Chad Perrin: SOB

16 August 2006

ITLOG Import: Elegance

Filed under: Cognition,Geek,Popular — apotheon @ 05:03

The following is imported from a now effectively defunct weblog of mine called ITLOG, and was a featured "Soapbox" item at TechRepublic. It is reproduced here with some modifications.

> **elegant (adj.): characterized by a lack of the gratuitous**
>
> **C.A.R. Hoare:** *"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."*

There's a long tradition of referring to the elegance of a system. In the IT industry, this tends most commonly to be applied to source code, and it is generally accepted that the more elegant it is, the better. Elegance is differentiated from other superficially good things in a number of ways, including the common assumption that elegance goes deeper, and applies more universally, while these other "good" things are only good within certain constraints.

For instance, "clever" source code is good for its cleverness, but can be bad for maintainability — mostly because clever code is often difficult to understand. Cleverness also falls short because of a simple principle famously articulated in an email signature of Brian Kernighan's: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Another example is object oriented programming. Probably ninety-some percent of the competent programmers out there are thoroughly sold on the concept that OOP is the holy grail of programming techniques, and any further advances in programming techniques are just fine-tuning OOP techniques. I think this common perception is an outgrowth of twenty years of corporate influence on the evolution of programming, where large numbers of mediocre programmers end up handling the same codebase over the course of its lifespan. Two orthogonal systems of minimizing the damage a mediocre programmer can do to a project have been introduced to programming practice with a great deal of success: version control and object oriented programming.

Object oriented programming isn't the holy grail, though. It doesn't in any way aid with the creation of truly excellent code, in and of itself. It simply aids in the avoidance of truly atrocious code, and even then only in an aggregate view of a complex project. When you start drilling down to the individual bits and pieces of a complex software project that has passed through the hands of a great many mediocre programmers, you'll start seeing atrocious bits of code that limp along just well enough to keep working, as long as they're strictly encapsulated and separated from the rest of the codebase (except for its API, of course). Encapsulation and modularity are good things in general, but they aren't immutable axioms of goodness.

One of the trade-offs with object oriented programming is that it encourages repetitive action and tedious effort in writing code. Have a look at some "enterprise" class Java source code some time and start paying attention to how much of it is actual program logic, as contrasted with how much of it is scaffolding imposed on the source by Java's object-orientedness. In fact, if you really want to understand what's going on with object oriented programming and other superficially "good" things in programming, I recommend you start comparing how easily one can produce short, elegant code in various languages, and pay attention to why one language produces a shorter, more elegant solution than another. I think you'll find some surprising facts come to light.

Of course, it's true that brevity is not strictly synonymous with elegance. In fact, Perl golf — the practice of passing code around between programmers to see how short a given algorithm can be made — is a thoroughly gratuitous sport, concerned little, if at all, with elegance. In pursuing elegance, it is more important to be concise than merely brief. In a general sense, however, brevity of code does account for a decent quick and dirty measure of the potential elegance that can be eked out of a programming language, with length measured in number of distinct syntactic elements rather than the number of bytes of code: don't confuse the number of keystrokes in a variable assignment with the syntactic elements required to accomplish a variable assignment. Armed with that definition of the term "shorter", you should be able to make some meaningful comparisons of the elegance possible when working with various programming languages.

In particular, you might notice that without using any object oriented techniques, Common Lisp and Perl produce much shorter examples of certain algorithms than Java and C++. Even if you cut out all the object oriented scaffolding in the Java and C++ examples, you still typically end up with a lot more code, as measured in discrete syntactic elements. Things like lexical variables and anonymous blocks (or, roughly equivalently, lambdas) tend to make for much simpler, more elegant solutions than imposing rigorous OOP structure. In fact, the more you examine the matter and make such comparisons, the more I suspect you'll come to realize that OOP itself has nothing to do with producing elegance, and everything to do with limiting opportunity for mediocre programmers to produce cruft and introduce bugs.

