use Moose;

$self->focus_on_actual_problem_solving

Raf Gemmail
@fiqus
Wellington.pm
June 2014

% perldoc perltoot

```
PERLTOOT(1)          User Contributed Perl Documentation          PERLTOOT(1)



NAME
       perltoot - Tom's object-oriented tutorial for perl

DESCRIPTION
       Object-oriented programming is a big seller these days.  Some managers
       would rather have objects than sliced bread.  Why is that?  What's so
       special about an object?  Just what is an object anyway?

       An object is nothing but a way of tucking away complex behaviours into
       a neat little easy-to-use bundle.  (This is what professors call
       abstraction.) Smart people who have nothing to do but sit around for
       weeks on end figuring out really hard problems make these nifty objects
       that even regular people can use. (This is what professors call
       software reuse.)  Users (well, programmers) can play with this little
       bundle all they want, but they aren't to open it up and mess with the
       insides.  Just like an expensive piece of hardware, the contract says
       that you void the warranty if you muck with the cover.  So don't do
       that.
```

% perldoc perltoot

```
package Person;
use strict;

#####################################################
## the object constructor (simplistic version)  ##
#####################################################
sub new {
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless($self);           # but see below
    return $self;
}

################################################
## methods to access per-object data          ##
##                                            ##
## With args, they set the value.  Without    ##
## any, they only retrieve it/them.           ##
################################################

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

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return @{ $self->{PEERS} };
}
```

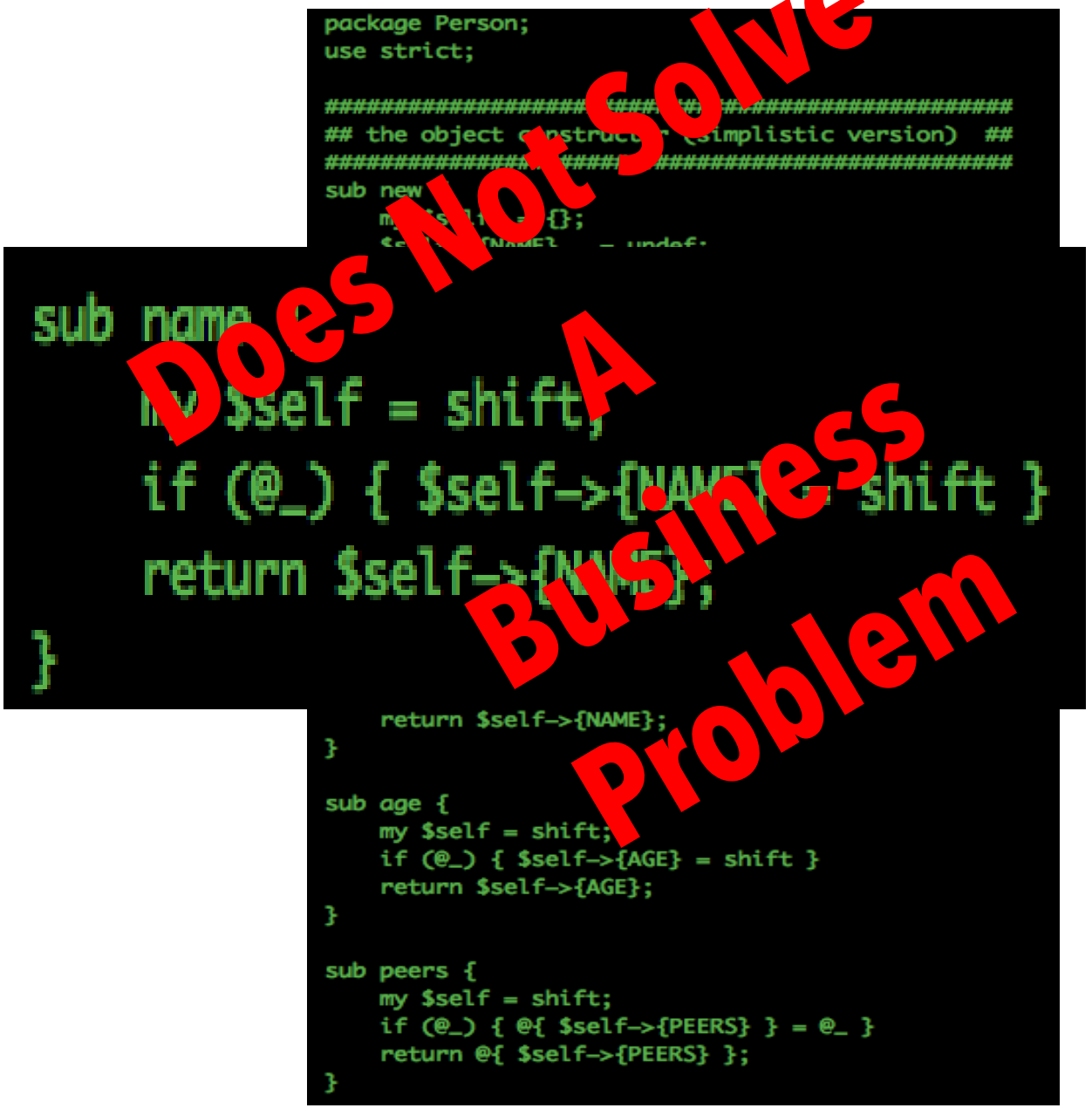
% perldoc perltoot

```
sub new {
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless($self);           # but see below
    return $self;
}
```

