Native Traits and Delegation

Delegation: the design pattern

Friday, June 7, 13

Delegation is a design pattern. Design patterns help you structure your program so that it remains flexible and maintainable.

```
package Account;
use Moose;

has expiration => (
    is  => 'ro',
    isa => 'DateTime',
);
```


Friday, June 7, 13

Say we have an attribute expiration that stores a DateTime object. Quite straightforward.

```
package Email::Subsystem;

say "Your account expires on " .
$account->expiration->ymd;

if ($account->expiration > DateTime->now) {
    say "Pay more money!";
}
```


Friday, June 7, 13

Then we consult an account's expiration date by calling methods on that DateTime object and comparing it to other DateTime objects. The places where we consult that DateTime object are highlighted in green.

```
package Email::Subsystem;

say "Your account expires on " .
$account->expiration_date;

if ($account->is_expired) {
    say "Pay more money!";
}
```


Friday, June 7, 13

The Delegation design pattern suggests that we hide that internal object by exposing methods in the Account class that call methods on the internal DateTime object. Now the Email::Subsystem doesn't receive a DateTime object. This is useful in a number of ways that will make up the bulk of this talk.

Delegation: external methods for internal objects

Friday, June 7, 13

So to sum up delegation succinctly, it's creating external methods for internal objects. That's the pattern you follow, instead of exposing internal objects directly.

~~$account->expiration->ymd~~

$account->expiration_date

Friday, June 7, 13

Using the delegation pattern means that you enable users call this simpler expiration_date method. expiration_date does the exact same work of looking up the DateTime object stored in the attribute and calling ymd on it. But that bit of detail is hidden in the simpler expiration_date method. So the API that your user consumes is simpler: call this method, get the date as a string out. Instead of: call this method, get a complete DateTime object out.

~~$account->expiration->ymd~~

$account->expiration_date

Friday, June 7, 13

It's generally less typing too, but in this case it's the same number of characters because DateTime's API is pretty concise.

```
$account->expiration
```

DateTime

Friday, June 7, 13

But here's the real kicker. Without delegation, you're exposing your internal DateTime object to your consumers. That's really an implementation detail.

```
$account->expiration
```

Date::Tiny

Friday, June 7, 13

What happens if down the road you decide you want to swap out the internal representation of an expiration date from DateTime to Date::Tiny? Maybe as an optimization. Well, you can't, really, because all of your consumers are expecting a DateTime object and might be using APIs that Date::Tiny can't support. So you're stuck with DateTime.

```
$account->expiration
```

1342566000

Friday, June 7, 13

Maybe you want to store the expiration as an integer of epoch seconds to trim down on your memory usage. Again, this would totally break the expectations of your downstream users so you can't

```
$ perldoc DateTime | wc -l
```


Friday, June 7, 13

Anyone want to hazard a guess as to how many lines of documentation DateTime has?

```
$ perldoc DateTime | wc -l
2074
```


Friday, June 7, 13

Answer: a lot. That's an overwhelming amount of documentation.

```
$ perldoc DateTime | perl -ple '$_ = length' | sort -n | tail
106
107
115
117
122
129
134
144
164
266
```


Friday, June 7, 13

And what's more, some of those lines can get pretty long.

DateTime

- floating timezones
- durations
- leap seconds
- nanoseconds
- infinite datetimes
- `year_with_secular_era`
- `day_of_quarter`

Friday, June 7, 13

DateTime's API supports a LOT. That's why we all use it. But it's overwhelming when you have to skip past all this stuff when you just want to know what year-month-day this DateTime represents.
Does your expiration date really need to support any of this? If not, why are you putting that in your users' laps?

expiration_date

- floating timezones
- durations
- leap seconds
- nanoseconds
- infinite datetimes
- year_with_secular_era
- day_of_quarter

Friday, June 7, 13

Your simple expiration_date method doesn't expose any of this so your users will be able to move on with their job that much quicker. And you won't have to worry about supporting any of this extraneous behavior. You can choose which bits of DateTime you want to support by exposing methods that interact with just those bits.
If you just expose a DateTime object, users can't really be blamed for changing its timezone, because that's something DateTime supports.

