Extending Moose for Applications

Shawn M Moore

Best Practical Solutions

Presented YAPC::NA, 2009-06-23.
CMU, Pittsburgh, PA.

Twitter!

@sartak Hey wait, what is "meta"?

People can be shy about asking questions in a room full of strangers. So if you want, you can ask me a question on Twitter and I'll eventually get to it. Just direct your questions at @sartak. That way you don't forget it or worry about interrupting me. Got the idea from Giles Bowkett: http://gilesbowkett.blogspot.com/2009/04/how-to-take-questions-at-tech.html

(or IRC)

Of course, many people dislike Twitter, so I'll check out at IRC as a last resort. I'm Sartak there too.

You don't have to use either of these. If you have a burning question or correction, shoot your hand up at any time.

Extending Moose for Applications

The title of this talk is Extending Moose for Applications. It means just what it says. We want to extend Moose to be more useful and expressive for the problem at hand.

Domain-specific Metaprogramming

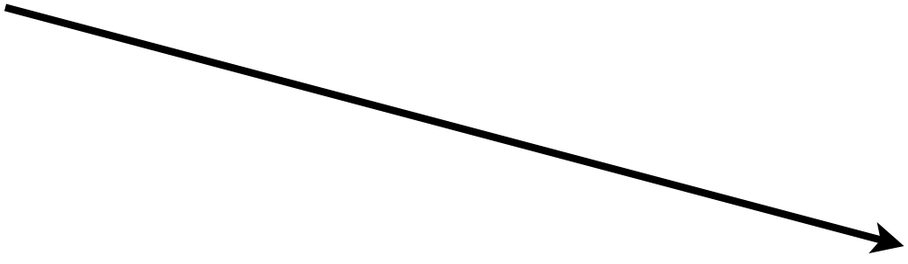
But I really wanted to call the talk Domain-specific Metaprogramming. It's more honest. But it's pretentious and a little scary. Hell, with that title I would have been able to sleep in today.

Extending Moose for Applications

Domain-specific Metaprogramming

But these two titles are the same talk.

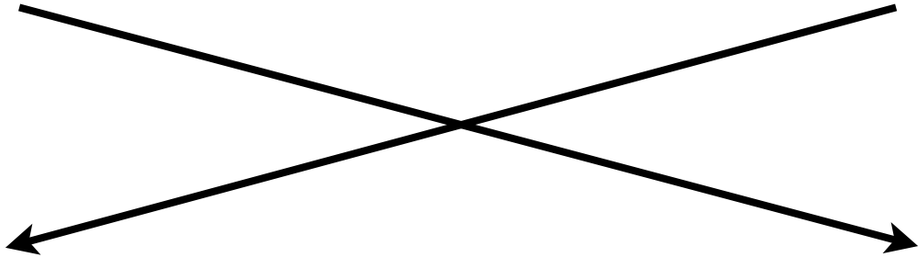
Extending Moose for Applications



Domain-specific Metaprogramming

You extend Moose through metaprogramming, which is what the talk is all about.

Extending Moose for Applications



Domain-specific Metaprogramming

And an application is a domain-specific system. I want to teach you new concepts and techniques so that you can solve your business needs.

In March I was at a Lisp conference with jrockway. I'm not much of a Lisper but I do appreciate Lisp for the huge impact it has had on language design. The conference was very enjoyable. What stuck out most was CLOS, which is Common Lisp's version of Moose. It is 15 years older.

Context-oriented Programming with ContextL

Towards a Secure Programming Language: An Access Control System for CommonLisp

Rhapsody: How CLOS simplifies System-in-a-Package Design

When I was there, quite a few talks began with "I extended CLOS to do X". I was so impressed that immediately after the first day of the conference, I submitted this talk for YAPC. I think Moose is ready to let many more people seriously dig into its guts to make their applications much better.

Metaprogramming

What do I actually mean by "metaprogramming"?

Metaprogramming

```
$obj->can('method_name')
```


How many of you have asked an object if it performs a particular method?

Metaprogramming

```
$obj->can('method_name')
```

```
$obj->isa('Class::Name')
```


Or if an object is an instance of a particular class?

Metaprogramming

```
$obj->can('method_name')

$obj->isa('Class::Name')

$obj->DOES('RoleName')
```


Or if an object does a particular role? (chromatic?)
This is a relatively new one, added in 5.10.

