably much better suited to web

applications where numerous vars and objects need be created and recreated with each page.

I found several references to prototypes, but I like this one the best so far:

<http://lieber.www.media.mit.edu/people/lieber/Lieberary/OOP/Delegation/Delegation.html>

It's lengthy, but was the first detailed explanation that really showed the differences between classes & prototypes.

### **Here's the response/code that Fletcher subsequently posted:**

The built-in data types use prototypes, but custom data types do not. A custom data type is serialized with the definitions of all of its member tags. This allows a custom data type to be unserialized anywhere, even on another Lasso server, with all of its member tags still available.

If you are creating a whole series of objects and are not worried about them being able to be unserialized anywhere then you could create your own prototype system where the member tags of the type are references to member tags stored elsewhere. For example, you could use the unknown tag functionality so your type has only one tag that dispatches to a global type.

Here is a quick example of how to create a prototype for a custom type. It's pretty simple.

All the custom tags are created as compound expressions and stored in a map in a global variable called 'prototype\_object'. This and the definition of the custom type 'object' would probably be in Lasso Startup so they are available on any page in the server.

For comparison purposes the custom type 'object' is defined using a prototype and the custom type 'object2' is defined without using a prototype. The size of the serialized object, speed of creation, and speed of serialization / unserialization (session performance) are tested.

The prototyped object tends to win. It saves about 40% in serialized size, is created about 33% faster, and serialized / unserialized about 33% faster. I imagine that with a more complex data the difference in performance between the two methods is going to widen.

```
<?LassoScript
```

```
Global: 'prototype_object' = (Map:  
  'alpha' = { Return: Self->'storage'-(Get: 1); },  
  'beta' = { Return: Self->'storage'-(Get: 2); },  
  'gamma' = { Return: Self->'storage'-(Get: 3); },  
  'onConvert' = { Return: String: Self->'storage'; }  
);
```

```
Define_Type: 'object';
```

```
  Local: 'storage'=(params);
```

```
  Define_Tag: '_unknowntag';
```

```
    Return: (Global: 'prototype_object')->(Find: Tag_Name)->(Run:  
      -Name=(Tag_Name), -Params=(Params), -Owner=(Self));
```

```
  /Define_Tag;
```

```
/Define_Type;
```

```
Define_Type: 'object2';
```

```
  Local: 'storage'=(params);
```

```
  Define_Tag: 'alpha';
```

```
    Return: Self->'storage'-(Get: 1);  
  /Define_Tag;
```

```
  Define_Tag: 'beta';
```

```
    Return: Self->'storage'-(Get: 2);  
  /Define_Tag;
```

```
  Define_Tag: 'gamma';
```

```
    Return: Self->'storage'-(Get: 3);  
  /Define_Tag;
```

```
  Define_Tag: 'onConvert';
```

```
    Return: String: Self->'storage';  
  /Define_Tag;
```

```
/Define_Type;
```

```
?>
```

```
<b>Object Test</b>
```

```
[var: 'test' = (object: 'aaa','bbb','ccc')]
```

```
<br>a [$test->alpha]
```

```
<br>1 [var: 'test2' = null][$test2->(unserialize:
```

```
$test->serialize)][$test2->alpha]
```

```
[var: 'test' = (object2: 'aaa','bbb','ccc')]  
<br>b [$test->beta]  
<br>2 [var: 'test2' = null][$test2->(unserialize:  
$test->serialize)][$test2->alpha]
```

---

**Serialize Size**

```
<br>Object with prototype [(object: 'aaa','bbb','ccc')->serialize->size]
```

```
<br>Object without prototype [(object2: 'aaa','bbb','ccc')->serialize->size]
```

