Code Evasion

MOST OF US have trouble reading other people’s code, or even our own code if we haven’t looked at it for a while. The art of writing more or less self-documenting code that any competent programmer can understand effortlessly is all too rare. Why?

One reason programs tend to lose their structure and clarity as they move from concept to product is the addition of error handling. Often, more than half of a code base ends up dedicated to various types of error detection and recovery, obscuring the nominal control flow that defines the basic structure.

Obscuring the Flow
Consider a typical fragment of C code that allocates memory to a buffer, opens a file for reading, and reads data from that file into the buffer. Taking care of the possibly failing operations, that code would look something like Figure 1a. Hiding in these 14 lines of code is a small nominal execution path of just three statements, which looks like Figure 1b.

Clearly, the second version is a lot easier to read than the first, although it wouldn’t be usable in this form. Figure 1a still keeps things simple, with any error triggering a straight exit from the program. In larger programs, the error-handling code could take up a lot more space if instead we try to recover from each type of error and continue executing. By doing so, we could end up obscuring the nominal program flow even more.

Restoring Flow
If we adopt a single uniform policy for handling all errors, we can rewrite the code to clarify the nominal flow. We can do so by defining little interface functions for each potentially failing function call. For instance, for the call to the malloc routine, we can define an interface function that handles the out-of-memory condition internally, without bothering the caller (see Figure 2a). This then lets us switch to a simple call to e_malloc without having to disrupt the flow of the main code when an error occurs, as the final version in Figure 2b shows.

This approach no longer works if we want to divert the execution to alternate execution paths, depending on the type of error detected, when we’re trying to implement more complex strategies for error recovery.

For the example in Figure 1a, four execution paths are possible, depending on which (if any) operation fails. Yet, the control-flow graph appears to have eight paths; visually, that’s also the number your eyes start tracing through the code when you first look at it. This apparent flow determines the code’s apparent complexity.

The alternative version in Figure 2b still has four possible executions, but on first inspection, your eyes see just the single nominal path. We could do that in this case not by ignoring errors but by standardizing the response to them. This turns out to be an important strategy in safety-critical-system design.

Avoiding Failure
You’d be tempted to think that to make a system reliable, we must study every conceivable source of error and define detailed strategies to
handle each one, so that nominal execution can be restored in all cases. This is only partly true. It’s true that before we build a system, we must have a good understanding of how it might fail under a broad range of conditions. As civil engineer and author Henry Petroski recently said, “The short definition of engineering is the avoidance of failure.”

But this doesn’t mean that reliable code is doomed to be inscrutable. As I showed before, it’s not the handling of errors in itself that necessarily complicates code structure, it’s how the errors are handled. The safest method isn’t always to devise elaborate mitigation strategies that can return the system to nominal execution under all imaginable circumstances. Sometimes the best strategy is to punt and merely move the system into a known state, so that an external user or operator can diagnose the problem with the benefit of a broader perspective. Doing so can reduce a system’s complexity, which can make the system safer.

Unmanned spacecraft contain algorithms that continuously check the system’s health, hunting for any type of anomalous condition. When an anomaly is detected, the simplest of these algorithms make no attempt to fix the problem. Instead, they place the spacecraft into a known safe mode, so that operators on the ground can diagnose the problem and devise a solution.

Safe mode is a system state with only the minimal functionality needed for a spacecraft to remain commandable, and therefore repairable. Stripping away the code needed to self-diagnose and repair all foreseen and unforeseen problems autonomously can significantly reduce the overall complexity (and improve the predictability) of system execution.

This approach to system safety has deep roots. When the software for the first moon landings was prepared at MIT in the mid ’60s, uncontrolled complexity seemed to be putting the Apollo missions’ time schedule at risk. NASA manager Bill Tindall reported in detail on the problems and how they were being addressed in his “Tindallgrams”—short updates he sent with some frequency to NASA mission planners in Houston. In his first Tindallgram from 31 May 1966, he wrote,

> I am still very concerned about unnecessary sophistication in the program and the effect of this “frosting on the cake” on schedule and storage. It is our intention to go through the entire program, eliminating as much of this sort of thing as possible, I am talking about complete routines, such as “Computer Self-checks”...".