`expiration_date` is “real”

- `expiration->ymd` is ethereal
- `expiration_date` can be documented
- You can tweak `expiration_date`
- Better greppability, tab completion

Friday, June 7, 13

expiration_date is a real method. It’s a name and humans are really good with names. expiration->ymd is kind of artificial. It’s a method call.
expiration_date can have domain-specific documentation in the Account module.
You can tweak the behavior of expiration_date if you need to. For example maybe you need special behavior because of leap years, adding or subtracting a day.
You can’t tweak DateTime’s ymd method unless you monkeypatch or subclass and then you’re in big trouble.
You can easily grep for, and tab complete, expiration_date.

Implementing delegation

Friday, June 7, 13

So far we've seen how delegation is a useful design pattern, but let's look at what it takes to implement delegation.

handroll

Friday, June 7, 13

First let's look at how to implement delegation yourself.

```
package Account;
use Moose;

has expiration => (
    is  => 'ro',
    isa => 'DateTime',
);
```


Friday, June 7, 13

As a reminder, here's our starting point. We have an Account class with an expiration attribute that stores DateTime objects. We want an expiration_date method that calls ymd on the expiration DateTime object and return it.

```
package Account;
use Moose;

has expiration => (
    is  => 'ro',
    isa => 'DateTime',
);

sub expiration_date {
    my $self = shift;
    return $self->expiration->ymd;
}
```


Friday, June 7, 13

Well that's about as simple as a method can get! Delegation is not at all difficult to implement.

handles

Friday, June 7, 13

It turns out that Moose offers tools for quickly implementing delegation. The “has” attribute creator has an option called “handles” for setting up delegation.

```
package Account;
use Moose;

has expiration => (
    is      => 'ro',
    isa     => 'DateTime',
    handles => {
        expiration_date => ‘ymd’,
    },
);
```


Friday, June 7, 13

Here’s what handles looks like. This declaration tells Moose to create an expiration_date method that calls ymd on the DateTime object in the expiration attribute, pretty similar to what we implemented by hand. You can of course put more methods in this hash to declare an entire façade in one fell swoop.

handles

- error checking
- less code (= fewer bugs!)
- maintainers know: `handles` == delegation
- declarations and conventions more maintainable than *DIY*

Friday, June 7, 13

There's a couple things handles offers you over implementing delegation by hand.
There's additional error checking which your handrolled implementation did not have.
There's less code that you have to write by hand. And there's plenty of research suggesting that number of defects increases with lines of code.
The people maintaining your code, yourself included, know exactly that handles means delegation, with all the secondary implications that delegation implies. They may not make that same inductive leap with the handrolled delegation. This happens to be where a lot of Moose's power comes from: declarations and conventions are way more maintainable (and introspectable) than Do-It-Yourself.

handles

- Cannot delegate expiration_date to ymd because the value of expiration is not defined
- Cannot delegate expiration_date to ymd because the value of expiration is not an object (got 'HASH(0x7fe15c029898)')

Friday, June 7, 13

As I mentioned, handles has some error checking in addition to what your handrolled implementation offered. Moose throws the first error when you try to call expiration_date when the expiration attribute is unset.
The latter error happens when expiration doesn't have an object in it. This is a very rare error in practice because you're almost always going to have a type constraint to protect you from this particular error scenario.

handles

- passes all parameters along
  - and you can curry in some parameters
- returns the return value
- does not impose scalar/list/void context
- does not check object's class (`isa`)

The method that Moose compiles for you follows this set of rules. Whatever parameters you pass into the external generated method will be passed along to the internal object's method. And as you'll see a bit later that Moose lets you set up some additional parameters that'll always be passed in. If you want anything more complicated than that you can still handroll your own delegation. Moose does not look at the return value of the internal object's method, it just returns it straight up to your caller. It doesn't impose context either, your caller does. Finally Moose does not consult the object's class in your delegation, just that what's in there is an object. Use a type constraint to get this checking in the right place (at assignment time).

handles => …

Friday, June 7, 13

handles is actually pretty complicated, syntactically. It's easily the most complex option to "has". There are a lot of ways you can specify what methods you want to delegate. You can pass a lot of different data types into handles, so let's quickly look at each one in turn.

handles => { … }

- key: method you want Moose to create
- value: method you want to call
- handles => { external => ‘internal’ }

Friday, June 7, 13

We already saw passing a hash reference to handles. The hash reference maps the external methods that you want Moose to create to the methods you want to call on the object in your attribute.

handles => […]

- shortcut when external names == internal names
- each item is both the external method and the method you want it to call
- useful, but don’t be lazy
  - $account->ymd??
- handles => [‘begin’, ‘rollback’]

