Object-Oriented Programming: Why You're Doing It Wrong

Toby Inkster

Three weird tricks to make your object-oriented code more encapsulated, more reusable, and more maintainable.

Toby Inkster (TOBYINK)

- Type::Tiny
- MooX::late
- Moops / Kavorka
- Test::Modern
- Pry
- Object::Util
- PerlX::Maybe
- Syntax::Collector

Object-Oriented Programming

- Examples in this presentation use Moo.
- Moo is a lightweight version of Moose.
  - Most of these examples can be rewritten to use Moose with only minor changes.
- Moo is still Perl
  - You could implement any of this with just core Perl OO if you were so inclined.

Stop creating mutable objects

http://www.diylol.com/

Stop creating mutable objects

- Perl Best Practices recommends creating methods called `get_foo` and `set_foo`.

```
my $obj = Pony->new(name => 'Pinkie Pie');

$obj->set_name('Twilight Sparkle');

say $obj->get_name();
```

Stop creating mutable objects

- Perl Best Practices recommends creating methods called `get_foo` and `set_foo`.
- Moose standard practice is to have a single accessor called `foo` that allows you to either get or set the attribute value.

```
my $obj = Pony->new(name => 'Pinkie Pie');

$obj->name('Twilight Sparkle');

say $obj->name();
```

Stop creating mutable objects

- Perl Best Practices recommends creating methods called `get_foo` and `set_foo`.
- Moose standard practice is to have a single accessor called `foo` that allows you to either get or set the attribute value.
- **These are both wrong.**

Stop creating mutable objects

```
my $alice = Person->new(
   name      => 'Alice',
   best_pony => Pony->new(name => 'Twilight Sparkle'),
);

my $bob   = Person->new(
   name      => 'Bob',
   best_pony => $alice->best_pony(),  # It's what brought us together
);

$alice->best_pony->set_name('Sunset Shimmer');

say $bob->best_pony->get_name();      # Spooky action at a distance
```

Stop creating mutable objects

```
my $conference = Event->new( start => DateTime->new(...) );
my $keynote    = Event->new( start => $conference->start );

# We need the keynote to be at the end of the conference
$keynote->start->add(seconds => 5*60*60);

# D'oh!
print $conference->start, "\n";
```

Stop creating mutable objects

- Make your accessors read-only.
- Don't allow an object's attribute values to be changed after it's been constructed.
- Save yourself from spooky action at a distance.

Stop creating mutable objects

- Moose and Moo:

```
is  => 'ro'
```

- Plain old Perl:

```
sub foo { $_[0]{foo} }
```

Stop creating mutable objects

- ~~Make your accessors read-only.~~
- ~~Don't allow an object's attribute values to be changed after it's been constructed.~~
- ~~Save yourself from spooky action at a distance.~~

- Sometimes you really need to model a changing world.

Stop creating mutable objects

```
my $alice = Person->new(
   name      => 'Alice',
   best_pony => Pony->new(name => 'Twilight Sparkle'),
);

my $bob   = Person->new(
   name      => 'Bob',
   best_pony => $alice->best_pony(),
);

$alice->best_pony->set_name('Princess Twilight Sparkle');    # SPOILER ALERT!

say $bob->best_pony->get_name();
```

Stop creating mutable objects

```
package Pony {
   use Moo;
   has name => (
      is      => 'ro',
      writer  => 'rename',
   );
}

# Better than name() or set_name() because it's clear that this is
# a method. It's a verb. It 'does something'.
$pony->rename('Princess Twilight Sparkle');
```

Stop creating mutable objects

- If you really need to model a changing world:
  - Make attributes mutable:
    - Only after careful consideration, not by default!
    - Not if they are part of the object's intrinsic identity.
  - Consider naming the writer method something that doesn't sound like an attribute.

Stop writing 'private' methods

We can actually see you.

https://www.flickr.com/photos/a_gods_child/4553482717/

Stop writing 'private' methods

- Methods named with a leading underscore are not really private.
- Subclasses can call them.
- Subclasses can override them.
- Even accidentally!

