Role Usage Patterns

Shawn M Moore
@sartak

1

Presented 2012-08-22, Universität Frankfurt, Frankfurt, Germany, YAPC::EU 2012

How to Reuse Code

Let’s start by discussing inheritance and where I think it fails us. Specifically using inheritance as a way to reuse code.

How *NOT* to Reuse Code

Using inheritance to reuse code is going to cause problems.

Inheritance is tight-coupling

I believe 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

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`

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. If you suddenly start returning false where an ancestor returned true, you can no longer use your subclass in place of its superclasses, which means you don't get a lot of the benefits of inheritance.

Multiple inheritance

- ☹ Perilous
- ☹ The Diamond problem
- ☹ Superclass order matters
- ☹ *Unnecessary* inheritance

People often reach for multiple inheritance not to model the actual relationships in your application, but simply to reuse code.
If you want to achieve horizontal reuse, which is sharing code across many different classes in your program, especially classes that aren't directly related, it's hard to do that with inheritance. One solution is multiple inheritance. But there are many problems with multiple inheritance and it's widely considered to be a bad idea. Lots of languages don't even support it because it's fraught with problems.
The Diamond is a well-known problem with multiple inheritance. Basically, what happens if you have a method defined in your two superclasses but you don't override it in your subclass? Which code should be run?
The answer usually depends on the order you list your superclasses. And the order that all of your ancestor classes list THEIR superclasses. Which you can't always change.

Single inheritance

- ☹ put common code in base class
- ☹ or copy/paste it across your classes

Single inheritance itself is not a great tool for reusing code. If you need to share behavior across unrelated classes, you end up putting common code into the most common ancestor (which is hopefully not UNIVERSAL!). This leads to other classes in your hierarchy having methods and behavior that they shouldn't simply because that was the traditional way to reuse code.
Or you could copy and paste the method into your various classes. Which is just obviously bad.

Inheritance poor for Reuse

I conclude that inheritance is a poor design pattern for reusing code. It’s often good enough but I think there’s a better answer out there.

Role Theory

Now that I’ve sufficiently demolished inheritance let’s talk about what roles are and how they work.

```
package Worker::Logging;
use Moose::Role;

has logger => (
    is      => ‘ro’,
    isa     => ‘Logger’,
    builder => ‘_build_logger’,
    handles => [‘log’],
);

sub _build_logger {
    my $self = shift;
    return Logger->new($self->_log_level);
}

requires ‘_log_level’;

before do_work => sub {
    my ($self, $name) = @_;
    $self->log(“About to do $name”);
};
```

So to quickly cover what a role is, a role is a special kind of package. A role is not a class, because you can’t instantiate a role and roles do not participate in inheritance. Instead, the way you interact with roles is fundamentally different.
A role has a set of methods, method modifiers (like before, after, and around, which you’ve probably used in Moose). Roles can also have attributes and all that attributes support like a type constraint, a default value, laziness, etc.
Finally roles also support method requirements which is a way for the role to declare that anything that uses the role must fulfill some requirements.

```
package Worker::Logging;
use Moose::Role;

has logger => (
    is      => ‘ro’,
    isa     => ‘Logger’,
    builder => ‘_build_logger’,
    handles => [‘log’],
);

sub _build_logger {
    my $self = shift;
    return Logger->new($self->_log_level);
}

requires ‘_log_level’;

before do_work => sub {
    my ($self, $name) = @_;
    $self->log(“About to do $name”);
};
```

So to quickly cover what a role is, a role is a special kind of package. A role is not a class, because you can’t instantiate a role and roles do not participate in inheritance. Instead, the way you interact with roles is fundamentally different.
A role has a set of methods, method modifiers (like before, after, and around, which you’ve probably used in Moose). Roles can also have attributes and all that attributes support like a type constraint, a default value, laziness, etc.
Finally roles also support method requirements which is a way for the role to declare that anything that uses the role must fulfill some requirements.

```
package Worker::Logging;
use Moose::Role;

has logger => (
    is      => ‘ro’,
    isa     => ‘Logger’,
    builder => ‘_build_logger’,
    handles => [‘log’],
);

sub _build_logger {
    my $self = shift;
    return Logger->new($self->_log_level);
}

requires ‘_log_level’;

before do_work => sub {
    my ($self, $name) = @_;
    $self->log(“About to do $name”);
};
```

So to quickly cover what a role is, a role is a special kind of package. A role is not a class, because you can’t instantiate a role and roles do not participate in inheritance. Instead, the way you interact with roles is fundamentally different.
A role has a set of methods, method modifiers (like before, after, and around, which you’ve probably used in Moose). Roles can also have attributes and all that attributes support like a type constraint, a default value, laziness, etc.
Finally roles also support method requirements which is a way for the role to declare that anything that uses the role must fulfill some requirements.

```
package Worker::Logging;
use Moose::Role;

has logger => (
    is      => ‘ro’,
    isa     => ‘Logger’,
    builder => ‘_build_logger’,
    handles => [‘log’],
);

sub _build_logger {
    my $self = shift;
    return Logger->new($self->_log_level);
}

requires ‘_log_level’;

before do_work => sub {
    my ($self, $name) = @_;
    $self->log(“About to do $name”);
};
```

So to quickly cover what a role is, a role is a special kind of package. A role is not a class, because you can’t instantiate a role and roles do not participate in inheritance. Instead, the way you interact with roles is fundamentally different.
A role has a set of methods, method modifiers (like before, after, and around, which you’ve probably used in Moose). Roles can also have attributes and all that attributes support like a type constraint, a default value, laziness, etc.
Finally roles also support method requirements which is a way for the role to declare that anything that uses the role must fulfill some requirements.

```
package Worker::Logging;
use Moose::Role;

has logger => (
    is      => ‘ro’,
    isa     => ‘Logger’,
    builder => ‘_build_logger’,
    handles => [‘log’],
);

sub _build_logger {
    my $self = shift;
    return Logger->new($self->_log_level);
}

requires ‘_log_level’;

before do_work => sub {
    my ($self, $name) = @_;
    $self->log(“About to do $name”);
};
```

So to quickly cover what a role is, a role is a special kind of package. A role is not a class, because you can’t instantiate a role and roles do not participate in inheritance. Instead, the way you interact with roles is fundamentally different.
A role has a set of methods, method modifiers (like before, after, and around, which you’ve probably used in Moose). Roles can also have attributes and all that attributes support like a type constraint, a default value, laziness, etc.
Finally roles also support method requirements which is a way for the role to declare that anything that uses the role must fulfill some requirements.

```
package Worker::Logging;
use Moose::Role;

has logger => (
    is      => ‘ro’,
    isa     => ‘Logger’,
    builder => ‘_build_logger’,
    handles => [‘log’],
);

sub _build_logger {
    my $self = shift;
    return Logger->new($self->_log_level);
}

requires ‘_log_level’;

before do_work => sub {
    my ($self, $name) = @_;
    $self->log(“About to do $name”);
};
```

