



Moose, CPAN and Legacy Code

Moosifiying

present participle of moos·i·fy (Verb)

Verb:

1. The act of refactoring Perl code to use Moose.
2. Mutating an organism, typically through the use of magic or some form of cosmic radiation, to become more Moose-like.

Typical Scenario

Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

Now lets actually talk about a typical scenario. You have an existing code base, it was built and has been running for years maybe. It followed the best practices of the time, but over the years has been maintained by different programmers and subject to tight enhancement deadlines, etc etc etc. Even the best code eventually bit-rots and the job of maintaining that can be tedious and even demoralizing.

At some point, it no longer makes sense to continue down this path, and so a decision is made, ... we need to use Moose, follow the best practices of the day (PBP, etc) and update our APIs to be better suited for the business needs.

However, very rarely does it make good business sense to throw away the accumulated knowledge and experience of an older codebase, no matter how horrible. It is also rare that you can start completely from scratch. So there will exist a need for new code to be able to work with old code. In my experience, the best approach to this (with the least risk) is ...

New Code

- ▶ Moose
- ▶ Modern Coding Standards
- ▶ Modern APIs

Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

New Code

- ▶ Moose
- ▶ Modern Coding Standards
- ▶ Modern APIs



Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

New Code

- ▶ Moose
- ▶ Modern Coding Standards
- ▶ Modern APIs

Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

Class

Module

Class

Class

Class

Class

New Code

- ▶ Moose
- ▶ Modern Coding Standards
- ▶ Modern APIs

Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

Class

Module

Class

Class

Class

Class

New Code

- ▶ Moose
- ▶ Modern Coding Standards
- ▶ Modern APIs

Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

Class

Module

Class

Class

Class

Class

New Code

- ▶ Moose
- ▶ Modern Coding Standards
- ▶ Modern APIs

Old Code

- ▶ No Moose
- ▶ Old Coding Standards
- ▶ Legacy APIs

Class

Class

Module

Class

Class

Class

Class

**Moose was
designed to
work well with
non-Moose
code**

support for partial
and incremental
moosification
is a key feature

Origins of Moose

- ▶ Writing lots of Perl 6 using Pugs
- ▶ In comparison, Perl 5 was frustrating and tedious
- ▶ Explored moving to Ruby (yuk!)
- ▶ Too much invested in Perl 5 (20-30K LoC)
- ▶ Ended up just writing Moose
- ▶ Which lead to lots of Moosification

Conversion & Extension

```
package Person;

use strict;
use warnings;

sub new {
    my ($class, $name) = @_;
    my $self = bless { name => $name } => $class;
    return $self;
}

sub getName {
    my ($self) = @_;
    return $self->{'name'};
}

sub setName {
    my ($self, $name) = @_;
    die "You must provide a name" unless $name;
    $self->{'name'} = $name;
}
```



```

package Person;

use strict;
use warnings;

sub new {
    my ($class, $name) = @_;
    my $self = $class->new;
    return $self;
}

sub getName {
    my ($self) = @_;
    return $self->{name};
}

sub setName {
    my ($self, $name) = @_;
    die "You must provide a name" unless $name;
    $self->{name} = $name;
}

```

```

package Person;
use Moose;

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

```

\$class;

```
package Person;
use Moose;

sub new {
    my ($class, $name) = @_ ;
    my $self = bless { name => $name } => $class;
    return $self;
}

sub getName {
    my ($self) = @_ ;
    return $self->{'name'};
}

sub setName {
    my ($self, $name) = @_ ;
    die "You must provide a name" unless $name;
    $self->{'name'} = $name;
}
```

Friday, June 7, 13

Importing blessed and confess typically are not an issue, these are well known functions and so not typically used as method names.

As for the Moose sugar, we chose these names carefully to avoid conflicts. And the ‘meta’ method does not often conflict either.