Passing an array reference to handles is a handy shortcut for when the methods you want to generate in your class have the same names as the methods they’re calling on your internal object. It’s a shortcut for when it makes sense, not a shortcut to let you be lazy. What the heck would $account->ymd mean? Expiration time, or signup time, or last login, or what. You still want to fit your class’s domain-specific naming convention, jargon, etc.

handles => '…'

- **NOT** a shortcut for delegating one method
- specify a **ROLE** name
- delegate the role's methods and requirements

- external names == internal names
- `handles => 'Backend::Transactional'`

Friday, June 7, 13

You can also pass a string to handles. You might guess that this is a shortcut for delegating a single method, but that's not what it does. Moose interprets the string as a role name. Then it sets up delegation for each of the method names that the role implements, but also for each of the method names that the role requires.
We'll talk about this pattern and how it's useful later.

handles => qr/…/

- matches the methods you want to delegate
- requires a class name in `isa`
- be **VERY** careful with this one

- external names == internal names
- `handles => qr/^process_/`

Friday, June 7, 13

Passing a qr regular expression to handles lets you specify which methods you want to delegate. This pattern requires you to have a class name in isa so that Moose knows which method list to run the regex over. The other ones don't.
Be very careful with this one because you could easily shoot yourself in the foot.
You might notice that external names == internal names in most of these patterns. If you need to rename methods then the only way to do that is to use a hashref to tell Moose how to name your generated methods.

handles => qr/.*/

- delegate **EVERY** method in the internal class
- possibly useful for wrapping another class
  - poor man's inheritance
    - or rich man's inheritance
- dangerous. avoid!

Friday, June 7, 13

Of course since you can specify a regular expression, you can specify dot-star as your regular expression which will match everything. This can sort of act like inheritance since you can now respond to all method calls that the other class can.
But you're combining namespaces like this is tight coupling and a maintenance nightmare. What happens when the other class adds another method which now matches one of your method names?
While it's a nice spectacle, I strongly discourage you from using this. Just specify the methods you want to delegate.

handles => Duck

- handles => duck_type([‘begin’, ‘rollback’])
- esoteric
- external names == internal names

Friday, June 7, 13

Moose has this concept of a duck-type type constraint. A duck type constraint has a list of method names. Any object that has all those methods will pass the type constraint.
Since a duck type has a list of methods we can use it to set up delegation.
You probably want to use roles for this instead, but duck typing is good for dealing with other people’s maybe-non-Moose code.

`handles => sub {…}`

- esoteric
- requires a class name in `isa`
- the `sub` gets the metaclass of what's in `isa`
- return a hash (like `handles => { ... }`)
- delegate based on arbitrary criteria

Friday, June 7, 13

You can also pass a subroutine to handles. This is so esoteric I've not only never used it, but I also never knew what it did. I had to research it for this talk. You'll probably never need it but in the interests of completeness I'll describe it quickly. The sub gets the metaclass of whatever class name is in isa, which is required. Then it returns a hash (not a reference) which acts just like the handles => hashref case: each key is a method you want Moose to generate and each corresponding value is the method that will be called on your internal object. This allows you to delegate based on arbitrary criteria. This is sort of like the regular expression case, except the regex is limited to delegating based on the method name. Sub can delegate based on anything, such as methods having a particular trait, or being part of the specific local class instead of being inherited from a superclass.

Currying

- pre-set parameters the method will always get
- they come before parameters that user passes
- handy for generating convenience methods
- `@_ = ($object, @curried, @specified)`

Friday, June 7, 13

Currying is a misused computer science term that means fixing some parameters to a function call.
The parameter list that the internal method will get always starts with the object. Then come the curried parameters, which is usually going to be empty. Then finally come the parameters that your caller specified directly.

Currying

```
has expiration => (
    is => 'ro',
    isa => 'DateTime',
    handles => {
        expiration_date => [
            'ymd',
            '/',
        ],
    },
);
```


Friday, June 7, 13

Here's what currying looks like. Everything is the same except instead of specifying the internal method we want to call as a string, we pass in an array reference. The first value is the method name, but after that come all the parameters. Here we're saying pass in the string / as a parameter.

