BOOK REVIEWS

The Productive Programmer

reviewed by Peter Scott, Peter@psdt.com

The Productive Programmer lands on the desk carrying the added weight of a huge promise: to lay bare the mechanics of why one programmer can be a hundred times as productive as another. We've known and talked about this statistic for decades, but never has a book taken such square aim at the bullseye of the holy grail for developers: how to raise your game to the level of the programming masters. That's enough to make anyone keep turning the pages, but also demands a brutally honest assessment of how well the book delivers on its promise in return for your attention.

I started this book dearly hoping it wouldn't be another lesson in refactoring Java because I want to be as productive as the masters. First, the bad news. Most of the examples in the book are in Java, with a few in Ruby, and the only mention of Perl I could find was quoting Larry Wall on the virtues of a programmer in the foreword. Now, I don't want to be a language bigot, and certainly nothing about this book promised it would have *anything* to do with Perl, but it just bugs the heck out of me when people claim that *this* (taken from the book) is user-friendly syntax:

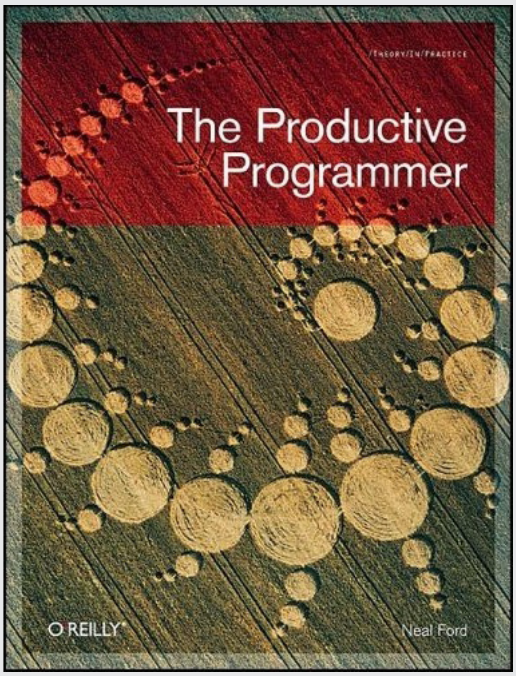
```
@Test public void factors_for_1() {
  List<Integer> expected =
     new ArrayList<Integer>(1);
  expected.add(1);
  Classifier c = new Classifier(1);
  assertThat(c.getFactors(), is(expected));
  }
```

The Perl equivalent is task-oriented and hides most of the details behind a sensible interface:

```
use Test::More;
is_deeply
  [ get_factors(1) ],
  [ 1 ];
```

Or, if the factorizer simply must be object-oriented, the equivalent is still much nicer to read:

```
use Test::More
is_deeply
  [ Factorize->new(1)->get_factors ],
  [ 1 ];
```



The Productive Programmer
Neal Ford
ISBN 978-0-596-51978-0
O'Reilly Media, 2008
http://productiveprogrammer.com

But enough venting. Before the book gets heavily into code, it gives advice that will be dear to the hearts of many of our readers. Starting from the aphorism that "Eye candy looks good but isn't nutritious," (I'm sure many of you who haven't heard that one before will turn it into a catchphrase), the author nails his colors to the wall with exhortations to embrace the command line, shun GUIs, and prefer typing over mousing. He explores keyboard accelerators, application launchers and macro generators (he says that hotkeys have problems on Windows but I've found Macro Express works in everything I use). I didn't know that Windows even has command history searching and tab completion.

Ford is unfailingly polite in giving platforms equal time and Windows, Macintosh, and *nix users will all find help. I've needed a way to make true filesystem links on Windows for a while and this book was worth getting just to learn from it about a utility called Junction that does that.

These tidbits are wrapped in the contexts of accelerating your keyboard performance, reducing distractions, and automating common tasks. It then heads into the major category of Practice, which is where it waxes on various object-oriented programming philosophies.