Lastly, adding Moose::Object to the @ISA will only be an issue if you have methods named like these. Again we chose them carefully and we keep them minimal for a reason. It important to note that if @ISA is already set, it will actually do nothing

▸ imports strict and warnings

- imports strict and warnings
- Imports Scalar::Util::blessed and Carp::confess

- imports strict and warnings
- Imports Scalar::Util::blessed and Carp::confess
- Imports the Moose sugar
 - extends, with, has
 - before, after, around
 - super/override, inner/augment

- imports strict and warnings
- Imports Scalar::Util::blessed and Carp::confess
- Imports the Moose sugar
 - extends, with, has
 - before, after, around
 - super/override, inner/augment
- Adds 'meta' method

- imports strict and warnings
- Imports Scalar::Util::blessed and Carp::confess
- Imports the Moose sugar
 - extends, with, has
 - before, after, around
 - super/override, inner/augment
- Adds 'meta' method
- Adds 'Moose::Object' to the @ISA (only if @ISA is empty)
 - new, does, dump
 - BUILDARGS, BUILDALL, DEMOLISHALL

```

package Person;
use Moose;

sub BUILDARGS {
    my $class = shift;
    if ( @_ == 1 && !ref $_[0] ) {
        return $class->SUPER::BUILDARGS( 'name' => $_[0] );
    }
    else {
        return $class->SUPER::BUILDARGS( @_ );
    }
}

sub getName {
    my ($self) = @_;
    return $self->{'name'};
}

sub setName {
    my ($self, $name) = @_;
    die "You must provide a name" unless $name;
    $self->{'name'} = $name;
}

```

BUILDARGS

BUILDARGS

- ▶ Moose standardizes constructor arguments
 - ▶ %hash or \%hash_ref

BUILDARGS

- ▶ Moose standardizes constructor arguments
 - ▶ %hash or \%hash_ref
- ▶ BUILDARGS runs ***before*** the object is constructed

BUILDARGS

- ▶ Moose standardizes constructor arguments
 - ▶ %hash or \%hash_ref
- ▶ BUILDARGS runs ***before*** the object is constructed
- ▶ recommended usage:
 - ▶ use ‘around’ or SUPER::BUILDARGS
 - ▶ only handle *your* exception case

```

package Person;
use Moose;

around 'BUILDARGS' => sub {
    my $orig = shift;
    my $class = shift;
    if ( @_ == 1 && !ref $_[0] ) {
        return $class->$orig( 'name' => $_[0] );
    }
    else {
        return $class->$orig( @_ );
    }
};

sub getName {
    my ($self) = @_;
    return $self->{'name'};
}

sub setName {
    my ($self, $name) = @_;
    die "You must provide a name" unless $name;
    $self->{'name'} = $name;
}

```



```

package Person;
use Moose;

around 'BUILDARGS' => sub {
    my $orig = shift;
    my $class = shift;
    if ( @_ == 1 && !ref $_[0] ) {
        return $class->$orig( 'name' => $_[0] );
    }
    else {
        return $class->$orig( @_ );
    }
};

has 'name' => (
    isa      => 'Str',
    reader   => 'getName',
    writer   => 'setName',
);

```

Tao of Conversion

Tao of Conversion

- Small step.

Tao of Conversion

- ▶ Small step.
- ▶ Test. (it is okay if they fail)

Tao of Conversion

- ▶ Small step.
- ▶ Test. (it is okay if they fail)
- ▶ Small step.

Tao of Conversion

- ▶ Small step.
- ▶ Test. (it is okay if they fail)
- ▶ Small step.
- ▶ Test and test again. (it is still okay if they fail)

Tao of Conversion

- ▶ Small step.
- ▶ Test. (it is okay if they fail)
- ▶ Small step.
- ▶ Test and test again. (it is still okay if they fail)
- ▶ Small step.

Tao of Conversion

- ▶ Small step.
- ▶ Test. (it is okay if they fail)
- ▶ Small step.
- ▶ Test and test again. (it is still okay if they fail)
- ▶ Small step.
- ▶ Test, test again and then test some more! (...)

```
package Person;
use Moose;

with 'MooseX::OneArgNew' => {
    type      => 'Str',
    init_arg => 'name',
};

has 'name' => (
    isa      => 'Str',
    reader   => 'getName',
    writer   => 'setName',
);
```

```
package Person;
use Moose;

with 'MooseX::OneArgNew' => {
    type      => 'Str',
    init_arg => 'name',
};

has 'name' => (
    isa      => 'Str',
    accessor => 'name',
    reader   => 'getName',
    writer   => 'setName',
);
```

2 APIs & No Code!

```

package ConfigLoader;
use strict;
use warnings;
use YAML qw[ LoadFile ];

sub new {
    my ($class, %args) = @_;
    my $self = bless {} => $class;
    if ( exists $args{'-config'} ) {
        $self->{'_config'} = $args{'-config'};
    }
    elsif ( exists $args{'-config_file'} ) {
        $self->{'_config'} = LoadFile( $args{'-config_file'} );
    }
    return $self;
}

sub is_config_ready { $_[0]->{'_config'} ? 1 : 0 }

sub get {
    my ($self, $name) = @_;
    return $self->{'_config'}->{ $name };
}

sub username { $_[0]->get('username') }
sub password { $_[0]->get('password') }

