Repeated-Code-Is-a-Mistake

Table of Contents
Repeated-Code-Is-a-Mistake

Variations on a Theme ...

Repeated Code Is a Mistake

YAPC::CA
Carlton University
Ottawa, Ontario
Friday, 16 May 2003
James E. Keenan
http://www.concentric.net/~Jkeen/repeated/

Inspiration

- From June 2000 to the present, I've attended talks and read articles by Philadelphia Perl master **Mark-Jason Dominus** (MJD) (http://perl.plover.com).
- The principal lesson I've drawn from these talks is:

Repeated Code Is a Mistake!

Locus Classicus

In his November 2000 article ''Program Repair Shop and Red Flags,'' MJD calls repeated code

```
the red flag of all red flags:  Any time a program does something twice,
look to see whether you can get away with doing it only once.
```

He emphasizes the point later:

```
The Cardinal Rule of Computer Programming is that if you wrote the
same code twice, you probably did something wrong.  At the very least,
you may be setting yourself up for a maintenance problem later on
when someone changes the code in one place and not in another. ...
Each time you see you have written the same code more than once,
give serious thought to how you might eliminate all but one instance.
```

Source: http://www.perl.com/lpt/a/2000/11/repair3.html

Obsession

- Since then, I've become obsessed with eliminating repeated code from my work.
- Today, I will try to lure you into become equally obsessed. I hope you will see that eliminating repeated code improve's your code's:
  - readability
  - maintainability
  - reusability
- But first, one more citation from MJD.

Avoiding Code Duplication as a General Principle of Computing Languages

Mark notes in the same article:

```
Programming languages are chock-full of features designed to
prevent code duplication from the very lowest levels (features such
as $a[3] += $b instead of $a[3] = $a[3] + $b) to the very highest
(features such as DLLs and pipes).  In between are essential
features such as subroutines and modules.
```

In this class, we focus on using subroutines and modules to eliminate repeated code.

Mode of Presentation

- You'll see examples from my own code as it has evolved over the last three years.
- **Warning!** Some of the code is not very good.
- But it has gotten better over time, mainly because I've applied what MJD and others drummed into me at various YAPC conferences.
- Fear not! You, too, can improve your code.

Subroutine as Basic Unit of Reusable Code

Rule of Thumb

Our general rule of thumb:
When you see repeated code, you should sense the opportunity to extract that code and place it in a subroutine.

A Database Report Problem

Suppose the following:

- You get canned reports from a legacy database system, but you can't directly access the database to compose your own new reports.
- You can, however, save the database reports as plain-text files.
- The reports present data in a row/column or matrix format.

Data::Presenter

- I encountered this problem on my day job over two years ago. I wrote a module called ***Data::Presenter*** to solve this problem.
- It's object-oriented. Wow!
- On CPAN: http://search.cpan.org/author/JKEENAN/Data-Presenter-0.62
- But this isn't a talk about Data::Presenter.

`get_data_count()`

Here's the original version of a Data::Presenter method which returns the number of data records in the object.

```
sub get_data_count {
    my $self = shift;
    my %data = %$self;
    my $count = 0;
    foreach (keys %data) {
        unless ($reserved{$_}) {
            $count++;
        }
    }
    return $count;
}
```

`print_data_count()`

And here's the original version of a similar Data::Presenter method which get the number of data records in the object and prints it to STDOUT.

```
sub print_data_count {
    my $self = shift;
```

```
        my %data = %$self;
        my $count = 0;
        foreach (keys %data) {
            unless ($reserved{$_}) {
                $count++;
            }
        }
        print "Current data count:  $count\n";
    }
```

What's the *diff*?

To compare, `get_data_count()` and `print_data_count()`, I first use ***perltidy*** to eliminate variation due to different ways of formatting Perl code.
I then use the Unix utility ***diff*** to compare the two subroutines.
The small number of different lines indicates that much of the code is repeated.

```
3c3
< sub get_data_count {
---
> sub print_data_count {
12c12
<     return $count;
---
>     print "Current data count:  $count\n";
```

I smell a subroutine!

First, Design the Interface

I'm going to write a subroutine that will replace the repeated code. First, I decide how `get_data_count()` and `print_data_count()` will each call the new subroutine:

```
sub get_data_count {
    my $self = shift;
    _count_engine($self);
}

sub print_data_count {
    my $self = shift;
    print 'Current data count:  ', _count_engine($self), "\n";
}
```

In other words, I design the **interface** to the new subroutine even before I write its code. Note how much more readable these subroutines have become.

