Table of Contents

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Chapter 5. Object Hacks

Hacks 43-50

Perl has objects, you bet! Beyond the oddity of `bless`, the repurposing of subroutines, packages, and references, OO Perl has a lot of power and tremendous flexibility. Maybe you've only blessed hash references because you need record objects—but have you considered the benefits of stronger encapsulation, automatic serialization, and enforced access control?

The more you know about Perl, the more options you have for creating and using higher-level abstractions. The next time your coworkers have a nasty problem they just can't solve, look in your bag of OO tricks and smile and say, "Don't worry. We can do anything with Perl."

Hack 43. Turn Your Objects Inside Out

Encapsulate your attributes strongly.

Perl 5's object orientation is minimalistic. It gives you enough to get the job done while not preventing you from doing clever things. Of course, the default approach is usually the simplest one (or the cleverest), not the cleanest or most maintainable.

Most objects are blessed hashes, because they're easy to understand and to use. Unfortunately, they can be difficult to debug and they don't really provide any encapsulation, thus tying you to specific implementation schemes.[1]

[1] See "Seven Sins of Perl OO Programming" in *The Perl Review* 2.1, Winter 2005.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Fortunately, fixing that is easy.

The Hack

An object in Perl needs two things, a place to store its instance data and a class in which to find its methods. A blessed hash (or array, or scalar, or subroutine, or typeglob, or...) stores its data *within* the object you pass around. If you dereference the reference, you can read and write that data from anywhere, even outside the class.

An inside out object stores its data elsewhere, often in a lexical variable scoped to the class. From outside the lexical scope, you can't (*usually*—see "Peek Inside Closures" [Hack #76]) access that data without using the object's accessors.

Damian Conway's first book, *Object Oriented Perl* (Manning, 2000) showed various ways to use closure-based encapsulation. His recent *Perl Best Practices* (O'Reilly, 2005) recommended using them as a best practice. The Perl hacker known simply as Abigail has also touted the virtues of inside out objects for several years. See the documentation of `Class::Std` for a fuller treatment of the issue.

Running the Hack

A simple, naïve inside out object implementation for a record class might be:

```
# create a new scope for the lexicals
{
    package InsideOut::User;

    use Scalar::Util 'refaddr';

    # lexicals used to hold instance data
    my %names;
    my %addresses;

    sub new
    {
        my ($class, $data) = @_;

        # bless a new scalar to get this object's id
        bless \\(my $self), $class;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
        # store the instance data
        my $id              = refaddr( $self );
        $names{     $id }   = $data->{name};
        $addresses{ $id }   = $data->{address};

        return $self;
    }

    # accessors, as $self->{name} and $self->{address} don't work
    sub get_name
    {
        my $self = shift;
        return $names{ refaddr( $self ) };
    }

    sub get_address
    {
        my $self = shift;
        return $addresses{ refaddr( $self ) };
    }

    # many people forget this part
    sub DESTROY
    {
        my $self = shift;
        my $id   = refaddr( $self );
        delete $names{     $id };
        delete $addresses{ $id };
    }
}

1;
```

That's a little more typing, but it's definitely a lot cleaner. Now you can subclass or reimplement `InsideOut::User` without having to use a blessed hash—just follow the interface this defines and your code will work.

Of course, the more complex the object, the more typing you have to do. Wouldn't it be nice to automate this?

Hacking the Hack

`Class::Std`, `Class::InsideOut`, and `Object::InsideOut` are three current modules on the CPAN that take some of the work out of inside out objects for you. They all have various tricks and features. `Class::Std` is nice in that it automatically creates accessors and mutators, calls better constructors and destructors, and uses a declarative attribute-based syntax [Hack #45].