So to quickly cover what a role is, a role is a special kind of package. A role is not a class, because you can’t instantiate a role and roles do not participate in inheritance. Instead, the way you interact with roles is fundamentally different.
A role has a set of methods, method modifiers (like before, after, and around, which you’ve probably used in Moose). Roles can also have attributes and all that attributes support like a type constraint, a default value, laziness, etc.
Finally roles also support method requirements which is a way for the role to declare that anything that uses the role must fulfill some requirements.

```
package Worker::Lazy;
use Moose;

with ‘Worker::Logging’;

sub _log_level { ‘WHINE’ }

sub do_work {
    my $self = shift;
    return “Nahh...”;
}
```

So then in a class you can...

```
package Worker::Lazy;
use Moose;

with ‘Worker::Logging’;

sub _log_level { ‘WHINE’ }

sub do_work {
    my $self = shift;
    return “Nahh...”;
}
```

Consume the role...

```
package Worker::Lazy;
use Moose;

with ‘Worker::Logging’;

sub _log_level { ‘WHINE’ }

sub do_work {
    my $self = shift;
    return “Nahh...”;
}
```

Implement the method that the role requires...

```
package Worker::Lazy;
use Moose;

with ‘Worker::Logging’;

sub _log_level { ‘WHINE’ }

sub do_work {
    my $self = shift;
    return “Nahh...”;
}
```

Then finally implement the method that the role wraps.

Now Worker::Lazy has a logger attribute, and when you call do_work, it WHINEs a message automatically.

Role Theory

- alternative to inheritance
- “horizontal” not “vertical”
- composition model is *inlining*
- roles combine

Role composition is an alternative to inheritance. There is some overlap in what they offer, but roles optimize for different things.

Role Theory

- alternative to inheritance
- “horizontal” not “vertical”
- composition model is *inlining*
- roles combine

We think of inheritance as vertical because you have this hierarchy of classes that is always present. That hierarchy influences every method call and the creation of new subclasses.
We think of what roles offer as horizontal, because they do not participate in the inheritance hierarchy. In fact a class kind of jams them in, and then from the outside you don’t see the roles (unless you choose to). You can also use roles throughout separate parts of your inheritance hierarchy, without hitting the pitfalls that using inheritance would cause.

Role Theory

- alternative to inheritance
- “horizontal” not “vertical”
- composition model is *inlining*
- roles combine

The way horizontal composition works is roles inline methods. When a class “consumes” a role it copies the methods from the role directly into itself. This means we don’t build up additional structure when consuming roles, so we don’t need to change the way methods work to support roles.
Contrast this with the composition model for inheritance, which is traversing the class tree searching for a method during method dispatch.

Role Theory

- alternative to inheritance
- “horizontal” not “vertical”
- composition model is *inlining*
- roles combine

Finally, you can combine roles to form new roles. This property has several implications, which we’ll discuss.

Roles: abstract unit of behavior

One way you can think of roles is that they're an abstract unit of behavior. Typically some behavior is implemented with a few related methods and attributes, so a role is a perfect fit for implementing that behavior. It doesn't make sense to instantiate a behavior or inherit from a behavior, but adding behavior to a class does make sense.

Role Composition

Finally the last bit of role theory is that of role composition. In other words, the ways that roles combine.

Composition

- forms a new role
- union of each role's methods, attributes, etc.
- might satisfy some method requirements
- can generate conflicts
  - which are compile time errors
    - unless they are resolved

When you compose or combine multiple roles, this creates a new role. You can choose to give this role a name if it's meaningful, or what typically happens is Moose generates one for you based on the individual role names.
The new role will have all the methods, attributes, method modifiers, and requirements that each component role has.
Except that some of the method requirements might disappear if they're provided by other roles in the set.
Finally if two roles provide a method of the same name, that generates a conflict. The conflict becomes a compile time error, which is hugely useful, but can be resolved by whoever is consuming the role.

Method Priority

1) local class method

2) superclass method

There’s one last consideration that I need to discuss which impacts the design of roles. What happens when a method name is defined by multiple classes in a hierarchy? Who wins? In most OO languages, and Perl is no exception, the local class wins, and then if needed, inherited methods come after. This means a class can override the methods provided by its superclasses, which has a huge impact on how classes are structured.

Method Priority

1) local class method

2) superclass method

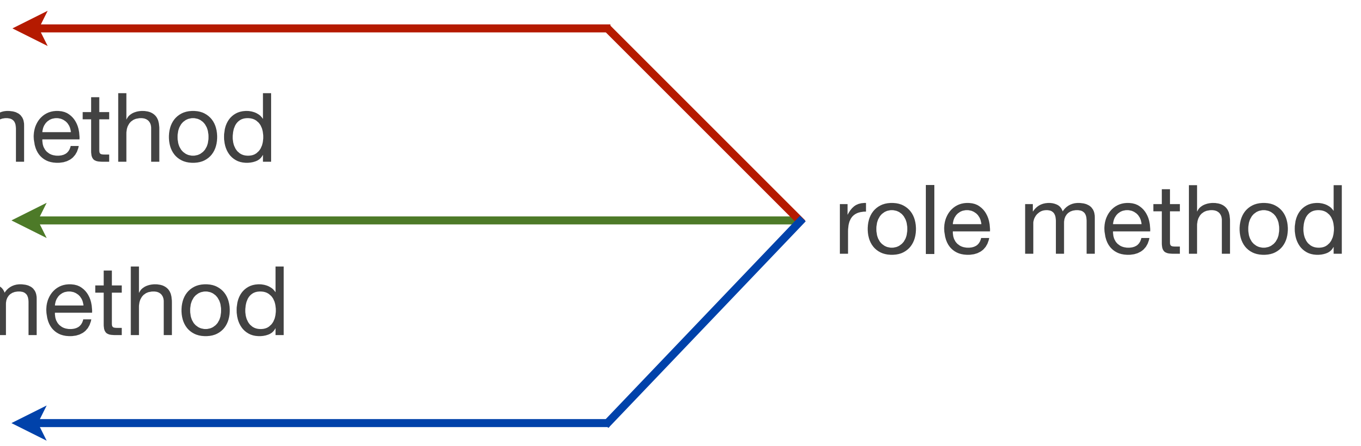
role method

What happens when we add roles into the mix? Where do they come into the priority list? Do they override methods defined in the class that consumed the role, do inherited methods trump role methods, or what?