---

**Creation Speed**

```
[var: 'start'=_date_msec]  
[Output_None][loop: 1000][var: 'test' = (object:  
'aaa','bbb','ccc')][/loop][Output_None]  
<br>Object with prototype [output: _date_msec - $start]
```

```
[var: 'start'=_date_msec]  
[Output_None][loop: 1000][var: 'test' = (object2:  
'aaa','bbb','ccc')][/loop][Output_None]  
<br>Object without prototype [output: _date_msec - $start]
```

---

**Serialization Speed**

```
[var: 'start'=_date_msec]  
[Output_None][loop: 1000][var: 'test' = null][$test->(unserialize: (object:  
'aaa','bbb','ccc')->serialize)][/loop][Output_None]  
<br>Object with prototype [output: _date_msec - $start]
```

```
[var: 'start'=_date_msec]  
[Output_None][loop: 1000][var: 'test' = null][$test->(unserialize: (object2:  
'aaa','bbb','ccc')->serialize)][/loop][Output_None]  
<br>Object without prototype [output: _date_msec - $start]
```

---

[Date]

That's it. The page takes several seconds to load because of the looped tests.

[fletcher]

## Article #6 (#5 intentionally skipped)

**From:** "Greg Willits" <gw@gregwillits.ws>

**Date:** Mon Jan 13, 2003 12:48:43 AM America/Los\_Angeles

**To:** Multiple recipients of corraltalk <corraltalk@corralmethod.org>

**Subject:** Lasso and OOP #6: MVC

### Installment #6: Model-View-Controller (MVC) and frameworks

I spent the day researching more stuff in detail and thinking about framework code. Like all complex knowledge bases, iterative reading is necessary. There is so much to OO software design, and there's a lot of circular references. (Learn A to understand B. Learn B to understand C. Learn C to understand A.)

After spending several days scribbling notes and boxes and arrows, and thinking about how to remap my general frameworks based on OOP, I have spent today revisiting MVC (Model-View-Controller) and patterns. I also ordered some books.

Brett Kirksey initially brought MVC to my attention, and I did some reading, but nothing that satisfied me. Today I was talking with Seth, my son, and he's also been reading about it with respect to doing some Cocoa programming (Cocoa is steeped in the MVC architecture). He has a book that had a nice discussion about MVC that clicked (I think mostly because I've been doing more of the structural thinking lately). And interestingly was also good at explaining Patterns. I'll spin patterns into a future installment.

Virtually every discussion I found of MVC had different slants as to definition. So, I'll offer another to add to what Brett wrote.

First off, MVC is an abstract architecture used to classify objects into one of three categories (a model, a view, or a controller). By using this classification method, the goal is to design objects with a purity of purpose, and also to segregate application code into bins which are inherently more vs less likely to be reusable. Models and Views are designed with reusability in mind, Controllers tend to be where the non-reuable stuff goes and we don't worry about it.

Models, view, and controllers are all objects, but they are objects designed with a specific role in mind. I'm going to use the example of web based article management system to illustrate examples we as web/Lasso folk are concerned with.

#### Models

Models = object(s) of application data structures and core algorithms. This would be an LDML CType which defines all the data structure(s) used in our articles (title, author, copy, images, captions, sidebars, etc). All of our application data is stored in Model object(s).

Additionally, we would have core code unique to our articles here. Perhaps our data structure includes a declaration of a related table of images per article and those images have a sort order defined. We then need a procedure to re-locate an image in that order. The code to do that would be a member tag in our CType (a method in our prototype/class). This object would be a "model" in this MVC architecture.

## **Views**

Views = object(s) which provide the visual displays/interfaces for our model. These would be HTML/LDML code chunks that would draw the forms we need for article admin, and also the lists, search forms, and page layouts for browsing. These objects do not store application data. They might temporarily cache data, or store dynamic preferences, so it isn't correct to say that views don't store any data, they just do not store any main application data. Therefore in Lasso, we would *not* create a custom type that fetched an article from the database, then also had member tags to display that data in various formats. That type of design would be contrary to MVC principles. We would separate the data storage into one object, and those displays into another object.