Then, Design the Engine

I like to think of code shared between two subroutines or methods as the 'engine' that powers those functions.
Here's how `_count_engine()` powers the two previous methods.

```
sub _count_engine {
    my $self = shift;
    my %data = %$self;
    my ($count);
    foreach (keys %data) {
        $count++ unless ($reserved{$_});
    }
    return $count;
```

```
}
```

A Side Benefit from Extracting Repeated Code

Extracting repeated code enabled me to see other places in the original methods where I could eliminate superfluous code.

Data::Presenter version 0.43:

```
sub get_data_count {
    my $self = shift;
    my $count = '';
    $count = _count_engine($self);
    return $count;
}
```

Data::Presenter version 0.44:

```
sub get_data_count {
    my $self = shift;
    _count_engine($self);
}
```

Eliminate Synthetic Variables

- MJD would call `$count` in v0.43 a **synthetic variable**:
  ```
  an artifact of the way we solve the problem ...
  inessential to the problem itself.
  ```
- `$count` was useful for readability early in Data::Presenter's development.
- But `_count_engine()` improves the readability so much that `$count` can now be eliminated.
- Assuming there's no loss of readability, eliminating synthetic variables improves the code's readability.

Two Methods Sharing an Engine

Here's a slightly more complicated example of an engine that powers two methods, one which prints data records to STDIN and one to file.
Once again, we first look at the interfaces or 'wrappers'.

```
sub print_to_screen {
    my $class = shift;
    my %data = %$class;
    _print_engine(\%data, \%reserved);
}

sub print_to_file {
    my ($class, $outputfile) = @_;
    my %data = %$class;
    my $oldfh = select OUT;
    open(OUT, ">$outputfile") || die;
    _print_engine(\%data, \%reserved);
    close(OUT) || die;
    select($oldfh);
}
```

The Engine That Powers the Two Methods

```
sub _print_engine {
    my ($dataref, $reservedref) = @_;
    my %data = %$dataref;
```

```
        my %reserved = %$reservedref;
        foreach my $i (sort keys %data) {
            unless ($reserved{$i}) {
                print $_, ';' foreach (@{$data{$i}});
                print "\n";
            }
        }
    }
```

Wrappers and Engines

- As the programming tasks we face become more complicated, leveraging the value of storing repeated code in subroutines becomes more challenging.
- We've already seen one useful technique: Write simple 'wrapper' functions around 'engine' functions which contain the repeated code and which do the heavy lifting.
- A wrapper's interface should be designed for user-friendliness. The test of user-friendliness:
- **Does the interface prompt the user to pass the correct number and type of arguments to the engine?**

User-Friendly Wrappers

To solve a different problem on my day job, I wrote a package called ***Mall::Instructor***. Each instructor on the treatment mall has a profile. To get an instructor's profile, I offer the user these choices:

- **Output Options:** On screen (STDOUT)? Or print to file?
- **Selection Options:** All instructors? Or just individually named instructors?

```
my $m1 = Mall::Instructor->new();

$m1->display_profile('Adams', 'Jones'); # selected data records
$m1->display_profile();                 # all data records

$m1->write_profile('Adams', 'Jones');   # selected data records
$m1->write_profile();                   # all data records
```

How many different cases do I have to account for?

Wrappers Pass Arguments to Engines

2 wrappers and 1 engine handle 4 different cases. Here are the wrappers:

```
sub display_profile {
    my $class = shift;
    if (@_) {
        my @requested = @_;
        _profile_engine($class, \@requested);
    } else {
        _profile_engine($class, 'all');
    }
}

sub write_profile {
    my $class = shift;
    if (@_) {
        my @requested = @_;
        _profile_engine($class, \@requested, 'write');
```

```
        } else {
            _profile_engine($class, 'all', 'write');
        }
    }
```

The Profile Engine (part 1)

The first half of the engine handles the cases where all instructors are selected.

```
    sub _profile_engine {
      my $self = shift;
      my $request_ref = shift;
      my $write = shift if ($_[0]);
      my %data = %$self;
      my (@sought);

      # if no arguments are provided, default to all members of category
      if ($request_ref eq 'all') {
        push(@sought, $_) foreach (sort keys %data);

        # if we're just displaying results on screen ...
        if (! $write) {
          $self->_profile_subengine(\%data, $_) foreach (@sought);
        }

        # but if we're writing to file (only one file, in this case) ...
        else {
          my $output = 'all_in_category.txt';
          my $oldfh = select OUT;
          open OUT, ">$output" or die;
          $self->_profile_subengine(\%data, $_) foreach (@sought);
          close OUT or die;
          select($oldfh);
        }
      }
```

The Profile Engine (part 2)