```

```

package ConfigLoader;
use Moose;
use YAML qw[ LoadFile ];

has '_config' => (
    init_arg => '-config',
    is       => 'rw',
    isa      => 'HashRef',
    lazy     => 1,
    default  => sub { +{} },
    predicate => 'is_config_ready'
);

sub BUILD {
    my ($self, $params) = @_;
    if ( exists $params->{'-config_file'} ) {
        $self->_config( LoadFile( $params->{'-config_file'} ) );
    }
}

sub get {
    my ($self, $name) = @_;
    return $self->_config->{ $name };
}

sub username { $_[0]->get('username') }
sub password { $_[0]->get('password') }

```

```

package ConfigLoader;
use Moose;
use YAML qw[ LoadFile ];

has '_config' => (
    init_arg   => '-config',
    is         => 'rw',
    isa        => 'HashRef',
    lazy       => 1,
    default    => sub { +{} },
    predicate  => 'is_config_ready'
);

sub BUILD {
    my ($self, $params) = @_;
    if ( exists $params->{'-config_file'} ) {
        $self->_config( LoadFile( $params->{'-config_file'} ) );
    }
}

sub get {
    my ($self, $name) = @_;
    return $self->_config->{ $name };
}

sub username { $_[0]->get('username') }
sub password { $_[0]->get('password') }

```

```

package ConfigLoader;
use Moose;
use YAML qw[ LoadFile ];

has '_config' => (
    init_arg => '-config',
    is       => 'rw',
    isa      => 'HashRef',
    lazy     => 1,
    default  => sub { +{} },
    predicate => 'is_config_ready'
);

sub BUILD {
    my ($self, $params) = @_;
    if ( exists $params->{'-config_file'} ) {
        $self->_config( LoadFile( $params->{'-config_file'} ) );
    }
}

sub get {
    my ($self, $name) = @_;
    return $self->_config->{ $name };
}

sub username { $_[0]->get('username') }
sub password { $_[0]->get('password') }

```


BUILD

BUILD

- runs *after* the object is constructed

BUILD

- ▶ runs ***after*** the object is constructed
- ▶ gets HASH ref of constructor parameters

BUILD

- ▶ runs ***after*** the object is constructed
- ▶ gets HASH ref of constructor parameters
- ▶ has access to fully initialized \$self

BUILD

- ▶ runs ***after*** the object is constructed
- ▶ gets HASH ref of constructor parameters
- ▶ has access to fully initialized \$self
- ▶ is called for all classes in the inheritance graph by BUILDALL

BUILD

- ▶ runs ***after*** the object is constructed
- ▶ gets HASH ref of constructor parameters
- ▶ has access to fully initialized \$self
- ▶ is called for all classes in the inheritance graph by BUILDALL
- ▶ ... but when possible, use default or builder

```

package ConfigLoader;
use Moose;
use YAML qw[ LoadFile ];

has '_config' => (
    traits => [ 'Hash' ],
    init_arg => '-config',
    is => 'rw',
    isa => 'HashRef',
    lazy => 1,
    default => sub { +{} },
    predicate => 'is_config_ready',
    handles => {
        'get' => 'get',
        'username' => [ 'get', 'username' ],
        'password' => [ 'get', 'password' ]
    }
);

sub BUILD {
    my ($self, $params) = @_;
    if ( exists $params->{'-config_file'} ) {
        $self->_config( LoadFile( $params->{'-config_file'} ) )
    }
}

```

handles

handles

- can delegate to embedded objects

handles

- ▶ can delegate to embedded objects
- ▶ using native traits can delegate to built-in Perl types (Hash, Array, Str, Number, etc).

handles

- ▶ can delegate to embedded objects
- ▶ using native traits can delegate to built-in Perl types (Hash, Array, Str, Number, etc).
- ▶ useful for building well encapsulated APIs

handles

- ▶ can delegate to embedded objects
- ▶ using native traits can delegate to built-in Perl types (Hash, Array, Str, Number, etc).
- ▶ useful for building well encapsulated APIs
- ▶ ‘*curried*’ delegations can be used to create complex relationships very simply

```

package ConfigLoader;
use Moose;
use YAML qw[ LoadFile ];

has '_config' => (
    traits => [ 'Hash' ],
    init_arg => '-config',
    is => 'rw',
    isa => 'HashRef',
    lazy => 1,
    default => sub { +{} },
    predicate => 'is_config_ready',
    handles => {
        'get' => 'get',
        'username' => [ 'get', 'username' ],
        'password' => [ 'get', 'password' ]
    }
);

sub BUILD {
    my ($self, $params) = @_;
    if ( exists $params->{'-config_file'} ) {
        $self->_config( LoadFile( $params->{'-config_file'} ) )
    }
}