Metaprogramming

```
$obj->can('method_name')

$obj->isa('Class::Name')

$obj->DOES('RoleName')

eval "code"
```


Who here has programmatically generated code? String eval has a lot of well-deserved stigma, but it's occasionally necessary.

Metaprogramming

```
my $code = setup();
$code .= important_stuff();
$code .= teardown();

eval $code;
```


String eval is interesting because there's an obvious separation of the code that is doing the generating (all these function calls and concatenation) and the code that is generated (the contents of $code). We could say that the first three lines are metaprogramming, since it's code whose domain is other code. The contents of $code would be the "base" level.

Metaprogramming

```
__PACKAGE__->meta->make_immutable
```


Here's another example of metaprogramming. Many of you have cargo culted this for your Moose classes to make them faster. (Somehow!)

Have you ever stopped to just look at this? Obviously the whole expression means to make the current class immutable.

Metaprogramming

```
__PACKAGE__->meta->make_immutable
```


There are two method calls. What is that first one actually doing?

Metaprogramming

```
my $meta = __PACKAGE__->meta;

$meta->make_immutable;
```


In other words: What is $meta?

Let's print it!

Metaprogramming

```
print $meta;

Moose::Meta::Class=HASH(0x966910)
```


It's an object. Its class is Moose::Meta::Class.

Moose::Meta::Class

Attributes	Methods
name	new_object
version	clone_object
attributes	rebless_instance
methods	subclasses
superclasses	linearized_isa
roles	add_attribute
attribute_metaclass	has_method
method_metaclass	get_all_method_names
constructor_name	is_immutable
constructor_class	calculate_all_roles

Moose::Meta::Class is a class *for* classes. Like every ordinary class, Moose::Meta::Class has attributes and methods.

An instance of Moose::Meta::Class *describes* some class. A class has a name, a list of superclasses, a list of methods.

You can also create a new_object of a class, or add an attribute to a class, or get the complete list of roles the class does.

Metaprogramming

```
my $code = setup();
$code .= important_stuff();
$code .= teardown();

eval $code;
```


If we go back to the string eval example, we can see some parallels. Moose itself is like the first section, which is the meta level. Your class is similar to the contents of $code, which is the base level.

In Moose, there is far more interaction between the two layers than in this example.

package Point;

```
use Moose;

has 'x' => (is => 'rw', isa => 'Int');
has 'y' => (is => 'rw', isa => 'Int');

sub clear { ... }
```


This is the Point class again, but with Moose. Before I can talk about *extending* Moose, you have to know a bit about how it works internally.

Point->meta

```
my $point = Point->meta;
```


We now have the class metaobject for Point in $point. We could call make_immutable on it if we wanted to make Point faster. But instead, let's inspect it.

Point->meta

```
my $point = Point->meta;

$point->name # ?
```


If we look at the name of the class metaobject, we'll get...

Point->meta

```
my $point = Point->meta;

$point->name # Point
```


Point, which is the name of the class that $point describes.

Point->meta

```
my $point = Point->meta;

$point->name # Point

$point->get_attribute_list # ?
```


And the list of attributes?

Point->meta

```
my $point = Point->meta;

$point->name # Point

$point->get_attribute_list # y, x
```


y and x. You might've expected x and y, since that was the order we added them to the class. However, the attribute and method lists are unordered. The canonical representation of these lists in Moose is a hash. The list methods just perform "keys" on that hash. This way we get free name collision resolution.

Point->meta

```
my $point = Point->meta;

$point->name # Point

$point->get_attribute_list # y, x

$point->has_method('clear') # ?
```


And finally does Point have a clear method?

Point->meta

```
my $point = Point->meta;

$point->name # Point

$point->get_attribute_list # y, x

$point->has_method('clear') # 1
```


Yes.

package Point3D;

```
use Moose;

extends 'Point';

has 'z' => (is => 'rw', isa => 'Int');

after clear => sub { ... };
```


And next up we have the Point3D subclass of Point. It has an additional attribute 'z' and a method modifier on the 'clear' method.

Point3D->meta

```
my $point3d = Point3D->meta;
```


Let's start by asking the same questions.

Point3D->meta

```
my $point3d = Point3D->meta;

$point3d->name # ?
```


Name?