It seems counterintuitive that error-handling code would end up being the culprit of so much system complexity. But note that this code generally aims to defend against the highly unpredictable types of errors and failures that real life can throw at us. If that’s almost doomed to failure, the safest strategy might be to practice extreme frugality.

Hey, Reboot!
The original design of Unix also avoided elaborate error-handling code to reduce complexity, which might be partly why it so quickly became popular with developers. Early versions of the Unix kernel source code are legendary among programmers for their clarity and readability, even inspiring the well-known line-by-line exegesis by John Lions. A developer on another project, Tom Van Vleck, once described a discussion he had with Dennis Ritchie about the difficulty of error handling. He said, “I remarked to Dennis that easily half the code I was writing in Multics was error recovery code.” Dennis answered, “We left all that stuff out. If there’s an error, we have this routine called panic, and when it is called, the machine crashes, and you holler down the hall, ‘Hey, reboot it.’”

The simple strategy to reboot a misbehaving system is familiar to anyone who has used a computer, no matter what OS it runs, or any device that uses software, for that mat-

```c
void *
e_malloc(size_t n)
{
    void *ptr = malloc(n);
    if (!ptr) {
        fprintf(stderr, “out of memory\n”);
        exit(1);
    }
    return ptr;
}

buf = e_malloc(N);
f = e_fopen(fnm, “r”);
e_read(fd, buf, N);
```
The strategy is fine and works most of the time, unless of course the error is hiding in the reboot code itself. This scenario might sound far-fetched, but it does occur. It indirectly caused the infamous Sol-18 problem that struck the first of two Mars Exploration Rovers shortly after it successfully landed on Mars in January 2004. Correctly diagnosing and fixing a problem such as this can be a true challenge even for humans back on Earth, as Glenn Reeves and Tracy Neilson documented in nail-biting detail. 5

Defect Rates
It’s worth considering yet another point that can make us extra cautious in writing elaborate error-handling code. Error-handling code by its very nature executes infrequently, not just in system operation but also in system testing. Naturally, code that’s not tested thoroughly tends to have a larger fraction of residual defects. This means that if we write large amounts of code to recover from obscure error conditions, that error-handling code will likely contain more defects than the code it’s trying to protect. Hitting one of those new defects while recovering from an earlier error can aggravate the situation and put a system in a state that’s even harder to diagnose and repair.

Faulty Fault Protection
I noted earlier that fault protection systems can increase complexity and introduce entirely new failure modes into a system. An easy noncomputational example of this phenomenon is the familiar “Always Alert, Nobody Hurt” sign you can trip over in the dark.

An even better example dates back about two thousand years. In Roman times, marble columns were key components of most monumental buildings. The best columns were cut from a single piece of stone and transported to the worksite, where they had to be stored (horizontally) until they could be used.

For storage, the columns would sometimes be placed on small supports, which could prevent discoloration from prolonged contact with the ground. This, however, could cause long columns to break in the middle, perhaps just by the sheer force of gravity, or helped by playing Roman children who would no doubt jump up on the columns.

To prevent the breakage, a clever builder thought of placing an extra support at each column’s midpoint, where breaks typically occurred. Ironically, this protective measure could introduce a new failure mode that was much more likely to strike. The added support could cause the column to break in the middle, this time in the opposite direction. Because a straight line goes through two points and not three, the fault protection increased the odds of breakage instead of reducing it. Figure 3, from a book by Galileo that first appeared in 1638, illustrates this problem.

FIGURE 3. Fault protection can introduce new failure modes. 6 Instead of protecting the column, adding a third support could add another reason for the column to break in two.
The lesson in all this is that unless the cause of an error is well understood, well-intended remedial actions could actually make matters worse. In those cases, doing nothing might well turn out to be the better strategy. Doug McIlroy, the head of the department at Bell Labs where Unix, C, and C++ were born, once phrased it as, “The real hero of programming is the one who writes negative code” (www .azquotes.com/quote/819506).

The challenge in writing reliable code is similarly to find ways to remove code from an application by simplifying and generalizing, rather than continuing to add more. After all, the only code that can’t fail is the code that isn’t there to begin with.

References
6. G. Galilei, Discourses and Mathematical Demonstrations Relating to Two New Sciences, 1638.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.

IEEE Software