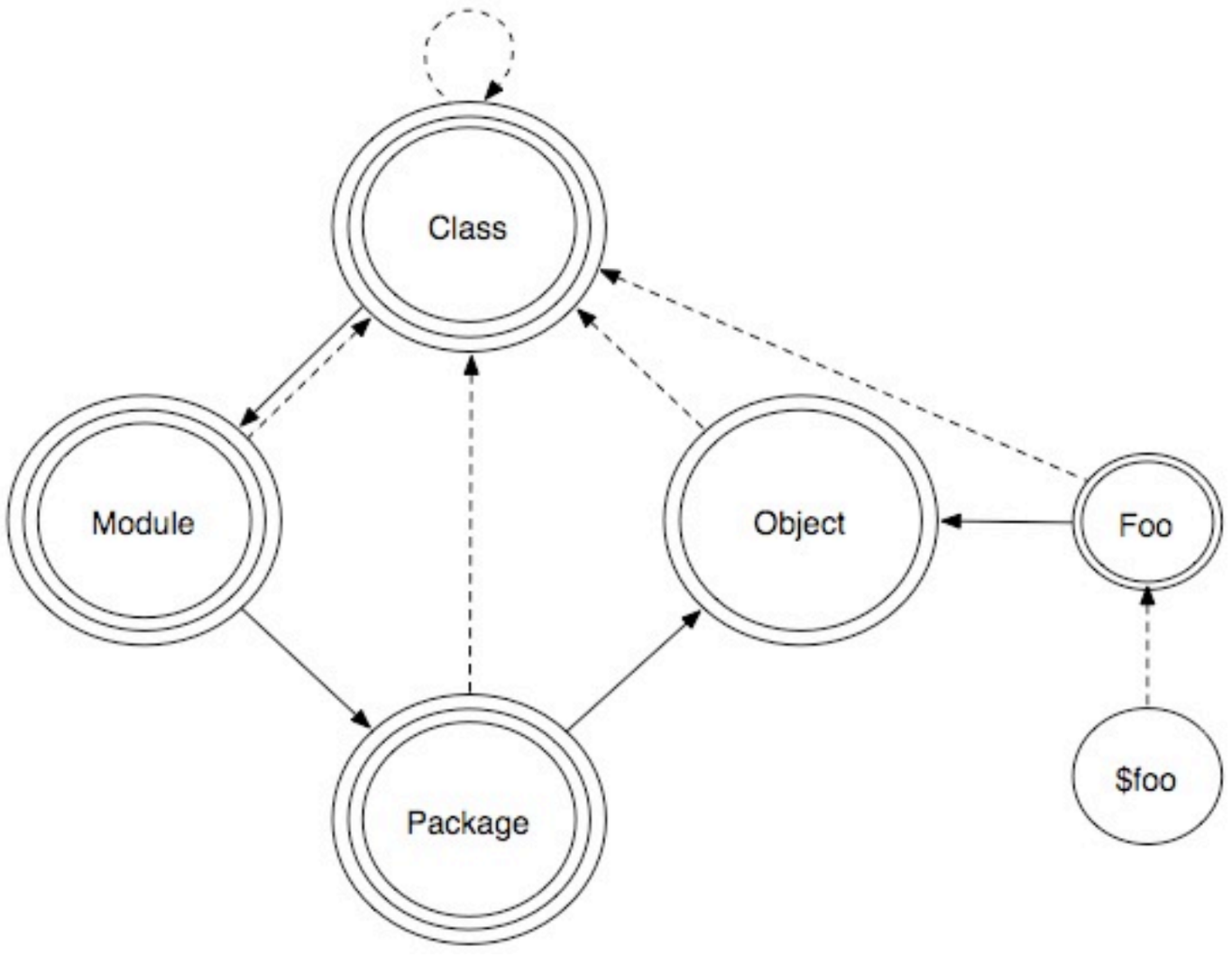
MooseX

Tour

Meta Circular



Friday, June 7, 13

So, Moose is meta-circular, which means, Moose is bootstrapped such that Moose can be extended with Moose. This seems a little esoteric, but in reality is is really useful. And this is proved by the rich Moose Ecosystem which has grown in the past few years

200+ MooseX::* on CPAN

Friday, June 7, 13

As of the writing of this talk, there were more than 200 MooseX:: modules on CPAN. Some are good, some are bad, some are just plain weird.

