

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.





This talk is primarily about Moosifying, and all the best practices, pitfalls and gotchas that come along with it.

Typical Scenario





Old Code

- No Moose
- Old Coding Standards
- Legacy APIs



YAPC::NA

Friday, June 7, 13

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. 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 ...

- **▶** Moose
- ▶ Modern Coding Standards
- ► Modern APIs

Old Code

- No Moose
- Old Coding Standards
- Legacy APIs



YAPC::NA

Friday, June 7, 13

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 ...

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



Old Code

- No Moose
- ▶ Old Coding Standards
- Legacy APIs





Friday, June 7, 13

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. 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 ...

- **Moose**
- ► Modern Coding Standards
- ► Modern APIs

Old Code

- No Moose
- Old Coding Standards
- Legacy APIs

Class

Module
Class Class
Class





Friday, June 7, 13

Divide and conquer.

Even the worst legacy code bases typically have some componentization. Sometimes, this is just the class level, sometimes it is a set of cooperating classes (hopefully organized into some kind of module). The key is to start to look at these as isolated units and if you have tests, improve them, if you don't have tests, write them.

5

Then you simply convert them to Moose, incrementally as dictated by the needs of the new codebase. Meanwhile providing the old interface for the legacy code (which is verified by your tests).

- **Moose**
- ► Modern Coding Standards
- ► Modern APIs

Old Code

- No Moose
- Old Coding Standards
- Legacy APIs

Class

Module
Class Class
Class



YAPC::NA

Friday, June 7, 13

Divide and conquer.

Even the worst legacy code bases typically have some componentization. Sometimes, this is just the class level, sometimes it is a set of cooperating classes (hopefully organized into some kind of module). The key is to start to look at these as isolated units and if you have tests, improve them, if you don't have tests, write them.

5

Then you simply convert them to Moose, incrementally as dictated by the needs of the new codebase. Meanwhile providing the old interface for the legacy code (which is verified by your tests).

- **▶** Moose
- ► Modern Coding Standards
- ► Modern APIs

Old Code

- No Moose
- Old Coding Standards
- Legacy APIs

Class

Module
Class Class
Class



YAPC::NA

Friday, June 7, 13

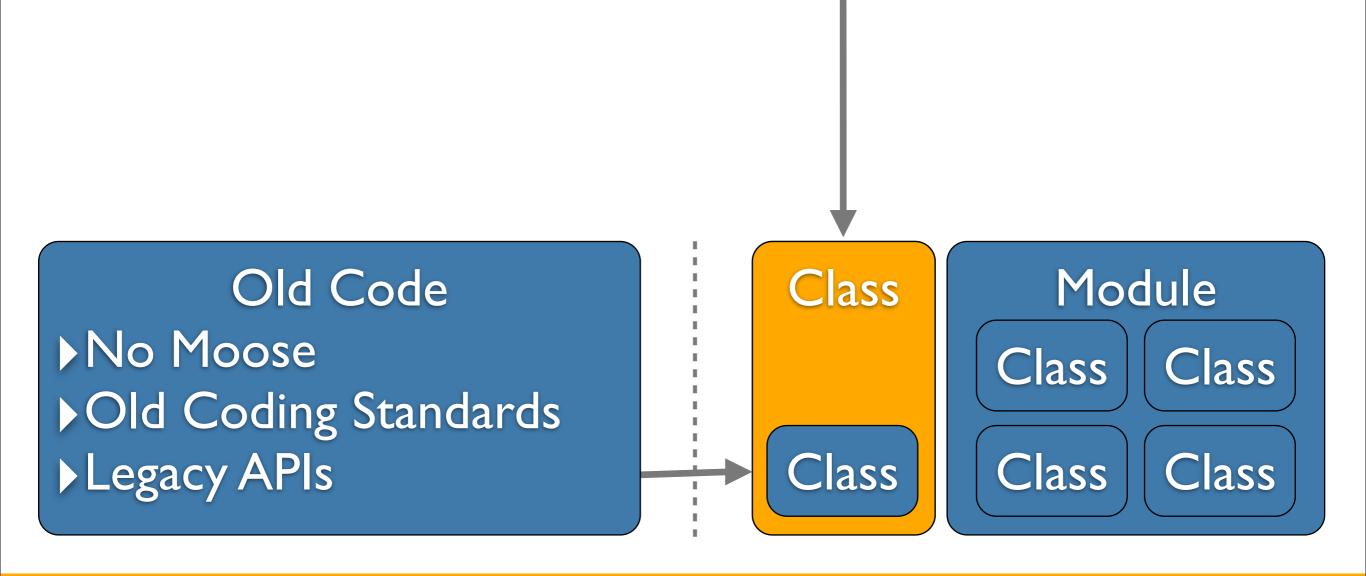
Divide and conquer.

Even the worst legacy code bases typically have some componentization. Sometimes, this is just the class level, sometimes it is a set of cooperating classes (hopefully organized into some kind of module). The key is to start to look at these as isolated units and if you have tests, improve them, if you don't have tests, write them.

5

Then you simply convert them to Moose, incrementally as dictated by the needs of the new codebase. Meanwhile providing the old interface for the legacy code (which is verified by your tests).