The same class using `Class::Std` is:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
{
    package InsideOut::User;

    use Class::Std;

    my %names     :ATTR( :get<name>    :init_arg<name>    );
    my %addresses :ATTR( :get<address> :init_arg<address> );
}
```

This code automatically generates the `get_name( )` and `get_address( )` accessors as well as a constructor that pulls the initial values for the objects out of a hash reference with the appropriate keys. The syntax isn't *quite* as nice as that of Perl 6, but it's much, much shorter than the naïve Perl 5 version—and provides all of the same features.

Hack 44. Serialize Objects (Mostly) for Free

Store object data without mess, confusion, or big blobs of binary data.

Some programs really need persistent data, and sometimes mapping between objects and multiple tables in a fully-relational database is just too much work. This is especially true in cases where being able to edit data quickly and easily is important—there's no interface more comfortable than your favorite text editor [Hack #12].

Instead of hard-coding configuration in a program, wasting your precious youth creating the perfect database schema, or doing XML sit-ups, why not serialize your important object data to YAML?

The Hack

If you use hash-based objects, it's very easy to serialize the data—just make a copy of the hash and serialize it:

```
use YAML 'DumpFile';

sub serialize
{
    my ($object, $file) = @_;
    my %data            = %$object;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    DumpFile( $file, \\%data );
}
```

This assumes, of course, that `$object` is the object you want to serialize and `$file` is the path and file to which to save the object.

If you use inside out objects [Hack #43], you have a bit more work to do:

```
package Graphics::Drawable;
{
    use Class::Std;

    my %coords_of     :ATTR( :get<coords>   :init_arg<coords>   );
    my %velocities_of :ATTR( :get<velocity> :init_arg<velocity> );
    my %shapes_of     :ATTR( :get<shape>    :init_arg<shape>    );

    sub get_serializable_data
    {
        my $self  = shift;

        my %data;

        for my $attribute (qw( coords velocity shape ))
        {
            my $method = 'get_' . $attribute;
            $data{ $attribute } = $self->$method( );
        }

        return \\%data;
    }
}
```

Now your `serialize( )` function can avoid breaking encapsulation and call `get_serializable_data( )` instead. An object at the origin (coordinates of `(0, 0, 0)`) with a velocity of one unit per time unit along the X axis (`(1, 0, 0)`) and a `Circle` shape serializes to:

```
---
coords:
  - 0
  - 0
  - 0
shape: Circle
velocity:
  - 1
  - 0
  - 0
```

If you want to make more objects, copy the file to a new location and modify it. Just be careful to keep the code valid YAML.[2]

[2] A task much easier than writing valid XML by hand...

Restoring objects is easy; just use `YAML`'s `LoadFile( )` method:

```
use YAML 'LoadFile';

sub deserialize
{
    my ($class, $file) = @_;
    my $data           = LoadFile( $file );
    return $class->new( $data );
}
```

Assuming your class constructor takes a hash reference keyed on attribute names (as `Class::Std` does), you're all set. Of course, this all presumes some sort of object factory that can manage instances, map files and paths to classes, and store and retrieve objects, let alone handle errors. `Class::StorageFactory` on the CPAN handles this.

If you have all of this—and only need data from an object's public interface (both constructor attributes and data accessible through accessors) to recreate the object—serializing to YAML or another simple plain-text format (JSON?) is fast, easy, flexible, and almost free.

Hack 45. Add Information with Attributes

Give your variables and subroutines a little extra information.

Subroutines and variables are straighforward. Sure, you can pass around references to them or make them anonymous and do weird things with them, but you have few options to change what Perl does with them.

Your best option is to give them attributes. Attributes are little pieces of data that attach to variables or subroutines. In return, Perl runs any code you like. This has many, many possibilities.

The Hack

Suppose that you have a class and want to document the purpose of each method. Some languages support *docstrings*—comments that you can introspect by calling class methods.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Perl's comments are pretty boring, but you can achieve almost the same effect by annotating methods with subroutine attributes.

Consider a `Counter` class, intended to provide a default constructor that counts the number of objects created. If there's a `Doc` attribute provided by the `Attribute::Docstring` module, the class may resemble:

```
package Counter;

use strict;
use warnings;

use Attribute::Docstring;

our $counter :Doc( 'a count of all new Foo objects' );

sub new :Doc( 'the constructor for Foo' )
{
    $counter++;
    bless { }, shift;
}

sub get_count :Doc( 'returns the count for all foo objects' )
{
    return $counter;
}

