Red Flag Summary

June 2, 2002

REPEATED CODE

Symptom

The same code appears twice.

Generic Treatment

Most all features of languages and many features of operating systems exist primarily to reduce repeated code. Investigate how similar problems have been solved in the past.

Small-Scale Treatments

Assignment operators:

```
# before                                    # after
$x->{q} = $x->{q} * 1.03;                   $x->{q} *= 1.03;
```

The `x` operator:

```
# before                                    # after
@a = ($z, $z, $z, $y, $y, $y);              @a = (($z) x 3, ($y) x 3);

# before                                    # after
@a = ($z, $y, $z, $y, $z, $y);              @a = ($z, $y) x 3;
```

Small-scope temporary variables:

```
        # before
        if (($zr{$ml{aban_TopItem_id_toptem}}{ibeg}
eq $zr{$ml{aban_TopItem_id_toptem}}{iend}) &&
($zr{$ml{aban_TopItem_id_toptem}}{ibeg} =~ /\.5/)) {
            $zr{$ml{aban_TopItem_id_toptem}}{beg} =
($zr{$ml{aban_TopItem_id_toptem}}{ibeg} - .5);
            $zr{$ml{aban_TopItem_id_toptem}}{end} =
($zr{$ml{aban_TopItem_id_toptem}}{iend} + .5);
            $zr{$ml{aban_TopItem_id_toptem}}{lop} = "^";
```

```
        }
        else {
            $zr{$ml{aban_TopItem_id_toptem}}{beg} =
$zr{$ml{aban_TopItem_id_toptem}}{ibeg};
            $zr{$ml{aban_TopItem_id_toptem}}{end} =
$zr{$ml{aban_TopItem_id_toptem}}{iend};
            $zr{$ml{aban_TopItem_id_toptem}}{lop} = '..';
        }
        ...

        # after
        { my $ID = $ml{aban_TopItem_id_toptem};
          my %zrh;
          $zr{$ID} = \%zrh;
          if (($zrh{ibeg} eq $zrh{iend}) && ($zrh{ibeg} =~ /\.5/)) {
              $zrh{beg} = ($zrh{ibeg} - .5);
              $zrh{end} = ($zrh{iend} + .5);
              $zrh{lop} = "^";
          }
          else {
              $zrh{beg} = $zrh{ibeg};
              $zrh{end} = $zrh{iend};
              $zrh{lop} = '..';
          }
        }
```

Medium-Scale Treatments

Use a loop:

```
# before
code1
code2
code3
code4
code5

# after
for (1..5) {
  code$_
};
```

Use a subroutine:

```
   # before
   if {
```

```
  ...
  CODE
  ...
} else {
  ...
  CODE
  ...
}

# after
if {
  ...
  func(...);
  ...
} else {
  ...
  func(...);
  ...
}
...

sub func {
  CODE
}
```

In programs written in an object-oriented style, use an inheritance or delegation structure.

Large-Scale Treatments

Factor the repeated code into a new module.

Turn the repeated code into a separate program and use an IPC mechanism (files, pipes, sockets shared memory, etc.) to commonucate with it.

Arbitrary Numeric Constants

Symptom

Numeric constants other than 0 and 1.

Treatment

Try to generalize the code to reduce the dependence on the constant. Where does it come from?

If it's in the outside world, comment it.

```
# 86400 = number of seconds in a day
$days = time() / 86400;
```

If it's in the problem specification, add an abstraction such as a named constant:

```
# The data file format specification says that the file
# will always have exactly four ACCOUNT sections
$num_account_sections = 4;
```

If it's an unwarranted assumption, fix it, or add an assertion to check for it:

```
die "Input was supposed to have one section per month, has \$sections instead"
  unless $sections == 12;
```

Array length variables

Symptom

```
$array[$i++] = VAL;
```

Treatment

```
push @array, VAL;
```

Backslashed double-quote

Symptom

```
"...\"...\"..."
```

Treatment

```
qq{..."..."...}
```

Comments (in general)

Symptom

```
# IMPORTANT: values of beta will give rise to dom!
```

Treatment

Folks tell you that if you do anything tricky or peculiar, you should put in a comment to explain. But it's better if you can find a way to accomplish the same goal *without* doing anything tricky or peculiar. Good code explains itself.

Safe uses for comments are at the head of a function or module to explain what it does, how to call it, and what it returns, and comments explaining the purpose of a variable and what kind of values it will take on.

Comments, End-Of-Block

Symptom

```
} # end of while statement
```

Treatment

Be sure the block is indented properly. Braces should line up consistently.

If the block is properly indented, then this comment probably means the block is too large, too complicated, or both. Turn some of the code into one or more subroutines and call the subroutines from the block.

If the code is too tangled to be factored into subroutines, then nobody was able to understand it anyway. It should be rewritten.

Finally, if you are still having trouble figuring out which brace goes with which, get a better editor program. Programming editors will tell you automatically which braces match which others.