Method Priority

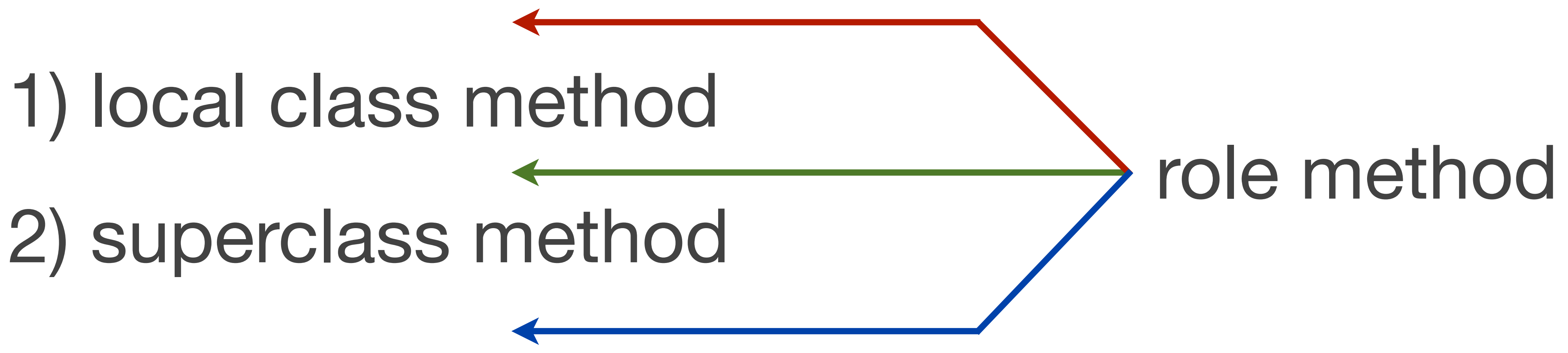
1) local class method

2) superclass method



What happens when we add roles into the mix? Where do they come into the priority list? Do they override methods defined in the class that consumed the role, do inherited methods trump role methods, or what?

Method Priority



30

Please ignore this gratuitous animation slide. :)

Method Priority

1) local class method

2) role method

3) superclass method

Please ignore this gratuitous animation slide. :)

Method Priority

1) local class method

2) role method

3) superclass method

Role methods override inherited methods, but the local class overrides methods that the role provides. This has important implications for the design of your roles.
Effectively this means that a role can offer a generic, least-common-denominator method, but a class can override that to do its work faster when possible, or by using a different method specific to that class.
This decision also has effects on conflict resolution as we'll see.

⚔ Conflict 争

Detection & Resolution

Next I’d like to talk about conflict detection and resolution since this is a huge selling point for roles as horizontal reuse mechanism that is safer and more maintainable than multiple inheritance.

```
package Role::REST;
use Moose::Role;

requires ‘endpoint’;

sub create { ... }
sub read   { ... }
sub edit   { ... }
sub delete { ... }
```

Let’s say we have a REST role. This role takes an endpoint method from its consumer, basically the URL of the resource you want to manipulate. Then it provides you with methods create, read, edit, and delete. These will make HTTP requests to perform those actions on the server.

```
package Role::TextEditor;
use Moose::Role;

sub edit {
    my $text = shift;
    my $file = tempfile($text);
    system(‘vim’, $file);
    return slurp($file);
}
```

Now let’s say we also have another role, this time one that lets you edit some text in your favorite text editor which is vim. It takes some text as a parameter, it saves it to a temporary file, invokes vim on that file, then when you’re done editing, returns the contents of the file.

```
package Role::REST;
sub edit { ... }

package Role::TextEditor;
sub edit { ... }
```

Both of these roles provide a method named edit. This isn’t a problem in itself, any more than two separate classes having methods of the same name. The problem comes up when you combine these two roles into a class.

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’;
```

Here we have a Bugsy::Ticket class for interacting with tickets in some fake bug tracker I’ve dubbed Bugsy.
It starts out as a simple REST client with methods named create, read, edit, delete, just like you’re used to.
But then your boss comes along and says he wants to edit tickets in his favorite text editor which is vim.

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’,
     ‘Role::TextEditor’;
```

So you start out by adding the existing Role::TextEditor to your class and firing up your code again.

```
Due to a method name conflict in roles
'Role::REST' and 'Role::TextEditor', the
method 'edit' must be implemented or
excluded by 'Bugsy::Ticket'
```

Because the roles we’re combining both have a method named edit, Moose doesn’t know which edit method the class should get. It flags this error at compile time. It tells you the roles that were involved, the method that generated the conflict, and the class that was consuming the roles.

```
Due to a method name conflict in roles
'Role::REST' and 'Role::TextEditor', the
method 'edit' must be implemented or
excluded by 'Bugsy::Ticket'
```

The error also tells you how to resolve the conflict. Let’s look at both options that Moose offers: implement the method or exclude it.

implementing `edit`

First we’ll look at what implementing “edit” looks like.

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’,
     ‘Role::TextEditor’;

sub edit { ... }
```

Let’s start by implementing the method named edit. The principle behind this type of conflict resolution is that, as we saw with the funky animation, local class methods override methods declared in roles they consume. So by declaring an “edit” method we are overriding the two “edit” methods brought in by the roles. Which resolves the conflict because the class is calling the shots.
So what should our new edit method do?

```
sub edit {
    my $self = shift;
    my $id   = shift;

    # fetch ticket content
    my $contents = $self->read($id);

    # fire up text editor
    $contents = $self->edit($contents);

    # push new content back up
    return $self->edit($id, $contents);
}
```

How about the edit method takes a ticket ID, fetches its contents, fires up the editor with those contents, and then pushes the edited content back up to the server. I think this makes the most sense in combining the two distinct features named “edit” into one method.

```perl
sub edit {
    my $self = shift;
    my $id   = shift;

    # fetch ticket content
    my $contents = $self->read($id);

    # fire up text editor
    $contents = $self->edit_text($contents);

    # push new content back up
    return $self->put_edit($id, $contents);
}
```

The only sane way to disambiguate is to call the methods by different names. Here we say the first edit method from the text-editor role will be called by the name edit_text. Then the REST edit method will be called put_edit (after the HTTP verb PUT it would use).

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’ => {
         -alias => { edit => ‘put_edit’ }
     },
     ‘Role::TextEditor’ => {
         -alias => { edit => ‘edit_text’ }
     };

sub edit { ... }
```

Here’s how we tell Moose to do exactly that. When we’re consuming the roles, we can instruct Moose to provide additional names for methods by using the “alias” option. We use “alias” to request disambiguated method names from each role, then our implementation can use them in the composite “edit” method that uses the behaviors of each role’s “edit” method.
And that is how you can resolve a method conflict by implementing a method with that name. The standard way is with aliases like this.

```
# fire up text editor
$contents =
    $self->Role::TextEditor::edit(
      $contents
    );

# push new content back up
$self->Role::REST::edit($id, $contents);
```

An alternative way is to specify the method names directly. One of Perl’s little-known features is the ability to specify which package you want method lookup to occur in. So instead of using “alias” to provide disambiguated method names, you can use this to call the methods you want directly.
However, my recommendation right now is to stick with “alias”, since that is the standard way. But this direct method invocation may become the new standard way in a future version of Moose, so I wanted to mention it here.

excluding

`edit`

So that was one resolution strategy, implementing the conflicting method. Moose offers another possible resolution which is excluding the method.