Stop writing 'private' methods

```
package Employee {
   use Moo;
   has name => (is => 'ro');
   sub _type { 'employee' }
   sub output { shift; say @_ }
   sub introduce_myself {
      my $self = shift;
      $self->output(
         'My name is ', $self->name, ' and I am an ', $self->_type,
      );
   }
}
```

Stop writing 'private' methods

```
package Typist {
   use Moo;
   extends 'Employee';
   ...;
}

my $obj = Typist->new(name => 'Moneypenny');
$obj->introduce_myself();
```

Can't call method "press_button" on an undefined value at Typist.pm line 16

Stop writing 'private' methods

```
package Typist {
   use Moo;
   extends 'Employee';
   has default_keyboard => (is => 'lazy', builder => sub { Keyboard->new });
   sub output {
      my $self = shift;
      my $text = join '', @_;
      $self->_type($self->default_keyboard, $text);
   }
   sub _type {
      my $self = shift;
      my ($kb, $text) = @_;
      for (my $i = 0; $i < length $text; $i++) {
         $kb->press_button( substr($text, $i, 1) );
      }
      $kb->press_button('Enter');
   }
}
```

Stop writing 'private' methods

- How can we fix this?

Stop writing 'private' methods

```
package Employee {
   use Moo;
   has name => (is => 'ro');
   sub _type { 'employee' }
   sub output { shift; say @_ }
   sub introduce_myself {
      my $self = shift;
      $self->output(
         'My name is ', $self->name, ' and I am an ', $self->_type,
      );
   }
}
```

Stop writing 'private' methods

```
package Employee {
   use Moo;
   has name => (is => 'ro');
   my $_type = sub { 'employee' };
   sub output { shift; say @_ }
   sub introduce_myself {
      my $self = shift;
      $self->output(
         'My name is ', $self->name, ' and I am an ', $self->$_type,
      );
   }
}
```

←a lexical method is just a coderef

Stop writing 'private' methods

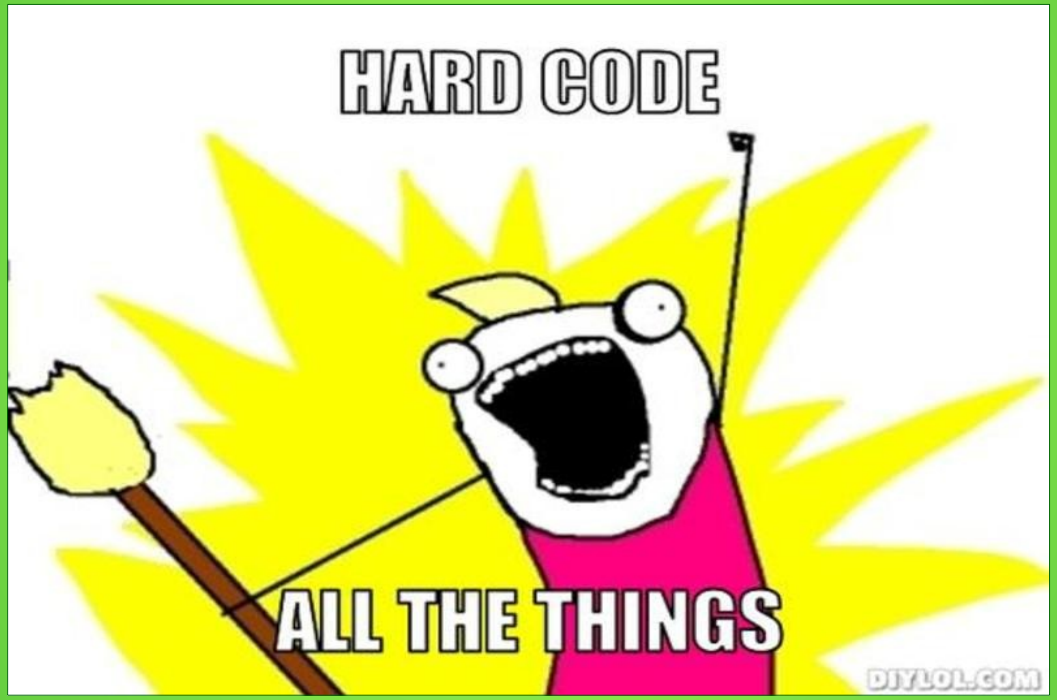
```
package Employee {
   use Moo;
   has name => (is => 'ro');
   sub type { 'employee' }
   sub output { shift; say @_ }
   sub introduce_myself {
      my $self = shift;
      $self->output(
         'My name is ', $self->name, ' and I am an ', $self->type,
      );
   }
}
```

