The Zero Law of Debugging or
Structured Programming—
How Far Structured?

G. Semid

The amazing statistics showing new
"peaks" in debugging to coding effort
ratios have already yielded several
so-called "Laws of Debugging," all
aimed at the ultimate goal of "quick-
as-possible" debugging. On the other
hand, formal approaches-program
verification and proofs—have been
developed, which bear the idea of a
clean, bug-free design.

Let's state an alternative approach, so
trivial that it should be called "The Zero
Law of Debugging," which states:

Don't debug! Recode!

In order to assign some meaning to
this strange statement, please direct your
attention to the paragraphs below.

If for just one moment we
programmers would lift our heads from
our never ending coding sheets and look
at the hardware environment around us,
we would see how an electrician takes
care of his system. After locating the
wrong circuit-card or whatever unit,
does he debug it? Does he try to
pin-point the exact failure? We would
probably find him replacing the entire
unit with a new, tested one.

What do we do with a malfunction in
our dear television set? We probably
"debug" it (or pay to somebody to do
it). But what do we do with our
malfunctioning pocket transistor radio?
We probably replace it.

So let's draw the conclusion from this
that for small enough units, the
replacement approach proves
itself economical.

And here we come back to our
professional environment—if we could
locate malfunctions in a small enough
module in our software, and if we were
stubborn enough to maintain a strict
structuring of our system down to the
level of that module, then recoding that
module could be less cumbersome than
debugging it. This suggestion gains weight
if the module was programmed by
someone else, or had insufficient
"comments" or "remarks" in the code,
or was coded in an unfamiliar language.

This, then, is a replacement, i.e.,
recoding, strategy as opposed to a
debugging approach. In order to make it
work, however, we must invest in some
extra organizational effort. 2

The preliminary indispensable condition
for module replacement is modular,
structured programming down to the
level of that module. In other words, we
need to be able to sketch the module as
a "black box," clearly stating its I/O
interfaces to the rest of the program and
its processes (HIP0). If we do so, we
may contribute to the current discussions
of how far to structure our software.
This leads us to a refinement of the
"Zero Law":

Structure your system to that level
where it would be economical to recode
a module rather than debug it.

In some environments such a module
may be equivalent to 50 PL/I statements,
or even 200 statements, as some experts
believe.  4 In any case, one can see that
the economical recoded module could
have either less or many more state-
ments, depending on the environment.
Computational modules may include
numerous temporary and interim
variables, and may be composed of
several nested "do loops," so that even
modules with only 20 or 30 statements
could be a considerable headache for
the reader.

The comments above on how far to
go with structuring may frequently
coincide with the standard approach that
says: "If you can code your system in a
straightforward manner, don't structure
it any further." However, it is not
necessarily an identical approach. In
recoding, some extra considerations
may be involved which are not relevant
to regular coding. These mainly concern
any cost and damage resulting from an
inoperative system.

A second necessary condition for the
recording strategy involves documenta-
tion, which should have detail sufficient
to enable a programmer to recode a
module. In the HIP0 approach a
description should define a module so
that all possible pieces of codes that
satisfy the documentation definitions will
be adequate for the parent system.
One may extend the analogy between modular hardware and software practice:

We should be prepared for software malfunctions by having alternative modules ready.

Some crucial systems can’t afford long “down” periods. Even recoding, much less classical debugging, may take too long.

Crucial systems could be equipped with alternative independent modules (coded by completely different programmers but satisfying the same requirements). So if something happened and a module was suspected of not supporting current input data, one could simply switch to an alternative module, link-edit again, and go! Some military and large commercial systems fall under the category of “crucial systems,” where double coding done without any pressure of emergency could contribute considerably to system reliability and correct long-run production.

To be practical one must admit that the “Zero Law of Debugging” cannot entirely replace the two existing approaches, classical debugging and program verification. But it can claim its fair share. Some complex modules are extremely difficult to “verify,” and some may conceal quite stubborn bugs—in such cases a fresh coding is the needed relief.

Of course the verification approach is the cleanest one, but in many practical cases one must choose either debugging or recoding. Successive debuggings turn the program into a progressively more complex patch work, while recoding tends to stream modernized coding practice into the program. The program continually improves because its modules are changed for the better.

Last to be mentioned, but not least in importance, is the positive psychological impact on programmers who face coding work rather than debugging. The “Zero Law of Debugging” then, simply turns out to be zero debugging.