The second half of the engine handles the cases where we select individual operators to appear in the output.

```
      # but if arguments are provided, handle them properly:
      # make sure argument is actually in the database
      else {
        push(@sought, $_) foreach (sort keys %data);

        # if we're just displaying results on screen ...
        if (! $write) {
          foreach (@sought) {
            defined $data{$_}
              ? $self->_profile_subengine(\%data, $_)
              : print "$_ not found.\n\n";
          }
        }

        # but if we're writing to file(s) (one file per element) ...
        else {
          foreach (@sought) {
            if (defined $data{$_}) {
              my $output = $_ . '.txt';
```

```
            my $oldfh = select OUT;
            open OUT, ">$output" or die;
            $self->_profile_subengine(\%data, $_);
            close OUT or die;
            select($oldfh);
          } else {
            print "$_ not found.\n\n";
          }
        }
      }
    }
```

Engines Can Have Subengines

What's that `< $self->_profile_subengine(\%data, $_)` doing there?

```
    defined $data{$_}
      ? $self->_profile_subengine(\%data, $_)
      : print "$_ not found.\n\n";
```

... and in another spot ...

```
      open OUT, ">$output" or die;
      $self->_profile_subengine(\%data, $_);
      close OUT or die;
```

Is it the same subroutine holding repeated code?

Subengines Deferred

Yes and no. But we'll come back to this later when we discuss polymorphism.

From Subroutines to Modules

When you realize that a subroutine you've written for one script can be used with little or no modification in another script, you're ready to write your first module.

My First Module

Directory::Tools. A set of subroutines which determine whether, from a given directory, you have certain specified subdirectories, whether they are empty or not,etc.

```
  use Directory::Tools qw(kill_subdirectories);

  @subdirs_matched = check_dir_structure('subdir1', 'subdir2');

  @subdirs_added   = add_subdir('subdir4', 'subdir5');

  @subdirs_added   = add_subdir_only_if_new('subdir6', 'subdir7');

  @subdirs_killed  = kill_subdirs('subdir9', 'subdir10');
```

A Regular Module Exports Its Subroutines

Directory::Tools is a regular (*i.e.*, non-object-oriented) Perl module. It exports subroutines either automatically or on specific request. Those subroutines become directly callable in the main package; they are not method calls.

```
    package Directory::Tools;
    use Exporter;
    @ISA = ("Exporter");
    @EXPORT    = qw( check_dir_structure
                     add_subdir
```

```
                  add_subdir_only_if_new );
@EXPORT_OK = qw( kill_subdirs           );
$VERSION = 0.4;
```

Your First Module Won't Be Your Greatest

- Directory::Tools is very limited. It examines only one level of subdirectories beneath the current directory.
- Directory::Tools was largely born out of ignorance. I didn't know how to use standard Perl modules like ***File::Find*** that can do the job more effectively.
- I don't use it in any new work that I do, but I still use it in one script that I run every day.

My First Object-Oriented Module

Two years ago I wrote my first object-oriented module, ***Timerecorder***. It's a crude stopwatch which I apply to various parts of a script.

```
use Timerecorder;
my $tr = Timerecorder->new;

$tr->start_recording();
$tr->record_time('check_dir_structure');
$tr->stop_recording();
$tr->print_to_stats_file("$this_key");
$tr->print_overall_time();
```

A Primitive Profiler

- Timerecorder is a primitive profiler: a program used to examine the speed or efficiency or another program.
- But I didn't even know that term until I attended a talk by Mark Jason Dominus last month, in which he discussed better profilers available on CPAN.
- Its structure is taken directly from Damian Conway's *Object Oriented Perl*.
- Not repeating code is one way of re-using code.
- So is stealing ... er, borrowing it from an expert.

Suddenly, I Was Teaching Perl!

- I whipped together a course using HTML slides whose order was controlled with a simple plain-text file:

```
lists.slide.txt
arrays.slide.txt
last_index.slide.txt
qw.slide.txt
list_assignment.slide.txt
```

- I was constantly comparing the list of file names above with the files I had actually created in the current working directory.
    - Which slides were present?
    - Which slides were missing?
    - Which slides were unused?
- I figured there must be some standard recipe for comparing lists. There is.

To Find a Recipe, Look in the Cookbook

- The *Perl Cookbook*, by Tom Christiansen and Nat Torkington, shows in Recipes 4.6 and 4.7 how to construct ''seen hashes'' to compare the contents of two lists.

```
foreach (@A) { $seenA{$_} = 1 };
foreach (@B) { $seenB{$_} = 1 };
foreach (keys %seenA) {
    if ( exists $seenB{$_} ) {
        push @both, $_;
    } else {
        push @Aonly, $_;
}
foreach (keys %seenB) {
    push @Bonly, $_ unless exists $seenA{$_};
}
```

- But once I had written this code in two different scripts, I wondered whether there already was a module to take care of it?
- I searched CPAN but could find no **simple** implementation of this code. So I wrote my own.