% perldoc perltoot

```
sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}
```

% perldoc perltoot



Class::Accessor

```
package Person;
use strict;
use base qw(Class::Accessor);
Person->follow_best_practice;
Person->mk_accessors(qw(name age peers));
1;
```

```
my $person = Person->new({ name=>12, age=>'Raf', peers=>['apple'] });
```

Class::Accessor

```
package Person;
use strict;
use base qw(Class::Accessor);
Person->follow_best_practice;
Person->mk_accessors(qw(name age peers));
1;
```

```
my $p = { name => 12, age => 'Raf', peers => ['apple'] };
```

Why Moose?

- Declarative
- Focus TDD(esign) and Single Responsibility
- Standard Paradigm (Mo[ou].* and Perl6)
- Reflective (MOPS)
- Extensible and extended (MooseX)
- Focus on returning business value / solution

Moose

```
package Person;
use Moose;
```

Moose

```
package Person;
use Moose;
```

- *strict* and *warnings* pragmas enabled

Moose

```
package Person;
use Moose;
```

- *strict* and *warnings* pragmas enabled
- Constructor : Person->new

Moose

```
package Person;
use Moose;
has 'name' => ( is => 'ro',       #read only
                isa => 'Str',     #string
                required => 1, ); #mandatory

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

1;
```

- *strict* and *warnings* pragmas enabled
- Constructor : Person->new

Moose

```
package Person;
use Moose;
has 'name' => ( is => 'ro',       #read only
                isa => 'Str',     #string
                required => 1, ); #mandatory

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

1;
```

- *strict* and *warnings* pragmas enabled
- Constructor : Person->new({name=>'Raf'})

Moose

```
package Person;
use Moose;
has 'name' => ( is => 'ro',       #read only
                isa => 'Str',     #string
                required => 1, ); #mandatory

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

1;
```

- *strict* and *warnings* pragmas enabled
- Constructor : Person->new({name=>'Raf'})
- $person->name() immutable
- $person->age(5)
- $age = $person->age()

Moose

```
package Person;
use Moose;
use MooseX::FollowPBP;

has 'name' => ( is => 'ro',       #read only
                isa => 'Str',     #string
                required => 1, ); #mandatory

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

1;
```

- *strict* and *warnings* pragmas enabled
- Constructor : Person->new
- get_name, get_age, set_age

Moose

```
package Person;
use Moose;
use MooseX::FollowPBP;

has 'name' => ( is => 'ro',       #read only
                isa => 'Str',     #string
                required => 1, ); #mandatory

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

1;
```

- *strict* and *warnings* pragmas enabled
- Constructor : Person->new
- get_name (immutable), get_age, set_age
- Type checking (isa)

Attributes

Attributes

- is (mutability)

Attributes

- is (mutability)
  - ro
  - rw

Attributes

- is (mutability)
- isa (type validation)

Attributes

- is (mutability)
- isa (type validation)
  - Class (or Role) name eg. Person

Attributes

- is (mutability)
- isa (type validation)
  - Class (or Role) name eg. Person
  - Object, Int, Str, Bool, Undef, Def,