Currying

```
sub expiration_date {
    my $self = shift;
    $self->expiration->ymd('/', @_);
}
```


Friday, June 7, 13

Here's what the method Moose compiles looks like. The ymd method in DateTime gets the string slash as its first parameter, then whatever else the caller passed to expiration_date.
The first parameter of ymd specifies what separator to use between the year month and day components, so what this currying does is override that default seperator of hyphen with a slash.

Currying

```
$self->expiration_date();
    # $self->expiration->ymd('/')
-> 2012/07/08

$self->expiration_date(':')
    # $self->expiration->ymd('/', ':')
-> 2012/07/08
```


Friday, June 7, 13

Here's what some method calls with currying look like. If we pass no arguments to expiration_date, then we pass just the curried slash to ymd.
If we do try to pass an argument, it will be added to the parameter list when calling ymd, but ymd will ignore it because it only looks at its first parameter.

Moose delegation patterns

Friday, June 7, 13

Now let's get into some interesting delegation patterns we can set up by using a confluence of Moose features.

Simple delegation

Friday, June 7, 13

We covered simple delegation already, but let's review quickly.

Simple delegation

- Your attribute stores an object of a given class
- Make façade methods for that class's methods
- Provide convenience methods
- Guide your user's interactions

Friday, June 7, 13

In the simplest delegation pattern, you have an attribute which stores objects of a particular class. So you set up methods in your class that calls methods on that object.
You can enhance a standard attribute by generating convenience methods that your users will be calling a lot.
This guides your user's interactions with your attribute, because its methods have domain-specific names which you've documented, etc.

Delegation *only*!

Friday, June 7, 13

To take this pattern a little bit further you can make it so the only way users can interact with your attribute is through delegation.

Delegation *only*

- Keep your internal object hidden
- Don’t provide an accessor
- `handles` methods only
- `init_arg => undef` (overkill)

The basic idea is that you keep your internal object hidden.
You do this by not providing a general read or read/write accessor that returns the object. Or if you do, it’s private to your class.
The only interactions that the outside world can have with your internal object is by calling your methods, which means the ball is in your court.
If you want to go all out with this, you could say specify init_arg => undef which forbids the outside world from passing in an object through the constructor. This way you have complete control.

```
has expiration => (
    isa     => ‘DateTime’,
    handles => {
        expiration_date => ‘ymd’,
    },
);
```


Friday, June 7, 13

Here’s what the delegation-only pattern looks like. Notice that you haven’t specified an option for “is” which means that Moose will not create a method named “expiration” that would return the DateTime object.
Now, I’ve been writing Moose code for like six years. When I see this code I have a very specific reaction. My brain has a Moose code linter and here’s what it reports:

```
has expiration => (
    is       => ‘ro’,
    isa      => ‘DateTime’,
    handles => {
        expiration_date => ‘ymd’,
    },
);
```


Friday, June 7, 13

“Oh! The author of this code forgot to declare is => ‘ro’. Let me add that in for them.”
Well now I’ve gone and removed all the benefits of the delegate-only pattern. The original author didn’t want an accessor because exposing that DateTime object leads to too many problems.

```
has expiration => (
    # Hey! No accessor on purpose!
    # is => 'ro',

    isa     => 'DateTime',
    handles => {
        expiration_date => 'ymd',
    },
);
```


Friday, June 7, 13

One way to avoid this problem is by leaving a comment telling your maintainers that you indeed really meant to have no accessor.

```
has expiration => (
    is       => 'bare',
    isa      => 'DateTime',
    handles => {
        expiration_date => 'ymd',
    },
);
```


Friday, June 7, 13

The other way to avoid this problem is to use a special value for the “is” option called “bare”. This declares your intent to Moose, and to your maintainers, that you explicitly want no standard accessor for this attribute.
“is => ‘bare’” was added as a special-case to suppress the “you have no accessors for your attribute” warning that you would see if you left off the handles too, to catch the common case of forgetting an “is” declaration without forcing experienced developers to silence warnings in uncouth ways.
But in this case we wouldn’t be getting a warning anyway, since we have the methods generated by handles. But “bare” is there, and it’s useful for this too.

Delegation *only*

- Declare exactly the API you support
- Freedom to refactor internals
- Fewer bugs
- Looser coupling

Friday, June 7, 13

Delegation-only attributes give you the benefits of delegation that we opened this talk with. You can declare exactly the API you want to support and no more. This means users get to interact with your class on your terms, as it should be. Users don't see the internal object. They only see the API you're providing. So as long as you can maintain that API, you can refactor your internals as much as you want. Replacing DateTime with a simple string becomes possible. Fewer bugs because your users aren't using APIs from that class that you don't directly control and might not even account for. For example, what happens if your consumer calls set_year on your DateTime object? This also avoids tight coupling because you can swap out the implementation for something lighter, or better, as long as the methods you expose fulfill the API.

