

This file is copyright © 2006 Mark Jason Dominus.  
Unauthorized distribution in any medium is  
absolutely forbidden.

---

1. Conciseness
  2. Families of Variables
- 

## Introduction

This book is about writing better programs, which is something that I hope we would all like to try to do.

Specifically, it's about common mistakes that people make when they're writing programs. I've looked at a lot of Perl code in the past ten years or so, and I've discovered that there are a lot of mistakes that almost everyone makes that make their programs worse than they could be.

These errors are "low-hanging fruit". If you want to get all the apples from the apple orchard, you might have to go to a lot of trouble and effort to climb up and get all the apples. But if you're content to get half the apples, you can pick just the low-hanging apples, and you'll get half the apples for far less than half the effort. The errors I'll show in this book are easy to find, and when you've found them, they're easy to fix. It's hard to write the perfect program, and doing it might take a lot of trouble and effort. But with a very small effort you can learn to find and fix the "low-hanging fruit", and make your programs a lot better with very little trouble or effort. You do not need an extensive knowledge of obscure Perl features to do this.

This book is emphatically *not* about clever tricks. My last book was clever tricks from one cover to the other, and I have a reputation for clever tricks. I have worked very hard to eliminate all the cleverness from this book. That's because nobody can be clever every day. We all have days when we got up too early, or drank too much bourbon the night before, or something else, and we are not at our best. This book is about being a good programmer and writing good programs even when you are hung over. Perhaps it requires cleverness to be a great programmer. But I don't think that any cleverness is required to be a *good* programmer, and that's what this book will help you do.

## Conciseness

I believe very strongly that if all other things are equal, the shorter program is better than the long one. If there's less code, there's less chance that the author got some of it wrong, and there's less for the maintenance programmer to understand or to maintain.

Clearly, there are exceptions to this shorter-is-better doctrine. That's why I qualified it by saying "if all other things are equal". But often, all other things *are* equal. When they're not, it's usually pretty clear that something strange is going on.

We will measure conciseness by counting lines of code. When I say this in my conference talks, someone often complains that counting lines makes no sense. But it *does* make sense, for several reasons.

First, everyone understands it. If I mentioned a 700-line Perl program, people have a pretty fair idea of how much code is involved. Sure, maybe my idea of a 700-line program actually corresponds to your idea of an 800-line or a 600-line program. But the goal here is not to come up with a really delicate unit of measurement. We are only looking for a gross measurement. If one program is 700 lines and another is 750, I am likely to say that they are about the same size; the difference in line count gets lost in the noise. But if one program is 700 lines and another is 350 lines, the 700-line program is clearly much bigger, even if the line counting is not very accurate.

In conference talks, someone usually raises their hand at this point to complain:

**But you can make any program smaller just by squeezing out all the whitespace.**

Exactly! This just goes to show that everyone understands what it means to count lines. *Everyone* agrees that when you are counting lines, you must not count the white space.

**For conciseness, whitespace does not count.**

*Everyone* agrees that this should not count as a 4-line program:

Download code for `obfuscated-japh.pl`

```
@P=split//, ".URRUU\c8R";@d=split//, "\nrekcah xinU / lreP rehtona tsuJ";sub p
@p{"r$p", "u$p"}=(P,P);pipe"r$p", "u$p";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f|or
($p{$_})&6];$p{$_}=/^$P/ix?$P:close$_}keys%p}p;p;p;p;p;p;map{$p{$_}=~/^[P.]/&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)if/\S/;prin
```

No, of course not. It is a 24-line program that has been squashed.

So we can count lines as a measure of conciseness, as long as we are careful not to let irrelevant whitespace matters throw off our count. We do not want these two blocks to score differently:

```
# Block 1
if (not $page->param('round'))
{
    $round = 0;
}
else
{
    $round = $page->param('round');
}

# Block 2
if (not $page->param('round')) { $round = 0 }
else { $round = $page->param('round') }
```

They are doing the exact same thing in the exact same way, so we should consider them equally concise. In the line-counting regime I will use in this book, both blocks will count as **3** lines of code, not as 8 or 2. Here's why:

```
if (not $page->param('round'))      # 1
{
    $round = 0;                      # 2
}
else
{
    $round = $page->param('round');  # 3
}
```

The basic principle is that any line with new semantic content counts. The `if` line counts, because you have to understand the meaning of the condition. And the assignment in the body of the `if` counts, because you have to understand the purpose of the `$round` variable and you have to understand why it is being changed. But the `else` line does not count, because once you have understood the condition on line 1, you know everything there is to know about the `else`---it is just the opposite. The body of the `else` contributes a third line. The other lines are just punctuation and don't count. And whitespace doesn't count, so this code is also three lines:

```
unless ($page->param('round')) { $round = 0 } # 1-2
else { $round = $page->param('round') }      # 3
```

White space doesn't count, and neither do curly braces, or lines with just parentheses or other punctuation. Simple statements count as one line each, and control clauses such as `while`, `if`, and `for` also count as one line each, because you have to understand the condition. `else` does not count, as explained above, and neither does `do`, for similar reasons.

The really important thing to remember about the line counting is that the details aren't too important. Perhaps we will overcount or undercount a program by a few lines. But I will show how to reduce the line count of every program in this book by at least 30%, not by squeezing out the white space or any such underhanded trick, but by really using less code to do the same jobs in simpler ways. Maybe the counts are off a little bit, so I am really reducing the programs by 25% or 35% instead of 30%. But the point still stands: the programs are much more complicated than they could have been, and we can make them a lot simpler.