Attributes

- is (mutability)
- isa (type validation)
  - Class (or Role) name eg. Person
  - Object, Int, Str, Bool, Undef, Def,
  - Ref (ScalarRef, ArrayRef, HashRef)

Attributes

- is (mutability)
- isa (type validation)
  - Class (or Role) name eg. Person
  - Object, Int, Str, Bool, Undef, Def,
  - Ref (ScalarRef, ArrayRef, HashRef)
  - ArrayRef[HashRef], ArrayRef[Person]

Attributes

- default
  - default => $scalar || $sub_ref

Attributes

- default
  - default => $scalar || $sub_ref

```
has 'name' => ( is => 'ro',       #read only
                isa => 'Str',     #string
                default => 'John Doe', );
```

Attributes

- builder, lazy
  - builder => method name

Attributes

- builder, lazy
  - builder => method name

```
has 'year_of_birth' => ( is => 'rw', isa => 'Int',
                         lazy    => 1,
                         builder => '_build_year_of_birth' );

# derive birth year from age
sub _build_year_of_birth {
  my $self = shift;

  DateTime->now()->subtract(
    years => $self->get_age )->year;
}
```

Attributes

- lazy_build => 1

```
has 'year_of_birth' => ( is => 'rw', isa => 'Int',
                         lazy_build      => 1, );

# derive birth year from age
sub _build_year_of_birth {
  my $self = shift;

  DateTime->now()->subtract(
    years => $self->get_age )->year;
}
```

Attributes

- handles
  - Proxies methods

Attributes

```
package TVShow;
use Moose;

has 'title' => (
    is       => 'ro',
    isa      => 'Str',
    required => 1,
);
has 'guide' => (
    is         => 'ro',
    isa        => 'WWW::EZTV::Show',
    lazy_build => 1,
    handles    => {
        get_episodes      => 'episodes',
        get_episode_count => 'has_episodes',
        find_episode      => 'find_episode'
    },
);

sub _build_guide {
    my $self = shift;
    my $title = shift;
    WWW::EZTV->new->find_show(
      sub { $_->name =~ /$title/i } );
}
no Moose;
1;
```

Attributes

```
package TVShow;
use Moose;

has 'title' => (
    is       => 'ro',
    isa      => 'Str',
    required => 1,
);
has 'guide' => (
    is          => 'ro',
    isa         => 'WWW::EZTV::Show',
    lazy_build => 1,
    handles     => {
        get_episodes      => 'episodes',
        get_episode_count => 'has_episodes',
        find_episode      => 'find_episode'
    },
);

sub _build_guide {
    my $self = shift;
    my $title = shift;
    WWW::EZTV->new->find_show(
      sub { $_->name =~ /$title/i } );
}
no Moose;
1;
```

Attributes

```
package TVShow;
use Moose;

has 'title' => (
    is       => 'ro',
    isa      => 'Str',
    required => 1,
);
has 'guide' => (
    is           => 'ro',
    isa          => 'WWW::EZTV::Show',
    lazy_build => 1,
    handles    => {
        get_episodes      => 'episodes',
        get_episode_count => 'has_episodes',
        find_episode      => 'find_episode'
    },
);

sub _build_guide {
    my $self = shift;
    my $title = shift;
    WWW::EZTV->new->find_show(
      sub { $_->name =~ /$title/i } );
}
no Moose;
1;
```

```
use Test::More;
use TVShow;
use WWW::EZTV;

my $got = TVShow->new({
  title => 'Game Of Thrones'
});

isa_ok( $got, 'TVShow' );

cmp_ok(
  $got->get_episodes->has_links->size,
  '>', 10,
  'Got > 10 episodes' );

done_testing;
```

Attributes

- reader, writer
  - reader => 'get_vm_config'
  - writer => 'persist_vm_config'

Attributes

- reader, writer
  - reader => 'get_vm_config'
  - writer => 'persist_vm_config'
- clearer => 'clear_attribute_name'

Attributes

- reader, writer
  - reader => 'get_vm_config'
  - writer => 'persist_vm_config'
- clearer => 'clear_attribute_name'
- predicate => 'is_verbose'