Point3D->meta

```
my $point3d = Point3D->meta;

$point3d->name # Point3D
```

Point3D->meta

```
my $point3d = Point3D->meta;

$point3d->name # Point3D

$point3d->get_attribute_list # ?
```


What attributes does Point3D have?

Point3D->meta

```
my $point3d = Point3D->meta;

$point3d->name # Point3D

$point3d->get_attribute_list # z
```


Just one, z! get_attribute_list works based on the *local* class. It does not consider inheritance. When you're using metaclasses you need to know whether inherited entities are included or ignored.

Point3D->meta

```
my $point3d = Point3D->meta;

$point3d->name # Point3D

$point3d->get_attribute_list # z

$point3d->has_method('clear') # ?
```


Clear method?

Point3D->meta

```
my $point3d = Point3D->meta;

$point3d->name # Point3D

$point3d->get_attribute_list # z

$point3d->has_method('clear') # 1
```


Yes, Point3D does have a clear method.

```
package Point3D;

use Moose;

extends 'Point';

has 'z' => (is => 'rw', isa => 'Int');

after clear => sub { ... };
```


Even though in this case "after clear" is just a method modifier, that is good enough for has_method. has_method, like has_attribute, checks only the local class, it ignores inherited methods.

Finding Things

Local	Global
`has_attribute`	`find_attribute_by_name`
`has_method`	`find_method_by_name`
`get_attribute_list`	`get_all_attributes`
`get_method_list`	`get_all_methods`
`superclasses`	`class_precedence_list`

For many metaclass methods, you have to know whether inheritance is considered. Generally, methods that ignore inheritance have shorter names than methods that include inheritance. The word "all" is a good indicator as well. You also need to know whether the method returns names or objects, though that's less easy to screw up. When you run your code the first time, the wrong choice will be immediately evident.

REST Interface

```
my $class = url2class($url);
my $meta = $class->meta;

for ($meta->get_all_attributes) {
    my $name = $_->name;
    my $tc = $_->type_constraint;
    my $default = $_->default;
    if ($_->is_required) { ... }
}
```


One of the uses of this sort of introspection could be a REST interface. In addition to the usual CRUD operations, you could dump the class to YAML or JSON as documentation. Another program could use that description to generate classes.

You have all this data available when you write classes with Moose, so reusing it in your application is far better than declaring it several times. Don't repeat yourself.

~~use Moose;~~

Don't get the wrong idea about this slide!

I'm going to define the Point class again, but without actually using the Moose sugar. I want to demonstrate that metaclasses do more than just describe classes, we can change them too. And Moose users do that all the time when they use the friendly functions like "has"

```
my $point = Moose::Meta::Class->create(
    'Point',
);
```


We start by creating the metaclass. Note that the name is the first parameter with no hash key. This interface sucks, we know. :(

```
my $point = Moose::Meta::Class->create(
    'Point',
);

$point->superclasses('Moose::Object');
```


Then we set the superclass. If we don't do this then we'll get no "new" method. This is one of the things "use Moose" does transparently for us.

```
my $point = Moose::Meta::Class->create(
    'Point',
);

$point->superclasses('Moose::Object');

$point->add_attribute(
    'x',
    is  => 'ro',
    isa => 'Int',
);
```


The "x" attribute. Again we pass the name without a key, but the other options are the same.

```
...

$point->add_attribute(
    'y',
    is  => 'ro',
    isa => 'Int',
);
```


The "y" attribute. It's the same as x but for the name, so maybe a loop would be better.

```
...

$point->add_attribute(
    'y',
    is  => 'ro',
    isa => 'Int',
);

$point->add_method(clear => sub {
    ...
});
```


And finally the clear method.

All of Moose's sugar functions are thin wrappers for metaclass methods. "has" and its friends actually form a very small part of Moose.

Ecosystem

Classes
Attributes
Methods
Roles
Type Constraint
Type Coercion

These are all the important entities that Moose provides.

Ecosystem

Classes
Attributes
Methods
Roles
Type Constraint
Type Coercion

`Moose::Meta::Class`

We've seen that Moose::Meta::Class objects each describe a class.

Ecosystem

Classes	`Moose::Meta::Class`
Attributes	`Moose::Meta::Attribute`
Methods	`Moose::Meta::Method`
Roles	`Moose::Meta::Role`
Type Constraint	`Moose::Meta::TypeConstraint`
Type Coercion	`Moose::Meta::TypeCoercion`

We also have classes for all the other interesting things in the program. Moose's devs like OO, so it makes sense that Moose itself is designed using object-oriented programming.