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’,
     ‘Role::TextEditor’;
```

Here’s where we’re starting from. This is a conflict because both roles we’re consuming provide an “edit” method.

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’ => {
         -excludes => [‘edit’],
     },
     ‘Role::TextEditor’;
```

What we can do is tell Moose to exclude the method named “edit” from one of the roles. This resolves the conflict because now only one of the roles is providing a method named “edit”.
But this particular choice of excluding edit from the REST role isn’t very good: it means when you call edit on a ticket, it just pops up a text editor with the ID of the ticket you wanted to change. When you save and leave the editor, nothing happens. We never push anything to the server because the REST functionality for “edit” was excluded.

```
package Bugsy::Ticket;
use Moose;

sub endpoint { ‘http://bugsy.com/ticket’ }

with ‘Role::REST’,
     ‘Role::TextEditor’ => {
         -excludes => [‘edit’],
     };
```

So the other alternative is to exclude “edit” from the text editor role. But that’s no good either because the whole point of consuming this TextEditor role is so your boss can edit tickets in his favorite text editor which is vim. If you exclude that edit method then you’re excluding the very functionality you wanted from the role and your class isn’t doing what was needed.

```
Due to a method name conflict in roles
'Role::REST' and 'Role::TextEditor', the
method 'edit' must be implemented or
excluded by 'Bugsy::Ticket'
```

So only one of these two strategies was appropriate for our class. Sometimes it’ll go the other way and exclusion is the appropriate strategy. You have to decide which one makes more sense for the situation.

alias & excludes

alias and excludes are powerful features intended to oil the role composition engine. When you're using roles heavily you're going to run into name conflicts and alias/excludes help you to manage the complexity that involves.
But you might be tempted to use these features freely, outside of the context of role conflict resolution. Perhaps you simply want a better domain-specific name for a method, so you use alias to create a new name for that method, and excludes to delete the old name.

alias
&
excludes

code smell!

Well that's a problem! By consuming the role you promised that the method the role provides will be there, and it will be a specific behavior. Renaming the method, or having different behavior from what the role specifies, basically lies to your users.
Don't lie to your users. Don't break the role's contract.

KiokuDB

- object database
- like Neo4j, AllegroCache, etc.
- many backends
  - SQLite, JSON, BerkeleyDB, hashref, etc.
  - different limitations and capabilities
- good role design

The rest of the talk will draw many examples from a project called KiokuDB. KiokuDB is an object graph database. Basically you put complete objects into the database and you can get them back out as objects. It's vaguely like an ORM, but designed from the ground-up to support objects. If you've used neo4j, AllegroCache, it's like those. Anyway, Kioku can store data in many different backends. Each backend has its own specific set of capabilities and limitations. Kioku does a good job of modeling those capabilities and limitations with roles.

Types of Roles

Now that we've covered all of the features of roles and their importance, let's get into the meat of the talk, which is discussing some patterns for how roles can be used to build robust, flexible systems. My goal here is to explain some of the common Design Patterns that roles enable. I made up most these names, because there's no standard naming convention for the patterns. That's another reason why I'm giving this talk, to try to give consistent name these patterns, so that we can discuss the ideas more easily.

Tag Role

The first and simplest role pattern is the tag role.

Tag Role

- no methods
- no attributes
- no method requirements
- for `->does()` only
- documentation-as-code

A tag role has no methods and no attributes. The tag role also requires no specific methods of its consumers.
So how is it useful? Recall that you can ask a class or object whether it does a particular role by calling does and passing the role name as a parameter.
You might think of this as documentation but in code form. The class is declaring something about itself using a bit of code.

BinarySafe

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

1;
```

Let's look at a concrete example pulled from KiokuDB for how this is useful. This is the complete role definition. It's just a package statement, a use Moose::Role, and then a 1 to finish the package. Code-wise there's nothing missing or elided here. The package itself will probably contain some documentation explaining what the role means and under what conditions you can consume it.

BinarySafe

```
package Backend::Hash;
use Moose;
with ‘Backend::BinarySafe’;
...

package Backend::JSON;
use Moose;
~~with ‘Backend::BinarySafe’;~~
...
```

Here we have two backend classes. The first one is for storing objects in an in-memory hash. This can be useful for testing or for tiny projects that don’t need to persist data. Perl hashes can accept binary data, specifically null bytes, just fine.
On the other hand, the JSON backend, which uses the JavaScript Object Notation format, cannot handle binary data, specifically null bytes. So if it declared that it was BinarySafe, that would be a lie and things would eventually break.

BinarySafe

```
if ($backend->does('Backend::BinarySafe')) {
    $backend->store($data);
}
else {
    $backend->store(base64_encode($data));
}
```

Here’s how it’s useful. When we’re trying to store arbitrary data into a backend, we need to know if the backend is binary-safe or not. If it’s binary-safe, then we can store our arbitrary data in it as-is. But if the backend is NOT binary-safe, then we must encode the data so that it doesn’t include literal null bytes.

BinarySafe

```
my $data = $backend->retrieve($key);
if (not $backend->does(‘Backend::BinarySafe’)) {
    $data = base64_decode($data);
}
return $data;
```

On the other side, when we’re pulling data out of the backend, we need to know if it was encoded or not. If the backend is binary safe then we can just return it, otherwise we may need to decode it.

Interface Role

The next role pattern I'd like to discuss is the interface role. This pattern gets its name because you're using roles similar to the way that Java interfaces work.

Interface Role

- require a set of methods
- now you have a *name* for that set
- with a place to put documentation

The interface role simply requires a set of methods. It does not provide any methods or attributes itself. The benefit of the interface role is that you now have a specific name for the behavior that those methods encapsulate. You can use that specific name in “does” or “handles” which is a nice shortcut. Finally that name is a package name, and we as a community have lots of practice adding documentation to package names.

Interface Role

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

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

Here we’re again borrowing an example from Kioku. We’re declaring a role called Backend::Transactional that requires methods named begin, commit, and rollback. A backend may or may not be transactional, so it may or may not have these methods. If a backend has these methods it can use this role to declare that it supports transactions.
Obviously the SQL backends will support these operations natively, but we might implement them by hand for a Hash backend too. We could create a temporary, working-copy of the original data at “begin” time, then during “commit”, we move it into place (blessing it as canonical), and during “rollback” we throw away the temporary working-copy hash.

Interface Role

```
has backend => (
    is   => ‘ro’,
    does => ‘Backend::Transactional’,
);
```

Now anywhere we specifically need a backend that supports transactions, we can simply consult whether the class does the role. Now we know anything that makes it into this “backend” attribute not only has the begin, commit, and rollback methods, we also know that those methods have the specific semantics described in Backend::Transactional’s documentation.

Interface Role

```
if ($backend->does(‘Backend::Transactional’)) {
    $backend->begin;
    $do_work->();
    $backend->commit;
}
else {
    $do_work->();
}
```

65

12年8月22日水曜日 65

We can also imagine explicitly asking the backend whether it’s transactional. If it is, then we know we can do some work with the safety net that transactions provide. If the backend does not support transactions then we can just go ahead and do some operation and cross our fingers that nothing will go wrong.

Duck Typing