- **▶** Moose
- ▶ Modern Coding Standards
- ► Modern APIs



Friday, June 7, 13

Divide and conquer.

Even the worst legacy code bases typically have some componentization. Sometimes, this is just the class level, sometimes it is a set of cooperating classes (hopefully organized into some kind of module). The key is to start to look at these as isolated units and if you have tests, improve them, if you don't have tests, write them.

Then you simply convert them to Moose, incrementally as dictated by the needs of the new codebase. Meanwhile providing the old interface for the legacy code (which is verified by your tests).

YAPC::NA

Moose was designed to work well with non-Moose code





support for partial and incremental moosification is a key feature





Friday, June 7, 13

So, in addition to what I said in the previous slide, this point needs to be made too.

Moose would not be nearly as useful if it required everything else to also use Moose. After all, CPAN is a large reason to use Perl 5, so it was important that Moose meld easily with older style Perl 5 OO. And throwing away thousands (or tens of thousands) of LoC is very often just not an option.

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 {
              package Person;
   my ($cl
   my $sel
                                         $class;
              use Moose;
    return
}
              has 'name' => (
sub getName
                   is => 'rw',
   my ($se
                   isa => 'Str'
    return
              );
}
sub setName
   my (self, sname) = @_;
   die "You must provide a name" unless $name;
    $self->{'name'} = $name;
```





| |

```
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





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 Scalar::Util::blessed and Carp::confess





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 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;
```





Friday, June 7, 13

One of the promises of Moose is that it removes the need for you to think about the details of object construction. So, lets first work on the constructor.





Friday, June 7, 13

Moose standardizes constructor params to be just a hash or hashref, this is not always workable with older APIs, and this is where BUILDARGS comes in.

BUILDARGS always runs first, you do not have a \$self, only an unblessed \$class. BUILDARGS is expected to produce a canonicalized HASHref, however ...

It is recommended to always either use 'around' or SUPER::BUILDARGS and pass the call back to the superclasses, this is typically Moose::Object, but might not always be. Because of this, it is always recommended that you try to detect your specific exception case and only handle that, everything else can be passed up the chain to the superclass.

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





Friday, June 7, 13

Moose standardizes constructor params to be just a hash or hashref, this is not always workable with older APIs, and this is where BUILDARGS comes in.

BUILDARGS always runs first, you do not have a \$self, only an unblessed \$class. BUILDARGS is expected to produce a canonicalized HASHref, however ...

It is recommended to always either use 'around' or SUPER::BUILDARGS and pass the call back to the superclasses, this is typically Moose::Object, but might not always be. Because of this, it is always recommended that you try to detect your specific exception case and only handle that, everything else can be passed up the chain to the superclass.

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





Friday, June 7, 13

Moose standardizes constructor params to be just a hash or hashref, this is not always workable with older APIs, and this is where BUILDARGS comes in.

BUILDARGS always runs first, you do not have a \$self, only an unblessed \$class. BUILDARGS is expected to produce a canonicalized HASHref, however ...

It is recommended to always either use 'around' or SUPER::BUILDARGS and pass the call back to the superclasses, this is typically Moose::Object, but might not always be. Because of this, it is always recommended that you try to detect your specific exception case and only handle that, everything else can be passed up the chain to the superclass.

- 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





Friday, June 7, 13

Moose standardizes constructor params to be just a hash or hashref, this is not always workable with older APIs, and this is where BUILDARGS comes in.

BUILDARGS always runs first, you do not have a \$self, only an unblessed \$class. BUILDARGS is expected to produce a canonicalized HASHref, however ...

```
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;
}
```





Friday, June 7, 13

Or we can use the Moose 'around' sugar, which is the more idiomatic approach.

```
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',
);
```





Friday, June 7, 13

So, the next thing to do is to remove those accessors, however typically Moose wants to just use the 'name' method for the accessor, which would not keep the API compatible. However you can tell Moose attributes what you actually want to call the reader and writers, as we do here.





Friday, June 7, 13

Small step.





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





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





- 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. (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! (...)









Friday, June 7, 13

So, lets do one last thing here, that BUILDARGS is kind of tedious, so lets replace it with a module written by RJBS that adds support for one-arg constructors. What this basically says is, if I get one-arg and it is a Str, assume it is the 'name' variable and pass it along accordingly.

19

At this point, we should have a fully functioning replacement class that conforms to the old API.





Friday, June 7, 13

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 \Rightarrow 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') }
```





Friday, June 7, 13

So, lets look at our first revision, and there are a couple things to note.

