# Object::Exercise

Keeping your objects healthy.

Steven Lembark
<slembark@cheetahmail.com>

## Objects Need Exercise

- Both in development and “normal” use.
  - Error & regression testing.
  - Benchmarking and capacity planning.
  - Setup for normal operations.
  - Bulk data processing.
- This is a description of how the Object::Execute harness evolved.

# Common Execution Cycle

- Run a command.
- Check $@, optionally aborting if it is set.
- Compare the return value to a known one or run a code snipped to evaluate it, optionally aborting on a mismatched value.
- During development, set a breakpoint if the operation or comparison fails.
- Run the next command…

# The Cycle is Repetitive

- We've all seen – or written – code like this:

```
my $object = Some::Class->construct( @argz );
my $sanity = '';
$sanity = $object->first_method( @more_argz );
'HASH' eq ref $sanity or die “Not a hashref...”;
$sanity = $object->second_part( @even_more_argz );
$DB::single = 1 unless $sanity;
$sanity->[0] eq 'foobar' or die “No foobar...”
$sanity = $object->third_part( @yet_more_argz );
$DB::single = 1 if 'ARRAY' ne ref $sanity;
'ARRAY' eq ref $sanity or die “Not an arrayref”;
$sanity = $object->third_part( 'a' .. 'z' );
```

# Replacing the Cruft

- This is a good example MJD's “Red Flags”:
  - Most of the code is framework, probably updated via cut+paste.
  - You can easily spend more time writing and debugging the framework code than the module itself.
  - It is difficult to generate the code so you end up having to code it all manuall.
- The trick to cleaning it up is abstracting out the framework and leaving the data.

## A better way: Lazy Exercise

- Being lazy means not doing something the computer can do for you or having to do it yourself more than once.
- Perl's OO handles this gracefully by allowing `$object->$name( @argz )` notation: I can store the method calls as data instead of hard-coding them.
- This allows storing the operations as data instead of code.

# Using Object::Exercise

- O::E is basically a “little language” for object execution – a very little language.
- Instructions are either arrayref's that are executed by the object or a small-ish set of text instructions that control the harness.
- Upcoming improvement is allowing Plugin::Installer to easily add add your own instructions.

## Object Execution: Arrayref

- The most common thing you'll do is call a method, possibly checking the results.
- To run a method and ignore anything other than failure use: [ $method => @argz ] (i.e., a method and list of arguments).
- $method can be either text or a coderef, which allows dispatching to local handler code.

# Pass 1: Storing Operations

- The previous code could be reduced to a list of arrayref's each one with a method name (or subref) and some arguments.
- All I need then is an object:

```
sub pass1
{
   my $obj = shift;
   for( @_ )
   {
      my $method = shift @$_;
      $obj->$method( @$_ )
   }
}
```

## Dealing With Failure

- Adding an eval{} allows checking the results and stopping execution on a failure:

```
sub pass2
{
   my $obj = shift;
   for( @_ )
   {
      my $method = shift @$_;
      eval { $obj->$method( @$_ ) };

      print $@ if $@;
      $DB::single = 1 if $@;
      0
   }
}
```

## Taking a Closure Look

- Setting $DB::single = 1 in the Perl debugger starts a breakpoint.
- $DB::single = 1 if $debug sets a breakpoint conditional on $debug being perly-true.
- Adding “our $debug” means it can be localized to true if $@ is set or or cmp_deeply returns false.
- This allows “&$cmd” to stop right before the method is called.

```
sub pass3
{
   my $obj = shift;
   for( @_ )
   {
      my ( $method, @argz ) = @$_;
      my $cmd = sub{ $obj->$method( @argz ) };

      eval { &$cmd };
      if( $@ )
      {
         print $@;
         $DB::single = 1;
         0                    # &$cmd here re-runs
      }
   }
}
```

## Adding A Breakpoint

- I wanted to stop execution before dispatching the method call on re-runs. But “`&$cmd`” immediately runs the code.
- Adding a breakpoint to the closure allows stopping before the method is called:

```
our $debug = '';
sub pass3
{
   my $obj = shift;
   for( @_ )
   {
      my ( $method, @argz ) = @$_;
      my $cmd
      = sub
      {
         $DB::single = 1 if $debug;
         $obj->$method( @argz )
      };

      eval { &$cmd };
      if( $@ )
      {
         print $@;
         local $debug = 1; # &$cmd stops before the method.
         $DB::single  = 1;
         0                 # harness execution stops here.
      }
   }
}
```

# Getting Testy About It

- Usually you will want to  check the results of execution.
- Instead of a single arrayref, I nested two of them: one with the method the other expected results:

```
[
   [ $method => @argz ],
   [ comparison value ],
]
```

## How methods are executed

- If there is data for comparison then the return is saved, otherwise it can be ignored.

```
for( @_ )
{
    my ( $operation, $compare )
    = 'ARRAY' eq ref $_->[0] ? @{ $_ } : ( $_ );

    ...

    if( $@ )
    {
        # set $debug, $DB::single = 1.
    }
    elsif( ! cmp_deeply $result, $compare, $message )
    {
        # set $debug, $DB::single = 1.
    }
    elsif( $verbose )
    {
        print ...
    }
}
```

```
sub pass3
{
   my $obj = shift;
   for( @_ )
   {
      my ( $method, @argz ) = @$_;
      my $cmd
      = sub
      {
         $DB::single = 1 if $debug;
         $obj->$method( @argz );
      };

      eval { &$cmd };
      if( $@ )
      {
         print $@;
         local $debug = 1; # &$cmd stops at $obj->method()
         $DB::single  = 1;
         0                 # execution stops here
      }
   }
}
```

## Doing it your way...

- Q: What if the execution requires more than just $obj->$method( @argz )?
- A: Use a coderef as the method.
- It will be called as $obj->$coderef with a sanity check for $@ afterward. This can be handy for updating the object's internals as operations progress.

```
[
   [ make_hash => ( 'a' .. 'z' ) ], [ { 'a' .. 'z' } ]
],
[
   [ $reset_object_params, qw( verbose 1 )
],
[
   ...
```

## Control statements

- There is a [very] little language for controlling execution.
- These are non-arrayref entries in the list of operations:
  - 'break' & 'continue' set $debug to true/false.
  - 'verbose' & 'quiet' set $verbose for log messages.

## The Result: Object::Exercise

- An early version is on CPAN.
- Supports benchmark mode (no comparison, full speed execution w/ Benchmark & :hireswallclock.
- Few more control statements.
- Plenty of POD.