Ecosystem

```
my $x =
    Point->meta->get_attribute('x')

$x->name                 # x
$x->get_read_method # x
$x->type_constraint # Int
```


We can get the attribute metaobject with the "get_attribute" method.

get_read_method returns the name of a method that can be used to read the attribute's value. We call the method it returns ("x") on an instance of Point.

Ecosystem

```
my $clear =
    Point->meta->get_method('clear')

$clear->name          # clear
$clear->package_name  # Point
```


There are also method metaobjects. There's not much to them though. All you can really do with methods is invoke them.

Extending

1. Extend a metaclass
2. Use it

There are two steps to extend Moose. You extend a particular metaclass with subclassing or role application. Then you just use it.

We definitely prefer role application over subclassing. That lets extensions compose. If everything subclassed, then you'd only be able to use one extension at a time.

Extending

Class that counts its instances

We want to create a class that counts the number of instantiations.

```
package HasCounter;
use Moose::Role;

has count => (
    is      => 'rw',
    isa     => 'Int',
    default => 0,
);

sub increment {
    my $self = shift;
    $self->count(
        $self->count + 1
    );
}
```


We start by defining a role that has a counter attribute and a method to increment it. This is a role that you could consume in your ordinary classes. It's not special in any way.

```
package CountInstances;
use Moose::Role;

with 'HasCounter';

after new_object => sub {
    my $self = shift;
    $self->increment;
};
```


Now we define a new role that we expect to apply to Moose::Meta::Class that hooks up the counter role to the "new_object" method.

```
package Point;
use Moose -traits => [
    'CountInstances',
];

has x => (
    is  => 'ro',
    isa => 'Int',
);

...
```


Now all we have to do use tell Moose we want to use this role on the metaclass. After the special -traits option to use Moose, the rest is just the same.

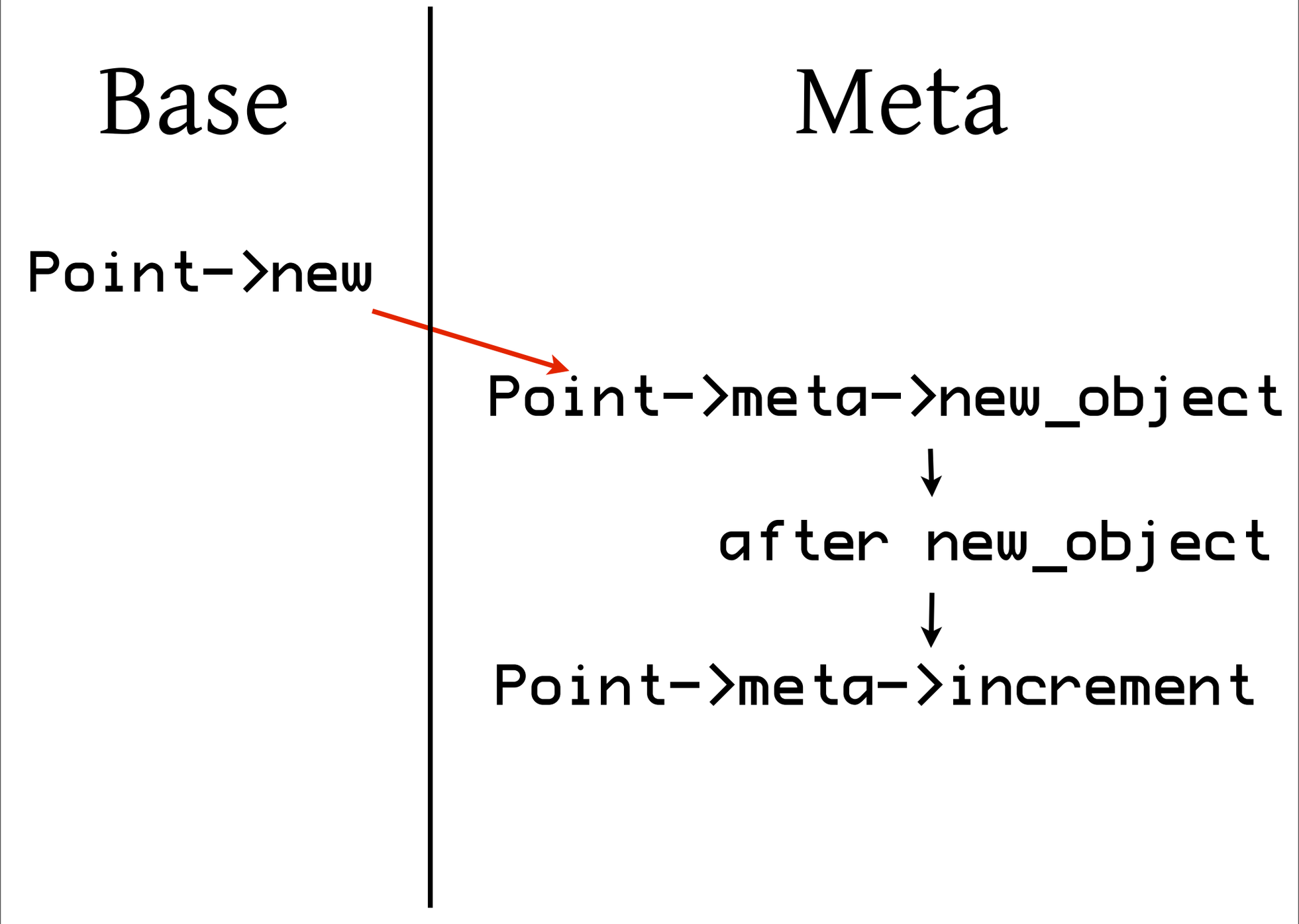
The words "trait" and "role" are mostly synonymous. There are vague differences but not worth covering. When you're adding roles to metaclasses in Moose, they're called traits.

```
Point->meta->count # 0

Point->new
Point->meta->count # 1

Point->new for 1 .. 5
Point->meta->count # 6
```