```
if ($backend->can('begin')
 && $backend->can('commit')) {
    $backend->begin;
    $do_work->();
    $backend->commit;
}
else {
    $do_work->();
}
```

You're probably familiar with a pattern called duck typing. Duck typing fulfills a similar need here where we're propping up the transactional safety net but only if we can. But I have some objections with duck typing and I think if you can use interface roles instead, you absolutely should.

Role vs Duck

- interface role is **explicit** and **declarative**
- duck typing is overly optimistic
- roles have discoverable documentation
- duck typing can match coincidentally

I like interface roles better because they're explicit. Duck typing is more implicit. Even if it "walks like a duck and quacks like a duck", it might be an overweight snake oil salesman.
Roles have discoverable documentation: you already know how to find the documentation of a particular package. That documentation can list expectations, requirements, common usage patterns, etc.
Finally, a duck type can accidentally match whereas consuming an interface role is never an accident.

handles

- sets up delegation for an attribute
- usually a list of method names
- but! it can take a role name

You've probably used the "handles" option when creating attributes before. It creates methods in your class that calls methods in the object that the attribute holds. Usually people pass in an array reference or hash reference of method names to generate.
But you can also pass in a role name.

`handles => ‘Role’`

- delegates the provided methods
- delegates the required methods
- great with `does => ‘Role’`

When you pass a string to handles, Moose interprets it as a role name. It then generates delegate methods for the methods that the role provides, and also the methods that the role requires.
If you’re using “handles” with a role you’re almost certainly going to want to use “does” with the same role name for your type constraint.

handles => ‘Role’

```
package Database;

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

Here’s what that looks like. Thanks to handles, Moose generates methods called “begin”, “commit”, and “rollback” methods in the class that call the same methods on the backend. We also know that any object that is stored in the backend attribute has passed the “Backend::Transactional” type constraint, so it has the “begin”, “commit”, and “rollback” methods. But other than that, we don’t care about the class of the object, or what its superclasses are, or whether it has other methods. We’re exactly as specific as we need to be, and no more.

handles => ‘Role’

```
my $dbh = Database->connect(...);

$dbh->begin;
    # $dbh->backend->begin;
$dbh->commit;
    # $dbh->backend->commit;
$dbh->rollback;
    # $dbh->backend->rollback;
```

Now the net effect of that is that we can call begin, commit, or rollback directly on the database handle. These delegate methods that Moose generated will call the same methods on the backend attribute that it’s storing internally.

```
if ($obj->can('bark')) {
    $obj->bark(at => 'mailman');
}
```

You've probably heard of the "bark" problem before. The idea is that you check whether an object supports a "bark" method and if it does, go bark at the mailman.
Well bark can mean the sound a dog makes OR the skin of a tree. Obviously it doesn't make sense to skin-of-a-tree at the mailman.

bark

```
package Role::Doglike;
use Moose::Role;
requires ‘bark’;

package Role::Treelike;
use Moose::Role;
requires ‘bark’;
```

We can solve this bark problem with interface roles. We create two roles with the method, each requiring the ambiguous method. Then the classes that implement the method can choose which interface they adhere to, and declare that they “do” that role.

```
if ($obj->does(‘Role::Doglike’)) {
    $obj->bark(at => ‘mailman’);
}
```

Now we can ask the object not if it can bark, but if it fulfills the doglike role. If they do the “doglike” role then they have the “bark” method since that is a requirement of the role. Furthermore, the role documents what the “bark” method means. In this case, making a loud sound.

Behavior Role

The next pattern is behavior roles. The idea behind behavior roles is simply that we want to share behavior across multiple classes.

reuse code
without
unnecessary inheritance

Behavior roles let you reuse code across multiple different classes. If you don't have roles, you might design your application such that you use unnecessary inheritance to achieve the behavior reuse that you need.
You'd be in good company: the core Smalltalk collection classes did exactly this.

reuse code without *multiple* inheritance

Similarly you might be tempted to use multiple inheritance to reuse code across many otherwise-unrelated classes, but as we discussed multiple inheritance leads to pain.

reuse code without *polluting base classes*

Finally you may end up putting the code into some common ancestor, but that also leads to pain when you have classes below that ancestor that do not want or deserve that behavior.
Again, by doing this you'd be in good company with the designers of Smalltalk.
You might even be tempted to commit the especially egregious sin of polluting the UNIVERSAL class, which is the topmost ancestor of all classes in Perl.

Behavior Role

```
has schema => (
    is       => ‘ro’,
    isa      => ‘DBIx::Class::Schema’,
    required => 1,
    handles  => ...,
);
```

Let’s look at a concrete example of a behavior role. So, applications that have a database connection typically need access to it throughout many classes in the application. Your web application controllers, your email subsystem, your job queue, command-line scripts, etc. Many such classes might have an attribute called schema which is required, is of type DBIx::Class::Schema, etc.
But that means we need to have this 6+ lines of code in every class, or we need to put it into some common base class, or we need a HasDBICSchema class that everyone multiply inherits from. All are crappy solutions.

Behavior Role

```
package HasDBICSchema;
use Moose::Role;
has schema => (
    is       => ‘ro’,
    isa      => ‘DBIx::Class::Schema’,
    required => 1,
    handles  => ...,
);
```

So what if we put this attribute declaration into a role called HasDBICSchema?

Behavior Role

```
with ‘HasDBISchema’;
```

Now all you need to say is “with ‘HasDBICSchema’” in any class to get access to that required schema attribute, and then that class can interact with the database.
The only problem is that now anywhere you instantiate objects of the classes that use HasDBISchema, you need to pass in the schema object. This might be totally OK for your problem.

Behavior Role

```
package HasDBICSchema;
use Moose::Role;
requires ‘dsn’;
has schema => (
    is       => ‘ro’,
    isa      => ‘DBIx::Class::Schema’,
    ~~required => 1,~~
    default  => sub { Foo->connect(shift->dsn) },
    handles  => ...,
);
```

What you can choose to do instead is to let classes construct the schema object themselves. No longer does the role demand the schema from whoever’s constructing the object with “required”. Instead, the role requires the class to provide a “dsn” method which specifies the database to connect to. Then the role consults that “dsn” method in the default to create a connection.

CPAN Behavior

- `Throwable`
- `MooseX::Getopt`
- `MooseX::Role::Matcher`

There’s a couple handy behavior roles on CPAN.
Throwable is a role which adds methods and attributes to throw an object of your class as an exception. It provides a ->throw method, tools for enabling identification, capturing stack traces, etc.
MooseX::Getopt gives your class a new constructor that will use the @ARGV command-line arguments to fill in your class’s attributes.
MooseX::Role::Matcher adds a few methods to your class for matching values in attributes and methods.

Plugin Role

The next pattern I'd like to highlight is a plugin role. This idea developed in the Dist::Zilla project by Ricardo Signes which is for packaging and releasing Perl code. That project has a very specific way of using role features that is very effective.

Dist::Zilla

```
     AllFiles
   ExtraTests
  InstallDirs
      License
    MakeMaker
     Manifest
 ManifestSkip
     MetaYAML
   PkgVersion
     PodTests
   PodVersion
   PruneCruft
       Readme
 UploadToCPAN
```

85

Here’s a list of plugins used by a typical Dist::Zilla-based distribution.

Dist::Zilla

```
     AllFiles
   ExtraTests
  InstallDirs
      License
    MakeMaker
     Manifest
 ManifestSkip
     MetaYAML
   PkgVersion
     PodTests
   PodVersion
   PruneCruft
       Readme
  UploadToCPAN
```

```
$_->gather_files
for
$self->plugins_with(
-FileGatherer
);
```

Dist::Zilla itself occasionally calls methods like this. The key bit is “plugins_with”.