## **Controllers**

Controllers = the glue code that provides communication between models and views. If we aren't going to have display code as member tags of a particular data type such as articles, how do we connect the model and view objects? We create another object with the necessary code to do that. Controllers contain the application and framework specific code which puts the models and views to work. Controllers do not generally contain any data. In Lasso, we might pass the article ID as a form parameter, a session variable, by cookie, or even by token. The controller contains the code that determines that. The model knows that it has to have an article ID available (a CType instance variable), and the view knows that it has to receive an article ID (probably as \$var), but how the two are connected is up to the controller. We delve into why below.

## **Code Reusability and MVC**

With these three classifications, hopefully you can already see that models and views are targeted to be inherently reusable, and the controller is not.

If we consider the article again, the only info in the model objects would be the core data structure and some core algorithms to perform unique manipulation of that data structure. We would not define any HTML, nor even make any assumptions about our code framework (Corral, OneFile, CMS, or anything).

Thinking about the View objects, we won't make any assumption as to what database is used, or how we get the article ID info. The view knows there's an

'\$article->id' attribute, and can acquire it, but where it comes from, the View doesn't know.

If I wrote an article application for my own framework which used individual page stubs, but followed the MVC architecture, the model and view objects would be reusable by someone else who used a database driven OneFile framework. That developer could take my controller objects and use what was useful, but could also write from scratch their own controller objects following the practices of their own framework. Done properly there would be zero need to rewrite anything in the model objects, and if the view objects were written with a proper "skins" API, then even the view objects would not need altered, while still being adaptable to unique site designs (classic applications would be concerned with specific GUI implementations). However, assuming my view objects were written in plain HTML, someone else could pick up the model objects and write their own CSS based view objects. The two view object sets should be interchangeable.

This structure means that my article model components would be truly universally sharable among LDML users -- for that matter even if they were not following OOP standards or MVC -- because that library of CTypes makes no assumptions about interfaces or communication frameworks. THAT is how we could finally be able to share each other's code without this ridiculous rewriting we all do.

If a person wanted some extra fields to my initial data structure, there would be no need to rewrite my CTypes. Rather the person would write some new CTypes which used my CTypes as parents and added the attributes and behaviors (instance vars and member tags) that they wanted to add.

### **Coding for Models, Views, Controllers**

Now, how do we code a model, a view, or a controller? Well, there's certainly nothing in LDML that says an object is one kind or another, and it doesn't appear that any language does. These are arbitrary and abstract concepts. What makes an object a model object is merely our decision to call it so and to follow the guidelines about functionality that are described above (and others published by more qualified sources than me). So when creating our applications we have to consciously divide our objects in M, V, C types and program them with the appropriate functionalities.

### **Talking with Databases**

It seems also that our models should not make assumptions as to what databases are being used. The actual database interaction should be written in general purpose controller objects--some general routines which accept parameters in order to create database statements. As I presented at Summit and within FW: -Pro, I've created some methods for abstracting my INSERT, UPDATE, DELETE, and SELECT routines. (see my recent post on LassoTalk <http://www.listsearch.com/lassotalk.lasso?id=107542>). I have now figured out how

to rewrite these as CType member tags so I can use them like an abstract superclass. Thus all my model objects can inherit those generic "autoSQL" routines. Each model does not need to have that code integrally (and redundantly) written. In theory I should be able to write my model so that it does not assume a specific database, although it might go ahead and assume an SQL base. My general purpose "autoSQL" controller objects take the data structures in the model and interpret them into MySQL specific commands. If I change databases, then I rewrite a new set of plug and play autoSQL controller objects. My models wouldn't have to change. [**update note:** I didn't know it then, but I was discovering n-tier. See the Article #7]

## **Application Frameworks and MVC**

The section above brings us to frameworks. While we might code MVC objects to specifically deal with articles, we're also going to need application level MVC components: libraries of general purpose routines just like we all tend to use now. These will be written as controller objects. Objects that don't store data, but perform general purpose functions and procedures. These are essentially what we would write today as custom tags, but we'd reformat them as custom types, and implement them as controllers.