```

```

package ConfigLoader;
use Moose;
use MooseX::Aliases;
use MooseX::Types::Path::Class;
use YAML qw[ LoadFile ];

has 'config_file' => (
    alias      => '-config_file',
    is         => 'ro',
    writer     => 'reload_config_file',
    isa        => 'Path::Class::File',
    coerce     => 1,
    trigger    => sub { $_[0]->_config( LoadFile( $_[1] ) ) }
);

has 'config' => (
    traits     => [ 'Hash' ],
    alias      => '-config',
    accessor   => '_config',
    isa        => 'HashRef',
    lazy       => 1,
    default    => sub { +{} },
    predicate  => 'is_config_ready',
    handles    => {
        'get'      => 'get',
        'username' => [ 'get', 'username' ],
        'password' => [ 'get', 'password' ]
    }
);

```

```

package ConfigLoader;
use Moose;
use MooseX::Aliases;
use MooseX::Types::Path::Class;
use YAML qw[ LoadFile ];

has 'config_file' => (
    alias      => '-config_file',
    is         => 'ro',
    writer     => 'reload_config_file',
    isa        => 'Path::Class::File',
    coerce     => 1,
    trigger    => sub { $_[0]->_config( LoadFile( $_[1] ) ) }
);

has 'config' => (
    traits     => [ 'Hash' ],
    alias      => '-config',
    accessor   => '_config',
    isa        => 'HashRef',
    lazy       => 1,
    default    => sub { +{} },
    predicate  => 'is_config_ready',
    handles    => {
        'get'      => 'get',
        'username' => [ 'get', 'username' ],
        'password' => [ 'get', 'password' ]
    }
);

```

```

package ConfigLoader;
use Moose;
use MooseX::Aliases;
use MooseX::Types::Path::Class;
use YAML qw[ LoadFile ];

has 'config_file' => (
    alias      => '-config_file',
    is         => 'ro',
    writer     => 'reload_config_file',
    isa        => 'Path::Class::File',
    coerce     => 1,
    trigger    => sub { $_[0]->_config( LoadFile( $_[1] ) ) }
);

has 'config' => (
    traits     => [ 'Hash' ],
    alias      => '-config',
    accessor   => '_config',
    isa        => 'HashRef',
    lazy       => 1,
    default    => sub { +{} },
    predicate  => 'is_config_ready',
    handles    => {
        'get'      => 'get',
        'username' => [ 'get', 'username' ],
        'password' => [ 'get', 'password' ]
    }
);

```


MooseX::*

There is a good chance
you are not the first person
to have this problem.

MooseX::*

MooseX::*

► MooseX::Aliases

MooseX::*

- ▶ MooseX::Aliases
- ▶ MooseX::OneArgNew

MooseX::*

- ▶ MooseX::Aliases
- ▶ MooseX::OneArgNew
- ▶ MooseX::UndefTolerant

MooseX::*

- ▶ MooseX::Aliases
- ▶ MooseX::OneArgNew
- ▶ MooseX::UndefTolerant
- ▶ MooseX::Attribute::TypeConstraint::CustomizeFatal

MooseX::*

- ▶ MooseX::Aliases
- ▶ MooseX::OneArgNew
- ▶ MooseX::UndefTolerant
- ▶ MooseX::Attribute::TypeConstraint::CustomizeFatal
- ▶ MooseX::Emulate::Class::Accessor::Fast

MooseX::*

- ▶ MooseX::Aliases
- ▶ MooseX::OneArgNew
- ▶ MooseX::UndefTolerant
- ▶ MooseX::Attribute::TypeConstraint::CustomizeFatal
- ▶ MooseX::Emulate::Class::Accessor::Fast
- ▶ MooseX::NonMoose

MooseX::*

- ▶ MooseX::Aliases
- ▶ MooseX::OneArgNew
- ▶ MooseX::UndefTolerant
- ▶ MooseX::Attribute::TypeConstraint::CustomizeFatal
- ▶ MooseX::Emulate::Class::Accessor::Fast
- ▶ MooseX::NonMoose
- ▶ ... and more