Role-based delegation

Friday, June 7, 13

The next pattern I'd like to discuss is role-based delegation. The basic idea is that you create a role which declares the API you want. Then you delegate that API, and you can restrict your attribute's values to only objects which have the role, and thus the API you want.

```
package Backend::Transactional;
use Moose::Role;

requires ‘begin’, ‘commit’, ‘rollback’;

sub txn_do {
    my ($self, $callback) = @_;
    $self->begin;
    $callback->();
    if (...error...) {
        $self->rollback;
    } else {
        $self->commit;
    }
}
```


Friday, June 7, 13

Here we have a Backend::Transactional role that requires three methods: begin, commit, rollback. It also implements a txn_do using those three methods. The idea of txn_do, which is implemented by a lot of systems like DBIx::Class and KiokuDB, is that it wraps your code in a transaction. If all goes well it will commit for you. If something goes wrong it will rollback for you.

```
package Database;
use Moose;

has backend => (
    is       => ‘bare’,
    required => 1,
    does     => ‘Backend::Transactional’,
    handles  => ‘Backend::Transactional’,
);
```


Friday, June 7, 13

Now we have a class Database that relies on Backend::Transactional. It has a backend attribute which, thanks to does, must be an object that does the Backend::Transactional role. Since it does the role, we know that the object has the methods begin, commit, and rollback, since those were requirements of the method. We also know that the object has the txn_do method since the role implements it.
And then we delegate those four methods, simply and easily.
Notice we also declare “is => ‘bare’” because we don’t want to expose the backend implementation directly.

```
$db->begin;
$db->rollback;

$db->txn_do(sub {
    ...
});
```


Friday, June 7, 13

What this enables is calling those four methods directly on the database object.

Role delegation

- `does` & `handles` are buddies
- reify some behavior with a role
- duck typing for professionals
- reduces coupling
- supports third-party plugins

Friday, June 7, 13

So to sum up how role-based delegation work, you specify your does and handles with the same role name.
This lets you factor out behavior into a new namespace but still maintain control over how the API should work by publishing the role. It's sort of like saying "I just want an object with this specific behavior. I don't care how I get it." Which is sort of what the goal of duck typing is, but in a more safe and structured way. See also the "Role Usage Patterns" talk. As usual with delegation, the user doesn't know or care that the Database's backend is a separate object, she just consumes the easy begin, commit, rollback, txn_do API directly from the Database. Furthermore anything that does the Backend::Transactional role will be allowed to support the Database, which reduces coupling and is a natural way to support third-party plugins.

Delegation, not inheritance

Friday, June 7, 13

The next pattern I'd like to talk about is delegation as an alternative to inheritance. We briefly covered that when talking about handles => qr/.*/, but I can get into more detail here.

Inheritance is tight-coupling

Friday, June 7, 13

I'm going to start off with this strong, potentially inflammatory declaration that inheritance inevitably leads to tight coupling.

Inheritance

- Can’t hide superclass’s behavior
- Can’t remove methods
- Hope your superclass doesn’t change

Friday, June 7, 13

The first big point is that you can’t hide your superclass’s behavior. If your superclass supports a method your subclass better support it as well. You can’t easily remove a method from your subclass. You can shadow it with a method that just throws an error, but then you’re breaking code that expected that method to work.
You also have to hope your superclass doesn’t change, because if it suddenly starts using a new method that happens to collide with a method in your subclass, then you’re going to have a lot of subtly broken behavior.

Superclass demands

- instance type (hashref, globref, opaque C pointer)
- attribute and method names
- `->isa` and `->DOES`

Friday, June 7, 13

Superclasses also impose a lot of demands of their subclasses. The instance type must be the same because all of the accessors in the superclass still need to be able to pull their attributes out when dealing with your subclass. Hopefully the superclass chose an instance type that allows extra attributes. The superclass also expects that your attribute and method names will not clash with your subclass's attribute and method names. This is because there's a single namespace shared between the superclass and subclass, so if there's an accidental collision, then you're going to have broken expectations and bugs. Collisions can happen by updates to your superclass, even in undocumented private attributes and methods, so this requires vigilance by all parties. Your ->isa and ->DOES and methods are influenced by your superclass's hierarchy and roles.