## Families of Variables

The low-hanging fruits we will be picking are errors I call "red flags". They are red flags because they are easy to see. Once you have been trained to see red flags, you can *see* them sticking up out of your program, waving at you. You can learn to notice red flags, and when you notice them, you can pause and ask yourself "I wonder if there is a way I could have done this better?" Sometimes the answer is "no". But often there *is* an easy way to make the program better, and then you have an opportunity.

The first red flag we'll see is **families of related variable names**. If you have a family of similarly-named scalar variables, such as `$x1`, `$x2`, `$x3`, and so on, that is a red flag. In almost all such cases, you should prefer to use an array, and use `$x[0]`, `$x[1]`, `$x[2]`, and so on, instead.

What benefit does `$x[0]`, `$x[1]`, `$x[2]`, ... have over `$x1`, `$x2`, `$x3`? One benefit is that if you need for

some reason to treat the family as a group, and do something with the entire family at once, Perl has a convenient notation for the entire family: @x. If you have used scalar variables, you will have to list all the variables explicitly. For example,

```
for (@X) {
    # do something ...
}
```

instead of

```
for ($X1, $X2, $X3, $X4, $X5, $X6, $X7, $X8, $X9,
     $X10, $X11, $X12, $X13, $X14, $X15, $X16, $X17) {
    # do something ...
}
```

I think it's clear that in this case nothing is lost by preferring the more concise version. With the longer version, the maintenance programmer has to carefully examine the target of the `for` and check to make sure that the numbers increase in sequence. In the array version, the maintenance programmer doesn't have to check; they know without checking.

The essential problem with the family is that although the variables are related, Perl does not know that, so it cannot give you any help in treating them as a family. The only relation between `$x1` and `$x2` is in your mind. You have a convention about the way the variables will be used. The convention isn't apparent to Perl and it may not be apparent to the maintenance programmer. It's best to make such things explicit because that way Perl can help you do what you want. Perl's way of doing that is with an array.

Once you have used an array, Perl can help you manage the family of variables. For example, you might find at run time that your program needs to add another member to the family. With the scalars, there is no convenient way to do that. [1] With the array, you just use `push` to add an item to the array.

I should mention at this point that every red flag in this book appears in real code. I have been collecting code examples for years. Whenever I get an idea about something that I think might be a red flag, I check through my hundreds of code examples to see if it is really a common error. Usually I am right, but sometimes it turns out that the mistake is not all that common, and in those cases I didn't put the red flag into the book. This habit of using families of variables instead of arrays is amazingly common, and I have dozens of examples of it. Here is one of my favorites:

```
$sth = $dbh->prepare("SELECT * FROM info");
$sth->execute();
my($one, $two, $three, $four, $five, $six, $seven, $eight,
   $nine, $ten, $eleven, $twelve, $thirteen, $fourteen, $fifteen,
   $sixteen);
$sth->bind_columns(undef, \$one, \$two, \$three, \$four,
                 \$five, \$six, \$seven, \$eight, \$nine, $ten, $eleven,
                 $twelve, $thirteen, $fourteen, $fifteen, $sixteen);
while ($sth->fetch) {
    $SqlStatement4 = ("INSERT INTO Customer_Information " .
" (CustomerNumber, Name, Address1, Address2, City, State, ZipCode, Email, Default
" VALUES ('$one', '$two', '$three', '$four', '$six', '$seven', '$eight', '$nine
$db->Sql($SqlStatement4);
}
```

Usually when someone wants to perpretrate a family of variables names, they do it by writing `$x1`, `$x2`,

\$x3, and so on, as I discussed above. The author of this code is unusually gifted! He has written \$one, \$two, \$three instead!

I swear this is real code. In fact, all the code in this book is real, collected from internet discussion groups over the years. This is for two reasons. First, I was afraid that if I let myself make up examples, I would make up examples to show errors that people don't really make, and that would be a waste of your time. If I can't find real code to show examples of an error, then it's not a real error, and I won't bother you with it. But second, I would never have the gall to make up examples that are half as bad as the programs that people really write. I would take one look at them and say to myself "no, nobody will ever believe that! I had better tone it down." And if I *didn't* tone it down, you might not believe me either! By showing only real examples, I can discuss some really bad code without anyone thinking that I am exaggerating what really happens.

I found the example above on PerlMonks (<http://www.perlmonks.org/>.) PerlMonks is a great source of rotten code. (It's also a great source of helpful and kind advice from wise and experienced programmers, and I heartily recommend it.)

So how can we clean up the code above? The first thing to try is to replace the family of variables with an array:

```
$sth = $dbh->prepare("SELECT * FROM info");
$sth->execute();
my $sth4 = $dbh->prepare(qq{INSERT INTO Customer_Information
    (CustomerNumber, Name, Address1, Address2,
     City, State, ZipCode, Email, DefaultPassword)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?) });
while (my @data = $sth->fetchrow_array()) {
    $sth4->execute(@data[0,1,2,3,4,6,7,8,15]);
}
```

We have eliminated 15 variables here. That is 15 fewer things that the maintenance programmer will have to understand or remember.

This advice, about eliminating families of related scalars, also extends to cases where the scalars have names instead of numbers. For example, if you have \$user\_name, \$user\_age, \$user\_hair\_color, and so on, consider replacing them with a hash called %user, and then use:

```
$user{name}
$user{age}
$user{hair_color}
```

instead. The good advice here is **store related values together**. The name, age, and hair color are all related, so they should share space in the same data structure, if at all possible.

And now we begin.

---

[1] You can use `eval`, or you can use symbolic references.

---

TOP