MooseX::NonMoose

MooseX::NonMoose

- ▶ Sometimes you can't Moosify

MooseX::NonMoose

- ▶ Sometimes you can't Moosify
- ▶ Delegation is typically the right answer
(but not always the practical one)

MooseX::NonMoose

- ▶ Sometimes you can't Moosify
- ▶ Delegation is typically the right answer
(but not always the practical one)
- ▶ Subclassing non-Moose classes with Moose is
actually pretty simple
(especially with MooseX::NonMoose)

```
package Plack::Handler::FCGI::Engine;
use Moose;
use MooseX::NonMoose;

use Plack::Handler::FCGI::Engine::ProcManager;

our $VERSION    = '0.18';
our $AUTHORITY = 'cpan:STEVAN';

extends 'Plack::Handler::FCGI';

has 'manager' => (
    is      => 'ro',
    isa     => 'Str | ClassName',
    default => sub { 'Plack::Handler::FCGI::Engine::ProcManager' },
);
```

```
package Jackalope::REST::Util::HashExpander;
use Moose;
use MooseX::NonMoose;

our $VERSION    = '0.01';
our $AUTHORITY = 'cpan:STEVAN';

extends 'CGI::Expand';

sub separator { ':' }
```

```
package Plack::App::Path::Router::PSGI;
use Moose;
use MooseX::NonMoose;

our $VERSION    = '0.04';
our $AUTHORITY = 'cpan:STEVAN';

extends 'Plack::Component';

has 'router' => (
    is      => 'ro',
    isa     => 'Path::Router',
    required => 1,
);

sub call {
    my ($self, $env) = @_;
    # ...
}
```



```

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;
}

```

Delegation

Delegation

- ▶ Sometimes you can't subclass
(Inside-Out classes, AUTOLOAD or some other weirdness)

Delegation

- ▶ Sometimes you can't subclass
(Inside-Out classes, AUTOLOAD or some other weirdness)
- ▶ Moose by default requires HASH based instances

Delegation

- ▶ Sometimes you can't subclass
(Inside-Out classes, AUTOLOAD or some other weirdness)
- ▶ Moose by default requires HASH based instances
- ▶ Delegation *is* the right answer

```

package Data::Riak::HTTP::Response;
use Moose;

use overload '""' => 'as_string', fallback => 1;

has 'http_response' => (
    is      => 'ro',
    isa     => 'HTTP::Response',
    required => 1,
    handles => {
        code      => 'code',
        status_code => 'code',
        message   => 'content',
        value     => 'content',
        is_success => 'is_success',
        is_error  => 'is_error',
        as_string => 'as_string',
        header    => 'header',
        headers   => 'headers'
    }
);

has 'parts' => (
    is      => 'ro',
    isa     => 'ArrayRef[HTTP::Message]',
    lazy    => 1,
    default => sub {
        my $self = shift;
        [ $self->_deconstruct_parts->( $self->http_response ) ]
    }
);

```

```

package Data::Riak::HTTP::Response;
use Moose;

use overload '""' => 'as_string', fallback => 1;

has 'http_response' => (
    is      => 'ro',
    isa     => 'HTTP::Response',
    required => 1,
    handles => {
        code      => 'code',
        status_code => 'code',
        message    => 'content',
        value      => 'content',
        is_success  => 'is_success',
        is_error    => 'is_error',
        as_string   => 'as_string',
        header      => 'header',
        headers     => 'headers'
    }
);

has 'parts' => (
    is      => 'ro',
    isa     => 'ArrayRef[HTTP::Message]',
    lazy    => 1,
    default => sub {
        my $self = shift;
        [ $self->_deconstruct_parts->( $self->http_response ) ]
    }
);

```

```

package Data::Riak::HTTP::Response;
use Moose;

use overload '""' => 'as_string', fallback => 1;

has 'http_response' => (
    is      => 'ro',
    isa     => 'HTTP::Response',
    required => 1,
    handles => {
        code      => 'code',
        status_code => 'code',
        message    => 'content',
        value      => 'content',
        is_success  => 'is_success',
        is_error    => 'is_error',
        as_string   => 'as_string',
        header      => 'header',
        headers     => 'headers'
    }
);

has 'parts' => (
    is      => 'ro',
    isa     => 'ArrayRef[HTTP::Message]',
    lazy    => 1,
    default => sub {
        my $self = shift;
        [ $self->_deconstruct_parts->( $self->http_response ) ]
    }
);

```


AUTOLOAD

Friday, June 7, 13

So, AUTOLOAD is an interesting thing, it can be useful, but really, it is Evil. It is evil largely because it is so flexible and there are so many ways in which it can be used and abused. It is also expensive, both performance wise and conceptually.