We'll also probably need some application level View objects. For me, while I will design custom visitor browsing pages, I tend to stick some visual standards for the back end admin code. This greatly accelerates site development. If that is an option, then framework level View objects are encouraged. In fact this is where we can even write some configurable View objects that are capable of being customized through skins or parameters.

One example from my side is the method I use to make sortable lists of found records sets. You've probably seen what I do: the link button on the left, a title row with clickable sort buttons, alternating row colors and a highlight color for the current sorted column. Well, they're a pain the arse to write because there is so much going on. I've started writing a routine that generates all that code based on a bunch of passed parameters (column names, fields, colors, widths, etc). There's a lot of parameters, but I only have to set them up once. Making small changes to a table then becomes a minor config editing task -- just like the autoSQL routines. Well, that kind of routine can be a framework level View object. I would write an application specific controller to define and invoke that view, but the view object code is reusable as a framework level resource.

## **Fuzzy MVC Boundries**

As with all arbitrary and abstract boundaries, there's cases which could be classified more than one way. As I read about MVC there were some good examples given (interestingly the Undo feature is one). I'll let you discover those as you search for backup material.

## MVC and Lasso

All this sounds well and good, but when the rubber hits the road and we try to implement it, there's some things that feel very weird.

Models are easy to conceive in Lasso. A straight forward CType. Instance vars to hold the data, and some member tags to manipulate that data. When you want an object of article #93, you create variable of the article type, and when that happens our CType will go load an article from the db. From then on we use instance vars and member tags.

Views and Controllers aren't so straight forward. In both cases we want to wrap the code in a CType in order to treat it as an object. To use that code we have to instantiate an object just like we would for our model. So instead of calling an include, or using a custom tag, we first have to instantiate a variable as the object type of our view controller, then we can use the member tags of that object to effect the desired screen page layout routines. So we have to create a variable in order to run our code which resides in the variable (so to speak). This just plain feels weird.

Further, in order to tie our model and our views together we need to accomplish that using a controller. Yet another CType with member tags. So we have to instantiate a controller to use those member tags which would in turn likely instantiate the model to establish our data structure, then in turn instantiate our view object(s) to gain access to those member tags and display something.

As I alluded to earlier, using the full breadth of object oriented design means we have to do things with Lasso that are very different from the traditional procedural perspective we're used to. It wouldn't feel as weird in another language because we would assume it normal. Using LDML like this won't feel normal. I think we're going to probably have to invent the most efficient way to do certain things as we go.

Our code is going to be organized very differently, and our "stubs," "siteConfig," and "pageblocks" possibly won't be cascading includes, I'm not sure. Objects are designed to send messages to other objects. How is that compatible with Corral which is a procedural model of building pages by nesting code chunks? I'm not sure there either.

One thing I need to do is dig into the "patterns" aspect of OOP. These questions about where to put some of the fuzzy boundary items, and how to separate the code into various responsibilities is what patterns is all about. There may be some answers to the several structural questions I have by reading through that stuff.

MVC is certainly not the only architectural model, but it is one that has been around since the beginning of OOP and appears to be a prominent model. So it seems only logical to give it a go. One of the things I need to do is build a teeny example of everything to get a feel for it. [see my exploratory notes with this file set: [ldml.gregwillits.ws/downloads/FWProMVC\\_concepts1.sit](http://ldml.gregwillits.ws/downloads/FWProMVC_concepts1.sit) and [.zip](#)]

## Article #7

**From:** Greg Willits <gw@gregwillits.ws>

**Date:** Tue Jun 17, 2003 7:53:46 AM America/Los\_Angeles

**To:** Multiple recipients of corraltalk <corraltalk@corralmethod.org>

**Subject:** N-Tier Architecture

Somewhat of a diary entry for the on going topic of OOP, MVC, and all things "advanced software architecture" oriented...

