Redispatch in Perl
Part I — NEXT.pm

Paul Fenwick
<pjf@perltraining.com.au>

13th February, 2001

Goals of this Presentation

The goal of this presentation is to demonstrate the use of Perl's redispatch mechanisms to solve some common and not-so-common programming problems.

A basic knowledge of object oriented Perl is assumed.

Dispatch

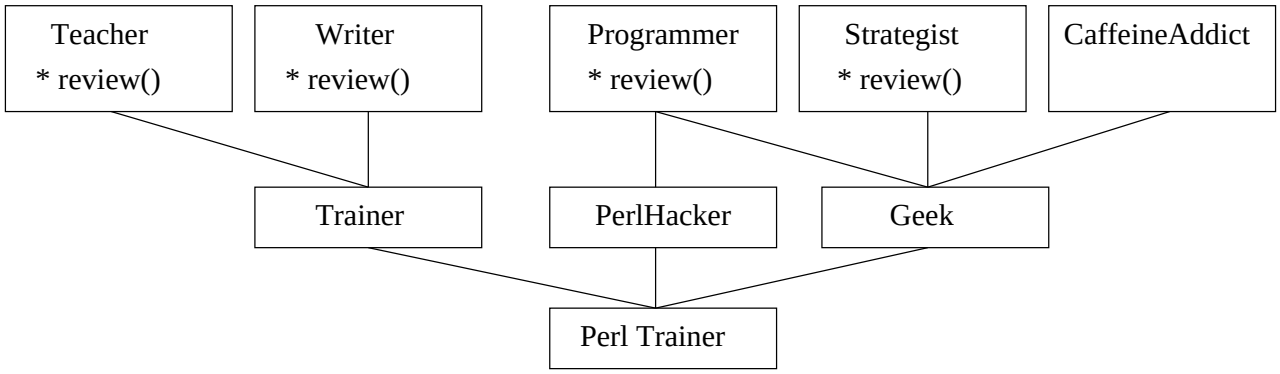
- Dispatch is the process where an appropriate subroutine is found and executed in response to a method call on an object or class.

- Involves searching a class' inheritance tree, `UNIVERSAL` subroutines, and `AUTOLOAD`ed methods.

- Part of Perl's object oriented features.

- Standard dispatch uses the "leftmost ancestor wins" rules.

Dispatch Example



In this example, a call like `$trainer->review` ends up being directed to `Teacher::review`.

Redispatch and Multiple Dispatch

- Redispatch is the process where a method call is dispatched a second or subsequent time, possibly using an alternate dispatch mechanism.

- Redispatch has limited usefulness in single-inheritance languages (such as Java), but many applications in multiply-inherited languages such as Perl.

- Multiple Dispatch is when a method dispatched depends not only upon its invoking object, but also upon the arguments of the method call.

- Multiple Dispatch is covered more in the sequel to this talk.

Constructors and Inheritance — a common problem

One problem that is worth of discussion is the correct way to handle the construction of objects in a class which has single or multiple inheritance.

A commonly seen approach is for a child class to call its parent's constructor and then modify the resulting object.

```
package PerlHacker;
use base 'Programmer';

sub new {
        my ($class, @args) = @_;
        my $this = Programmer->new(@args);

        # Modify $this to create a PerlHacker.

        return $this;
}
```

Problems with the constructor-only approach

- `Programmer->new(@args)` will return a `Programmer` object, not a `PerlHacker`. This requires either re-blessing the object or careful coding to avoid.

- Most importantly, this approach plain doesn't work when we're dealing with multiple inheritance. Each parent will return a (different) object, and attempting to combine them into a single object is often a very Bad Idea.

Constructors and Initialisers

A much better solution is to break up construction from initialisation.

```
package Programmer;

sub new {
        my ($class,@args) = @_;
        my $this = bless({},$class);
        $this->_init(@args);
}

sub _init {
        # Programmer's initialiser.
}

package PerlHacker;
use base 'Programmer';

sub _init {
        my ($this, @args) = @_;
        $this->Programmer::_init(@args);
        # PerlHacker's initialiser.
}
```

The advantages of a separate initialiser

- Inheritance becomes easy. `$this->_init` in `Programmer::new` calls the appropriate `_init` method from the *derived* class. In this case, `PerlHacker::_init`.