The Good

MooseX::StrictConstructor

- by default Moose ignores constructor params that don't match attributes
- this extension reverses that behavior
- written and maintained by Moose team
- likely to be merged into Moose core

```
package Foo;
use Moose;
use MooseX::StrictConstructor;

has 'bar' => ( is => 'ro' );

# ...
```

```
my $foo = Foo->new( baz => '...' ); # BOOM!
```

MooseX::UndefTolerant

- Moose treats `undef` as a value
  (instead of the lack of a value)
- this extension reverses that behavior
- written and maintained by Moose team
- likely to be merged into Moose core

Friday, June 7, 13

Moose treats undef as a explict value, rather than an implicit lack of a value. This aligns more with the ML and Haskell school of thought about types then it does the Java/C school of thought. This is not how everyone thinks though, so this module was created.

```
package Foo;
use Moose;
use MooseX::UndefTolerant;

has 'bar' => (
    is  => 'ro',
    isa => ‘ArrayRef’,
);

# ...
```

```
my $foo = Foo->new( baz => undef );

# OR

my $foo = Foo->new( baz => $some_undefined_value );
```


Friday, June 7, 13

This is particularly useful when passing in a pre-built hash of values to a constructor, potentially coming from another source (CGI params, etc.) and not having to worry about undefined values can be useful.

MooseX::Constructor::AllErrors

- by default, Moose dies on the first error it encounters while constructing an object
- this extension collects ***all*** the errors
- and ***then*** returns them
- written and maintained by Moose team

Friday, June 7, 13

In many cases it is useful to see all the errors that happen, rather then to discover them one by one. One key thing to note about this module is that it throws an exception object, not just a string.

```
package Foo;
use Moose;
use MooseX::Constructor::AllErrors;

has 'bar' => (
    is  => 'ro',
    isa => ‘ArrayRef’,
);

has 'baz' => (
    is  => 'ro',
    isa => ‘Number’,
);

# ...

my $foo = Foo->new( baz => undef, bar => [] ); # BOOM!
```

MooseX::Params::Validate

- builds on Params::Validate
- supports many Moose type features
  - including coercion
- straightforward and simple
- written and maintained by Moose team

Friday, June 7, 13

This module is intentionally not fancy, and does not try to provide any approximation of method signatures. It is purely for validating the values in @_.

```perl
package Foo;
use Moose;
use MooseX::Params::Validate;

sub test {
    my ($self, $this) = validated_hash(\@_,
         this => { isa => 'HashRef' },
         that => { isa => 'Boolean', optional => 1 }
    );
    # ...
}
```

```perl
# ...

$foo->test( this => [] ); # BOOM!

$foo->test( this => {}, that => 1 );
```


Friday, June 7, 13

There is also a validated_list version, which validates positional parameters.

MooseX::Getopt

- allows attributes to be set via command line
- makes scripts “inheritable”
- written and maintained by Moose team
- more complex CLI apps should look at MooseX::App::Cmd

Friday, June 7, 13

This module changes the way you approach writing a command line application and simplifies the capturing of the arguments. It actually allows your scripts to be built up using inheritance or roles, which can make for much more flexiblility. It does however have a limit and after that something like MooseX::App::Cmd is highly recommended.

```
package Foo;
use Moose;

with 'MooseX::Getopt';

has 'verbose'    => ( is => 'ro', isa => 'Bool' );
has 'company_id' => ( is => 'ro', isa => 'Int' );

sub run { ... }

# ...

Foo->new_with_options->run;

# ...

% perl foo.pl --verbose --company_id 10
```


One key thing to note is that you almost want to write an "interface" class with MooseX::Getopt, which itself delgates to a class or classes which do the bulk of the work.

MooseX::Types

- default Moose types are global
- default Moose types are strings
- this extension solves these problems (mostly)
- written and maintained by Moose team

Friday, June 7, 13

Moose types are global and so best practice dictates that you should namespace them, just as you would classes and package names. This works because Moose types are string, which internally get parsed and turned into Moose::Meta::TypeConstraint objects. In most cases however, you can substitute a Moose::Meta::TypeConstraint instance for a Moose string type and things will Just Work. MooseX::Types expliots this by creating subroutines that return pre-built, properly namespaced Moose::Meta::TypeConstraint instances. This has the benefit of catching type-name typos at compile time, as well as making it very easy to package types up into libraries.