1;
```

The prototype comes after the name of the subroutine and has a preceding colon. Otherwise, it looks like a function call. The documentation string is the (single) argument to the attribute.

Running the Hack

The easiest way to create and use attributes is with the `Attribute::Handlers` module. This allows you to write subroutines named after the attributes you want to declare. The implementation of `Attribute::Docstring` is:

```
package Attribute::Docstring;

use strict;
use warnings;

use Scalar::Util 'blessed';
use Attribute::Handlers;

my %doc;

sub UNIVERSAL::Doc :ATTR
{
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
    return if $symbol eq 'LEXICAL';

    my $name                    = *{$symbol}{NAME};
    $doc{ $package }{ $name } = $data;
}

sub UNIVERSAL::doc
{
    my ($self, $name) = @_;
    my $package       = blessed( $self ) || $self;

    return unless exists $doc{ $package }{ $name };
    return               $doc{ $package }{ $name };
}

1;
```

To make the `Doc` attribute available everywhere, the module defines a subroutine called `UNIVERSAL::Doc`. This subroutine itself has an attribute, `:ATTR`, which identifies it as an attribute handler.

For any subroutine or variable that declares a `Doc` attribute, the subroutine receives several pieces of information. Here, the important ones are the package containing the subroutine, the symbol—from which the typeglob access can retrieve the name, and the data provided to the attribute. In the `Counter` class, the attribute handler receives a package name of `Counter` and the typeglob with the name `new` for the symbol when Perl finishes compiling the `new( )` method. It then stores the attribute data (the docstring itself) in a hash keyed first on the name of the package and then on the name of the symbol.

Because of the difference between how Perl treats lexical and global variables, the handler can't do much if it receives a lexical symbol (that is, when `$symbol` is `LEXICAL`). Then again, these are private to the package so they're not worth documenting in this way anyway.

The similarly named `doc( )` method works on any class or object, so that calling `Counter->doc( 'new' )` or `$counter->doc( 'get_count' )` both return the docstring for the appropriate method. It simply looks up the docstring in the appropriate package for the given name and returns it.

Hacking the Hack

One potential enhancement is to add the appropriate sigil to the name, so that the docstrings for a variable named `$count` and a method named `count( )` will not overwrite each other. That would require a change to `UNIVERSAL::doc( )` so that `$name` contains the sigil (or, with no sigil, defaults to the method).

Another possibility is to take `UNIVERSAL::Doc( )` out of `UNIVERSAL`, instead importing it into any package that uses this module. That unclutters `UNIVERSAL` somewhat at the expense of cluttering calling classes. That may or may not be a useful tradeoff.

Attributes may span lines, but you cannot use heredocs, unfortunately.

Hack 46. Make Methods Really Private

Enforce encapsulation with a little more flair.

Perl's object orientation is powerful in many ways, allowing the creation and emulation of almost any kind of object or class system. It's also very permissive, enforcing no access control by default. Any code can poke and prod methods and parents into any class at any time and can call even ostensibly private methods regardless of the intent of the code's original author.

By convention, the Perl community considers methods with a leading underscore as private methods that you shouldn't override or call outside of the class or rely on any specific semantics or workings. That's usually a good policy, but there's little enforcement and it's only a convention. It's still possible to call the wrong method accidentally or even on purpose.

Fortunately, there are better (or at least scarier) ways to hide methods.

The Hack

One easy way to manipulate subroutines and methods at compile time is with subroutine attributes [Hack #45]. The `Class::HideMethods` module adds an attribute to methods named `Hide` that makes them unavailable and mostly uncallable from outside the program:

