

from Code Complete, by Steve McConnell:

Software and Religion

Religion appears in software development in numerous incarnations – as a dogmatic adherence to a single design-method, as unswerving belief in a specific formatting or commenting style, as a zealous avoidance of go-tos. It's always inappropriate.

Software Oracles

Unfortunately, the religious attitude is decreed from on-high by some of the more prominent people in the profession. It is important to popularize new methods before they can be fully proven. There's a difference, however, between disseminating a new methodology and selling object-structured snake oil. *"Forget everything you've already learned because this new method is so great it will improve your productivity 100% in everything."* But rather than latching on to the latest miracle fad, use a mixture of methods. Experiment with the exciting recent methods, but bank on the old and dependable ones.

Eclecticism

Blind faith in one method precludes the selectivity you need if you're to find the most effective solutions to programming problems. If software development were a deterministic algorithmic process, you could follow a rigid methodology to your solution. Software development isn't a deterministic process however. It's heuristic – which means that rigid processes are inappropriate and have little hope of success. In design, for example, sometimes top-down decomposition works well. Sometimes an object-oriented approach, a bottom-up composition, or a data-structure approach works better. *Sometimes, a few different approaches are equivalent.* You have to be willing to try several approaches, knowing that some will fail and some will succeed but not knowing which ones will work until you try them. You have to be eclectic.

Adherence to a single method is also harmful in that it makes you force-fit the problem to the solution. If you decide on a solution method before you fully understand the problem, you act prematurely. You over-constrain the set of possible solutions, and you might rule out the most effective solutions.

You'll be uncomfortable with any new methodology initially, and the advice that you avoid religion in programming isn't meant to suggest that you should stop using a new method as soon as you have a little trouble solving a problem with it. Give the new method a fair shake and the old methods their fair shakes too.

A dogmatic stance on any aspect of design and programming conflicts with the eclectic toolbox approach to software construction. It is incompatible with the attitude needed to build high-quality software.

from the coding standards document, of a local (but very large) corporation:

Go-tos, pointers, and issues of clarity

Go-tos are not to be "avoided at all costs". It is, instead, *serpentine code* that needs to be avoided. Simplicity and clarity should override most other design decisions. A go-to, in particular, is a powerful tool when used as a direct, no-nonsense jump under well-stated conditions, and can very closely follow problem-space behavior if used with some planning and forethought (Ada, a language designed from scratch by smart French people, contains a goto keyword). On the other hand, the indirection of a pointer tends to be a computer-space construct, that is often confusing and - if honesty should prevail - unnecessary (Java, the latest geek programming language, does not allow the use of pointers).

Other clarity-based suggestions:

Use case statements instead of nested ifs, use arrays instead of linked lists, optimize through solid design rather than bit-tuning, get a faster CPU instead of writing assembler, pay for the extra memory, buy code if it's available.

Strive to develop clear code

Engineers should strive to develop code that is both clear, and efficient in its use of CPU time, memory, and other resources. However, when efficiency and clarity conflict, then clarity should take strong precedence over resource stinginess, unless it is proven that using the clear but less efficient method impairs the program critically. Micro-optimizations to small areas of code are especially to be avoided if they impair clarity in any way, since it is generally only the program's overall design that affect resource utilization significantly.

Steve McConnell's coverage of the go-to debate:

<http://www.stevemcconnell.com/ccgoto.htm>