Elegance is about the gratuitous — or, rather, avoiding the gratuitous. It's true that sometimes people disagree about which of two or more things is the "most elegant", but this arises from underlying assumptions rather than any true subjectivity of the principle. Each of us has a set of operating assumptions, some greater (meaning: bloated and cumbersome) than others. Where something conforms to one's expectations and assumptions, it is seen to not lack in elegance in that manner. Someone that does not have the same underlying assumptions might see the same thing as atrociously inelegant, but having a different set of assumptions would overlook similarly subjective quirks in another example that are, to the first person, inelegant.

Specifically, someone with assumptions derived from long indoctrination by the OOP crowd might overlook all the scaffolding imposed by a language like Java for using OOP techniques, and see something that takes up 50 lines of program logic and 150 lines of OOP scaffolding as elegant. Meanwhile, a long-time Perl hacker might take one look at that and see it as the inelegant monstrosity it is. This Perl hacker, on the other hand, might write 30 lines of procedural code to perform the same task, and the Java programmer might look at it and wonder why it isn't more modular, simplifying the program logic itself and making the whole thing more scalable for future code maintenance, thus rightly seeing the inelegance of the procedural hack the Perl programmer threw together.

This doesn't make elegance subjective: it only makes our individual perspectives on it subjective. If we can discard the assumptions of both the Java developer and the Perl hacker, and recognize the underlying principles of source code design that contributed elegance to each solution, we could probably turn the same set of solutions into something much, much simpler and more elegant, in terms of its program logic and cruft-weight. Unfortunately, languages like Java are not really suited to that sort of optimization for elegance: you really need a language more dynamic than that, such as Perl, Python, Ruby, or basically any Lisp. The more a language lets you define the language you're using on the fly, the more likely it is to allow an excellent programmer to produce elegance, which should really be the end goal of writing code, generally speaking: elegant solutions.

All really useful principles of programming, or systems design in general, seem to be practical, case-specific extrapolations from my fundamental definition of elegance. In short, they all seem to boil down to this one instruction: If it's gratuitous, find a way to get rid of it. For example, consider the Pragmatic Programmers' DRY principle — Don't Repeat Yourself. In short, it is a Good Thing to avoid repetitions of data and program logic in your code. Any time you find yourself having to repeat or rephrase something in your code, reinject data into your data model from wherever you have it stored, and so on, you're screwing up. Ask yourself whether DRY is really useful by reducing repetition in and of itself, or by reducing gratuitous repetition. After all, recursion and looping behavior might also fall within the definition of "repeat yourself", but I don't think anyone (sane) would ever recommend

eliminating all loops and recursion from all programs. Sometimes, you just need your program to perform a given set of instructions on a long list of slightly different items. Often, loops and recursion make source code more elegant.

This ties in very nicely with the more general, more philosophical concept of aesthetics, and that provides some understanding of why it is possible to look at source code and, without yet consciously knowing what's wrong with it, have an immediate intuitive reaction to its inelegance. That's not to say that something can't be aesthetically pleasing without being perfect in its elegance, of course. Instead, the ability to recognize some characteristics of elegance is what leads to an aesthetically pleasing perception of the subject.

Elegance is not about aesthetics. Rather, aesthetics is about elegance. Ostentation lacks aesthetic appeal, and is inelegant (read: "tacky"), because it's gratuitous. Simplicity is often not elegant either: if something is too simple, it is nonfunctional, and fails to achieve its aim. What makes something beautiful is not strictly simplicity, symmetry, complexity, or any other such characteristic. Instead, what makes something beautiful is that its characteristics are all appropriate to its purpose. Complexity can be exceedingly beautiful, as long as it's not gratuitous complexity, which is just chaos and confusion. Likewise, simplicity can be exceedingly beautiful, but if you make something gratuitously simple, you get dullness rather than beauty. Gratuitous simplicity is merely boring.

When you're writing source code, make it elegant. When you've written something, go back and look it over, and for each and every thing you've done you should take a moment to question whether it's really necessary, or even functionally desirable, to have it in there. You probably won't get it perfect, but you can at least make it awfully pretty, which is a good thing as long as you do so by addressing elegance rather than trying to disguise the inelegance of your code by conforming to formatting conventions without rethinking your program logic at all. Refactoring, in the end, is really just about looking with fresh eyes for any opportunities to introduce elegance by removing the gratuitous.

If you're unlucky, you may discover that making your code significantly more elegant might require rewriting it in a different language.