And, in this particular context, it makes delegation tricky.

AUTOLOAD

► Is Evil!

AUTOLOAD

- Is Evil!
- Is Expensive!

AUTOLOAD

- Is Evil!
- Is Expensive!
- Makes delegation tricky!

AUTOLOAD

- ▶ Is Evil!
- ▶ Is Expensive!
- ▶ Makes delegation tricky!
- ▶ Requires you to specify the list of methods, and does not play well with RegExp based delegation

```
package Person;
use strict;
use warnings;

sub new { shift; bless { @_ } }

sub AUTOLOAD {
    my $method = (split ':' => our $AUTOLOAD)[-1];
    return $_[0]->{ $method };
}

package Employee;
use Moose;

has 'human' => (
    is      => 'bare',
    isa     => 'Person',
    required => 1,
    handles => [qw[
        first_name
        last_name
    ]]
);
```

```
package Person;
use strict;
use warnings;

sub new { shift; bless { @_ } }

sub AUTOLOAD {
    my $method = (split ':' => our $AUTOLOAD)[-1];
    return $_[0]->{ $method };
}

package Employee;
use Moose;

has 'human' => (
    is      => 'bare',
    isa     => 'Person',
    required => 1,
    handles => {
        first => 'first_name',
        last  => 'last_name'
    }
);
```

```
package Person;
use strict;
use warnings;

sub new { shift; bless { @_ } }

sub AUTOLOAD {
    my $method = (split ':' => our $AUTOLOAD)[-1];
    return $_[0]->{ $method };
}

package Employee;
use Moose;

has 'human' => (
    is      => 'bare',
    isa     => 'Person',
    required => 1,
    handles => qr/.*_name$/
);

# Can't locate object method "first_name" via
# package "Employee" at 002-test.t line 37.
```


Tips, Tricks & Edge Cases

Constructors not named new

Tips, Tricks & Edge Cases

Constructors not named new

- ▶ Move *all* logic into Moose

Tips, Tricks & Edge Cases

Constructors not named new

- ▶ Move *all* logic into Moose
 - ▶ Including old API support with BUILDARGS

Tips, Tricks & Edge Cases

Constructors not named new

- ▶ Move *all* logic into Moose
 - ▶ Including old API support with BUILDARGS
 - ▶ Including initialization code with BUILD

Tips, Tricks & Edge Cases

Constructors not named new

- ▶ Move *all* logic into Moose
 - ▶ Including old API support with BUILDARGS
 - ▶ Including initialization code with BUILD
- ▶ Alias old constructor to new Moose based

Tips, Tricks & Edge Cases

Constructors not named new

- ▶ Move *all* logic into Moose
 - ▶ Including old API support with BUILDARGS
 - ▶ Including initialization code with BUILD
- ▶ Alias old constructor to new Moose based
 - ▶ use MooseX::Aliases to do this

Tips, Tricks & Edge Cases

Using method modifiers on Moose accessors

Tips, Tricks & Edge Cases

Using method modifiers on Moose accessors

- Can be really useful for decorated accessors

Tips, Tricks & Edge Cases

Using method modifiers on Moose accessors

- Can be really useful for decorated accessors
- But, Moose does ***not*** call accessors during object construction

Tips, Tricks & Edge Cases

Using method modifiers on Moose accessors

- ▶ Can be really useful for decorated accessors
- ▶ But, Moose does ***not*** call accessors during object construction
- ▶ Be careful

Tips, Tricks & Edge Cases

Aliasing methods

Tips, Tricks & Edge Cases

Aliasing methods

- The alias should always call the method properly

Tips, Tricks & Edge Cases

Aliasing methods

- The alias should always call the method properly
- Do not alias via the typeglob (`*foo = \&bar`)

Tips, Tricks & Edge Cases

Aliasing methods

- The alias should always call the method properly
- Do not alias via the typeglob (`*foo = \&bar`)
- Just use `MooseX::Aliases`

Tips, Tricks & Edge Cases

Aliasing methods

- ▶ The alias should always call the method properly
- ▶ Do not alias via the typeglob (`*foo = \&bar`)
- ▶ Just use `MooseX::Aliases`
- ▶ Nuff Said

Questions?