MooseX::Types

- MooseX::Types::Path::Class
- MooseX::Types::Uri
- MooseX::Types::UUID
- MooseX::Types::Digest
- ... and many more

Friday, June 7, 13

This has lead to a fairly good sized set of MooseX::Type::* extenstention modules cropping up on CPAN. Here is just a short (non-comprehensive) list of some that I have used myself. Many of these not only provide type constraints, but also include sensible coercions as well.

MooseX::NonMoose

- Moose was built to play well with non-Moose code
- this extension takes care of the details of subclassing
- Just Works ™
- written and maintained by Moose team

Friday, June 7, 13

Moose was always meant to work well with non-Moose code, particularly modules from CPAN. This extension takes much of the pain out of subclassing non-Moose classes and having them still behave like proper Moose classes.

```
package SAuth::Web::Consumer;
use Moose;
use MooseX::NonMoose;

extends 'Plack::Component';

has 'client' => (
    is       => 'ro',
    isa      => 'SAuth::Web::Consumer::Client',
    required => 1,
);

has 'automate_access' => ( is => 'ro', isa => 'Bool', default => 0 );
has 'token_lifespan'  => ( is => 'ro', isa => 'Int' );
has 'access_for'      => ( is => 'ro', isa => 'ArrayRef[Str]' );

sub BUILD {
    my $self = shift;
    ($self->token_lifespan && $self->access_for)
        || SAuth::Core::Error->throw("You must specify a token_lifespan ...")
            if $self->automate_access;
}

sub prepare_app { (shift)->check_client_status }

sub call {
    my $self = shift;
    my $r    = Plack::Request->new( shift );
    $self->check_client_status;
    $self->client->call_service( $r )->finalize;
}
```


Friday, June 7, 13

Simply importing MooseX::NonMoose will give you all that you need to just simply extend a non-Moose class. This means you get the full benefits of Moose constructors, including checking for required attributes, initialization with default and builder, and calling of BUILD.

MooseX::Aliases

- aliasing of attributes properly
  - accessors
  - init_arg
- aliasing of methods properly
- written and maintained by Moose team

Friday, June 7, 13

Aliasing can be a tricky problem in Perl. This module allows you not only to alias an attribute and its generated methods, but also provides a way to properly alias methods.

```perl
package MyApp;
use Moose;
use MooseX::Aliases;

has 'this' => (
    is    => 'rw',
    isa   => 'Str',
    alias => 'that',
);

sub foo { print $_[0]->that }

alias bar => 'foo';

# ...
```

```perl
my $o = MyApp->new( that => 'Hi Planet!' );
$o->foo; # prints 'Hi Planet!'

$o->this('Hello World');
$o->bar; # prints 'Hello World'
```

MooseX::Storage

- serialization library for Moose objects
- uses MOP to properly collapse and expand objects
- written and maintained by Moose team

Friday, June 7, 13

Moose, and more specifically the MOP, creates a ton of metadata about your class. This module uses this metadata to provide a simple way to collapse and expand your objects in a very controlled way.

```perl
package Point;
use Moose;
use MooseX::Storage;

our $VERSION = '0.01';

with Storage('format' => 'JSON', 'io' => 'File');

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

# ...

my $p = Point->new(x => 10, y => 10);

$p->pack; # { __CLASS__ => 'Point-0.01', x => 10, y => 10 }
my $p2 = Point->unpack({ __CLASS__ => 'Point-0.01', x => 10, y => 10 });


$p->freeze; # { "__CLASS__" : "Point-0.01", "x" : 10, "y" : 10 }
my $p2 = Point->thaw('{ "__CLASS__" : "Point-0.01", "x" : 10, "y" : 10 }');

$p->store('my_point.json');
my $p2 = Point->load('my_point.json');
```


Friday, June 7, 13

MooseX::Storage provides 3 levels of serialization. First is pack/unpack which works simply with Perl data structures. The second is freeze/thaw which work with the specified serialization format (JSON in this example, but YAML and Storable are both available too) and produce and consume strings. And the last is store/load which, using the specified serialization format, will read and write to a file.