The book has practical advice on source code analysis such as code coverage and automated duplication detection (something I do in my code review business). When it veers back into the Java world it makes the assertion that all objects should always have valid state and therefore should never be born naked, *i.e.*, constructors must populate the object with valid data (good luck with that), and mutating methods should be restricted so they cannot create an invalid state (like leaving a person with a missing city in their address) so that methods don't have to worry about the possibility of invalid state (again, good luck with that).

Here I am watching Olympic diving, where the competitors' scores are weighted by the degree of difficulty of what they attempt, and this book is attempting an armstand back triple somersault with a half twist, and just because it ends up over-rotating on entry doesn't stop it from scoring higher than a perfectly executed reverse double somersault. That's a convoluted way of saying, this book is worth reading, especially if you use Java or Ruby. It covers numerous up-to-the-minute technologies, and does explicitly embrace the concept of polyglot programming, including examples from Groovy and Jaskell (earning a high score from the Web 2.0 judge). There's an unorthodox use for Ant (cleaning up files for trainings). There's even a thought-provoking spoiler on the mechanics of the old PacMan game.

If, however, you were expecting an exhaustive exploration of the productivity gap and blueprints for a bridge, I don't think this book is going to get you more than a little of the way there. Jon Bentley's *Programming Pearls* and Gerald Weinberg's *The Psychology of Computer Programming* have set the bar pretty high already. This book provides some contemporary embellishments of their ideas, but you still need to get those classics under your belt.

It's tempting to think that *The Productive Programmer* was written as a response to *The Pragmatic Programmer*, but aside from the alliterative parallels, I couldn't come up with any others, so I'll just have to leave that one to the conspiracy theorists.

Peter Scott is the author of Perl Medic and Perl Debugged and gives corporate trainings in Perl. He'll be teaching a full day class on dealing with legacy code at the Pittsburgh Perl Workshop this October.

Core:: values

by brian d foy

Did you know that you can override Perl's built-in functions? Even if you didn't know that, you probably already know that if a feature exists, someone is going to use it. I want to ensure that when I use a Perl built-in, I get the real one, not the overridden one.

I've been using `Parallel::ForkManager` do handle some parallel processing, Instead of thinking about all of the details of `fork`-ing, counting child processes, and the like, this module handles it for me with a couple of method calls. The `start` method creates a new child process, and the `finish` method cleans up from within the child:

```
while( 1 ) {
  if( my $pid = $pm->start ) { # parent
    }
  else { # child
    $pm->finish;
    }
  }
```

Once I got that part going, I figured I'd hook up my program to a `Tk` interface so I could get a display that tracks the progress of the processing. Once I did that, I spent the next day trying to figure out why it wasn't working. The child processes weren't exiting correctly (or at all), and they started to talk to the window manager as they re-entered the code for the parent process. When several processes try to change the same `Tk` window, `Tk` starts complaining.

After a little bit of googling, I remembered that `Tk` overrides `exit()`, and that `Parallel::ForkManager` probably uses `exit()` to terminate the child process. Since `Tk` overrides `exit()`, any bare use of `exit()` anywhere else in the program is `Tk`'s version. I need to get back to the real `exit()`. No matter what any other part of the code does, I can get to the original built-ins through the `CORE::` namespace; `CORE::exit()` is always the one documented in *perlfunc*.

Although I've submitted a report of the problem to RT (*http://rt.cpan.org/Ticket/Display.html?id=38724*), I can also fix that while I'm waiting. I could simply editing the original source, but if I upgrade the module without the fix or move my program to another machine, the fix might disappear. Instead, I settle for some monkey patching.

I load the original `Parallel::ForkManager` first so it defines everything that it will define. I can't define my method first since loading the module would simply overwrite it. Once loaded, I redefine its `finish` method to use `CORE::exit()`, solving my problem:

```
BEGIN {
package Parallel::ForkManager;

sub finish { my ($s, $x)=@_;
  if ( $s->{in_child} ) {
    CORE::exit ($x || 0);
  }
  if ($s->{max_proc} == 0) { # max_proc == 0
    $s->on_finish($$, $x,
      $s->{processes}->{$$}, 0, 0);
    delete $s->{processes}->{$$};
  }
  return 0;
}
}
```