Superclass demands

- attribute defaults won't be set
- constructor parameters might not set attributes
  - if they are, they won't be type-checked
- `BUILD` won't be called
- nor `DESTRUCT`
- good luck with `MooseX::`

Friday, June 7, 13

The problem is even more acute when your subclass is Moose-based and your superclass isn't.
Your subclass's attribute defaults won't be set, because your superclass doesn't know about your "has ... default => ..." declarations. Constructor parameters might not even set attributes depending on how the superclass's constructor was written. If the superclass constructor was written a particular way, then parameters will make it into attributes, but then they won't be type checked, because the superclass doesn't know about your "has ... isa => ..." declarations. Similarly, BUILD and DESTRUCT are Moose additions so your superclass's constructor or destructor won't be calling them either. And if you use any MooseX modules, who knows if they'll work with non-Moose superclasses. This is especially pernicious because at first blush it will look like you can subclass non-Moose classes safely, but you silently lose all these features.

Superclass demands

- `MooseX::NonMoose`
- `MooseX::NonMoose::InsideOut`
- avoid these if possible

Friday, June 7, 13

There's an extension called MooseX::NonMoose which brings back a lot of these features for you by implementing "new" and "DESTROY" for you which do a lot of this work.
If your superclass's instance type doesn't allow additional attributes, for example because it is an opaque C pointer, then you can use MooseX::NonMoose::InsideOut which safely stores your attribute values outside the object, in the same way that Class::InsideOut does.
These modules work but they're giant hacks so if you can structure your program in a way that doesn't require them, so much the better.

Delegation!

Friday, June 7, 13

Thankfully there is a pattern that lets you avoid these problems. If you're subclassing a non-Moose class, you might have a lot more success if you use delegation instead.

Delegation > inheritance

- Choose which superclass methods you allow
- Methods, attributes, `->isa` not polluted
- Use any instance type
- Wrapped class can evolve
- Wrapping class can evolve

Friday, June 7, 13

With delegation, you can select exactly which superclass methods you will allow. You can use a regular expression like qr/.*/ if you really want to, but I'd avoid it.
Your method and attribute namespaces are not polluted by what happens to be in the superclass. You also can choose whether or not your ->isa method consults the wrapped superclass. You probably don't want to mess with that though.
Your instance type can be whatever you want. You're storing your attributes in your instance, they're storing their attributes in their instance, so there's no risk of collision and no hoops to jump through like Inside-Out objects.
The two classes can evolve independently without having to maintain that strict vigilance about accidentally colliding method names, since there are two separate namespaces for methods.

Native delegation

Friday, June 7, 13

There's another type of delegation that Moose supports that I'd also like to talk about today.

Delegation rocks

- Refactor your internals
- Provide a simpler API
- Document that API
- `handles` is concise and effective

Friday, June 7, 13

So far I think we can all agree that delegation is pretty useful. You can refactor your internals without risking breaking all your clients. You expose a simpler API which is under your control, not the control of some object you expose to the outside world. And Moose's handles is a good tool that concisely declares the kind of delegation you are doing.

Delegation

- Objects only!

Friday, June 7, 13

But there's a problem with delegation. handles is for objects only. But I bet a lot of your code has attributes with array references, hash references, strings, numbers, etc. in them. We can't use delegation for those because those NATIVE TYPES don't have methods.

```
package Queue;
use Moose;

has elements => (
    is      => 'ro',
    isa     => 'ArrayRef',
    default => sub { [] },
);
```


Friday, June 7, 13

Let's say we have a class that implements a queue. We'll call that class Queue because that's what it is. A queue has a list of elements in it.

Queue API

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`
- `my $next = shift @{ $q->elements }`

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`
- `my $next = shift @{ $q->elements }`
- `my $count = scalar @{ $q->elements }`

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`
- `my $next = shift @{ $q->elements }`
- `my $count = scalar @{ $q->elements }`
- `unshift @{ $q->elements }, $jumper`

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`
- `my $next = shift @{ $q->elements }`
- `my $count = scalar @{ $q->elements }`
- `unshift @{ $q->elements }, $jumper`
- `splice @{ $q->elements }, 10`

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`
- `my $next = shift @{ $q->elements }`
- `my $count = scalar @{ $q->elements }`
- `unshift @{ $q->elements }, $jumper`
- `splice @{ $q->elements }, 10`
- `@{ $q->elements } = …`

Friday, June 7, 13

Here's the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

- `push @{ $q->elements }, $new`
- `my $next = shift @{ $q->elements }`
- `my $count = scalar @{ $q->elements }`
- `unshift @{ $q->elements }, $jumper`
- `splice @{ $q->elements }, 10`
- `@{ $q->elements } = …`
- `tie $q->elements, ‘Whoa::There’;`

Here’s the API that our queue class supports.
We can add an element.
We can pull out the first element.
We can count the number of elements in the queue.