We can inspect the count which is an attribute of "Point->meta".



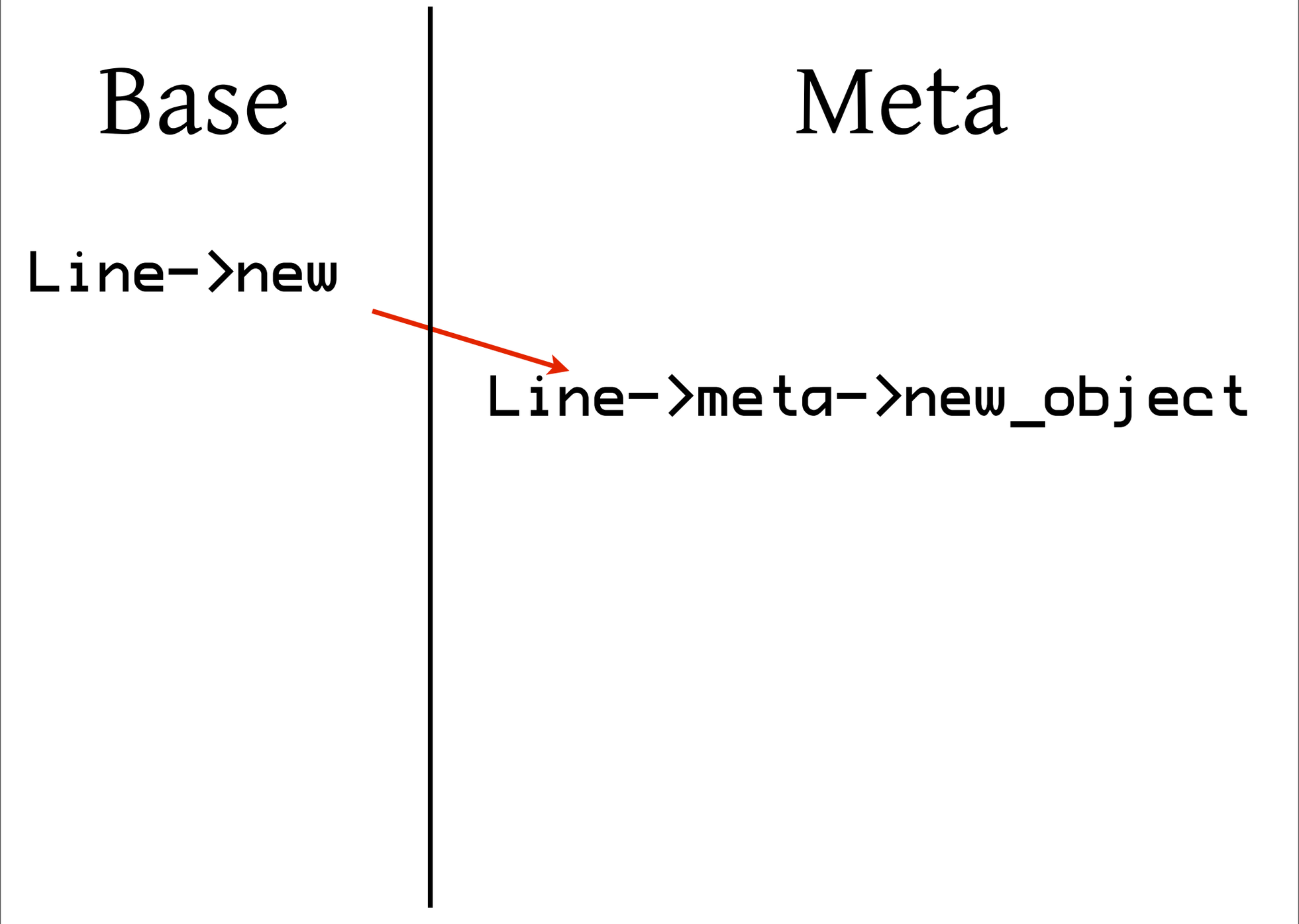
Point->new is a Moose::Object method that enters the meta-level, hence the red arrow. The black arrows indicate method calls that don't traverse the base/meta boundary.

```
package Line;
use Moose;

has start => (
    is  => 'ro',
    isa => 'Point',
);

has end => (
    is  => 'ro',
    isa => 'Point',
);
```