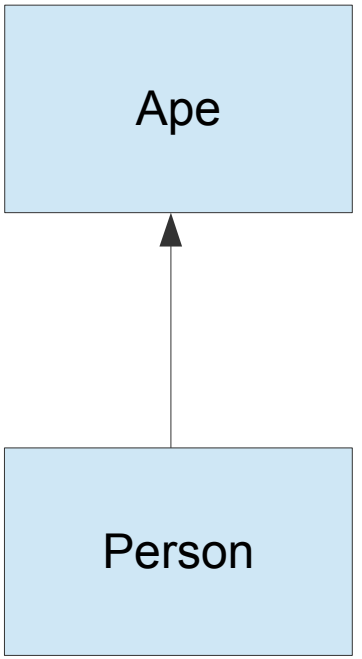
Attributes

- reader, writer
  - reader => 'get_vm_config'
  - writer => 'persist_vm_config'
- clearer => 'clear_attribute_name'
- predicate => 'is_verbose'
- trigger => \&sub_ref

Inheritance

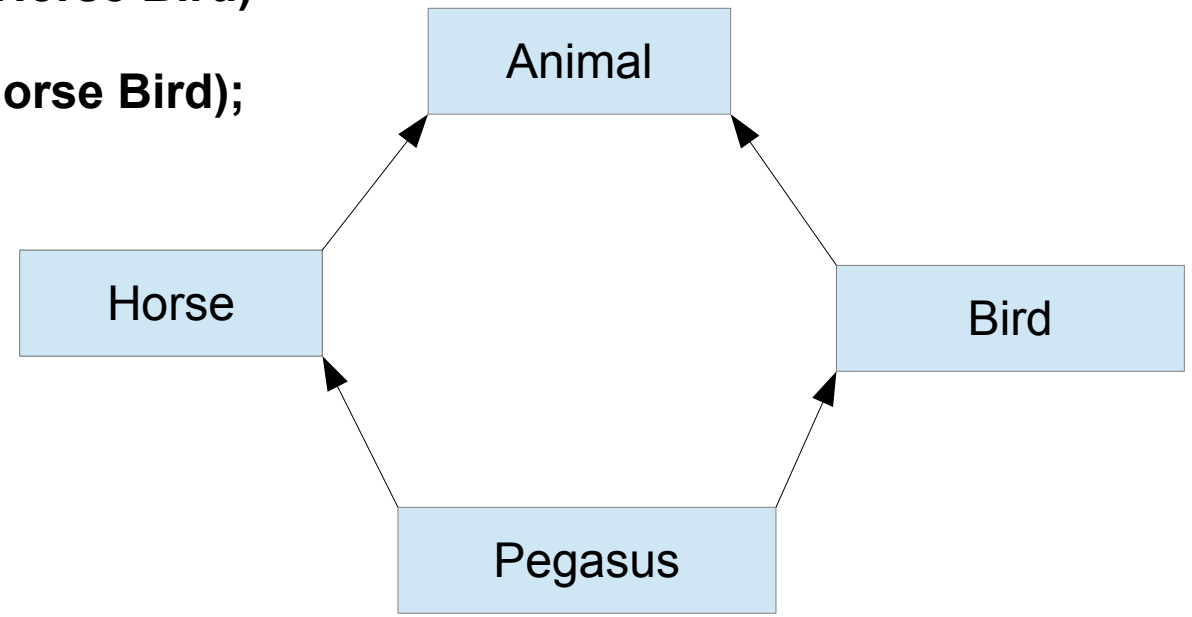
- extends

```
package Person;
use Moose;
use MooseX::FollowPBP;
use DateTime;
extends 'Ape';
```



Method Resolution Order

use base qw(Horse Bird)
vs
extends qw(Horse Bird);



- Perl depth-first: [Pegasus → Horse → Animal, Bird]
  - eg. Animal::brain has higher precedence than Bird::brain
- C3 MRO breadth first: [Pegasus, Horse, Bird, Animal]

Roles

- Cross cutting characteristics (attribs & methods)
- Like Interface and Abstract Classes
- Base class which defines a behavioural contract with an optional partial implementation