The following definitions are from Princeton WordNet:

> **elegant (adj.): of seemingly effortless beauty in form or proportion**
>
> **gratuitous (adj.): unnecessary and unwarranted**

Comments (22)

22 Comments

1. 

   As a follow-up, here are some examples of “meaningful comparisons” as noted in the above SOB entry:

   ### C:

   ```
   #include <stdio .h>
   main()
   {
       {
           printf ("Hello World!");
       }
   }
   ```

   This program contains two or three lines of code, depending on whether you count the include statement: I do, so I’d call it three. I’m going to be kind to languages that require multiple lines and use braces to enclose blocks of code, so I won’t count the braces in this example (or in other examples, like Java). More important than the lines of code, though, is the number of syntactic elements used to achieve your aim: in this program, depending on how you define discrete syntactic elements, there are three to five syntactic elements. I’d call it four.

   ### Haskell:

   ```
   main = putStrLn "Hello World!"
   ```

   As in the Perl, Python, and Ruby examples below, this requires only one line of code. It uses only four discrete syntactic elements (I don’t know Haskell well enough to be sure it can’t be golfed down a bit), a pretty damned good showing.

   ### Java:

   ```
   class HelloWorldApp {
       public static void main(String[] args) {
           System.out.println("Hello World!");
       }
   }
   ```

   This Java program contains three lines of code. In this program, depending on how you define discrete syntactic elements, there’s anywhere between eight and twelve syntactic elements involved.

   Perl, Python, and Ruby: I’m going to include all three of these in one explanation, because the three are amazingly similar when you’re

writing a Hello World, even though what's going on behind the scenes is quite different for the three.

Perl:

```
print "Hello World!";
```

Python:

```
print "Hello World!"
```

Ruby:

```
print "Hello World!"
```

Each of these languages provides a simple, one-line program as a Hello World, with exactly two syntactic elements — no more, and no less (and half as many as the Haskell example above). Perl requires semicolons as line terminators, while in Python you can't use semicolons as line terminators, and in Ruby you can use them or not at your leisure. Ultimately, however, the point here is the two syntactic elements required by each language, without scaffolding, and (less importantly) the ability to represent the entire operation in one line of code.

UCBLogo:

```
print [Hello World!]
```

As far as I've been able to determine thus far, UCBLogo is a complete Lisp dialect, with full macro support. Logo has been described in the past as "Lisp (without the parentheses)," but to my knowledge no other subdialect of Logo than UCBLogo and its forks has gone so far as to provide macro support. As with the Perl, Python, and Ruby examples above, UCBLogo's example is comprised of exactly two syntactic elements. In this case, however, they are not a function and a string, but a function and a list of lists. The `[Hello World!]` is a list containing the lists `"Hello` and `"World!`. The programmatic flexibility and power inherent in such a construction was really rather surprising to me the first time I started looking into any Lisplike languages. While I will certainly defend the impressive opportunity for elegance in programming that Perl makes possible, despite the fact that Perl code often looks like Snoopy swearing, the language itself is in no way I've seen particularly elegant; the list-based semantic structure of Lisp dialects like UCBLogo (and others such as Common Lisp and Scheme, example code for which appears in further comments below) makes the language itself elegant. It's a beautiful thing, really. I guess you could say I'm a fan.

Visual Basic .NET:

```
Imports System
Public Module modmain
   Sub Main()
     Console.WriteLine ("Hello World!")
   End Sub
End Module
```

This program contains either five or six lines of code, depending on whether you include the Imports statement. I do, making it six. As for discrete syntactic elements, this thing has between nine and fourteen syntactic elements, depending on your definitions. I call it about eleven to thirteen, depending on my mood on a given day — after all, I'm a hacker, not a linguist. C#.NET looks similar, but actually accomplishes the same with four lines of code and about ten syntactic elements, making it an easier language to use as well as being a technically better language by far than VB.NET, despite all the "VB is easy!" marketing hype to the contrary. The only thing easier about VB is this: it's easier to write bad code, because its OOP features are crap, and its structure is still fundamentally line-oriented under the hood.