I want to introduce another class, Line. This does NOT use the CountInstances trait.



The Line class's instances are not counted. The major point is that one class's extensions do not affect other classes. We're not monkeypatching in a method modifier or attribute or anything like that. With monkeypatching you could destroy Moose's own workings by redefining a particular method it uses internally.

This is Perl, we love to design modular, robust systems. Monkeypatching is the antithesis of that ideal.

```
package HasCounter;
use Moose::Role;
use MooseX::AttributeHelpers;

has count => (
    traits   => ['Counter'],
    provides => {
        inc => 'increment',
    },
);
```


We can also apply roles to other metaclasses, such as attributes. Many of you have used MooseX::AttributeHelpers; it works by extending the attribute metaobject. "provides" is an attribute of the AttributeHelpers Counter role which ultimately uses "add_method" on your class.

```
package FieldType;
use Moose::Role;
use Moose::Util::TypeConstraints;

has render_as => (
    is => 'ro',
    isa => (enum [
                'text',
                'textarea',
                'password',
                ...,
            ]),
);
```


We're writing a web application so it's important to know how to render every attribute as a form field. We'll use a role to let the application writer (usually you!) declare what type of <input> element each attribute uses.

```
package User;
use Moose;

has name => (
    traits    => ['FieldType'],
    is        => 'rw',
    isa       => 'Str',
    render_as => 'text',
);
```


We're writing a user class. The name is usually a short string of characters, so a regular text element makes sense for it.

```
...

has password => (
    traits    => ['FieldType'],
    is        => 'rw',
    isa       => 'Str',
    render_as => 'password',
);
```


For a password, we want to obscure the value of the element so that someone looking over your shoulder can't steal your password.

```
...

has biography => (
    traits    => ['FieldType'],
    is        => 'rw',
    isa       => 'Str',
    render_as => 'textarea',
);
```


And finally we let the user explain all about themselves in a long biography textarea.

```
traits => ['FieldType'],
render_as => 'text',

traits => ['FieldType'],
render_as => 'password',

traits => ['FieldType'],
render_as => 'textarea',
```


That's a lot of redundancy. We know that when we're writing a web application, most classes will eventually be rendered as a form some day.

```
traits => ['FieldType'],
render_as => 'text',

traits => ['FieldType'],
render_as => 'password',

traits => ['FieldType'],
render_as => 'textarea',
```


Ideally we'd be able to say just how each field is rendered, not that it is going to have a particular field type. This problem is more pronounced when you are using many metaclass extensions together.

