# Faults of Omission

Brian Marick
www.testing.com
marick@testing.com

First published in *Software Testing and Quality Engineering* Magazine, January 2000
(www.stqemagazine.com)

I don't know if this is a true story, but it's truly a story I've heard. A new jet fighter was being tested. The test pilot strapped in, turned the key in the ignition (or the equivalent), and flipped the switch to raise the landing gear. The plane wasn't moving, but the avionics software dutifully raised the landing gear. The plane fell down and broke.

I haven't seen the actual code, but it's reasonable to assume that it looked like the pseudocode in Figure 1. The bug is that some code is missing—code that in Figure 2 is shown underlined and in a different color.

Figure 1:
The buggy code

```
Loop forever:
  fetch new command or event
  if command is 'raise landing gear'
    turn on hydraulics (or whatever)
  else if command is something else
    do something else
  else if …
```

Figure 2: Code that corrects the bug (but other bugs may remain, since there may be other reasons not to raise the landing gear)

```
Loop forever:
  fetch new command or event
  if command is 'raise landing gear'
    if the plane is on the ground
      issue an error message
    else
      turn on hydraulics (or whatever)
  else if command is something else
    do something else
  else if …
```

This type of bug—one that is corrected by adding code that someone left out—is called a *fault of omission*. Numerous studies have shown it to be a common fault in production code.

In 1990, I studied bug fixes posted to USENET newsgroups. By comparing the fixed and original versions of the code, I could identify the underlying fault. 47% were faults of omission. A further 23% were complex faults, some of which could be said to contain a fault of omission. More recently, Roger Sherman, formerly of Microsoft, reported that 30% of bugs in one product were due to missing code. In the appendix, I cite five papers by other people. In their products, they found that from 22% to 54% of faults were omissions.

Because they're so common, I'm obsessed with faults of omission; and I think you should be too. We should think about how to prevent them, and how to detect them when prevention fails.

How? As Jerome Kagan has written in *Three Dangerous Ideas* (p. 2), "Much of our progress in the study of nature has occurred because investigators analyzed abstract concepts and replaced them with families of related but distinct categories." We need to do that with "fault of omission" by examining lots of examples, discovering natural categories, and thinking hard about how to handle them. Let's take a stab at that, even though our categories will inevitably be imperfect and fuzzy.

I'll choose three categories, which could be roughly labeled "coding omissions," "design omissions," and "requirements omissions." I resist those terms, though, because my experience is that they just lead to big arguments over what the first of each pair of words means, and I want to concentrate on the second word.

### *Failure to handle code details*

Sometimes the programmer's mistake is caused by details of specific code. For example, programmers often read textual data into fixed-size buffers without anticipating that the data might be too big to fit. When the data overflows the buffer, bad things happen. Such faults are a major cause of security problems on the Internet. (See Bob Johnson's Bug Report in *Software Testing and Quality* Engineering Magazine, January 1999.) Here's an example of a corrected buffer overrun bug:

```
fscanf(f, "%99s", command_buffer);
```

A version without the "99" would have no checks on the length of the input. The corrected version cuts off the input at 99 characters. That may not be ideal if the input was legitimately long, but it's less likely to be catastrophic.

One fault of omission in my USENET survey was due to memory allocation. Some ways of allocating memory initialize it to zero; because of that, it's easy to fall into the trap of assuming that *all* of them do. The faulty code allocated a `gdbm_file_info` structure in a way that left it full of garbage. The garbage was then processed as if it were meaningful data. The code was corrected by zeroing the structure just after it was allocated. The new code looks like the following, where `dbf` is the location of the new memory:

```
bzero((char *)dbf, sizeof(gdbm_file_info));
```

Bugs like these are not prevented by any abstract theory of good programming or good design. They're prevented when programmers are trained to think specific thoughts like "Always double-check that allocated memory is initialized" and "Never, never, never read into a buffer without guarding against overruns." When the code is important, you should supplement prevention with reviews by other programmers trained to think the same thoughts.

Thinking is easier when checklists are written down. It would be helpful if documentation for library functions were explicit about potential faults of omission. ("Watch out for buffer overruns!" in flashing red letters would be nice.) But they often aren't. Omissions are easy to make because the documentation is obscure about special cases, especially error cases.

But why force people to avoid errors that are begging to be made? Instead of relying on programmers to zero memory, have every memory allocation routine do it for them. Trade an occasional performance penalty for greater reliability. Error proofing, sometimes called "poka-yoke," is not always feasible. But it's used less often than it should be.

When prevention fails, you test. But conventional "black box" testing that doesn't look at the code is weak at finding these code-specific omissions. For many of these bugs (such as problems with memory allocation), there's no clue in the external interface or documentation that would prompt you to try a test that would fail. For others, there is. You could search for buffer overruns by looking for all places where you might enter variable-length data. But code inspections are a better use of your money.

Code-aware "clear box" testing isn't useful for finding faults of omission. When you see the call to fscanf and wonder about buffer overruns, you don't run a test—you simply check if the code to handle long inputs exists. (If the code does exist, you might run a test to check whether it works, but that's searching for a different fault.)

### *Mistakes applying programming clichés*

Now let's move away from the specifics of code to a slightly more abstract realm. When programmers write code that searches a collection of objects, they sometimes overlook special cases like these:

- The object being sought doesn't appear.
- The object being sought appears twice.
- The collection doesn't contain any objects to search through.

Notice that these mistakes are independent of code. It doesn't matter whether the search is done by library routines like (in *C*) bsearch and strrchr, or by hand-crafted searching code. That's why I say we're in a more abstract realm—we're dealing with abstract ideas like "searching" and "collection."