```
package Class::HideMethods;

use strict;
use warnings;
use Attribute::Handlers;

my %prefixes;

sub import
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
{
    my ($self, $ref)      = @_;
    my $package           = caller( );
    $prefixes{ $package } = $ref;
}

sub gen_prefix
{
    my $invalid_chars = "\\0\\r\\n\\f\\b";

    my $prefix;

    for ( 1 .. 5 )
    {
        my $char_pos = int( rand( length( $invalid_chars ) ) );
        $prefix     .= substr( $invalid_chars, $char_pos, 1 );
    }

    return $prefix;
}

package UNIVERSAL;

sub Private :ATTR
{
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;

    my $name    = *{ $symbol }{NAME};
    my $newname = Class::HideMethods::gen_prefix( $package ) . $name;
    my @refs    = map { *$symbol{ $_ } } qw( HASH SCALAR ARRAY GLOB );
    *$symbol    = do { local *symbol };

    no strict 'refs';
    *{ $package . '::' . $newname } = $referent;
    *{ $package . '::' . $name    } = $_ for @refs;
    $prefixes{ $package }{ $name }  = $newname;
}

1;
```

To hide the method, the code replaces the method's symbol with a new, empty typeglob. This would also delete any variables with the same name, so the code copies them out of the symbol first, and then back into the new, empty symbol. Now you know how to "delete" from a typeglob.

Running the Hack

Using this module is easy; within your class, declare a lexical hash to hold the secret new method names. Pass it to the line that `use`s `Class::HideMethods`:

```
package SecretClass;

my %methods;
use Class::HideMethods \\%methods;

sub new            { bless { }, shift }
sub hello :Private { return 'hello'   }
sub goodbye        { return 'goodbye' }

sub public_hello
{
    my $self  = shift;
    my $hello = $methods{hello};
    $self->$hello( );
}

1;
```

Remember to call all private methods with the `$invocant->$method_name` syntax, looking up the hidden method name instead.

To prove that it works, try a few tests from outside the code.

```
use Test::More tests => 6;

my $sc = SecretClass->new( );
isa_ok( $sc, 'SecretClass' );

ok( ! $sc->can( 'hello' ),        'hello( ) should be hidden'              );
ok( $sc->can( 'public_hello' ),   'public_hello( ) should be available'    );
is( $sc->public_hello( ),
    'hello', '... and should be able to call hello( )' );
ok( $sc->can( 'goodbye' ),        'goodbye( ) should be available'         );
is( $sc->goodbye( ), 'goodbye',    '... and should be callable'            );
```

Not even subclasses can call the methods directly. They're fairly private!

Inside the hack

Perl uses symbol tables internally to store everything with a name—variables, subroutines, methods, classes, and packages. This is for the benefit of humans. By one theory, Perl doesn't really care what the name of a method is; it's happy to call it by name, by reference, or by loose description.

That's sort of true and sort of false. Only Perl's *parser* cares about names. Valid identifiers start with an alphabetic character or an underscore and contain zero or more alphanumeric or underscore characters. Once Perl has parsed the program, it looks up whatever symbols it

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

has in a manner similar to looking up values in a hash. If you can force Perl to look up a symbol containing otherwise-invalid characters, it will happily do so.

Fortunately, there's more than one way to call a method. If you have a scalar containing the name of the method (which you can define as a string containing any character, not just a valid identifier) or a reference to the method itself, Perl will invoke the method on the invocant. That's half of the trick.

The other magic is in removing the symbol from the symbol table under its unhidden name. Without this, users could bypass the hidden name and call supposedly hidden methods directly.

Without the real name being visible, the class itself needs some way to find the names of private methods. That's the purpose of the lexical `%methods`, which is not normally visible outside of the class itself (or at least its containing file).

Hacking the Hack

A very clever version of this code could even do away with the need for `%methods` in the class with hidden methods, perhaps by abusing the `constant` pragma to store method names appropriately.

This approach isn't *complete* access control, at least in the sense that the language can enforce it. It's still possible to get around this. For example, you can crawl a package's symbol table, looking for defined code. One way to thwart this is to skip installing methods back in the symbol table with mangled names. Instead, delete the method from the symbol table and store the reference in the lexical cache of methods.

That'll keep out determined people. It won't keep out *really* determined people who know that the `PadWalker` module from the CPAN lets them poke around in lexical variables outside their normal scope [Hack #76]...but anyone who wants to go to that much trouble could just as easily fake the loading of `Class::HideMethods` with something that doesn't delete the symbol for hidden methods. Still, it's really difficult to call these methods by accident or on purpose without some head-scratching, which is probably as good as it gets in Perl 5.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hack 47. Autodeclare Method Arguments

You know who you are. Stop repeating your `$self`.

Perl's object orientation is very flexible, in part because of its simplicity and minimalism. At times that's valuable: it allows hackers to build complex object systems from a few small features. The rest of the time it can be painful to do simple things.

Though not everyone always calls the invocant in methods `$self`, everyone has to declare and manage the invocant and other arguments. That's a bit of a drag—but it's fixable. Sure, you could use a full-blown source filter [Hack #94] to remove the need to shift off `$self` and process the rest of your argument list, but that's an unnecessarily large hammer to swing at such a small annoyance. There's another way.

The Hack

Solving this problem without source filters requires three ideas. First, there must be some way to mark a subroutine as a method, because not all subroutines *are* methods. Second, this should be compatible with `strict`, for good programming practices. Third, there should be some way to add the proper operations to populate `$self` and the other arguments.

The first is easy: how about a subroutine attribute [Hack #45] called `Method`? The third is also possible with a little bit of B::Deparse [Hack #56] and `eval` magic. The second is trickier....

A surprisingly short module can do all of this:

```
package Attribute::Method;

use strict;
use warnings;

use B::Deparse;
use Attribute::Handlers;

my $deparse = B::Deparse->new( );

sub import
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
{
    my ( $class, @vars ) = @_;
    my $package          = caller( );

    my %references       =
    (
        '$' => \\undef,
        '@' => [ ],
        '%' => { },
    );

    push @vars, '$self';

    for my $var (@vars)
    {
        my $reftype                 = substr( $var, 0, 1, '' );

        no strict 'refs';
        *{ $package . '::' . $var } = $references{$reftype};
    }
}

sub UNIVERSAL::Method :ATTR(RAWDATA)
{
    my ($package, $symbol, $referent, undef, $arglist) = @_;

    my $code                = $deparse->coderef2text( $referent );
    $code                   =~ s/{/sub {\\nmy (\\$self, $arglist) = \\@_;\\n/;

    no warnings 'redefine';
    *$symbol                = eval "package $package; $code";
}

1;
```

All of the variables, including `$self`, have to be lexical within methods, lest bad things happen when calling one method from another, such as accidentally overwriting a global variable somewhere. The handler for the `Method` attribute takes the compiled code, deparses it, and inserts the `sub` keyword and the argument handling line before the rest of the code. All of the arguments to the attribute are the names of the lexical variables within the method.

Compiling that with `eval` produces a new anonymous subroutine, which the code then inserts into the symbol table after disabling the `Subroutine %s redefined` warnings.

Running the Hack

From any class in which you tire of declaring and fetching the same arguments over and over again, write instead:

```
package Easy::Class;

use strict;
use warnings;

use Attribute::Method qw( $status );