Moose::Exporter
Moose::Util::MetaRole

We have a solution that comes in two modules. These modules are the workhorses of extending Moose. They were written by Dave Rolsky after he realized that Moose extensions were not as composable as we wished. These modules let you do the right thing easily, which is the goal of every well-designed module.

Moose::Exporter

Moose::Util::MetaRole

Moose::Exporter is a module that lets you write modules like "Moose" and "Moose::Role". These modules create a class (or role) metaobject, and provide for the user some sugar functions. Moose and Moose::Role are themselves implemented with Moose::Exporter.

Moose::Exporter

Moose::Util::MetaRole

MetaRole is a module that lets you apply roles to the various metaclasses in your class. You use it inside of Moose::Exporter.

```
package MyWeb::OO;

use Moose ();
use Moose::Exporter;
use Moose::Util::MetaRole;

use FieldType;

Moose::Exporter->setup_import_methods(
    also => 'Moose',
);

sub init_meta {
    ...
}
```


Here we're defining a module MyWeb::OO that people use instead of Moose itself. We have to load a bunch of stuff, including the role we're going to apply to every attribute.

We then setup import methods. This particular invocation makes MyWeb::OO provide all of the Moose sugar. You could add new functions to that if you wanted to.

```
my $class = shift;
my %options = @_;

Moose->init_meta(%options);

Moose::Util::MetaRole::apply_metaclass_roles(
    for_class => $options{for_class},
    attribute_metaclass_roles => ['FieldType'],
);

return $options{for_class}->meta;
```


Here is the good stuff, inside the init_meta method. This is called to construct a metaclass for the user of MyWeb::OO. We let Moose's own init_meta do the heavy lifting. We then change it slightly so that FieldType is automatically applied to attribute metaclasses.

This is a pretty decent amount of code, but it's all well documented and could be abstracted further if you wanted to make this common case require less typing.

```
package User;
use MyWeb::OO;

has name => (
    is         => 'rw',
    isa        => 'Str',
    field_type => 'text',
);

...
```


Let's revisit the User class now that we have this shiny new exporter. We've extended the attribute in a useful way. The user doesn't need to know the traits invocation. They don't need to know about all this metaprogramming stuff. They just use this new option to "has" as if it were in Moose from the start.

```
package User;
use MyWeb::OO;
use MyWeb::OO::Persistent;
use MyWeb::OO::RESTful;
use MyWeb::OO::IncludeInAdminUI;
use MyWeb::OO::SpamTarget;

database_name 'person'; # legacy

has email => (
    is              => 'rw',
    isa             => 'Str',
    field_type      => 'text',
    spam_this       => 1,
    admin_editable => 0,
    primary_key     => 1,
);
```


Another important point of Moose::Util::MetaRole is that it composes with other extensions seamlessly. You could write all of these Moose extensions that inject all sorts of roles into the various metaclasses. For example, Persistent would inject both class and attribute roles, and provide a database_name keyword.

Immutability

Though it served as a useful launching-off point, immutability is the most irritating thing about extending Moose. If your extensions affect object construction or accessors, then you will probably need to care about immutability.

```
sub _initialize_body {
    my $self = shift;
    my $source = 'sub {';
    $source .= "\n" . 'my $class = shift;';

    $source .= "\n" . 'return $class->Moose::Object::new(@_)';
    $source .= "\n    if \$class ne '" . $self->associated_metaclass->name
            .  "';\n";

    $source .= $self->_generate_params('$params', '$class');
    $source .= $self->_generate_instance('$instance', '$class');
    $source .= $self->_generate_slot_initializers;

    $source .= $self->_generate_triggers();
    $source .= ";\n" . $self->_generate_BUILDALL();

    $source .= ";\nreturn \$instance";
    $source .= ";\n" . '}';
    ...
```


One of the ways we make Moose faster is by string evaling constructors and accessors. That certainly makes Moose faster, but for the .1% of users who want to extend Moose, it sucks. You need to hook the methods called here to add the string of code that you need. You can also turn off immutabilization, but that slows the class down. Damned if you do, damned if you don't.

KiokuDB
Fey::ORM

These are some projects that extend Moose to great effect. If I've convinced you that domain-specific metaprogramming is awesome, you should read through their source.