Comments, Excessively Decorated

Symptom

```
# +============================================================+
# I/------------------------------------------------------------\I
# I| !!!!!!!!!!!!!!!!!! Construct @a !!!!!!!!!!!!!!!!!!!! |I
# I\------------------------------------------------------------/I
# +============================================================+
```

Treatment

```
# Construct @a
```

The concatenation operator . (and , in `print`)

Symptom

```
"foo/[".$x[$y]."]bar:".$z->{blah}."\n"

print "<img src='", $url, "'>\n";
```

Treatment

```
"foo/[$x[$y]]bar:$z->{blah}\n"

print "<img src='$url'>\n";
```

The Condition that Ate Michigan

Symptom

```
sub func {
  if ($condition) {
    ...
  }
}
```

Treatment

```
sub func {
  return unless $condition;

  ...
}
```

Also see **Unbalanced `if`–`else` Blocks**, below.

C-style `for` loop and loop counter variables

Symptom

```
for ($i = 0; $i < @array; $i++) {
  ... $array[$i] ...
}
```

Treatment

```
for my $item (@array) {
  ... $item ...
}
```

If you really need `$i` not just for indexing the array, but for some numeric purpose inside the loop (for example, for printing out a numbered list of elements) then it's still preferable to use:

```
for my $i (0 .. $#array) {
  ... $array[$i] ...
}
```

Empty `if` and `else` blocks

Symptom

```
if ($condition) {
} else {
  ... statements ...
}
```

```
if ($condition) {
  ... statements ...
} else {
}
```

Treatment

```
unless ($condition) {
  ... statements ...
}
```

```
if ($condition) {
  ... statements ...
}
```

Families of related variable names

Symptom

```
my ($item1, $item2, $item3, $item4, $item5, $item6, $item7,
    $item8, $item9, $item10, $item11, $item12, $item13,
    $item14, $item15, $item16);

my ($user_name, $user_age, $user_city, $user_state,
    $user_phone_number $user_hair_color,
    $user_first_visit_date, $user_number_of_nostrils);
```

Treatment

```
my (@item, %user);
```

File-scope lexicals

Symptom

```
my($x, $foo, %bar, ...)
```

at the top of the file

Treatment

Restrict scope of variables where possible.

See 'Global variables' below.

Global variables

Symptom

Program doesn't run under 'strict vars'

Treatment

Redesign functions. Generalize functions. Use standard functional argument passing and value return techniques. Declare global variables as small-scope lexicals where possible.

Leaning toothpick syndrome

Symptom

```
if (/\/usr\/local\/lib\/perl\//) { ... }

s/\//-/;
```

Treatment

```
if (m{/usr/local/lib/perl/}) { ... }

s{/}{-};
```

Many consecutive `prints`

Symptom

```
print "...\n";
print "...\n";
print "...\n";
print "...\n";
```

Treatment

```
print qq{...
...
...
...
};
```

Making two passes

Symptom

Two nearby loops over the same file, hash, or array. Preprocessing phases.

Treatment

Try to merge them into one loop. But this may complicate the code excessively, so be sure to *Try it Both Ways*. Only merge the passes if it simplifies the code.

Many very long strings

Symptom

```
print qq{...
...
...
...
};
```

Treatment

Store strings in files. Use template modules where appropriate.

Single scalar variable in quotes

Symptom

```
"$foo"
```

Treatment

```
$foo
```

Special Case Tests

Symptom

A large block of code inside an `if` block with no `else`:

```
if (@items > 0) {
  for (@items) {
    ...
  }
}
```

Treatment

Ask "What would break if I got rid of the test?" If the answer is "Nothing", then get rid of the test.

If not, see *The Condition That Ate Michigan*, above.

Taking two steps forward and one step back

Symptom

Code whose only purpose is to undo the work done by other code

Treatment

Use more variables. Save the original, unchanged version of the data instead of changing it and then changing it back again.

Testing a boolean value against 0 or 1

Symptom

```
if (-d foo == 1) { ... }

if (foo() == FALSE) { ... }
```

Treatment

```
if (-d foo) { ... }

if (!foo()) { ... }
```

Unbalanced `if`–`else` Blocks

Symptom

```
if (CONDITION1) {
  if (CONDITION2) {
    if (CONDITION3) {
      ACTION;
    } else {
      warn ERROR3;
    }
  } else {
    warn ERROR2;
  }
} else {
  warn ERROR1;
}
```

Treatment

```
unless (CONDITION1) {
  warn ERROR1;
  BREAK;
}

unless (CONDITION2) {
  warn ERROR2;
  BREAK;
}

unless (CONDITION3) {
  warn ERROR3;
  BREAK;
}

ACTION;
```

`BREAK` here represents a non-local jump operator, such as `next`, `last`, `return`, `die`, or `exit`, as appropriate. If none of these will solve the problem, it is often appropriate to move the entire structure into a new subroutine and to use `return`.

After rewriting the nested block, check to see if two or more `unless` sections can be combined into one using the `&&` or `||` operators.

`else` with `unless`

Symptom

```
unless ($condition) {
  ... code A ...
} else {
  ... code B ...
}
```

Treatment

```
if ($condition) {
  ... code B ...
} else {
  ... code A ...
}
```

Variable use immediately follows assignment

Symptom

```
$x = EXPRESSION;
if ($x) { ... }
```

and then `$x` is never used again

Treatment

```
if (EXPRESSION) { ... }
```