Dist::Zilla

```
     AllFiles
   ExtraTests
  InstallDirs
      License
    MakeMaker
     Manifest
 ManifestSkip
     MetaYAML
   PkgVersion
     PodTests
   PodVersion
   PruneCruft
       Readme
 UploadToCPAN
```

```
$_->gather_files
for
$self->plugins_with(
-FileGatherer
);
```

plugins_with takes a role name as its parameter. The dash is just a bit of syntax that indicates that the role is in the usual Dist::Zilla::Plugin:: namespace.

Dist::Zilla

```
AllFiles
ExtraTests
InstallDirs
License
MakeMaker
Manifest
ManifestSkip
MetaYAML
PkgVersion
PodTests
PodVersion
PruneCruft
Readme
UploadToCPAN
```

```
$_->gather_files
for
$self->plugins_with(
-FileGatherer
);
```

plugins_with then selects all of the plugins in the that do the role name you specified. All of the highlighted plugins on the left do this “FileGatherer” role, which means the plugin adds files to the distribution.

Dist::Zilla

```
     AllFiles
   ExtraTests
  InstallDirs
      License
    MakeMaker
     Manifest
 ManifestSkip
     MetaYAML
   PkgVersion
     PodTests
   PodVersion
   PruneCruft
       Readme
  UploadToCPAN
```

```
$_->gather_files
for
$self->plugins_with(
-FileGatherer
);
```

Then finally this bit of code calls the gather_files method on each of these FileGatherer plugins, so they can go ahead and add their files to the distribution. Of course, the FileGatherer role requires the gather_file method.
“License”, “README”, and “MetaYAML” add the LICENSE, README, and META.yml files to the distribution. “AllFiles” adds each file that the author actually wrote. Finally “PodTests” adds POD-testing files.

Dist::Zilla

```
AllFiles
ExtraTests
InstallDirs
License
MakeMaker
Manifest
ManifestSkip
MetaYAML
PkgVersion
PodTests
PodVersion
PruneCruft
Readme
UploadToCPAN
```

```
$_->munge_files
for
$self->plugins_with(
-FileMunger
);
```

Dist::Zilla uses this architecture for all the interesting parts of building a CPAN distribution. Even the boring parts too, actually. You could think of Dist::Zilla itself as just a plugin-execution-engine, because all the real work is done by plugins.
A plugin of course could do multiple roles if it affects multiple stages of the pipeline.

Plugin Roles

- choice!
  - `ModuleBuild` or `MakeMaker`
  - `MetaYAML` or `MetaJSON`
- extensible
- project- and company-specific plugins

This design gives the user choice of which behavior she wants. And in my experience, users really really want that choice. I'm sure if Dist::Zilla declared "all Dist::Zilla projects will generate Module::Build" files a lot of potential users would be turned off.
This design is also extensible for free. I'm sure there are plenty of Dist::Zilla plugins that do things the author didn't imagine.
Finally it also, of course, enables project-specific and secret company-specific plugins. We have our own dzil plugin at Infinity that generates versioned packages from our very specific git repository layout.

Abstract Role

Next up is the Abstract Role. The “Abstract” refers to abstract base classes or ABC. This role pattern offers many of the features that people use ABC for, but with fewer downsides.

ABC

- no `->new` method
- provides methods
- requires methods (via `sub foo { die }`)
- often checked with `->isa`

An abstract base class does not have a ->new method. Or if it does, it throws an error about how you're not supposed to instantiate this class directly. That's what the "abstract" means.
The ABC typically provides some methods, and requires other methods that it will call. Of course, since classes can't require methods directly, typically what will happen is that it stubs out the method with a die statement telling you that your subclass needs to implement this method. At runtime.
Finally ABCs are typically used with ->isa to group sibling classes together.

Role

- no `->new` method
- provides methods
- requires methods (via `requires`)
- often checked with `->does`

Well roles have all those same features! You do not instantiate roles. They don't have a ->new method.
Roles can of course provide methods. Roles also require methods and failure to implement them is a compile-time error instead of a runtime error.
Finally you can check whether an object has this role with the ->does method.

Roles > ABC

Roles are better than abstract base classes because roles can catch missing method errors at compile time, but ABC missing methods are runtime errors in Perl. Roles don't offer a "new" method, and because they're roles, it's obvious that you don't instantiate them.
The MooseX::ABC documentation even says in big bold letters at the top "This module is almost certainly a bad idea. You really want to just be using a role instead.

Combinations!

Most of the roles you write will actually be a combination of the interface pattern and the behavior pattern. Let's at a couple examples of those.

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

requires ‘all_entries’;

sub root_entries  { ... }
sub child_entries { ... }
sub all_entry_ids { ... }
```

Here’s another example from KiokuDB. The role requires an all_entries method which is documented to take no parameters and must return a particular type. In this way it is an interface role.
Then the role provides several convenience methods for its consumers to use. root_entries greps through all_entries to find entries with no parents, all_entry_ids maps through to return a list of all the IDs, etc.

```
package Backend::Role::TXN::Memory;
use Moose::Role;

requires ‘commit_entries’,
         ‘get_from_storage’;

sub txn_begin    { ... }
sub txn_rollback { ... }
sub txn_commit   { ... }

sub insert       { ... }
sub delete       { ... }
```

TXN::Memory is a role that requires some methods and then builds transactional support on top of those. It also modifies how insert, delete, get, exists, etc. work to be transaction-aware.
This role is also acts as a tag role to declare that this backend works in memory, so it needs special consideration with garbage collection.

Semi ~~Poor Man's~~ Parameterized Roles

The next pattern I'd like to talk about is parameterized roles. The "poor man's" bit refers to the fact that the kind of role I'm about to describe is not a true parameterized role, but it does have ways to influence the behavior of the role. I crossed out the "poor man's" because if you can use this pattern instead of a true parameterized role, you should. Maybe I'll start calling them semi-parameterized roles.

Semi-parameterized Roles

- `requires`
- let the consuming class influence behavior

The idea with semi-parameterized roles is that you use the “requires” feature of roles to let the consuming class influence the behavior of the role.

Semi-parameterized roles

```
package RetryAble;
use Moose::Role;
requires ‘operation’, ‘retry_count’;

around operation => sub {
    my ($orig, $self, @args) = @_;
    for (1.. $self->retry_count) {
        last if eval { $self->$orig(@args); 1 };
    }
};
```

Here’s a basic semi-parameterized role RetryAble. The idea is that the role wraps the “operation” method, which can do anything, with a loop that retries however many times the class asked for. So we’re parameterizing on retry_count.
We’ll cover this more when we discuss real parameterized roles.

S-P Role Limits

- can only parameterize behavior in methods
- parameters are evaluated at runtime
- can’t hide those parameter methods

Semi-parameterized roles are useful because they use the existing role infrastructure in their design. But they have limits.
You can only parameterize the behavior of methods in the role. You can’t influence the types or names of attributes, just their default values (since “builder” does work in this pattern).
The parameters you specify for the role come in the form of methods that are called at runtime. And you can’t hide those methods because at any time the role may need them.

MooseX::Role::Parameterized

There’s a Moose extension called MooseX::Role::Parameterized. Unlike semi-parameterized roles, MooseX::Role::Parameterized lets you parameterize any aspect of the role that you want. The effect is that MooseX::Role::Parameterized can sort of act as a declarative class macro system.