Voila! My First CPAN Distribution

- ***List::Compare*** was my first module distributed on CPAN: http://search.cpan.org/author/JKEENAN/List-Compare-0.16/Compare.pm .

```
$lc = List::Compare->new(\@A, \@B);

@intersection = $lc->get_intersection;
@union        = $lc->get_union;
@Lonly        = $lc->get_unique;
@Ronly        = $lc->get_complement;
@LorRonly     = $lc->get_symmetric_difference;
```

- Now, not only do I not have to repeat the code needed to compare two lists, **no one else in the world does either!**

Reusing Code via Object Oriented Perl

- **Inheritance** and **Polymorphism** demystified.
- They're just ways of re-using code.

Inheritance in Object Oriented Perl

Damian Conway, *Object Oriented Perl*, p. 169:

```
Inheritance in Perl ... means nothing more than:  if you can't find
the method requested in an object's blessed class, look for it in the
classes that the blessed class inherits from.
```

Put even more simply: **If you can't find it here, this is how you find it.**

Remember Data::Presenter?

When you use my Data::Presenter module, you actually invoke it from a subclass which inherits many methods, including its constructor, from Data::Presenter itself.

```
use Data::Presenter;
use Data::Presenter::SampleCensus;
...
$dp1 = Data::Presenter::SampleCensus->new(
            $sourcefile, \@fields,\%parameters, $index);
```

```
$data_count = $dp1->get_data_count();
$dp1->print_data_count();
$keysref = $dp1->get_keys();

$dp1->print_to_screen();
$dp1->print_to_file($outputfile);
$dp1->print_with_delimiter($outputfile, $delimiter);
$dp1->full_report($outputfile);
```

Due to inheritance, none of the code for these methods needs to be repeated inside Data::Presenter::SampleCensus or any other subclass which inherits from Data::Presenter.

Why Subclass at All?

Each Data::Presenter subclass holds subroutines which are fine-tuned to the messy details of parsing data from the database which the subclass is designed to handle.

&Data::Presenter::new

```
sub new {
  ...
  $self = bless {}, ref($class) || $class;
  ...
  $dataref = $self->_init(
      $source, $fieldsref, $paramsref, $index, \%reserved);
  ...
  %$self = %$dataref;
  return $self;
}
```

&Data::Presenter::SampleCensus::_init

```
sub _init {
  ($self, $sourcefile, $fieldsref, $paramsref, $index) = @_;
  %data = ();
  $data{'fields'} = $fieldsref;
  $data{'parameters'} = $paramsref;
  $data{'index'} = [$index];
  ... # DATA MUNGING
  $data{$corrected[$index]} = \@corrected;
  return \%data;
}
```

An Inherited Constructor

- When you call
  ```
  $dp1 = Data::Presenter::SampleCensus->new()
  ```
  ... Perl notes that there is no subroutine called `new()` in Data::Presenter::SampleCensus.
- Perl then asks: From whom does Data::Presenter::SampleCensus inherit?
- The answer is found in `@Data::Presenter::SampleCensus::ISA`.
  ```
  @ISA = qw(Data::Presenter);
  ```

Initializer in the Invoking Subclass

- When we reach
  ```
  $dataref = $self->init(...);
  ```

... we are instructed to call the version of `_init()` found in the invoking subclass — in this case `Data::Presenter::SampleCensus::_init()`.

- That `_init()` function parses data from a specific sourcefile.
- We would write a **different** Data::Presenter subclass with its **own** `_init()` subroutine to parse a **different** sourcefile.
- But in each case we would re-use the parent Data::Presenter module's constructor and output methods.

Interface Polymorphism

```
foreach $datum (@data) { $datum->print_me(); }

@data = (
    XML::File->new("./lamasery.xml");
    HTTP::get->new("http://www.perl.org/news.html");
    Signature->new();
}
```

- As long as each object's class's interface provides a `print_me()` method, the method call will handle `< $datum->print_me();` correctly. (Damian Conway, *Object Oriented Perl*, p. 204.)
- In interface polymorphism, **we're re-using the interface!**

Inheritance Polymorphism

To print labels for your CD collection, you might design a one-size-fits-all label that would work for **any** CD, but then design labels for specific genres that printed extra information more appropriate to those genres.

```
$cd    = CD::Music->new(...);
$cdc   = CD::Music::Classical->new(...);
$cdj   = CD::Music::Jazz->new(...);

$cd->print_label();     # would work for any CD
$cdc->print_label();    # would work for classical CDs
$cdj->print_label();    # would work for jazz CDs
```

Inheritance polymorphism requires that objects provide a specific method **and** that they belong to classes in a common hierarchy.
We re-use the interface **and** we re-use the code in `< $cd->print_label()` **if** our invoking class does not define its own `print_label()` method.