We can add an element to the beginning of the queue.
We can throw out the first ten items in the queue.
We can set the queue to be a new list of items.
We can tie the array reference backing the queue to support all sorts of whacky behavior.

Queue API

Friday, June 7, 13

In reality, here's the API we actually want to support. We want a method in the class Queue called add that takes an element and pushes onto the end of the list. We want a next method that shifts the next item off the beginning of the list and returns it. And we want a count method that returns the number of items in the list.
And nothing else! We don't want to support queue jumping, or mass reassignment, or tie...

Queue API

- `$queue->add($new)`

Friday, June 7, 13

In reality, here's the API we actually want to support. We want a method in the class Queue called add that takes an element and pushes onto the end of the list. We want a next method that shifts the next item off the beginning of the list and returns it. And we want a count method that returns the number of items in the list.
And nothing else! We don't want to support queue jumping, or mass reassignment, or tie...

Queue API

- `$queue->add($new)`
- `my $next = $queue->next()`

Friday, June 7, 13

In reality, here's the API we actually want to support. We want a method in the class Queue called add that takes an element and pushes onto the end of the list. We want a next method that shifts the next item off the beginning of the list and returns it. And we want a count method that returns the number of items in the list.
And nothing else! We don't want to support queue jumping, or mass reassignment, or tie...

Queue API

- `$queue->add($new)`
- `my $next = $queue->next()`
- `my $count = $queue->count()`

Friday, June 7, 13

In reality, here's the API we actually want to support. We want a method in the class Queue called add that takes an element and pushes onto the end of the list. We want a next method that shifts the next item off the beginning of the list and returns it. And we want a count method that returns the number of items in the list.
And nothing else! We don't want to support queue jumping, or mass reassignment, or tie...

Queue API

- `$queue->add($new)`
- `my $next = $queue->next()`
- `my $count = $queue->count()`
- *no more!*

Friday, June 7, 13

In reality, here's the API we actually want to support. We want a method in the class Queue called add that takes an element and pushes onto the end of the list. We want a next method that shifts the next item off the beginning of the list and returns it. And we want a count method that returns the number of items in the list.
And nothing else! We don't want to support queue jumping, or mass reassignment, or tie...

Native delegation

Friday, June 7, 13

Ordinarily we'd have to implement those methods ourselves, just like handrolling delegation, to expose the behavior we want with the names we want. If you do implement it yourself you're still likely to get some of the details wrong. Does your "add" method check type constraints? Probably not.
Like with ordinary delegation, you want to expose only the part of the API you're comfortable letting your users have. But an arrayref doesn't have methods, it has builtin procedures that operate on them.
Moose provides a feature called Native Delegation which brings back all those benefits of handles but for Perl's native types. It's like delegation but for native types, so we call it native delegation.

```
package Queue;
use Moose;

has elements => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => 'ArrayRef',
    default => sub { [] },
    handles => {
        add   => 'push',
        next  => 'shift',
        count => 'count',
    },
);
```


Friday, June 7, 13

So here's what native delegation looks like. I'll walk through each piece in turn.

```
package Queue;
use Moose;

has elements => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => 'ArrayRef',
    default => sub { [] },
    handles => {
        add   => 'push',
        next  => 'shift',
        count => 'count',
    },
);
```


Friday, June 7, 13

The first thing you do is pass in a traits option that looks like this. This signals to Moose that you want to use an extension for this attribute. Specifically the extension called Array. This extension gives “handles” a slightly different meaning. Ordinarily handles would complain that your array reference is not a class when you tried to call a delegated method on it.

```
package Queue;
use Moose;

has elements => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => 'ArrayRef',
    default => sub { [] },
    handles => {
        add   => 'push',
        next  => 'shift',
        count => 'count',
    },
);
```