This has two practical consequences. First, it's an additional source of checklists. (The appendix points you to one.) As a programmer writing code, or a reviewer reading code, I can ask, "Can I interpret this code as doing a search?" If so, I refer to a checklist of special cases that the code should handle. I used the word "interpret" deliberately. Sometimes you can say, "If I look at the program from *this* perverse point of view and consider these two fairly unrelated operations, I could argue that there's a sort of a search going on. What, then, would happen if the object being 'sought' appears twice?" In my experience, perverse interpretations find important bugs. That makes sense: programmers

are more likely to overlook the special case if they don't realize that the general situation applies.

The second practical consequence is that black box testers do better at finding this kind of omission than the previous kind, which was so tightly tied to specific code constructs. Someone familiar with the internals is likely to unconsciously reject the notion that searching applies—after all, there's no searching code. Black box testers are not hampered by the same misleading knowledge.

(Please note that suggestions for handling the three types of omissions are summarized in a table in the appendix. I've also added a few others that don't fit here.)

### *Misunderstanding the environment*

A third type of fault of omission has nothing to do with either specific code or abstract programming concepts. It's what I call the "who would want to do *that*?" fault. The fighter fault was of this type. Programmers and designers build products to help people. Designers know something of what those people do before they have the product, but never enough. And designers' predictions of how the product will change people's behavior are always wrong. So, armed with this incomplete knowledge, programmers create a product. Having done that, they find it very difficult to break free from what are now ingrained assumptions. They don't anticipate other uses. When the users (or other actors in the environment, like other products on the same machine) behave in unexpected ways, that uncovers faults of omission.

Better up-front knowledge—better requirements analyses, more thorough and detailed user scenarios—will surely help. The state of the practice is shockingly bad. But even after it's improved, I believe the black box tester has a crucial role to play. Why? We humans are a species that manipulates—we learn by picking things up, shaking them, and seeing what happens. Using programs is eye-opening: so much that was hidden becomes obvious. Some of the things that become obvious are ways to use the program that the designers never intended—"Hey, that operation takes a long time, so why don't I browse over here while I'm waiting?…" <crash> Black box testers—if they have some domain expertise and if they pay attention to how people behave—are a concentrated and efficient way of churning through odd but inevitable uses.

The fault of omission is not your friend, but it will be your constant companion. Learn to keep it from dragging you down.

*I thank James Bach, Danny Faught, Payson Hall, Douglas Hoffman, Noel Nyman, Bret Pettichord, Johanna Rothman, Carol Stollmeyer, and Ned Young for critiquing a draft. Alyn Wambeke copyedited the final draft.*

## *Appendix*

Here are the three (admittedly rough) categories of faults of omission, together with comments about preventing or detecting them.

| | **Handling Specific Code (buffer overruns, etc.)** | **Applying Programming Abstractions (collections, searching, etc.)** | **Understanding the Product's Environment ("Who would want to do that?")** |
|---|---|---|---|
| Inspections | Yes, especially if based on checklists | Yes, especially if based on checklists. Inspections of design and other "upstream" documents can prevent omissions. | Inspections of the code are rarely useful. Inspections of requirements documents or use cases are definitely useful. |
| Code-aware testing | Inspections are better | Yes. Inspections are probably better. | Rarely useful. |
| "Black box" testing | Problems will for the most part be found by dumb luck. There are better ways. | Yes. The independent point of view will find bugs. | Yes, especially if testers have domain knowledge or are good at putting themselves in the shoes of the users. |
| Miscellaneous | Don't use predefined data structures and functions that invite error | | Do a better job of requirements analysis.<br>Zero in on undocumented and unexamined assumptions. |

---

The checklist mentioned in the article can be found at http://www.testing.com/writings/catalog.pdf. It is an improved version of the one in my book, *The Craft of Software Testing*.

---

Bibliography:
My Usenet survey was published in "Two Experiments in Software Testing," Brian Marick, University of Illinois Computer Science department report UIUCDCS-R-90-1644, November 1990. Sorry, the electronic version is gone.

More readily available are:

- Victor R. Basili and Barry T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984.
- Victor R. Basili and H. Dieter Rombach, "Tailoring the Software Process to Project Goals and Environments," in *Proceedings of the 9th International Conference on Software Engineering*, IEEE Press, 1987.
- Victor R. Basili, E.E. Katz, N.M. Panlilio-Rap, C.L. Ramsey, and S. Chang, "Characterization of an Ada Software Development," *IEEE Computer*, September, 1985.
- Thomas J. Ostrand and Elaine J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment," *Journal of Systems and Software*, Vol. 4, 1984.
- Robert L. Glass, "Persistent Software Errors," *IEEE Transactions on Software Engineering*, March 1981.

I first heard the Microsoft number in Roger Sherman's talk titled "Software Testing: Can We Ship It Yet?" at Quality Week '96. James Tierney (in a personal communication) reminded me of the exact number. It is from a project postmortem not available outside of Microsoft.

I talk about the implications that faults of omission have for code coverage in "How to Misuse Code Coverage" at http://www.testing.com/writings/coverage.pdf.

To find out about Poka-Yoke, go to "John Grout's Poka-Yoke Page" at www.campbell.berry.edu/faculty/jgrout/pokayoke.html. There are some neat examples. (The 3.5" floppy is slightly asymmetrical so that you can't put it in incorrectly.) Harry Robinson wrote a nice paper on its application to software, "Using Poka-Yoke Techniques for Early Defect Detection," www.campbell.berry.edu/faculty/jgrout/pokasoft.html.