Perl Reliability Workshop

How to Clean up Your Perl Code and Make It More Readable and Reliable.

Producing Better Perl

- strict / warnings / taint
- Eliminate Eliminated code
- Use quotes properly
- Use “Here” strings
- Remove Dangling **if** statement
- Use function prototypes
- Put in comments
- Avoid Spaghetti code

Basic Language Checks

- **`use strict;`**
- **`use warnings;`**
- **Taint mode**

  `#!/usr/bin/perl -T`

  - Taint mode should be turned on for every CGI program.

Security Problem

- What's wrong with the following code?

```
my $url = $cgi->param("url");
system("check_url $url");
```

Eliminate Eliminated code

```
34 ##XXX:REPLACED BY cgCDI:use Pg;
35 use rapid::confgen::cgCDI;
36 use iprism::OVRAMailSystem;
37 use POSIX qw(strftime);
38 use iprism::Categories;
39 ##XXX:REMOVE, NOT USED!:use UTILS;
40 ##XXX:REPLACED BY cgCDI:use confgen;
```

Eliminate Eliminated code

```
##XXX:REPLACED:#
##XXX:REPLACED:# execQuery
##XXX:REPLACED:################
##XXX:REPLACED:sub execQuery {
##XXX:REPLACED:  my $self = shift;
##XXX:REPLACED:  my $query = shift;
##XXX:REPLACED:
##XXX:REPLACED:  $self->connection()
##XXX:REPLACED           unless $self->{$CONNECTION};
##XXX:REPLACED:
##XXX:REPLACED:  my @results;
And so on for the rest of the subroutine
```

Eliminate Eliminated code

- It’s hard enough to read the code when it tells me what you are doing.
- CVS tells me what was eliminated.

Poor Use of Quotes

```
my %hash = (
     "FORM_TAGS" => "<input type
  = \"hidden\" name = \"action\"
  value= \"uploadFilters\“ >“
);
```

Strings can be enclosed in ``

Bad

```
$v  = " a \"quoted\ " word ";
```

Better

```
$v  = ' a "quoted" word ';
```

But what about mixed messages?

- How would you deal with:

```
Replaced "don't" with "do not".
```

- Perl has many different ways of quoting:

```
qq/Replaced "don't" with "do not"./
```

Use Here Documents

Bad

```
$error =
   “Probable user error.\n”.
   “User should correct data\n”.
   “and redo the operation.\n”;
```

Use Here Documents

GOOD

```
$error = <<EOF;
Probable user error.
User should correct data
and redo the operation.
EOF
```

The dangling **if**

```
do_something()
```

This statement should do something.

WRONG!

```
do_something()
    if ($maybe)
```

Code flow == thought flow

```
if ($condition) {
    do_something();
}
```

Much better.

Keep **if** code together

Wrong!

```
if ($condition) {
    … 1,000 lines
} else {
    print “ERROR:No Cond.\n”);
}
# End of function
```

Keep **if** code together

Better

```
if (not $condition) {
    print “ERROR:No Cond.\n”);
} else {
    … 1,000 lines
}
# End of function
```

Keep **if** code together

Best

```
if (not $condition) {
    print “ERROR:No Cond.\n”);
    return;
}
… 1,000 lines
# End of function
```

Function Definitions

Example:

$$ -- two scalar parameters

```
sub min($$) {
    my $n1 = shift;  # A number
    my $n2 = shift;  # Other number
```

Use shift to get the parameters

Comment each parameter

Prototype Parameters

$ -- Scalar

% -- Hash (must be last)

@ -- Array (must be last)

-- There’s some more advanced stuff

Passing Parameters -- Arrays

```
sub foo($@)
{
    my $size = shift; # Counter
    my @data = @_; # Data array
…
foo(15, “A”, “B”, “C”);
```

Passing Hashes

```
sub draw($%)
{
    my $shape = shift;
    my %options = @_;
```

Using hash passing

```
draw(“SQUARE”,
     -bg => “white”,
     -fg => “green”,
     -size => “very big”
);
```

Function Comments

1. Explain Function
2. Explain Function Return value
3. Describe all variables

Commenting Regular Expr.

```
 +---------- Beginning of line
 |+++------- One or more(+) sp.
 |||| ++---- Non space
 |||| ||+--- Zero or more
 ||||+|||+-- Put in $1
/^\s+(\S*)/
```

Spaghetti code

Wrong

```
if ($action eq “Up”) {
    … 300 lines
} elsif ($action eq “Down”) {
    … 400 lines
} elsif ($action eq “Back”) {
    … 200 lines
-- and so on for 60 actions --
```

Spaghetti code

Better

```
if ($action eq “Up”) {
   do_up()
} elsif ($action eq “Down”) {
   do_down()
} elsif ($action eq “Back”) {
   do_back()
-- and so on for 60 actions --
```

Best

```
my %act_func = (
    “Up” => \&do_up,
    “Down” => \&do_down,
    “Back” => \&do_back,
    … 60 more actions
);
if (defined($act_func{$action})) {
     $act_func{$action}();
} else {
     do_default_action();
}
```

Indentation

- Indent 4 spaces for each level. (Studies have show that 4 make the cost most readable.)

Indentation with vim

- The vim editor has a indent function.
  1. :set sw=4
  2. Go to the start of the block you wish to indent
  3. **v**
  4. Go to the end of the block. It will be highlighted
  5. **=**
- Other vim commands are available. Ask me.