sub new :Method
{
    bless { @_ }, $self;
}

sub set_status :Method( $status )
{
    $self->{status} = $status;
}

sub get_status :Method
{
    return $self->{status};
}

1;
```

For every method marked with the `:Method` attribute, you get the `$self` invocant declared for free. For every method with that attribute parameterized with a list of variable names, you get those variables as well.

Notice the strange and deep magic in `import( )` as well as the list of arguments passed to it; this is what bypasses the `strict` checking. If you use instead only the `refs` and `subs` strictures, you don't even have to pass the variables you want to `Attribute::Method`.

Hacking the Hack

Is this better than source filters? It's certainly not as syntactically tidy. On the other hand, attribute-based solutions are often less fragile than source filtering. In particular, they don't prevent the use of other source filters or other attributes. It also almost never fails—if your subroutines have errors, Perl will report them when compiling from the point of view of the original code before even calling the attribute handler. This technique works best in classes with several methods that take the same arguments.

Another *possible* way to accomplish this task is to rewrite the optree of the code reference (with `B::Generate` and a *lot* of patience) to add the ops to assign the arguments to the proper variables. Of course, you'll also have to insert the lexical variables into the pad associated with the CV, but if you know what this means, you probably know how to do it.

Finding and fixing any lexicals that methods close over isn't as bad in comparison. See "Peek Inside Closures" [Hack #76].

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

See Ricardo Signes's `Sub::MicroSig` for an alternate approach to the same problem.

Hack 48. Control Access to Remote Objects

Enforce access control to your objects.

Perl's idea of access control and privacy is politeness. Sometimes this is useful—you don't have to spend a lot of time and energy figuring out what to hide and how. Sometimes you *need* to rifle through someone else's code to get your job done quickly.

Other times, security is more important than ease of coding—especially when you have to deal with the cold, hostile world at large. Though you may need to make your code accessible to the wilds of the Internet, you don't want to let just anyone do anything.

Modules and frameworks such as `SOAP::Lite` make it easy to provide web service access to plain old Perl objects. Here's one way to make them somewhat safer.

The Hack

First, decide what kinds of operations you need to support on your object. Take a standard web-enabled inventory system. You need to fetch an item, insert an item, update an item, and delete an item. Then identify the types of access: creating, reading, writing, and deleting.

You *could* maintain a list in code or a configuration file somewhere mapping all the access controls to all the methods of the objects in your system. That would be silly, though; this is Perl! Instead, consider using a subroutine attribute [Hack #45].

```
package Proxy::AccessControl;

use strict;
```

```
use warnings;

use Attribute::Handlers;

my %perms;

sub UNIVERSAL::perms
{
    my ($package, $symbol, $referent, $attr, $data) = @_;
    my $method                                      = *{ $symbol }{NAME};

    for my $permission (split(/\\s+/, $data))
    {
        push @{ $perms{ $package }{ $method } }, $permission;
    }
}

sub dispatch
{
    my ($user, $class, $method, @args) = @_;

    return unless $perms{ $class }{ $method } and $class->can( $method );

    for my $perm (@{ $perms{ $class }{ $method } })
    {
      die "Need permission '$perm\\n'" unless $user->has_permission( $perm );
    }

    $class->$method( @args );
}

1;
```

Declaring permissions is easy:

```
package Inventory;

use Proxy::AccessControl;

sub insert :perms( 'create' )
{
    my ($self, $attributes) = @_;
    # ...
}

sub delete :perms( 'delete' )
{
    my ($self, $id) = @_;
    # ...
}

sub update :perms( 'write' )
{
    my ($self, $id, $attributes) = @_;
    # ...
}

sub fetch :perms( 'read' )
{
```

```
    my ($self, $id) = @_;
    # ...
}
```

You can also mix and match permissions:

```
sub clone :perms( 'read create' )
{
    my ($self, $id, $attributes) = @_;
    # ...
}
```

`Proxy::AccessControl` provides an attribute handler `perms` that registers a space-separated list of permissions for each marked method. It also provides a `dispatch( )` method—the barrier point into the system between the controller routing incoming requests and the actual Perl objects handling the requests.

The only thing left to do (besides actually writing the business logic code) is to make your controller run everything through `Proxy::AccessControl::dispatch( )`. This function takes three parameters. The first is a `$user` object that represents the access capabilities of the external user somehow. (Your code needs to allow authentication and creation of this object.) The `$class` and `$method` parameters identify the proper class and method to call, if the user has permission to do so.

Hacking the Hack

`dispatch( )` is a coarsely-grained approach to proxying. Perhaps creating dedicated proxies that speak web services or remote object protocols natively would be useful. Behind the scenes, they could take only one extra parameter (the user object) and, for each proxied method, provide their own implementation that performs the access checks before delegating or denying the request as necessary.

There's no reason to limit access control to permissions alone, either. You could control access to objects based on the number of concurrent accesses, the phase of the moon, the remote operating system, the time of day, or whatever mechanism you desire. Anything you can put in an attribute's data is fair game.

Hack 49. Make Your Objects Truly Polymorphic

Build classes based on what they do, not how they inherit.

Many tutorials and books declare confidently that inheritance is a central feature of object-oriented programming.

They're wrong.

Polymorphism is much, much more important. It matters that when you call `log( )` on an object that knows how to log its internal state it does so, not that it inherits from some abstract `Logger` class somewhere or that it calculates a natural log. Perl 6 encourages this type of design with roles. In Perl 5, you can either build it yourself or use `Class::Trait` to decompose complex operations into natural, named groups of methods.

That sounds awfully abstract—but if you have a complex problem you *can* decompose appropriately, you can write just a little bit of code and accomplish quite a bit.

The Hack

Imagine that you're building an application with properly abstracted model, view, and controller. You have multiple output types—standard XHTML, cut-down-XHTML for mobile devices, and Ajax or JSON output for RESTful web services and user interface goodness.

Every possible view has a corresponding view class. So far the design makes sense. Yet as your code handles an incoming request and decides what to do with it, how do you decide which view to use? Worse, if you have multiple views, how do you build the appropriate classes without going crazy for all of the combinations?

If you cheat a little bit and declare your views as traits, you can apply them to the model objects and render the data appropriately.

Here's an example model from which the concrete `Uncle` and `Nephew` classes both inherit:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
package Model;

sub new
{
    my ($class, %args) = @_;
    bless \\%args, $class;
}

sub get_data
{
    my $self = shift;
    my %data = map { $_ => $self->{$_} } qw( name occupation age );
    return \\%data;
}

1;
```