Friday, June 7, 13

Next we say “is => ‘bare’” so we don’t expose the internal object, which gives us all those benefits I keep talking about.

```
package Queue;
use Moose;

has elements => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => 'ArrayRef',
    default => sub { [] },
    handles => {
        add   => 'push',
        next  => 'shift',
        count => 'count',
    },
);
```


Friday, June 7, 13

The traits option does not set up an ArrayRef type constraint for you, you still need that. This is because you might want to say ArrayRef of DateTime objects or something. Or you might not want a type constraint for optimization purposes.
We're actually considering making Moose work the other way: if you say isa => 'ArrayRef', it'll set up this native delegation for you.

```
package Queue;
use Moose;

has elements => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => 'ArrayRef',
    default => sub { [] },
    handles => {
        add   => 'push',
        next  => 'shift',
        count => 'count',
    },
);
```


Friday, June 7, 13

Next up is default => sub { [] }. Again the traits => 'Array' doesn't set this up for you automatically because you might want your default to do more than just provide an empty array. Or you might want to make it required so that
It turns out that in this case, this default is actually unnecessary, because Perl's autovivification will kick in. But it's a good habit to specify it anyway because relying on dodgy side effects like that usually leads to pain.

```
package Queue;
use Moose;

has elements => (
    traits  => [‘Array’],
    is      => ‘bare’,
    isa     => ‘ArrayRef’,
    default => sub { [] },
    handles => {
        add   => ‘push’,
        next  => ‘shift’,
        count => ‘count’,
    },
);
```


And finally, handles. Like usual, you specify the methods you want generated on the left-hand side, and the methods you want to call on the right-hand side. The trick here is that those push, shift, and count aren’t methods, but Moose is smart enough to generate code that just uses the builtin array procedures and syntax instead of method calls.

Trait	“Methods”
Array	push, pop, shift, unshift, splice, count, uniq, elements, join, ...
Hash	set, get, delete, keys, values, kv, exists, count, ...
Counter	inc, dec, reset, ...
Bool	set, unset, toggle, not
String	length, chop, chomp, match, replace, append, substr, ...
Number	add, sub, mul, div, mod, abs, ...
Code	execute, execute_method

Here’s a list of the native traits Moose supports and the “methods” that you can delegate to.
Array we saw
Hash lets you get and set individual keys (handy when combined with currying)
Counter lets you go up and down or start over from zero
Bool lets you set or unset the value, toggle it, or return its “not”.
String and Number are obvious
Code lets you execute the function or execute as a method (which just means you’ll get $self)
There’s more native delegation you can get into, but this is a good taste of what you get out of it. It’s all in the documentation anyway.

Best Practices

Friday, June 7, 13

So let's wrap up with a set of best practices from the ideas I've discussed today.

Don’t expose internal implementation

Friday, June 7, 13

Avoid exposing your internal implementation. You don’t want to be bound by those decisions forever. It’s more work for your users because they have to learn the intricacies of the objects you’re returning.

Rule of thumb: don't leak references

Friday, June 7, 13

One way to achieve this is to avoid leaking references out of your API. You open yourself up to abuse, as we saw several times, like in the queue jumper example.

`is => 'ro'`

allows updates!

Friday, June 7, 13

Remember that even is => ‘ro’ attributes can be updated if they’re references. Users can do anything with a reference. Maybe you want is => ‘bare’.

Be strict on your users

If it isn't in the API, they shouldn't be doing it. You don't have to support their crazy hacks. If they work around your API, they're the ones being naughty. But they're probably doing it to get their job done, so consider their hacks feature requests.
You can yell at them for digging around the internals of your object, or calling underscore methods. You can't really yell at them for calling methods you didn't expect on objects your API returns.

More API later is OK

Friday, June 7, 13

Don't be afraid to limit your API when you're unsure. Don't just expose functionality willy-nilly. If it's not well-considered, it may bite you in the future. See also qr/.*/. The next version can always add the extra functionality. If your next version removes functionality, you may break your users' code and they made get angry.

Moderation

```
handles => {
    out => [ in => sub {
        ...
        # 50 lines of code
        ...
    }],
},
```


Just because you can jam everything into delegation doesn't mean you always should. Regular methods are OK too.

Use native delegation

Friday, June 7, 13

Use native delegation if only because the dereferencing syntax sucks. $queue->add is a lot better than push @{ $queue }.