Another Kind of Polymorphism

- Conway's discussion of polymorphism is couched in terms of **publicly callable methods**.
- But a publicly callable method in a parent class might internally call a subroutine which is located in the invoking child class — and is defined differently in each such child class.
- Remember this code from Mall::Instructor?

```
defined $data{$_}
  ? $self->_profile_subengine(\%data, $_)
  : print "$_ not found.\n\n";
```

... and in another spot ...

```
open OUT, ">$output" or die;
$self->_profile_subengine(\%data, $_);
close OUT or die;
```

‘Under-the-hood’ Polymorphism

- Actually, `_profile_engine()` is **not** a function found in Mall::Instructor.
- It‘s found in a package called ***Mall***, from which Mall::Instructor — and other subclasses such as Mall::Room and Mall::Schedule — inherit.
- Each Mall subclass displays its data in slightly different ways.
- So each such subclass has its own `_profile_subengine()` to handle the messy details.
- But each such `_profile_subengine()` has the same interface and similar internal structure.
- The `< $self->` means: Call the version of `_profile_subengine()` found in the invoking package.

Identically Named Subroutines in Different Invoking Classes

Here are two different functions with the same name but found in different invoking packages:

```
sub _profile_subengine {
    my ($self, $dataref, $current) = @_;
    my @record = @{${$dataref}{$current}};
    print <<INSTRUCTOR;
Instructor Profile for:\t$record[1] $record[0]
    Title:       $record[3]
    Department:  $record[2]
    Phone Ext.:  $record[5]
INSTRUCTOR
}

sub _profile_subengine {
    my ($self, $dataref, $current) = @_;
    my @record = @{${$dataref}{$current}};
    print <<SCHEDULE;
Profile for Group $current in Room $record[0] at Time Slot $record[1]
    Room:        $record[0]
    Time Slot:   $record[1]
    Group Name:  $record[2]
    Leader:      $record[5]
SCHEDULE
}
```

‘Under-the-hood’ Polymorphism in List::Compare

- My CPAN module List::Compare also features ‘under-the-hood’ polymorphism.
- Whether you wish to compare two lists or an arbitrary number of lists, the interface looks the same.

```
$lc   = List::Compare->new(\@A, \@B);
@comp = $lc->get_complement;

$lcm  = List::Compare->new(\@A, \@B, \@C);
@comp = $lcm->get_complement;
```

- But since the mathematical notions of ‘union‘, ‘intersection‘, ‘complement‘, etc. are quite different for three or more sets, the calculation of these relationships inside List::Compare is quite different as well.

Different Initializers for Different Numbers of Arguments

Under the hood, `List::Compare::new()` calls a different `_init()` routine depending on the number of arguments passed to it:

```
sub new {
    $class = shift;
    @args = @_;
```

```
        if (@args > 2) {
            $class .= '::Multiple';
            $self = bless {}, ref($class) || $class;
        } else {
            $self = bless {}, ref($class) || $class;
        }

        $dataref = $self->_init(@args);
        %$self = %$dataref;
        return $self;
    }
```

In Conclusion ...

What We've Learned (I)

- Format your code in a consistent manner (or use ***perltidy*** to make it so). Then, use ***diff*** to identify where code might differ in blocks that appear to be repeated.
- The subroutine is the basic unit of code re-use in Perl.
- Place code shared between two subroutines in an engine, then write a wrapper around the engine.
- Engines can have subengines.
- The test of an interface's user-friendliness is whether it guides the user to pass the correct number and types of arguments to a subroutine.

What We've Learned (II)

- While working a project, begin with synthetic variables to maximize readability.
- But as the project develops, eliminating synthetic variables may enable you to identify patterns of code repetition. If so, long-term maintainability and reusability will increase.

What We've Learned (III)

- When you've used the same subroutine in two different scripts, it belongs in a module.
- Regular Perl modules export their functions. Object-oriented Perl modules do not; they use method calls instead.
- Inheritance and polymorphism are methods of reusing code both in the way methods are called and in the way they are coded internally.

What We've Learned (IV)

- Avoiding repeated code increases readability, maintainability and reusability.

The End

James E. Keenan
http://www.concentric.net/~Jkeen/repeated
jkeenan@cpan.org (put 'repeated' in subject line)