One might argue in favor of the positive value of the additional syntactic elements present in the Java and VB.NET examples, for instance, by pointing out that they are part of what makes modularity and encapsulation possible. These characteristics of the code become increasingly important as the size of your program grows. Such a defense seems plausible on the face of it, but upon closer inspection one realizes that the linguistic *requirement* for such scaffolding is gratuitous. Perl is fully capable of OOP structure, if you wish to add it in for larger projects; Python tends more toward OOP than procedural programming by far, and provides excellent tools for achieving the same effects as those within the Java example (for instance), and Ruby is so object oriented by nature that it makes my teeth ache. It is the ability to eschew the unnecessary when it is not necessary that is part of what makes for elegant code.

Comment by apotheon — 16 August 2006 @ 05:43

2. 

Glad to see you brought this one over to a real blog, apotheon. Tagged.

Comment by SterlingCamden — 16 August 2006 @ 05:51

3. 

There are a number of things in ITLOG that need to be rescued, and reposted elsewhere. Some will get cleaned up, expanded, and submitted to TR as articles (once they're in good enough shape for that). Others

will end up here. I may even end up posting one or two to PerlMonks, or to some other similar location.

Comment by apotheon — 16 August 2006 @ 06:00

4. […] SOB: Scion Of Backronymics Â» ITLOG Import: Elegance apotheon imported his landmark post on elegance from his old blog ITLOG – definitely worth reading (tags: elegance programming language oop agile) Share and Enjoy:These icons link to social bookmarking sites where readers can share and discover new web pages. […]

   Pingback by links for 2006-08-17 -- Chip's Quips — 16 August 2006 @ 07:19

5. 

   Common Lisp: (format nil "Hello World!")

   Comment by karstensrage — 17 August 2006 @ 12:56

6. 

   Excellent! Thanks for the contribution, karstensrage. I see three syntactic elements in the Lisp example, putting it well within the 2-4 range of the languages most allowing for elegance in small programs in this series of examples.

   Comment by apotheon — 17 August 2006 @ 02:06

7. 

   Scheme:

   (display "Hello, world!")

   Comment by Egg Shen — 17 August 2006 @ 06:45

8. 

   Comparing "Hello World!" implementations is silly. Clearly, by this method, HQ9+ is the superior language. http://en.wikipedia.org/wiki/HQ9+

   HQ9+: H

   On a less flippant note, the Ruby, Perl and Python examples are also executable as Boo (Boo is surprisingly nice, if you like the CLR) or Lua

code. And the Perl code doesn't require the ending ';'.

Oz:

functor import Application System define {System.printInfo "Hello Oz!\n"} {Application.exit 0} end

C: /* includes are not required */ main() { puts("Hello C!"); }

Common Lisp: (princ "Hello Lisp!")

SML: val _ = print "Hello SML!\n"

Erlang: -module(hello). -export([hello_world/0]).

hello_world()-> io:format("Hello Erlang!~n").

Clean: module hello Start = "Hello Clean!"

Factor or IO: "Hello Factor!\n" print

Here's a more academic treatment of comparing languages: http://citeseer.ist.psu.edu/felleisen90expressive.html

Comment by Huh — 18 August 2006 @ 12:24

9. 

Thanks for the Scheme, Egg.

"Huh": The Perl could be run without the semicolon, but only if you take for granted that the "hello world" print statement is the only code in the program. I was speaking in general terms, with an assumption of normal linguistic idiom as required for reasonably good programming practice in a given language to provide a fairly straightforward solution to the "hello world" problem.

As a result, certain forms of trickery and the like are in violation of the spirit of the thing.

I don't really see a big enough difference between two and four syntactic elements to write home about the impressive savings in code, so anything from Perl to Haskell seems to pretty much be in the same barrel as far as I can tell. It's when we start approaching or, even worse, exceeding a dozen syntactic elements for a simple string output operation that I start rubbing my eyes and cursing at the absurdity of it all.

Also, this is meant as something of an "all else being equal" sort of comparison. As HQ9+ is not only not Turing-complete, but can't even tie

its own shoes, “all else” is definitely **not** equal. It would win, if it weren’t disqualified. That’s a little like trying to determine the best bicyclist by seeing who can complete the Tour de France the quickest, and including a well-trained falcon in the contest that finishes it in hours rather than weeks. Sure, it would win — if it were a bicyclist.