The views are pretty simple, too:

```
package View;

use Class::Trait 'base';

package TextView;

use base 'View';

sub render
{
    my $self = shift;
    printf( "My name is %s.  I am an %s and I am %d years old.\\n",
        @{ $self->get_data( ) }{qw( name occupation age )} );
}

package YAMLView;

use YAML;
use base 'View';

sub render
{
    my $self = shift;
    print Dump $self->get_data( );
}

1;
```

The text view displays a nicely formatted English string, while the YAML view spits out a serialized version of the data structure. Now all the controller class has to do is to create the appropriate model object and apply the appropriate view to it before calling `render( )`:

```
# use model and view classes

# create the appropriate model objects
my $uncle  = Uncle->new(
    name => 'Bob', occupation => 'Uncle', age => 50
);
```

```
my $nephew = Nephew->new(
    name => 'Jacob', occupation => 'Agent of Chaos', age => 3
);

# apply the appropriate views
Class::Trait->apply( $uncle,  'TextView' );
Class::Trait->apply( $nephew, 'YAMLView' );

# display the results
$uncle->render( );
$nephew->render( );
```

Running the Hack

The code produces:

```
My name is Bob.  I am an Uncle and I am 50 years old.
---
age: 3
name: Jacob
occupation: Agent of Chaos
```

Hacking the Hack

If that were all that traits and roles are, that would still be useful. There's more though! `Class::Trait` also provides a `does( )` method which you can use to query the capabilities of an object. If you could possibly receive an object that already has a built-in view (a debugging model, for example), call `does` to see if it does already do a view:

```
Class::Trait->apply( $uncle, $view_type ) unless $uncle->does( 'View' );
```

You also don't have to have your traits inherit from a base trait. If all of the code that uses objects and classes with traits checks `does( )` instead of Perl's `isa( )` method, you can have traits that do the right thing without having any relationship, code- or inheritance-wise, with any other traits.

This is especially useful for working with proxied, logged, or tested models and views.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hack 50. Autogenerate Your Accessors

Stop writing accessor methods by hand.

One of the Perl virtues is laziness. This doesn't mean not doing your work, it means doing your work with as little effort as possible. When you find yourself typing the same code over and over again, stop! Make the computer do the work.

Method accessors/mutators (getters/setters) are a case in point. Here's a simple object-oriented module:

```
package My::Customer;

use strict;
use warnings;

sub new { bless { }, shift }

sub first_name
{
    my $self            = shift;
    return $self->{first_name} unless @_;
    $self->{first_name} = shift;
    return $self;
}

sub last_name
{
    my $self           = shift;
    return $self->{last_name} unless @_;
    $self->{last_name} = shift;
    return $self;
}

sub full_name
{
    my $self = shift;
    return join ' ', $self->first_name( ), $self->last_name( );
}

1;
```

and a small program to use it:

```
my $cust = My::Customer->new( );
$cust->first_name( 'John' );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
$cust->last_name( 'Public' );
print $cust->full_name( );
```

That prints `John Public`.