```
package Queue;
use MooseX::Role::Parameterized;

parameter item_type => (
    isa => ‘Str’,
);

role {
    my $p = shift;
    my $queue_type = $p->item_type
                   ? ‘ArrayRef[‘ . $p->item_type . ‘]’
                   : ‘ArrayRef’;

    has elements => (
        isa     => $queue_type,
        default => sub { [] },
    );
};
```

Here’s how MooseX::Role::Parameterized works. There’s a lot of code here so let’s take it bit by bit.

```
package Queue;
use MooseX::Role::Parameterized;

parameter item_type => (
    isa => ‘Str’,
);

role {
    my $p = shift;
    my $queue_type = $p->item_type
                   ? ‘ArrayRef[‘ . $p->item_type . ‘]’
                   : ‘ArrayRef’;

    has elements => (
        isa     => $queue_type,
        default => sub { [] },
    );
};
```

First off to declare a role that can take parameters, you use MooseX::Role::Parameterized instead of Moose::Role. This is so MXRP can add additional sugar to your role, and set up the metaobject layer correctly.

```
package Queue;
use MooseX::Role::Parameterized;

parameter item_type => (
    isa => ‘Str’,
);

role {
    my $p = shift;
    my $queue_type = $p->item_type
                   ? ‘ArrayRef[‘ . $p->item_type . ‘]’
                   : ‘ArrayRef’;

    has elements => (
        isa     => $queue_type,
        default => sub { [] },
    );
};
```

Next is a set of parameter declarations. Each one takes the form of the keyword “parameter” followed by the parameter’s name, then a list of options. You might notice a superficial similarity to the “has” keyword. That was very much intentional and in fact, you can use all of the behavior that “has” provides. The only actual difference is that “parameter” has a default “is => ‘ro’” for convenience, because 99.99% of the time, that’s what you actually want.
Unfortunately Moose doesn’t have a type constraint for TypeConstraintName so we have to use the less specific Str instead.

```
package Queue;
use MooseX::Role::Parameterized;

parameter item_type => (
    isa => ‘Str’,
);

role {
    my $p = shift;
    my $queue_type = $p->item_type
                   ? ‘ArrayRef[‘ . $p->item_type . ‘]’
                   : ‘ArrayRef’;

    has elements => (
        isa     => $queue_type,
        default => sub { [] },
    );
};
```

Next up is a block of code starting with the keyword “role”. The block of code inside “role” is there to generate a role from the parameters that the user specified. This block will be executed anew every time the parameterized role is consumed, but with different parameters.

```
package Queue;
use MooseX::Role::Parameterized;

parameter item_type => (
    isa => ‘Str’,
);

role {
    my $p = shift;
    my $queue_type = $p->item_type
                   ? ‘ArrayRef[‘ . $p->item_type . ‘]’
                   : ‘ArrayRef’;

    has elements => (
        isa     => $queue_type,
        default => sub { [] },
    );
};
```

The first thing you do inside role is pull out the parameters that the consumer specified for this role. $p is actually an object, and the “parameter” declarations specify the attributes that $p will have. So if in your parameters you set defaults, handles, required, etc., all of that will apply to your $p object.

```
package Queue;
use MooseX::Role::Parameterized;

parameter item_type => (
    isa => ‘Str’,
);

role {
    my $p = shift;
    my $queue_type = $p->item_type
                   ? ‘ArrayRef[‘ . $p->item_type . ‘]’
                   : ‘ArrayRef’;

    has elements => (
        isa     => $queue_type,
        default => sub { [] },
    );
};
```

Then we consult $p’s value for the item_type attribute, which corresponds to the parameter that the user specified. We look at this value to decide what type constraint the elements attribute will have. If the user specified no value for item_type we use a plain ArrayRef type, but if the user specified a type then we build up a string of “ArrayRef[subtype]” for use in the attribute.

```
package MyApp::MovieQueue;
use Moose;

with ‘Queue’ => {
    item_type => ‘MyApp::DVD’,
};
```

Here’s how we can use a parameterized role. You don’t need any special extensions for classes that use parameterized roles. The only trick is that you simply specify parameters to the roles you’re consuming. We already saw this with -alias and -excludes, but now the role can declare any additional parameters that it needs.

```
package MyApp::MovieQueue;
use Moose;

with ‘Queue’ => {
    item_type => ‘MyApp::DVD’,
};
```

At “with” time, the block of code you specified in “role { ... }” is executed. MooseX::Role::Parameterized will use the parameters you specify here (item_type => ‘MyApp::DVD’) as constructor arguments create an object which you fleshed out with the “parameter” declarations.

```
package MyApp::MovieQueue;
use Moose;

with ‘Queue’ => {
    item_type => ‘MyApp::DVD’,
};

has elements => (
    isa     => ‘ArrayRef[MyApp::DVD]’,
    default => sub { [] },
);
```

To solidify what the parameterized role is doing, consuming it is creating an elements attribute in the MyApp::MovieQueue class with type ArrayRef[MyApp::DVD].
There’s no runtime penalty for using a parameterized role.

```
package MyApp::MovieQueue;
use Moose;

with ‘Queue’ => {
    item_type => [],
};

# -> Attribute (item_type) does not pass
the type constraint because: Validation
failed for 'Str' with value [  ]
```

It’s worth pointing out that parameterized roles are using all of the mechanics you already understand. Here we’re passing an array ref as the item_type parameter, which violates the type constraint, so you get an error telling you as much.

Parameterized Roles

- configure a role’s attributes
  - including name
- tell the role about your class
- additional validation

Here are some things you can use parameterized roles for. You can configure a role’s attributes, as we saw by configuring the “isa” type constraint in our Queue example. There’s also nothing stopping you from making the name of an attribute a parameter.
You can tell the role about your class. In this way you can have a role that wraps some method in your class that you specify. Anything you can think of as a parameter is fair game.
Finally you can also do some additional validation. Does your role require a class to have an attribute of a particular type? Does your role require a method with a particular signature?

Parameterized Roles

- powerful
- but dangerous
- beware breaking ->`does`
- maybe treat them like class macros
- use a regular role whenever possible

Parameterized roles are a very powerful feature but they can be dangerous because if you go overboard then ->does becomes useless.
If you treat them more like class macros than roles, then you might have a better understanding of their place in the Moose ecosystem. See also the Reflex project which definitely embraces this mindset.
Lisp programmers often warn that you shouldn't use a macro when an ordinary function would do. Same here: don't use a parameterized role when an ordinary role would do. Parameterized roles have much greater complexity and danger.

Parameterized Roles

- don’t use a **MACRO** when a **FUNCTION** would do
- don’t use a **PARAMETERIZED ROLE** when a **ROLE** would do

Lisp programmers have often said that you should never use macros when ordinary functions would work. This is because macros are fundamentally more difficult to understand and use correctly. Similarly, if you can use an ordinary role, even a semi-parameterized role, do that. Only reach for parameterized roles when what you’re doing cannot be achieved with ordinary roles.

Runtime
Role Application

One neat thing you can do with roles is apply them at runtime. And not only to classes, but also to individual objects!

Runtime Roles

- `apply_all_roles`
- `ensure_all_roles`
- `with_traits`
- `MooseX::Traits`
- `MooseX::ClassCompositor`

This is such a handy pattern that a number of tools have cropped up around applying roles at runtime. The first three are functions from Moose::Util, so they're bundled in the core Moose distrobution.
apply_all_roles takes a class or object and a list of role names and adds the roles to that class/object.
ensure_all_roles does the same but skips any roles that the class/object already has.
with_traits creates a new class, which will be a subclass of the class name you pass in, with the roles you specify. It will return the class name.
MooseX::Traits has similar behavior but it adds methods like ->with_traits to your class instead of being a utility function.
Finally MooseX::ClassCompositor is a tool that generates classes from detailed specifications including a list of roles.

Exceptions