- Initialisers modify an existing object, instead of creating a new object. As such, multiple inheritance is just a matter of calling all of one's parents initialisers.

- Since constructors just create an appropriate object and call its `_init` method, there's no need for a derived class to provide a constructor.

Using initialisers correctly

Initialisers don't get us completely out of the woods. Our previous example naively has the `Programmer` class hard-coded in. This could be a problem if the class stopped inheriting from `Programmer`, or began to inherit from new classes at a later date.

We'd like to just call our parents' `_init` methods, regardless of what those parents are. Perl *almost* gives us a good way of doing this.

The SUPER pseudo-class

```
package PerlHacker;
use base 'Programmer';

sub _init {
        my ($this, @args) = @_;
        $this->SUPER::_init(@args);
        # PerlHacker's initialiser.
}
```

`SUPER` is the form of redispatch most Perl programmers are familiar with. It performs a "leftmost ancestor wins" search through all the parents of the current class.

The problem with SUPER

Unfortunately for us, `SUPER` is not without fault:

- Does not deal with multiple inheritance, due to "left-most-ancestor wins" dispatch.
- Only ever looks *up* the current inheritance tree. Will never backtrack to try other ancestors of the original class.
- Requires the method to actually exist, and throws an exception if it does not.

`SUPER` is a fine solution for single-inheritance, but fails miserably for multiple inheritance.

A better solution, walking @ISA...

```
sub _init {
my ($this, %args) = @_;

# Call parents' _init methods.
foreach my $parent (@ISA) {
        $parent_init = $parent->can("_init");

        $this->$parent_init(%args)
                if $parent_init;
        }

# Do our own init...
}
```

Problems with walking @ISA

- A multiply-inherited class needs to have a dummy `_init` methods in order to avoid the rightmost inheritance trees being skipped.

- Some `_init` methods may be called multiple times in the case of diamond inheritance.

- A lot of work to do what should be a simple thing.

The NEXT solution

```
sub _init {
        my ($this, @args) = @_;
        $this->NEXT::_init(@_);

        # My initialisation goes here...
}
```

So what does NEXT do?

- Restarts the dispatch mechanism at the current position in the inheritance tree.

- Will backtrack and follow other branches, just like regular dispatch.

- Will invoke `AUTOLOAD`, just like regular dispatch.

- Quietly does nothing if no dispatch can occur.

- `NEXT` (like `SUPER`) is called a *pseudo-class*.

Advantages of NEXT

- Simple, easy to understand and use.
- Will backtrack down and across inheritance trees. Using `NEXT` when you have no parents still does the right thing.
- No need for dummy `_init` methods.
- Quietly behaves and does nothing if there is no `NEXT::_init` to call.

Problems with NEXT during initialisation

- Only works if everyone plays by the rules.
- Multiply inherited classes may be invoked multiple times (until now...).
- Doesn't come in the standard Perl distribution (but watch out for 5.8.x).
- Breaks if `UNIVERSAL::_init` exists (but we're working on that too).

The New NEXT

- `NEXT` has recently been improved, the current version is 0.50.

- Faster, Stronger, Better.

- More functionality for greater usefulness.

Old uses for NEXT

`NEXT` has always had three obvious uses:

- Initialisation (which we've seen).
- Destructor functions (really just the reverse of initialisation).
- `AUTOLOAD`s.

The property of `NEXT` to "silently disappear" when there's nowhere to go is not always a feature, particularly in the case where a class is trying to "pass on" an `AUTOLOAD` call.

Forcing redispatch with NEXT::ACTUAL

```
package PerlHacker;

sub AUTOLOAD {
    my ($this, @args) = @_;
    unless ($AUTOLOAD =~ /::hack_\w+$/) {
        # Ooops, not for us.
        return $this->NEXT::ACTUAL::AUTOLOAD(@_);
    }
    # Hack some stuff here.
}
```

`NEXT::ACTUAL` throws an exception if there's no next method to go to.