←a public, documented method

Stop writing 'private' methods

- If a method is useful for end-users, then promote it to a public method.
- If a method exists in your namespace, then document it.
- Otherwise, use 'lexical methods' – coderefs.
- For lexical accessors, see `Lexical::Accessor`.

Stop hard-coding stuff



Not a great idea.

http://www.diylol.com/

Stop hard-coding stuff

```
package MyAuth;
use Moo;

sub fetch_user_list {
   my $self = shift;
   my $ua   = LWP::UserAgent->new();
   return $ua->get(
      "http://example.com/users.txt",
   );
}
```

Stop hard-coding stuff

```
package MyAuth;
use Moo;

sub fetch_user_list {
   my $self = shift;
   my $ua   = LWP::UserAgent->new();
   return $ua->get(
      "http://example.com/users.txt",
   );
}
```

- URL
- User-agent

Stop hard-coding stuff

```
package MyAuth;
use Moo;

sub fetch_user_list {
   my $self = shift;
   my $ua   = LWP::UserAgent->new();
   return $ua->get(
      "http://example.com/users.txt",
   );
}
```

```
package MyAuth::Testing;
use Moo;
extends 'MyAuth';

sub fetch_user_list {
   my $self = shift;
   my $ua   = LWP::UserAgent::WithLogging->new();
   return $ua->get(
      "http://example.com/users.txt",
   );
}
```

Stop hard-coding stuff

```
package MyAuth;
use Moo;

sub fetch_user_list {
   my $self = shift;
   my $ua   = LWP::UserAgent->new();
   return $ua->get(
      "http://example.com/users.txt",
   );
}
```

```
package MyAuth;
use Moo;

has user_agent => (
   is      => 'lazy',
   builder => sub { LWP::UserAgent->new() },
);

has user_list_url => (
   is      => 'lazy',
   builder => sub { "http://example.com/users.txt" },
);

sub fetch_user_list {
   my $self = shift;
   $self->user_agent->get($self->user_list_url);
}
```

Stop hard-coding stuff

```
package MyAuth::Testing;
use Moo;
extends 'MyAuth';

sub _build_user_agent {
   LWP::UserAgent::WithLogging->new();
}
```

```
package MyAuth::Pony;
use Moo;
extends 'MyAuth';

sub _build_user_list_url {
   'http://example.com/everypony.txt';
}
```

Look, it's really easy to subclass now!

Stop hard-coding stuff

```
package MyAuth::Pony::Testing;
use Moo;
extends 'MyAuth';

sub _build_user_agent {
   LWP::UserAgent::WithLogging->new();
}

sub _build_user_list_url {
   'http://example.com/everypony.txt';
}
```

Stop hard-coding stuff

- Better for testing
- Better for extensibility

Stop hard-coding stuff

- Things that you might be hard-coding without realising:
  - File paths
    - Including the path to your config file
  - Object instances
  - Class names
    - `$class->new()` is better than `Class->new()`

Why you were doing it wrong

- You created mutable objects
- You wrote 'private' methods
- You hard-coded stuff

How to do it right

- Create immutable objects
  - `is => 'ro'`
- Avoid undocumented methods
  - If they seem useful enough, document them
  - Otherwise, make them coderefs so they stay private
- Stop hard-coding stuff
  - `is => 'lazy'`
  - `builder => sub { ... }`

That's all folks!