```
my $exception_class = with_traits('Exception',
     'Exception::File',
     'Exception::Timeout',
);

my $exception = $exception_class->new(
    file    => $filename,
    timeout => $duration,
);
```

Here's one kind of situation where applying roles at runtime makes sense. We had an error in our code so we want to throw an exception. The error condition specifically was that reading a file timed out, maybe because we're dealing with NFS. So we create a subclass of our generic Exception class and choose a-la-carte the roles Exception::File and Exception::Timeout. Then we construct an object from this class and pass in the specific details of the error case.

Exceptions

```
my $exception_class = with_traits(‘Exception’,
    ‘Exception::Validation’,
    ‘Exception::HTTP::400’,
);

my $exception = $exception_class->new(
    original_input => $input,
    expected       => $type,
);
```

This pattern go as broad as you want. Here we’re creating another exception which apparently was produced by invalid input. Note that we’re using a tag role here to tell the HTTP infrastructure what error code to present to the user.
Exceptions are a particularly common use case for this kind of thing because the data you want to include in the exception might change depending on the context from which you’re throwing that exception.

Runtime Roles

- creates anonymous classes
- classes that escape your API should have names

Runtime role application creates anonymous classes by default. This isn't necessarily a problem because anonymous classes *work* just fine, they're just a pain when a user has to look at them, because the class name is opaque and might even change across runs of the program.
A rule of thumb would be to specifically name any classes that you produce from your API, exceptions included.
This lets you write documentation, and gives some concrete code for the user to debug, instrument, tweak, etc.

Object + Role

- creates anonymous subclass
- adds the role to the subclass
- reblesses the object into that subclass
- avoids polluting the object's original class

When you add a role to a specific object, the specific operations that occur are: Moose creates an anonymous subclass, adding that role into it. Then Moose reblesses your object into that anonymous subclass. Now your object, and that object alone, has that role. Without that anonymous subclass + rebless, we wouldn't be able to add roles to individual objects without polluting all other objects of that class.

Object + Role

```
apply_all_roles($obj, ‘Role::Debug’)
    if $obj->name eq ‘Stevan’;
```

One way this is particularly useful is for debugging. Say you wanted to to instrument some code, but only for one particular instance of your class. We could hack up the class of the object, but that might be difficult (involving editing shared code) if the object is not directly part of your application’s classes. We’d also have to litter the class’s code with both checks to see if it’s the desired object and the logging itself. If we’re instrumenting more than one method this quickly becomes a hassle.

Object + Role

```
package Role::Debug;
use Moose::Role;
around do_work => sub {
    my $orig = shift;
    warn “-> do_work(@_)”;
    my $ret = $orig->(@_);
    warn “<- do_work(@_) [$ret]”;
    return $ret;
};
```

Here’s what that Role::Debug role might look like. It wraps the do_work method and adds some instrumentation to it. Note how we don’t need to check $self to make sure it’s the object we want -- we only have to do that once, when we apply the role.

```
package Role::Debug;
use MooseX::Role::Parameterized;

parameter spy_on => (
    isa      => ‘ArrayRef[Str]’,
    required => 1,
);

role {
    for my $method (@{ shift->spy_on }) {
        around $method => sub {
            ...
        };
    }
};
```

If we wanted to make that a little bit more generic we could make Role::Debug a parameterized role. It takes a list of method names to instrument.

```
if ($obj->name eq ‘Stevan’) {
    apply_all_roles(
        $obj,
        ‘Role::Debug’ => {
            spy_on => [
                ‘do_work’,
                ‘eat’,
            ],
        },
    );
}
```

And now to complete the circle we use this new role to spy on the do_work and eat methods but only for Stevan objects.

For more discussion of this idea please see the method modifiers talk.

Reuse

- inheritance unfit for code reuse
- multiple inheritance doubly so
- roles: abstract unit of behavior

So, to wrap up, I think I've shown that inheritance and especially multiple inheritance is unfit for code reuse. Roles are a better answer for that, because they're abstract units of behavior.

Role Features

- composition (combination)
- conflict detection and resolution
- methods, modifiers, attributes
- method requirements
- horizontal reuse
- runtime role application

Roles have a number of useful features including the ability to combine to form new roles.
Unlike multiple inheritance, roles can tell you about accidental method name conflicts - at compile time - and gives you tools like alias and excludes to resolve them.
Roles support methods, modifiers, and attributes. Roles can also require methods of their consumer, which they should do when they call or wrap methods they expect to be there.
And all these features work especially well because roles can be used all across your class hierarchy without polluting your classes with unnecessary inheritance.
And finally you can apply roles at runtime to classes or individual objects to generate classes or add behavior a-la-carte.

Role Patterns

- tag role
- interface role
- behavior role
- plugin role
- abstract role
- ~~poor man's~~ semi-parameterized roles
- parameterized roles

Tag roles let you declare facts about your classes.
Interface roles require a set of methods of your classes so you can name, document, and delegate that set of methods.
Behavior roles are the engine of reuse by adding useful methods and attributes to any class in your hierarchy.
Plugin roles is a pattern from Dist::Zilla that works very well for selecting a subset of your plugins to perform some operation.
Abstract base classes don't work very well but by using role features like "requires" you can get something that works even better.
Finally we discussed semi-parameterized roles using "requires", especially on builders, and real parameterized roles which, by virtue of generating roles at runtime, can let you parameterize any aspect of your role, and even operate as declarative class transformation macros.

See Also

- Traits paper
  - http://scg.unibe.ch/research/traits
  - Smalltalk collection class refactoring
- `KiokuDB`
- `Dist::Zilla`
- `Fey`
- `Reflex`

If you've fallen in love with roles and want to learn more, I highly recommend you read the famous/infamous Traits paper. This is a PhD thesis from the team that developed the traits idea.
In particular I recommend reading the Smalltalk collection class refactoring for a real-world example of refactoring a complex, gross set of collection classes implemented with inheritance into a fine-tuned, exact-fit traits-based system.
Finally I encourage you to read the source code of several modules that we discussed today, and some we didn't. KiokuDB, Dist::Zilla, Fey, and Reflex all use roles very effectively to get their jobs done.

slides@

@sartak