Of course, if this is really is a customer object, it needs to do more. You might need to set a customer's credit rating, the identity of a primary salesperson, and so on.

As you can see, the `first_name` and `last_name` methods are effectively duplicates of one another. New accessors are likely to be the very similar. Can you automate this?

The Hack

There are many modules on the CPAN which handle this, all in slightly different flavors. Here are two—one of the most widespread and one of the least constraining.

Class::MethodMaker

One of the oldest such module is `Class::MethodMaker`, originally released in 1996. It is very feature rich, and although the documentation can seem a bit daunting, the module itself is very easy to use. To convert the `My::Customer` code, write:

```
package My::Customer;

use strict;
use warnings;

use Class::MethodMaker[
            new    => [qw( new )],
            scalar => [qw( first_name last_name )],];

sub full_name
{
    my $self = shift;
    return join ' ', $self->first_name( ), $self->last_name( );
}
```

The constructor is very straightforward, but what's up with `first_name` and `last_name`? The arguments passed to `Class::MethodMaker` cause it to create two getter/setters which contain scalar values. However, even though this code appears to behave identically, it's actually much more powerful.

Do you want to check that no one ever set an object's `first_name`, as opposed to having set it to an undefined value?

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
print $cust->first_name_isset( ) ? 'true' : 'false';
```

Even if you set `first_name` to `undef`, `first_name_isset( )` will return true. Of course, sometimes you will want to be unset, even after you've set it. That works, too:

```
$cust->first_name( 'Ozymandias' );
print $cust->first_name_isset( ) ? 'true' : 'false'; # true
$cust->first_name_reset( );
print $cust->first_name_isset( ) ? 'true' : 'false'; # false
```

Class::BuildMethods

`Class::MethodMaker` also has built-in support for arrays, hashes, and many other useful features. However, it requires you use a blessed hash for your objects. In fact, most of the accessor builder modules on the CPAN make assumptions about your object's internals. One exception to this is `Class::BuildMethods`.

`Class::BuildMethods` allows you to build accessors for your class regardless of whether it's a blessed hash, arrayref, regular expression, or whatever. It does this by borrowing a trick from inside out objects [Hack #43]. Typical code looks like:

```
package My::Customer;

use strict;
use warnings;

use Class::BuildMethods qw(
              first_name
              last_name
            );

# Note that you can use an array reference, if you prefer
sub new { bless [ ], shift }

sub full_name
{
    my $self = shift;
    return join ' ', $self->first_name( ), $self->last_name( );
}

1;
```

Use this class just like any other. Internally it indexes the accessor values by the address of the object. It handles object destruction for you by default, but allows you to handle this manually if you need special behavior on `DESTROY` (such as releasing locks).

`Class::BuildMethods` is very simple, by design. Like most other accessor generators, it provides some convenience features, but only in the form of default values and data validation:

```
use Class::BuildMethods
  'name',
  gender => { default  => 'male' },
  age    => { validate => sub
  {
      my ($self, $age) = @_;
      carp 'You can't enlist if you're a minor'
          if ( $age < 18 && ! $self->is_emancipated( ) );
  }};
```

With this code, `gender( )` will return `male` unless you set it to some other value. `age( )` shows how to provide flexible validation. Because the `validate( )` method points to a subroutine reference rather than providing special validation handlers, the author's assumptions of how you should validate your code don't constrain you.

`Class::BuildMethods` always assumes that a setter takes a single value, so you must pass references for arrays and hashes. It also does not provide class methods (a.k.a. static methods). These limitations may mean this code doesn't fit your needs, but this module was designed to be simple. You can read and understand the docs in one sitting.

Running the Hack

Accessor generation, when it fits your needs, can remove a tremendous amount of grunt work. This is hacking your brain. As a Perl programmer, true laziness means that you waste less time on the fiddly bits so you can spend more time worrying about the hard stuff.