Roles

```
package Role::BreathLogger;
use Moose::Role;
use MooseX::FollowPBP;

# interface style contract
requires 'get_breath_count';

has 'logger' => (
  is      => 'ro',
  default => sub { Log::Any->new() },
);

sub log {
  my $self = shift;
  my $count = $self->get_breath_count;
  $self->get_logger->info("Breaths: $count");
}

no Moose;
1;
```

Role

```
use Moose::Role;
use MooseX::FollowPBP;
```

Roles

```
# interface style contract
requires 'get_breath_count';
```

Roles

```
t/01.t .. 'Role::BreathLogger' requires the method 'get_breath_count' to be implemented by 'Pe
sers/rafiq/perl5/lib/perl5/darwin-2level/Moose/Exception.pm line 37
        Moose::Exception::_build_trace('Moose::Exception::RequiredMethodsNotImplementedByClass
a19e05840)') called at reader Moose::Exception::trace (defined at /Users/rafiq/perl5/lib/perl5
vel/Moose/Exception.pm line 9) line 7
```

Roles

```
package Person;
use Moose;
use MooseX::FollowPBP;
use DateTime;
extends 'Ape';

with 'Role::BreathLogger';
```

Consume role after attribute declarations

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

Method Modifiers

- Aspects – non-core cross-cutting behaviour eg. logging
- Advice (do this) eg. A sub ref
- Point-cut (do the advice at these points of my execution) **Before**, **After** or **Around** some methods

Method Modifiers

```
before 'breath' => sub {
  my $self = shift;
  $self->get_logger->debug("Inhale");
};

after 'breath' => sub {
  my $self = shift;
  $self->get_logger->debug("Exhale");
};
```

```
with 'Role::BreathLogger';

sub breath {
  my $self = shift;
  $self->set_breath_count( $self->get_breath_count+1 );
}
```

Method Modifiers

```
before 'breath' => sub {
   my $self = shift;
   $self->get_logger->debug("Inhale");
};

after 'breath' => sub {
   my $self = shift;
   $self->get_logger->debug("Exhale");
};
```

```
# consume role
with 'Role::BreathLogger';

sub breath {
  my $self = shift;
  $self->get_logger->debug("Wheeze");
  $self->set_breath_count( $self->get_breath_count+1 );
}
```

Method Modifiers

```
before 'breath' => sub {
   my $self = shift;
   $self->get_logger->debug("Inhale");
};

after 'breath' => sub {
   my $self = shift;
   $self->get_logger->debug("Exhale");
};
```

```
# consume role
with 'Role::BreathLogger';

sub breath {
   my $self = shift;
   $self->get_logger->debug("Wheeze");
   $self->set_breath_count( $self->get_breath_count+1 );
}
```

Method Modifiers

```
$person->breath;
```

```
InhaleWheezeExhaleok 5 - Testing breathing
```

Method Modifiers

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

```
around 'think_of' => sub {
  my $target_method = shift;
  my $self = shift;
  my $args = shift;
  $self->get_logger->debug("Start thinking\n");
  my @thoughts = map{ $_ . " is great" } @$args;
  my $result = $self->$target_method(\@thoughts);
  $self->get_logger->debug("Done thinking\n");

  $result = 'censored' if $result eq 'poo';
  return $result;
};

sub think_of {
  my $self = shift;
  my $thoughts = shift;
  for (@$thoughts) {
    $self->get_logger->debug("Thinking of $_\n");
  }
  return "poo";
}
```

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

```
around 'think_of' => sub {
  my $target_method = shift;
  my $self = shift;
  my $args = shift;
  $self->get_logger->debug("Start thinking\n");
  my @thoughts = map{ $_ . " is great" } @$args;
  my $result = $self->$target_method(\@thoughts);
  $self->get_logger->debug("Done thinking\n");

  $result = 'censored' if $result eq 'poo';
  return $result;
};

sub think_of {
  my $self = shift;
  my $thoughts = shift;
  for (@$thoughts) {
    $self->get_logger->debug("Thinking of $_\n");
  }
  return "poo";
}
```

before

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

```
around 'think_of' => sub {
  my $target_method = shift;
  my $self = shift;
  my $args = shift;
  $self->get_logger->debug("Start thinking\n");
  my @thoughts = map{ $_ . " is great" } @$args;
  my $result = $self->$target_method(\@thoughts);
  $self->get_logger->debug("Done thinking\n");

  $result = 'censored' if $result eq 'poo';
  return $result;
};

sub think_of {
  my $self = shift;
  my $thoughts = shift;
  for (@$thoughts) {
    $self->get_logger->debug("Thinking of $_\n");
  }
  return "poo";
}
```