Why it's called NEXT::ACTUAL

The new facility is accessed as:

```
$self->NEXT::ACTUAL::method();
```

indicating that there actually must be an actual next method or you'll actually get an actual exception. ;-)

— *Damian Conway, private e-mail, 16 Nov 2001*

Stopping multiple invocations using NEXT::UNSEEN

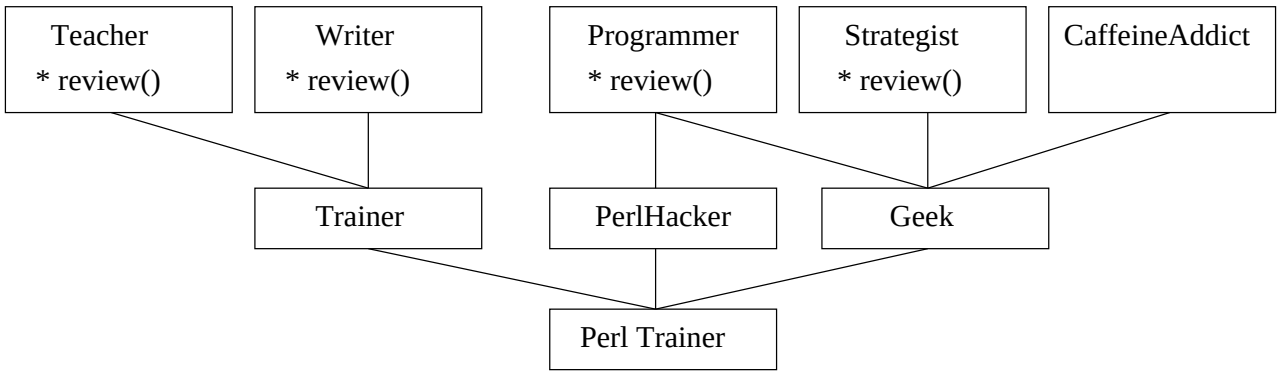
Using `NEXT::UNSEEN` instead of `NEXT` means that method-calls are skipped if they've already been made that dispatch-run.

```
sub _init {
        my ($this, @args) = @_;
        $this->NEXT::UNSEEN::_init(@_);

        # My initialisation goes here...
}
```

In case you're wondering, `NEXT::UNSEEN::ACTUAL` and `NEXT::ACTUAL::UNSEEN` both work as you would expect.

Using NEXT to do multiple dispatch



What happens if we call

```
$trainer->review(code=>"Finance/Quote.pm") ?
```

Multiple dispatch (continued)

```
$trainer->review(code=>"Finance/Quote.pm")
```

Due to Perl's "leftmost ancestor wins" dispatch, `Teacher::review` will be called. The Teacher class can review essays and exams, but not Perl modules. Obviously this line of dispatch is *not* what we intended.

However, it's possible for `Teacher::review` to "pass-on" the method call to other eligible classes. In fact, it can do this without even knowing that those other classes exist!

Multiple dispatch (continued)

```
package Teacher;

sub review {
    my $this = shift;
    my %args = @_;

    unless ($args{essay} or $args{exam}) {
            # Doesn't look like it's for us.
            return $this->NEXT::ACTUAL::review(@_)
    }

    # Review essays and exams here...
}
```

Multiple dispatch (continued)

- It's possible for each method to "pass" on method calls it doesn't understand, and catch those that it does.

- Methods can augment, transform, or otherwise process the results that come back after passing on a call.

- Possible for a group of ancestor classes to decide between themselves who handles what calls (with leftmost ancestors having first say).

The future of NEXT

- Standard module with Perl 5.8.0.
- Ability to perform redispatch on a breadth-first rather than depth-first basis.
- Eventual integration into `Class::MultiMethods`.

Summary

- A general problem exists when we wish to pass a method call to all classes in an inheritance tree.

- One good solution to this problem is the `NEXT` pseudo-class.

- `NEXT` has uses in initialisers, destructors, `AUTOLOAD`s, and multiple dispatch.

- `NEXT` will be entering the standard Perl distribution with Perl 5.8.0.

References

- **Object Oriented Perl** (training manual), Paul Fenwick. `http://perltraining.com.au/notes.html`

- **Redispatching Method Calls with NEXT**, Damian Conway. `http://use.perl.org/articles/01/12/05/1654222.shtml`