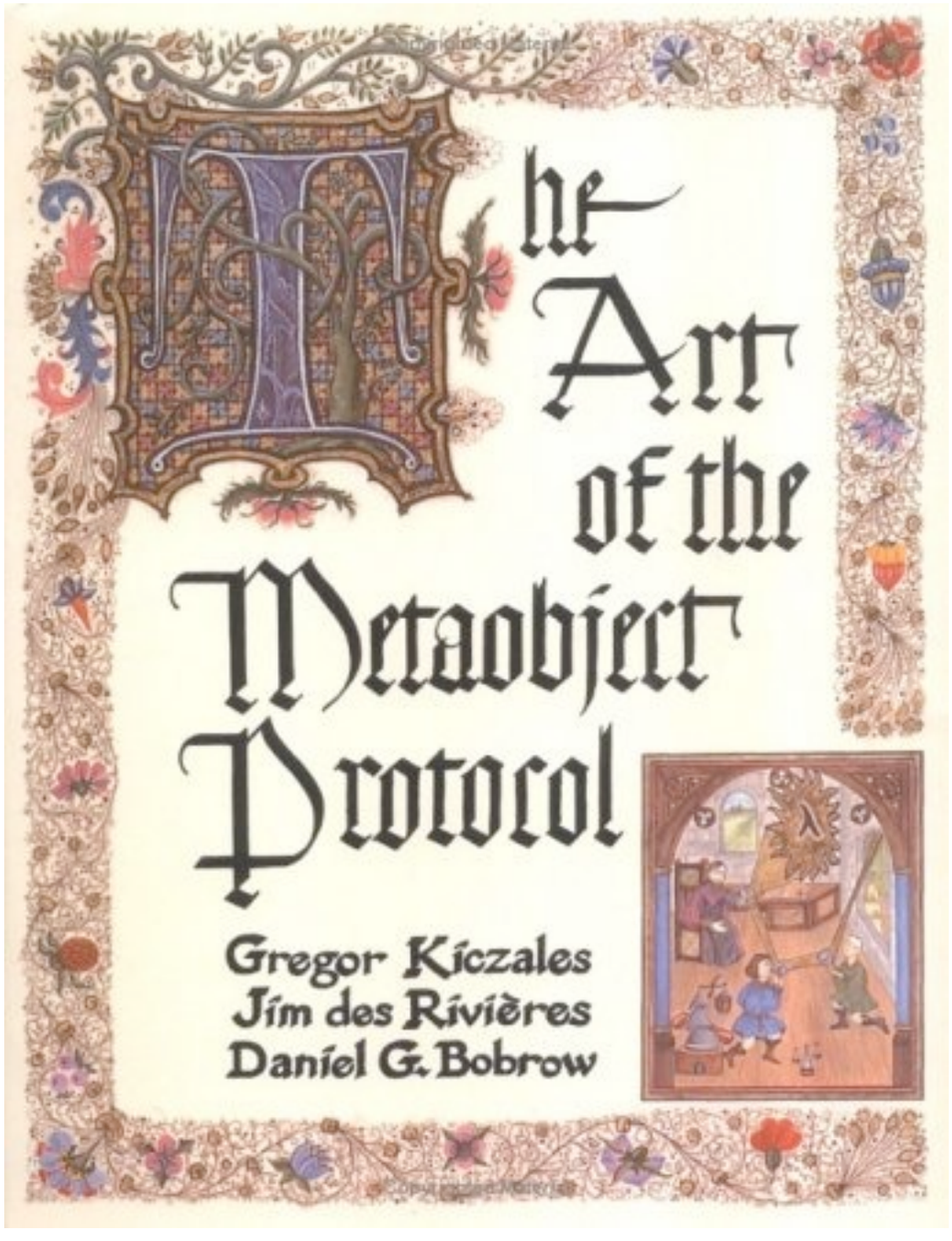
KiokuDB
Fey::ORM

ContextL (CLOS)

I'd also like to give a shout-out to Pascal Costanza's ContextL project. This is a pretty serious extension of CLOS's metaobject protocol. It provides things like layered classes where layers are dynamically turned on and off which alters the class definition. It's really neat and worth looking at.

ROLES!

Roles themselves would not have been easy or clean without Class::MOP's metaobject protocol.



Finally, if you really like this stuff, get this book. Alan Kay, inventor of OO, said "This is the best book anybody has written in ten years". He also says it's a very hard book to read because of its Lisp-centric nature, but hopefully that isn't too big a stumbling block. It's just an abstract syntax tree!

Thank you!