The moral of the story: try to provide meaningful comparisons of languages’ rigid scaffolding and obtuse source code requirements rather than trying to “get away with something”.

That aside, I’m pleased you provided a boatload of code examples from other languages. Next, I think I might examine the lexical closures across languages issue. Or not.

Comment by apotheon — 18 August 2006 @ 01:10

10. 

Those hello world examples don’t say anything. HQ9+ shows this with humor.

PHP: print(“hello world”);

One-line-hello-world compliant, but i wouldn’t want to do complex tasks with PHP.

You know you are elegant, when you can delete large parts of your code. ;)

Comment by beza1e1 — 18 August 2006 @ 06:31

11. 

Nor would I (want to do complex tasks in PHP), though I have had to from time to time. PHP lacks a lot of syntactic clutter under which certain other languages labor, but its syntax is anemic, and it is burdened with an overabundance of poorly-designed core functions — and its semantic structure seems to be entirely built around the idea that if you have a function for everything, you’re okay. It’s one-line Hello World compliant, to be sure, but it is not “all else being equal” compliant.

Thanks for contributing. I actually thought about adding PHP in my original list of short Hello World programs, but forgot about it, and didn’t feel like explaining its other shortcomings at the time anyway.

Comment by apotheon — 18 August 2006 @ 11:41

12. 

    You can use semicolons as line terminators in Python

    ```
    >>> print "Hello"; print "World!"; Hello World!
    ```

    This is handy for statements that go hand in hand, like when working with Twisted Deferreds.

    Comment by Colin Alston — 18 August 2006 @ 08:48

13. 

    Just to add to what some others have said. I think a better comparison would be a simple list/array opperation. Interpreted languages, in general, start to shine when it comes to working with arrays or hash tables (dictionaries in Python).

    Comment by Colin Alston — 18 August 2006 @ 08:52

14. […] I have recently addressed the matter of elegance in programming here at SOB. Within that discussion, and in a follow-up comment to it, I explained and demonstrated the varied potential for elegance as I defined it within the designs of various programmign languages. My initial, very simple, demonstration involved merely providing a series of “hello world” programs in a number of different languages so that the amount of scaffolding cruft necessary to achieve this simplest of operations — outputting a string — could be compared between languages. It was an intentionally an extremely simple example, as the point was to demonstrate the lack of elegance necessary to achieve such simple results in some languages. This is, I believe, a very salient and useful means of comparing the elegance of code a given set of languages provide for their users. […]

    Pingback by SOB: Scion Of Backronymics » Linguistic Elegance Test: Closures and Lists — 23 August 2006 @ 04:05

15. 

    I agree that code should be as elegant as possible. But I do not agree that non OOP code will be more elegent than OOP code. It all matters on what you want to create. I also consider using only objects, like with Java, creates strange and bad solutions when it comes to actions etc. An elegant product has a mix of objects, interfaces and global procedures/functions, as they all have their places where they work best.

    Comment by Atle — 2 October 2006 @ 11:52

16. 

    I don't think anyone was claiming that non-OOP code is *necessarily* more elegant than OOP code — just that it is often so, particularly in shorter programs, because of the infrastructure overhead that is generally imposed by an object oriented approach if for no other reason.

    By the way, thanks for the comment, and welcome to **SOB**.

    Comment by apotheon — 2 October 2006 @ 12:22

17. […] Joel decides to supplement the discussion on simplicity with a "new" term: elegance. Hmm, seems like I've heard that one somewhere before, only it didn't have much to do with avoiding "intrusion into the userâ€™s actual DNA-replication goals". […]

    Pingback by Chipping the web - "a funny number" -- Chip's Quips — 17 December 2006 @ 05:16

18. […] Dan's proposed solution was to branch around this logic if the entire field was blank, but I took this opportunity to optimize this code a bit. After all, performing iterative overlay moves of the entire field by one character doesn't really jingle my elegance bells. So I replaced it with this algorithm: […]

    Pingback by Drawing a blank -- Chip's Tips for Developers — 9 January 2007 @ 04:36