Transform Args

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

```
around 'think_of' => sub {
  my $target_method = shift;
  my $self = shift;
  my $args = shift;
  $self->get_logger->debug("Start thinking\n");
  my @thoughts = map{ $_ . " is great" } @$args;
  my $result = $self->$target_method(\@thoughts);
  $self->get_logger->debug("Done thinking\n");

  $result = 'censored' if $result eq 'poo';
  return $result;
};

sub think_of {
  my $self = shift;
  my $thoughts = shift;
  for (@$thoughts) {
    $self->get_logger->debug("Thinking of $_\n");
  }
  return "poo";
}
```

Real Method

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

```
around 'think_of' => sub {
  my $target_method = shift;
  my $self = shift;
  my $args = shift;
  $self->get_logger->debug("Start thinking\n");
  my @thoughts = map{ $_ . " is great" } @$args;
  my $result = $self->$target_method(\@thoughts);
  $self->get_logger->debug("Done thinking\n");

  $result = 'censored' if $result eq 'poo';
  return $result;
};

sub think_of {
  my $self = shift;
  my $thoughts = shift;
  for (@$thoughts) {
    $self->get_logger->debug("Thinking of $_\n");
  }
  return "poo";
}
```

After Invocation

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

```
around 'think_of' => sub {
  my $target_method = shift;
  my $self = shift;
  my $args = shift;
  $self->get_logger->debug("Start thinking\n");
  my @thoughts = map{ $_ . " is great" } @$args;
  my $result = $self->$target_method(\@thoughts);
  $self->get_logger->debug("Done thinking\n");

  $result = 'censored' if $result eq 'poo';
  return $result;
};

sub think_of {
  my $self = shift;
  my $thoughts = shift;
  for (@$thoughts) {
    $self->get_logger->debug("Thinking of $_\n");
  }
  return "poo";
}
```

```
Start thinking
Thinking of space is great
Thinking of cheese is great
Thinking of trek is great
Done thinking
ok 6 - Thoughts filtered
```

```
is( $person->think_of([qw(space cheese trek)]), 'censored', 'Thoughts filtered' );
```

Method Modifiers

- around [qw(method_1, method_2)]
- before qr/.*all_matching.*/
- HasLogger HasModel HasWebService

MooseX::Method::Signatures

- Type checking
- Method argument validation

```
use MooseX::Method::Signatures;

method mind_meld_with(Person $subject, Int $depth) {
  $self->think_of( $subject->get_thought );
}

has 'thought' => ( is => 'rw', isa => 'Str', default => 'doh' );
```

Moops

```
use Moops;
use Ape;

class HomoHabilis extends Ape {
  has 'grunt' => (
      is => 'rw',
      isa => 'Bool',
      default => 1,
      predicate => 'can_grunt',
  );

  # private
  has 'rage' => ( is => 'rw', isa => 'Bool', default=>1 );
  has 'xp' => ( is => 'rw', isa => 'Int', default=>100 );

  method adjust_xp( Int $value ) {
      $self->xp( $self->xp + $value );
  }

  method hit_noisy_ape( HomoHabilis $target  where {$_->can_grunt} ) {
      if ($self->rage) {
          $target->adjust_xp( -10 );
          $self->adjust_xp( +10 );
      }
  }
}
```

MooseX::Declare

```
use MooseX::Declare;
use HomoHabilis;

class HomoErectus extends HomoHabilis {
  has 'fire_skill' => (
  is => 'rw',
  isa => 'Bool',
        default => 1,
  predicate => 'can_burn_stuff',
  );

  method hit_smart_ape( HomoErectus $target  where {$_->can_burn_stuff} ) {
    if ($self->rage) {
      $target->adjust_xp( -10 );
      $self->adjust_xp( +10 );
    }
  }
}
```