Today we talk about N-tier, and my guest is... me :-)

So, I've started down the path of OOP to help in building stable, reusable code. I've also adopted MVC principles in some of my stuff to segregate my object definitions from the logic and display code used to interact with them. Whether by CType, Ctags, or Includes, I started segregating most things so that data structure, logic, and display code are in independent files. Having everything written as a CType would be "pure," but it just isn't practical for everything when building a web page

**Mini follow-up to OOP / MVC and how far to push it** -- web pages are like individual little applications because they're built from scratch every page, and with the middleware languages like Lasso specialized for this purpose it is a little ridiculous to create generic objects for things like templates and pages. It just add layers for objects which don't need to be isolated within an environment because they are the environment. Also, once you get three or more big ctypes talking to each other, things slow down a bit in LP6, not dramatic, but noticeable. I've seen similar comments by those proposing OOP for web apps using other languages, so it's possibly an issue for any scripting language. I really hope this will continue to improve with new LP versions (like it did between LP5 and LP6), and I believe it will. My number two feature request (after the ability to have query results deposited to various data types (arrays, maps, named inlines), is simply keep improving Lasso's processing efficiency in these areas so we can build better OOP systems with minimal speed penalty.

Anyway, I originally only expected to ever use MySQL because most of my projects will be ASP (not .asp!) oriented, so I can do what I want :-). Now I have this project which will require MS SQL Server, and it will push me to add a lot of new things to my core code which will be pretty cool actually. Not wanting to have that stuff exclusive to the MS SQL project, I have decided to go further with the concepts of N-tier. After having researched this, I have decided that for web developers, the principles behind N-tier are probably more important than OOP! So, I figured we could toss it about here, as it is similar to the principles behind Corral. In fact, I suspect many of you are using N-tier principles out of common sense and may not know it :-)

Let's set aside OOP & MVC for a moment and simply look at an application as a body of code which has "layers." This is somewhat similar to the ideas in MVC, but don't equate the two because they are different. As an example let's use the task of

retrieving data to display on the web page, and dissect the sequence of events into layers or tiers.

The code you'd write to populate and display an update page includes:

- a source of data (database, delimited text file, XML, etc)
- a core data retrieval routine to acquire the data. This would most likely be an inline to fetch a record from a database.
- surrounding that inline, you'll have some data retrieval error management code -- is there a record number? is found\_count > 0?, was there a database error?
- surrounding that error code you'll have some access rules -- is this user a valid user? is this user allowed to view this record?
- after retrieving the data successfully you might have some code to process the data before presenting it. Maybe there's some fields to combine, or some interpretation of simple data values into human readable forms.
- after processing the data, next you'll have some code to build a formatted view of the data. For the web page, we'll create an HTML table or CSS boxes, etc
- the data view (page block) will have to be combined with the environment view (template)
- the last thing we have nothing to do with, but the browser is used to actually display the final result

Now, the way I have broken this down, you can see that we could call each of these functions a layer or tier. Well, the geniuses that have trod this ground in the days of yore have decided we could opt to segregate this into as many tiers as we want - - maybe 2, 3, 4, 12, whatever. Because it is a variable, they, undoubtedly having math degrees and not marketing degrees, decided to call it N-Tier (N representing any number).

Why do we care about segregating and tiers and all this philosophical rot? Well, suppose you thought you were in control of your life and decided you'll only ever use MySQL (what dufus does that?). You write a bunch of application modules to handle typical things your clients want, and for the most part you can tweak 'em a little and reuse them for new apps and clients. You're set.

Next thing you know, someone likes your app, but must host it themselves and they want you to use another database. Doh! You have this great application already written, but now pieces of it all over the place need updated for the new database syntax.