19. […] Finally, the last thing on the first page of the programming reddit when I looked was 10 tips for developing deployment procedures (or: Deployment Is Development). Yes, I wrote it. In fact, it's the SOB post just before this one. I wasn't even aware it was on reddit until I saw it there tonight. I've also noticed, in the process of following links around, that both OOP and the death of modularity and The One Important Factor of programming languages are still getting some action, through links various sources. Include Elegance (which I'm almost embarrassed to admit has been called a "seminal work"), and I might have the beginning of a "Joel On Software" style of book in my future. Then again, maybe I'm just feeling too full of myself. […]

    Pingback by Chad Perrin: SOB » the goings-on in coderspace — 2 June 2007 @ 12:46

20. […] If people learn something from your site, and they use it, like it, and marvel at the simple elegance of its design, they'll probably want to hire you for related projects in the […]

    Pingback by Six ways IT consultants can build their reputation | IT Consultant | TechRepublic.com — 24 March 2008 @ 03:45

21. […] If people learn something from your site, and they use it, like it, and marvel at the simple elegance of its design, they'll probably want to hire you for related projects in the […]

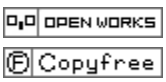
    Pingback by [Five tips for building a solid reputation as an IT consultant | Five Tips | TechRepublic.com](#) — 28 June 2010 @ [09:28](#)

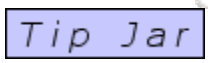
22. […] unexpected harmonies reveal themselves, and your design deals with unforeseen circumstances just as elegantly as it addresses the cases that inspired it. The satisfaction gained from that functional creativity […]

    Pingback by [The fine line between persistence and stagnation in IT consulting | TechRepublic](#) — 18 April 2012 @ [02:35](#)

[RSS feed for comments on this post.](#)

Sorry, the comment form is closed at this time.

- 

- Search for: Search
- 

- **Contact Page**
- Subscribe
  - Main RSS Feed
  - Main Atom Feed

- Pages
  - Contact/Statistics
  - Dead Letter Office
  - Images
  - Memory
    - The Bad
    - The Good
    - The Ugly
  - Off-Topic Commentary
  - privacy policy
  - public
  - Resume/CV
    - Online Publication Credits
- Categories

- Cognition
- Geek
- Humor
- inanity
- Liberty
- Lists
- Metalog
- Mezilla
- Miscellaneous
- Popular
- Profession
- Review
- RPG
- Security
- The Ebon Gate
- Tour de Lance
- Writing

- Autolatry
  - Apodictism
  - ITLOG
  - use Perl;
- Blogroll
  - A Thousand Cuts
  - All_is_onUSA
  - Ameliorations
  - Chip's Quips
  - Chip's Tips
  - Cryptome
  - IT Consultant
  - IT Security
  - Jaqui Greenlees
  - Keymaster
  - Labnotes
  - Nepotism
  - Pieces of Flair
  - Right to Create
  - Unqualified Reservations
- Hacklaw
  - Free State Project
  - Groklaw
  - Open Invention Network
- Mindhack
  - CCD CopyWrite
  - Taoism

February 2020

- **S M T W T F S**

« Jan

S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

« Jan

- Archives
  - January 2011
  - October 2010
  - September 2010
  - August 2010
  - May 2010
  - March 2010
  - February 2010
  - January 2010
  - December 2009
  - November 2009
  - October 2009
  - September 2009
  - August 2009
  - July 2009
  - June 2009
  - May 2009
  - April 2009
  - March 2009
  - February 2009
  - January 2009
  - December 2008
  - November 2008
  - October 2008
  - September 2008
  - August 2008
  - July 2008
  - June 2008
  - May 2008
  - April 2008
  - March 2008
  - February 2008
  - January 2008
  - December 2007
  - November 2007
  - October 2007
  - September 2007
  - August 2007
  - July 2007

- June 2007
- May 2007
- April 2007
- March 2007
- February 2007
- January 2007
- December 2006
- November 2006
- October 2006
- September 2006
- August 2006
- July 2006
- June 2006
- May 2006
- April 2006
- March 2006
- February 2006
- January 2006

- Meta
  - Log in
  - Entries RSS
  - Comments RSS
  - WordPress.org

*All original content Copyright **Chad Perrin**: Distributed under the terms of the **Open Works License***