Code Evasion

Gerard J. Holzmann
Jet Propulsion Laboratory, California Institute of Technology

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 can be understood effortlessly by any competent programmer is all too rare. Why is that?

One reason why programs tend to lose their structure and clarity as they move from concept to product is the addition of error handling. It is not unusual to find that more than half of a code base ends up being dedicated to various types of error detection and recovery, obscuring the nominal flow of control that defines the basic structure.

Consider the following typical fragment of C code, that first allocates some memory to a buffer, then opens a file for reading, and finally reads data from that file into the buffer. Taking care of the possibly failing operations, that code would look something like what is shown in Figure 1.

```
if ((buf = malloc(N)) == NULL) {
     fprintf(stderr, "out of memory\n");
     exit(1);
}
if ((fd = fopen(fnm, "r")) == NULL) {
     fprintf(stderr, "cannot open %s\n", fnm);
     exit(2);
}
if (read(fd, buf, N) != N) {
     fprintf(stderr, "read error\n");
     exit(3);
}
```

Figure 1. Standard Code Obfuscation

There are 14 lines of code here, and hiding inside is a small nominal execution path of just three statements, which looks like the code fragment in Figure 2.

```
buf = malloc(N);
fd = fopen(fnm, “r”);
read(fd, buf, N);
```

Figure 2. The nominal path from Figure 1.

Clearly the second version is a lot easier to read than the first, although it would not be usable in this form. In Figure 1, I still kept 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 obfuscating the nominal program flow much better still.

Restoring Flow

If we adopt a single uniform policy for handling all errors, it is possible to rewrite the code in a way that restores the nominal flow quite well. We can do so by defining little interface functions for each of the potentially failing function calls. 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, as shown in Figure 3.

```
void *
e_malloc(size_t n)
{ void *ptr = malloc(n);

  if (!ptr) {
     fprintf(stderr, “out of memory\n”);
     exit(1);
  }
  return ptr;
}
```

Figure 3. Simple interface function for `malloc`.

This then lets us switch to a simple call to emalloc without having to disrupt the main flow of the main code when there is an error, as shown in the final version of this fragment in Figure 4.

```
buf = e_malloc(N);
fd = e_fopen(fnm, “r”);
```

```
e_read(fd, buf, N);
```

Figure 4. The cleaned up code fragment.

This approach no longer works of course if we want to divert the execution to alternate execution paths, depending on the type of error that happened, when we want to attempt not just error detection but also error recovery.

For the example from Figure 1, there are four possible execution paths, depending on which operation fails (if any). Yet, in the control-flow graph of the code it would appear that there are eight paths, and visually that's also the number that your eyes start tracing through the code when you first look at it. The complexity of the code is determined by this apparent flow. In the alternative version shown in Figure 4 there are still four possible executions, but on first inspection your eye sees 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

One would be tempted to think that to make a system reliable we have to study every conceivable source of error and define detailed strategies to handle each one, so that nominal execution can be restored in all cases. But, this is only partly true. It is true that before we build a system we must have a good understanding of how it might fail under w broad range of conditions. As civil engineer and author Henry Petroski recently said: "*The short definition of engineering is the avoidance of failure*." [3]

But this does not mean that reliable code is therefore doomed to be inscrutable. As we showed above, it is not the handling of errors in itself that necessarily complicates code structure – it is the way in which the errors are handled. The safest method is not 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 the complexity of a safety critical system, which can make the system safer.

Unmanned spacecraft, for instance, contain algorithms that check the health of the system continuously – 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 are designed to place the spacecraft into a known "safe mode," so that operators on the ground can diagnose the problem and come up with a solution.

Safe mode is defined as a system state with only the minimal functionality that is needed to remain commandable, and therefore repairable. Stripping away the code that would be 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 sixties, uncontrolled complexity also seemed to be putting the time schedule for the Apollo missions at risk. NASA manager Bill Tindall reported in detail on the problems and how they were being addressed in his so-called "Tindalgrams" – short updates that he sent with some frequency to NASA mission planners in Houston. In his first Tindalgram from May 31, 1966, he wrote [7]:

> *"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 counter-intuitive that error handling code would end up being the culprit of so much system complexity, but note that the purpose of this code is generally to defend against the highly unpredictable types of errors and failures that real-life can throw at us. If that is almost doomed to failure, the safest strategy may be to practice extreme frugality.

Hey, Reboot!

The strategy of avoiding elaborate error handling code to reduce complexity was also used in the original design of the Unix operating system, and may be part of the reason why it became popular among developers so quickly. 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 [1].