MooseX::Traits

- Roles can be applied at runtime to objects
- this extension simplifies that syntax
- written and maintained by Moose team

Friday, June 7, 13

Roles can be applied at runtime to specific instances. This can be a very powerful tool, but the default Moose API for this is cumbersome and involves some deep MOP trickery. MooseX::Traits simplifies this syntax and provides ways to do this efficiently and effectively, without forcing your users to dive into the MOP.

```perl
package My::Role;
use Moose::Role;

has foo => ( is => 'ro', isa => 'Int' required => 1 );

# ...

package My::Class;
use Moose;

with 'MooseX::Traits';

# ...

my $o = My::Class->with_traits('My::Role')->new( foo => 42 );

# OR

my $o = My::Class->new_with_traits(
    traits => [ 'My::Role' ],
    foo    => 42
);

$o->isa('My::Class'); # true
$o->does('My::Role'); # true
$o->foo; # 42
```


Note that the results of the with_traits version above could be saved to a variable and used to construct instances. This allows you to control the caching of the generated anon-classes yourself.

MooseX::SetOnce

- Write Once / Read Often attributes
- does what it says on the tin

Friday, June 7, 13

Moose has 'ro' (readonly) and 'rw' (read/write), but sometimes you need something different. This module provides that and really just does exactly what it says on the tin.

```
package My::Class;
use Moose;
use MooseX::SetOnce;

has 'some_attr' => (
    traits => [ qw[ SetOnce ] ],
    is     => 'rw',
    isa    => 'Str'
);

# ...

my $o = My::Class->new;

$o->some_attr(10);  # ok
$o->some_attr(20);  # BOOM
```

MooseX::Role::Parameterized

- Roles are awesome
- Roles are fun
- this makes them scary fun and wicked awesome
- written and maintained by Moose team

Friday, June 7, 13

Moose roles are a powerful mechanism for re-use. Shawn Moore decided that wasn't enough and he wanted more. So MooseX::Roles::Parameterized was born. This actually works somewhat like generics in languages like Java/C++/etc. It allows you to create roles dynamically based on a template and a set of parameters. It is important to note that this module is **VERY** powerful, and with great power comes great responsibility. Use it sensibly and it will reward you.

```
package Counter;
use MooseX::Role::Parameterized;

parameter 'name' => ( isa => 'Str', required => 1 );

role {
    my $p = shift;
    my $name = $p->name;

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

    method "inc_$name" => sub {
        my $self = shift;
        $self->$name( $self->$name + 1 )
    };
};
```


Friday, June 7, 13

So here we have a Counter role which is parameterized by the name of the attribute in which you wish to store the count. It also provides an increment method on that counter, which is also appropriately named based on the parameter.

```
package MyGame::Weapon;
use Moose;
with Counter => { name => 'enchantment' };

package MyGame::Wand;
use Moose;
with Counter => { name => 'zapped' };

# ...

my $weapon = MyGame::Weapon->new( enchantment => 10 );

$weapon->inc_enchantment; # 11

my $wand = MyGame::Wand->new( zapped => 100 );

$wand->inc_zapped; # 101
```


Friday, June 7, 13

Here is just a simple example of how we can use this role to re-use the same functionality, but provide completely different APIs.

MooseX::Clone

- cloning can be tricky
- this extension allows full range
  - clone all the things!
  - clone selectively
- written and maintained by Moose team

Friday, June 7, 13

Cloning is complicated business, simple shallow clones are easily dont in Perl, but deep clones are not so simple. Additionally some objects might not want to be cloned and sometime an object might not want to clone all the values within it. This module provides a number of features that allow you to have sensible defaults as well as fine grained control over this whole process.

```
package My::Class::Bar;
use Moose;

with 'MooseX::Clone';

has 'name' => ( is => 'ro', isa => 'Str' );

has 'foo' => (
    traits => [ qw[ Clone ] ],
    is     => 'ro',
    isa    => 'My::Class::Foo',
);

has 'baz' => (
    traits => [ qw[ NoClone ] ],
    is     => 'ro',
    isa    => 'My::Class::Baz',
);

package My::Class::Foo;
use Moose;

sub clone {
    my ( $self, %params ) = @_;
    # ...
}
```


Friday, June 7, 13

So the first thing is to use the MooseX::Clone role. This adds a sensible 'clone' method to your class, which in general just does the right thing. It also makes the "Clone" and "NoClone" attribute traits available, which allow you control how the values contained within those classes are treated during the cloning process. In this specific example, the foo attribute will actually use the custom My::Class::Foo::clone method when it is being cloned.

```perl
my $bar = My::Class::Bar->new(
    name => 'Really Great Bar',
    foo  => My::Class::Foo->new
);

# ...

my $copy = $bar->clone;

my $copy = $bar->clone( foo => [ %args ] );

my $copy = $bar->clone(
    name => 'Best Bar Ever',
    foo  => [ %args ],
);
```


It is also worth noting that the default clone method provided by MooseX::Clone accepts key/value pairs of constructor params, which allow you to selectively override attributes during the cloning process if you so choose.