It's pretty common to find routines like this:

```
if: $usrAppvd;
  inline:
    -username='',
    -password='',
    -sql='';

    if: error_currenterror == 0;
      ..... oh goodie .....
      records;
      output:.....
    /records;
  else;
    ..... @$#! .....
  /if;
/inline;
/if;
```

Our first improvement would be to use named inlines to separate the logic and the display:

```
if: $usrAppvd;
  inline:
    -inlineName='better',
    -username='',
    -password='',
    -sql='bla bla bla';

    if: error_currenterror == 0;
      ..... oh goodie .....
    else;
      ..... @$#! .....
    /if;
  /inline;
/if;

//----- separate file -----//

[records: -inlinename='better']
  ..... yadda yadda .....
[/records]
```

Now it is easier to modify display code without introducing errors to the basic data acquisition routine.

However, our logic still has problems. Our business logic (albeit only one line in this case), error management, and data retrieval code is all intertwined. If the whole site was like this, switching database servers is going to require editing a bunch of

the site's main files. Not to mention, what if instead of a database file, your data needed to come from an XML transaction? N-tier philosophy dictates that these code layers should be separate. So, to separate the query from the business logic we could do the following:

```
if: $usrAppvd;

    include: 'myQuery.inc';

    if: $myQueryError == 0;
        ..... oh goodie .....
    else;
        ..... @%$#! .....
    /if;
/if;

//----- myQuery.inc -----//

inline:
    -inlineName='better',
    -username='',
    -password='',
    -sql='bla bla bla';
    var:'myQueryError' = error_currenterror;
/inline;

//----- separate file -----//

[records: -inlinename='better']
    ..... yadda yadda .....
[/records]
```

Granted, this is an extremely simple example, so hopefully you can picture parallel examples in your own code with more complex details.

There's an arguable separation here. The error management -- should it be with the query logic or in a sperate layer or combined with the business logic? That's pretty much a design decision. I prefer it above the query so my query layer only needs to set expected variables/properties. This keeps my error management standardized without a need to debug it again. Use generic variable names (instead of \$dbNotFound, use \$dataSrcNotFound and now it doesn't matter if the source was db, text, xml, etc).

At this point we could swap the myQuery.inc file with another one that retrieves data from another database server, XML, text file, or any other data source -- including URL parameters. None of the business logic or application error management layers need to change. We have a bit of a problem though with the presentation layer -- it expects [records] which is specific to databases.

This is one major reason why Lasso really needs to expand to allow multiple native data formats for query results. If we could put the data results into a standardized array, map, or custom type format (without the current overhead/memory penalty of doing so), then even our presentation layer could remain unchanged as a result of changing data sources from SQL to XML.

So, a common N-tier structure for web apps appears to be:

```
=====
GUI: browser, other client front end...
-----
Presentation Logic:
-----
Business Logic: auth, validation, data processing...
-----
Data Access Logic: SQL, file_tags, XML_tags...
-----
Data Storage: db, text, XML...
=====
```

The idea is to segregate the responsibilities of code and isolate the specific technologies used from each other. To be perfect, each tier must have an abstract view of the tiers around it. For example, a presentation tier should not be using [found\_count] to test if there is data. Rather, the Data Access Logic tier using a db routine should determine [found\_count] then convert it to an abstract variable like \$dataFound. Now the presentation code is not dependent on database technology for its data source.

Techniques like this allow you to develop a site in which you can literally pull a specific set of files out that ran your MySQL code, and drop in another set of files that will use Oracle, Postgres, MS SQL Server, etc. You could also replace the presentation tier (written for a traditional web browser) and replace it with one written for mobile phones - without rewriting any of the data access or business logic code!

As I said earlier, this structural concept may be more important to web site technologies than MVC, although I am sure you can see how they can be used together to gain the maximum in code reuse and reliability. MVC is how you structure your objects. The tiers are comprised of elements from these objects. Being discrete files, they are easily substituted.

I hope this helped provoke some thoughts for organizing your apps.