And they all play together

```
use Test::More;
use HomoErectus;
use HomoHabilis;

my $erectus = HomoErectus->new;
my $habilis = HomoHabilis->new;

$erectus->hit_smart_ape( $erectus );
$habilis->hit_noisy_ape( $erectus );

is(110, $habilis->xp);
is(90, $erectus->xp);

done_testing;
```

Custom Types

```
package CustomTypes;

use MooseX::Types::Moose qw/Int/;
use MooseX::Types -declare => [ 'LifeEventDate' ];
use DateTime;

subtype LifeEventDate, as class_type 'DateTime';
coerce LifeEventDate, from Int, via { DateTime->from_epoch( epoch=> $_ ) };



1;
```

Custom Types

```
package CustomTypes;

use MooseX::Types::Moose qw/Int/;
use MooseX::Types -declare => [ 'LifeEventDate' ];
use DateTime;

subtype LifeEventDate, as class_type 'DateTime';
coerce LifeEventDate, from Int, via { DateTime->from_epoch( epoch=> $_ ) };



1;
```

Custom Types

```
package CustomTypes;

use MooseX::Types::Moose qw/Int/;
use MooseX::Types -declare => [ 'LifeEventDate' ];
use DateTime;

subtype LifeEventDate, as class_type 'DateTime';
coerce LifeEventDate, from Int, via { DateTime->from_epoch( epoch=> $_ ) };



1;
```

Custom Types

```
package Person;
use Moose;
use MooseX::FollowPBP;
use DateTime;
extends 'Ape';
use MooseX::Method::Signatures;
use CustomTypes ':all';

has 'engagement_anniversary' => (
    is => 'rw', isa => 'CustomTypes::LifeEventDate', coerce => 1 );
```

Custom Types

```
package Person;
use Moose;
use MooseX::FollowPBP;
use DateTime;
extends 'Ape';
use MooseX::Method::Signatures;
use CustomTypes ':all';

has 'engagement_anniversary' => (
    is => 'rw', isa => 'CustomTypes::LifeEventDate', coerce => 1 );
```

Custom Types

```
$person->set_engagement_anniversary( 1407598440 );
is( $person->get_engagement_anniversary, '2014-08-09T15:34:00',
    'Engage!');
```

Avoid Starup Costs with Moo

- Pure Perl

Avoid Starup Costs with Moo

- Pure Perl
- Painless integration with Moose deps (auto upgrades and does meta-magic)

Avoid Starup Costs with Moo

- Pure Perl
- Painless integration with Moose deps (auto upgrades and does meta-magic)
- Fast

MooseX

- Lots of Extensions
- MooseX::Types
- MooseX::GetOpt
- MooseX::App
- MooseX::Aspect
- MooseX::Daemonize

MooseX::App::Simple

```
package CalcApp;
use v5.10;
use MooseX::App::Simple qw(Color);

parameter 'num1' => (
    is            => 'rw',
    isa           => 'Int',
    documentation => q[First number],
    required      => 1,
);

parameter 'num2' => (
    is            => 'rw',
    isa           => 'Int',
    documentation => q[Second number],
    required      => 1,
);

parameter 'operator' => (
    is            => 'rw',
    isa           => 'Str',
    documentation => q[operator],
    required      => 1,
);

option 'verbose' => (
    is => 'ro',
    isa => 'Bool',
    documentation => 'spew noise'
);
```

MooseX::App::Simple

```perl
sub run {
  my $self = shift;
  my $statement =
     sprintf("%d %s %d", $self->num1, $self->operator, $self->num2);
  say $statement if $self->verbose;
  say eval $statement;
}
```

```perl
#!/usr/bin/env perl
use CalcApp;
CalcApp->new_with_options->run;
```

```
usage:
    calc.pl [long options...]
    calc.pl --help

parameters:
    num1      First number [Required; Integer]
    num2      Second number [Required; Integer]
    operator  operator [Required]

options:
    --help -h --usage -?  Prints this usage information. [Flag]
    --verbose             spew noise [Flag]
```

Meta Magic

- Reflection
- __PACKAGE__->meta
  - > get_all_subclasses
  - > get_all_attributes
  - > add_attribute
  - > magix

Recommendations

- Use MooseX::FollowPBP
- Test::Routine – test with fixture roles and various permutations of test data.
- __PACKAGE__->meta->make_immutable;

```
no Moose;
```