The Sorta Good

Friday, June 7, 13

So, not all extensions are good, here are a few which are recommended as modules, but not as techniques.

MooseX::Singleton

- singletons are really global variables
  (don't be fooled!)
- sometimes globals are useful
  (sometimes)
- written and maintained by Moose team
  (well a few people really, the rest of us like Bread::Board)

Friday, June 7, 13

Singletons can be really useful at times, but generally are not recommended. By declaring your class a singleton, you are preventing it from being re-used in other contexts. In general you should not let the class itself control its own lifecycle, this should be a choice made by the user of the class, rather then the class itself.

MooseX::Singleton

Friday, June 7, 13

Singletons can be really useful at times, but generally are not recommended. By declaring your class a singleton, you are preventing it from being re-used in other contexts. In general you should not let the class itself control its own lifecycle, this should be a choice made by the user of the class, rather then the class itself.

MooseX::ClassAttribute

- creates class scoped attributes
- basically the same as a package scoped variable with an accessor
- written and maintained by Moose team
  (really just Dave, but sometimes others find this useful)

Friday, June 7, 13

The idea of a class holding state is useful, but again is a lifecycle decision perhaps best left to the users of your class.

MooseX::ClassAttribute

```
package MyApp;
use MooseX::Singleton;

has env => (
    is      => 'rw',
    isa     => 'HashRef[Str]',
    default => sub { \%ENV },
);

# ...
```

```
delete MyApp->env->{PATH};
my $instance = MyApp->instance;
my $same = MyApp->instance;
```


Friday, June 7, 13

The idea of a class holding state is useful, but again is a lifecycle decision perhaps best left to the users of your class.

MooseX::SemiAffordanceAccessor

- Moose accessor style is opinionated
- Not everyone shares those opinions
- this extension changes that
- written and maintained by Moose team
  (again, just Dave really)

Friday, June 7, 13

This breaks convention, which in an of itself is not a problem, the problem is that it still looks like it is using convention. Too much is hidden under the covers and the only indication is the "use MooseX::SemiAffordanceAccessor"

MooseX::SemiAffordanceAccessor

```
package Point;
use Moose;
use MooseX::SingleAffordanceAccessor;

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

# ...

my $point = Point->new( x => 10, y => 10 );

$point->x # 10
$point->set_x( 20 );
$point->x # 20
```


Friday, June 7, 13

This breaks convention, which in an of itself is not a problem, the problem is that it still looks like it is using convention. Too much is hidden under the covers and the only indication is the "use MooseX::SemiAffordanceAccessor"

MooseX::MethodAttributes

- subroutine attributes are tricky to deal with
- this extension provides a saner API
- used internally in Catalyst

Friday, June 7, 13

Subroutine attributes, not to be confused with Moose attributes, are a problematic feature at best. They provide a nice sugar syntax for things like Catalyst, but the API they provide is horrible. This module makes that horrible API palettable.

MooseX::MethodAttributes

```
package MyClass;
use Moose;
use MooseX::MethodAttributes;

sub foo : Bar Baz('corge') { ... }

# ...

my $attrs = MyClass->meta
                   ->get_method('foo')
                   ->attributes;
# ["Bar", "Baz('corge')"]
```


Friday, June 7, 13

Subroutine attributes, not to be confused with Moose attributes, are a problematic feature at best. They provide a nice sugar syntax for things like Catalyst, but the API they provide is horrible. This module makes that horrible API palettable.

The Not So Good

MooseX::Declare

- great idea upon an unstable foundation
- Devel::Declare == evil
- Devel::CallParser == sane-ish
- p5-mop project

Evaluation

- how recent is the latest version?
  - does it use the latest Moose APIs
- evidence of community input?
  - what does #moose have to say?
  - is tracked in the ‘x_conflicts’ section of Moose’s META.yml file?

Friday, June 7, 13

Moose has a strong community and is open to input and discussion. It only serves to the benefits of MooseX module authors to get involved and get feedback.

Questions?