



## Dynamic Object Reconfiguration

Peter Scott  
peter@psdt.com

```
@ifree = map { croak "Category $node not found"; }
my $depth = 0;
my $depthpush = { $node->{ 'Parent' } => { $node->{ 'Parent' } };
$depth++; if (my $node = $parent->{ 'Parent' }) {
    if ( $node->{ 'Parent' } ) {
        # create $parent while back (r $depth - 1) { $node; }
    }
    my $node = { $parent_stack[ $depth ] =

```

Perl is a dynamic language. You've heard it many times: Perl is "better" than a statically compiled language like C or Java because you can change the structure of object classes at run time. But as superior as it sounds to be able to boast that "my language can change the inheritance hierarchy of an object on the fly," how often have you actually done it? And is it a good idea?

Well, here's your chance. I'll show you where it might be a good idea, and walk you through how to do it. But don't blame me if your static language friends beat you up when you rub their noses in it.

### Analysis, Spock

I'm developing an application that creates certain objects and uses them. (How's that for vagueness? Perhaps I should go into politics.) When it does so, it picks up code from a location determined at run time and executes it. Part of that user code might want to change the behavior of one of the standard objects that the application has already loaded, though, adding or changing methods that it performs.

Here's a simpler example than what I'm working on that's easier to understand. Suppose I have a generalized card-game-playing application: it knows about hands and players and card decks, but the strategy of any given type of game is left up to a module chosen at run time, whether it's `Baccarat.pm` or `TexasHoldem.pm`. And, of course, other users might write new game modules later. The über-class instantiates a `$game` object that calls an internal method `deal()` to deal hands to players, getting the number of cards per hand from whatever specific module it has loaded, and then calls a `play()` method on a game strategy object instantiated from that module.

All goes well until the user comes up with `Fizzbin.pm`, whose rules require that one player gets a different number of cards from everyone else (<http://en.wikipedia.org/wiki/Fizzbin>). The `deal()` method did not anticipate this. Should you revise the

semantics of `deal()`, and how all the other specific game modules are called, just to accommodate a half-baked game played only on Beta Antares IV?

### I need alternatives, gentlemen

What if, instead, I could modify the `$game` object so that just this once, its `deal()` method came from `Fizzbin.pm` instead? Then the weirdness would be encapsulated where it belongs and no one else would have to change. As long as `Fizzbin.pm` can get at the `$game` object, all it would have to do would be to replace the `deal()` method at run time.

So there's the excuse, er, justification, for dynamic class reconfiguration. Now to the implementation. I was too lazy to come up with the code myself, so I pilfered most of it from a CPAN module. I've used `Test::MockObject::Extends`, written by chromatic, a fair bit in my test writing lately, and that module has the pleasant ability to replace, or *mock*, a method in an already-instantiated object. I studied its source code and took the parts I wanted.

I can't just overwrite the `deal()` method in whatever class it resides because some other game might be in progress simultaneously that wouldn't appreciate having the rug pulled out from under it. I need to change the `deal()` method just for the particular `$game` object I specify.

Methods are just subroutines, which means they live in packages, and if the `deal()` subroutine already exists in the `$game` object's package then my replacement will have to live in a different class. And that means that the `$game` object will then have to be re-blessed into that class so that when the generic code calls the `deal()` method it finds it there. But that new class will have to inherit from the class that `$game` is currently in so that it finds the other methods it needs.

It wouldn't do for this new class to have any other subroutines in it because they might get called by