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's answer was: *"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.'"* [2].

(sample illustration from: http://seandenigris.com/blog/wp-content/uploads/2010/11/panic-button.jpg)

The simple strategy to reboot a misbehaving system is familiar to anyone who has used a computer, no matter what operating system it runs, or any device that uses software for that matter. The strategy is fine, and works most of the time, unless of course the error is hiding in the reboot code itself. This may sound like a far-fetched scenario, 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 like this can be a true challenge even for humans back on

Earth, as was nicely documented by Glenn Reeves and Tracy Neilson in [4].

Defect Rates

There is yet another point that is worth considering, and that can make use extra cautious in writing elaborate error handling code. Error handling code is by its very nature executed infrequently, not just in system operation but also in system testing. Naturally, code that is not tested very thoroughly tends to have a larger fraction of residual defects. This means that if you write large amounts of code to recover from obscure error conditions, that error handling code is likely to contain more defects than the nominal code it is 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 is now 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 example of this phenomenon is the familiar “Always Alert, Nobody Hurt” sign that 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.

Until they were used, the columns would sometimes be placed on small pilons, which could prevent discoloration from prolonged contact with the ground. This, however, created a new problem that 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 pilon at the midpoint of each column, where breaks would typically occur. The irony is now that this protective measure could itself introduce a new failure mode, that was much more likely to strike. The added pilon could again cause the column to break in the middle, but

this time in the opposite direction. Since a straight line goes through two points and not three, the fault protection increased the odds of breakage, instead of reducing it. The problem is illustrated in Figure 5, from a book by Galileo which first appeared in 1638. [5]

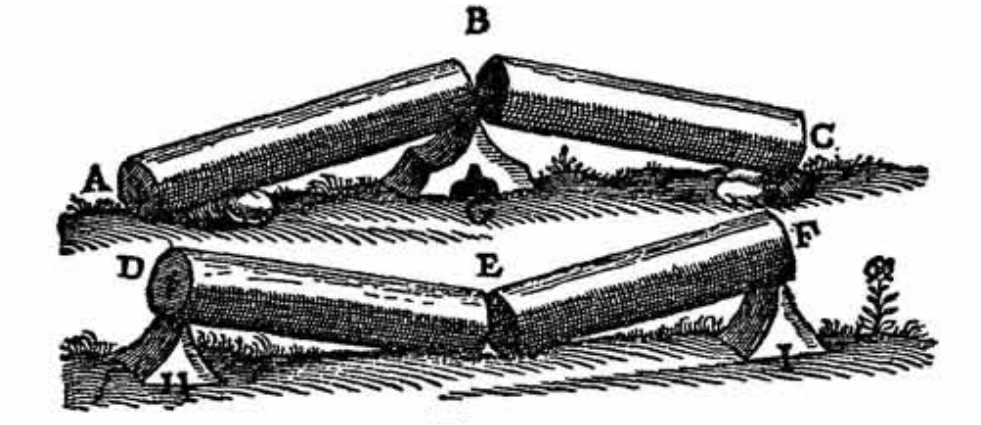

Figure 5. Fault protection can introduce new failure modes. *[public domain image, there are many copies of this, e.g. Fig. 29: http://oll.libertyfund.org/titles/753]*

Negative Code

The lesson in all this is that unless the cause of an error is well understood, well-intended remedial actions could well make matters worse instead of better. In those cases doing nothing may well turn out to be the better strategy. Doug McIlroy, the head of the department at Bell Labs where the Unix operating system and the C programming language were born, once phrased it as follows: *"The real hero of programming is the one who writes negative code."* [6]

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 cannot fail is the code that isn't there to begin with.

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] J. Lions, "Commentary on Unix' 6th Edition, with source code," Lecture Notes, University of New South Wales, 1976, republished by Peer-to-Peer Communications in 1996.

[2] T. van Vleck, http://www.multicians.org/unix.html

[3] R. Latanision and C. Fletcher, "An interview with Henry Petroski," in: *The Bridge*, Vol. 45, No. 1, National Academy of Engineering, Spring 2015, pp. 49-55.

[4] G. Reeves and T. Neilson, "The Mars Rover Spirit Flash Anomaly," Proc. IEEE Aerospace Conference, Big Sky, MT, 2005, pp. 1-14.

[5] Galileo, *Discourses and Mathematical Demonstrations Relating to Two New Sciences*, Leiden, 1638.

[6] http://www.azquotes.com/quote/819506

[7] H.W. Tindall Jr., http://web.mit.edu/digitalapollo/Documents/Chapter7/tindallgrams.pdf

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.