First, we moved the config to an attribute. We added the init_arg option, which tells Moose what to look for in constructor keys (beyond the default name). And then we added the predicate option, which tells Moose to make an predicate function that tells us if the config has been initialized.

```
package ConfigLoader;
use Moose;
use YAML qw[ LoadFile ];
has '_config' => (
    init_arg => '-config',
    is => 'rw',
    isa => 'HashRef',
    lazy => 1,
    default \Rightarrow 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',
           => 'rw',
    is
    isa => 'HashRef',
    lazy => 1,
    default \Rightarrow 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') }
```









Friday, June 7, 13

runs after the object is constructed





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





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





- 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





- 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'} ) )
}
```





Friday, June 7, 13

So, lets continue with our refactor here. The next step is to notice that the get, username and password methods can all be actually dealt with via the Native Traits features in Moose.





Friday, June 7, 13

So, lets review some of the handles information. Handles is used to delegate both to embedded objects, or using native traits can delegate to built in perl types. This feature is very useful for building well encapsulated APIs because it helps to hide the underlying implementation details via delegated methods. And with 'curried' delegations, it is possible to build fairly complex APIs in a very declarative way.

can delegate to embedded objects





Friday, June 7, 13

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





- 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





- 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 \Rightarrow 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 \Rightarrow 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 \Rightarrow sub \{+\},
     predicate => 'is_config_ready',
     handles => {
           'get' => 'get',
           'username' => [ 'get', 'username' ],
'password' => [ 'get', 'password' ]
);
```





Friday, June 7, 13

Next addition is the use of MooseX::Aliases, which allows us to actually add support for idiomatic constructor args for config_file and config attributes, while still keeping the dash prefixed old APIs intact.

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







Friday, June 7, 13

We have already seen MooseX::Aliases, but it can do more then just aliasing the init_arg, it can also alias methods properly (so that they work for subclasses) and more.

MooseX::OneArgNew we have also seen, it really just does this, not more. MooseX::UndefTolerant is something that will likely make its way into Moose core before long, it allows you to treat 'undef' as "lack of a value" rather then an undefined value. MX::A::TC::CustomizeFatal allows you to use the type constraint errors of Moose, but specify exactly how severe the error is (warning, error, default, etc). MX::E::C::A::F is actually a leftover from the original Catalyst port to Moose, but can be useful for converting things early on. And lastly MooseX::NonMoose, which is invaluable with extensions and which we will get into in more detail shortly.

MooseX::Aliases





Friday, June 7, 13

MooseX::Aliases

MooseX::OneArgNew





Friday, June 7, 13

MooseX::Aliases

MooseX::OneArgNew

MooseX::UndefTolerant





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





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





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





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





MooseX::NonMoose





Friday, June 7, 13

It is not always practical to Moosify; it might be a CPAN module, it might belong to another department (who can't use Moose), it might be just fine as it is and not need it. When it comes time to extend that code, you should first look into delegation, which we saw before and was covered in the class. But if delegation is not suitable, then subclassing is your answer.

MooseX::NonMoose

Sometimes you can't Moosify





Friday, June 7, 13

It is not always practical to Moosify; it might be a CPAN module, it might belong to another department (who can't use Moose), it might be just fine as it is and not need it. When it comes time to extend that code, you should first look into delegation, which we saw before and was covered in the class. But if delegation is not suitable, then subclassing is your answer.

MooseX::NonMoose

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





Friday, June 7, 13

It is not always practical to Moosify; it might be a CPAN module, it might belong to another department (who can't use Moose), it might be just fine as it is and not need it. When it comes time to extend that code, you should first look into delegation, which we saw before and was covered in the class. But if delegation is not suitable, then subclassing is your answer.

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' },
);
```





Friday, June 7, 13

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

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

extends 'CGI::Expand';

sub separator { ':' }
```





Friday, June 7, 13

This one came from a project I was working on a year or so ago, which has since been deprecated, but I thought this was interesting simply because CGI::Expand is one of those modules that has multiple APIS; it exports functions, it has a class method style, but no real object style. This subclass not only overrides the separator value, but it also still supports those two (very unMoose-like) APIs.

```
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' => (
         => 'ro',
    is
            => 'SAuth::Web::Consumer::Client',
    isa
    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

Sometimes you can't subclass

(Inside-Out classes, AUTOLOAD or some other weirdness)





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





- 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',
            => 'HTTP::Response',
    isa
    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 ) ]
);
```





Friday, June 7, 13

```
package Data::Riak::HTTP::Response;
use Moose;
use overload '""' => 'as_string', fallback => 1;
has 'http_response' => (
    is => 'ro',
            => 'HTTP::Response',
    isa
    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' => (
       => 'ro',
    is
          => 'ArrayRef[HTTP::Message]',
    isa
   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 ) ]
);
```





Friday, June 7, 13





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.

▶ Is Evil!





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.

- ▶ Is Evil!
- Is Expensive!





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.

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





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.

- ▶ 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' => (
           => 'bare',
    is
    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' => (
           => 'bare',
    is
    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]->{ $method };
}
package Employee;
use Moose;
has 'human' => (
           => 'bare',
    is
    isa => 'Person',
    required => 1,
    handles => qr/•*_name$/
);
# Can't locate object method "first_name" via
# package "Employee" at 002-test.t line 37.
```





Constructors not named new





Constructors not named new

Move all logic into Moose





Constructors not named new

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





Constructors not named new

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





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





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





Using method modifiers on Moose accessors





Using method modifiers on Moose accessors

Can be really useful for decorated accessors





Using method modifiers on Moose accessors

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





Using method modifiers on Moose accessors

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









Aliasing methods

The alias should always call the method properly



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



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



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





Questions?



