CHAPTER

Fixing the Defect 15

Once we have understood a failure's cause–effect chain, we know how the failure came to be. Still, we must find the place where the infection begins—that is, the actual location of the defect. In this chapter, we discuss how to narrow down a defect systematically—and having found the defect, how to fix it.

15.1 LOCATING THE DEFECT

Section 9.5 in Chapter 9 discussed a general strategy for narrowing down infection sites:

1. We start with the infected value that defines the failure (Figure 15.1a).
2. We determine the possible origins of the infected value, following dependences in the source code (Figure 15.1b).
3. Using observation, we check each single origin to determine whether it is infected or not (Figure 15.1c). Having found the earlier infection, we restart at step 2.

This loop goes on until we find an infection with origins that are all sane. The code producing this infection is the defect.

Although this process is guaranteed to isolate the infection chain, it is pretty tedious—especially if you consider the space and time of a simple program execution. This is where the induction and experimentation techniques discussed in Chapter 16 come into play. However, although these techniques can determine causes (or at least anomalies that correlate with failure) they cannot tell where the defect is—simply because they have no notion of correctness. Therefore, we must combine induction and experimentation with *observation* such that the programmer can tell (or specify) what is correct or not—and eventually fix the program. Our key question is:

HOW DO WE ACTUALLY LOCATE AND FIX THE DEFECT?

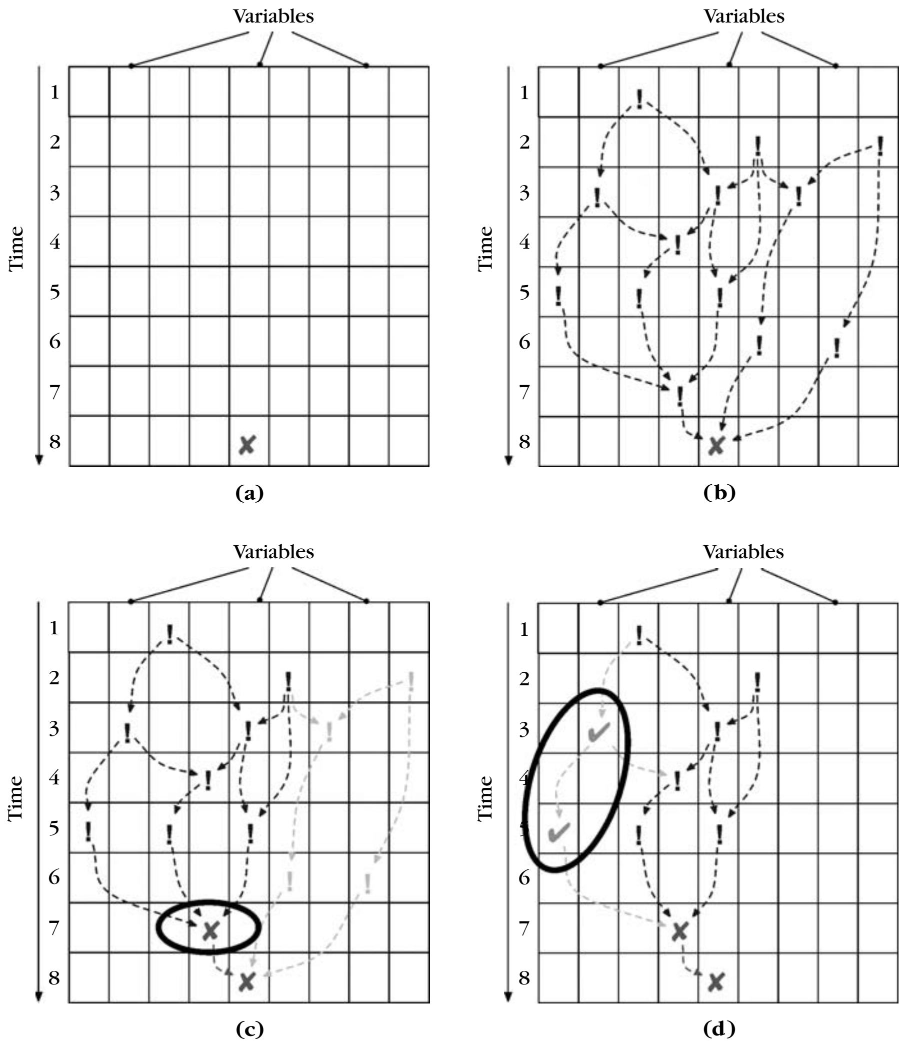

FIGURE 15.1

Narrowing down a defect: (a) the starting point, (b) following dependences, (c) observing state, and (d) asserting an invariant.

15.2 FOCUSING ON THE MOST LIKELY ERRORS

In the previous section, we resumed the general strategy for locating the defect along the infection chain. It turns out that *induction* and *experimentation* techniques nicely fit into this strategy. The key is to use them to *focus* on specific origins. Whenever we have a choice of multiple origins (or, more generally, hypotheses), we can use automatic induction and experimentation techniques to help us focus on the most likely origin.

As an example, let's reexamine the situation shown in Figure 15.1(c) and continue to locate the defect.

Assertions (Chapter 10) ensure data sanity over a long moment in time and a wide range in space. Any *failing* assertion by definition signals an infection. Of course, this is something we must focus on. On the other hand, whatever is covered by a *passing* assertion need no longer be considered. In our example, we can use an assertion to rule out possible infection origins—simply because the assertion guarantees that the state is sane (Figure 15.1d).

Anomalies (Chapter 11) are aspects of the execution the properties of which are correlated with failure, such as coverage (Section 11.2) or dynamic invariants (Section 11.5). Because of the correlation, it is wise to focus on such anomalies first.

In Figure 15.1(d), we still have the choice between two origins. Observing the one that is abnormal reveals an infection (Figure 15.2a).

Causes (Chapters 13 and 14) are aspects of the execution (such as input, state, or code) that are not only correlated with failure but actually cause the failure, as experimentally proven. Therefore, causes are even more likely to indicate the defect than anomalies.

In Figure 15.2(b) we have found a cause transition—a statement that causes the failure. As the origin is sane and the target is infected, we have a real defect here—and the complete infection chain.

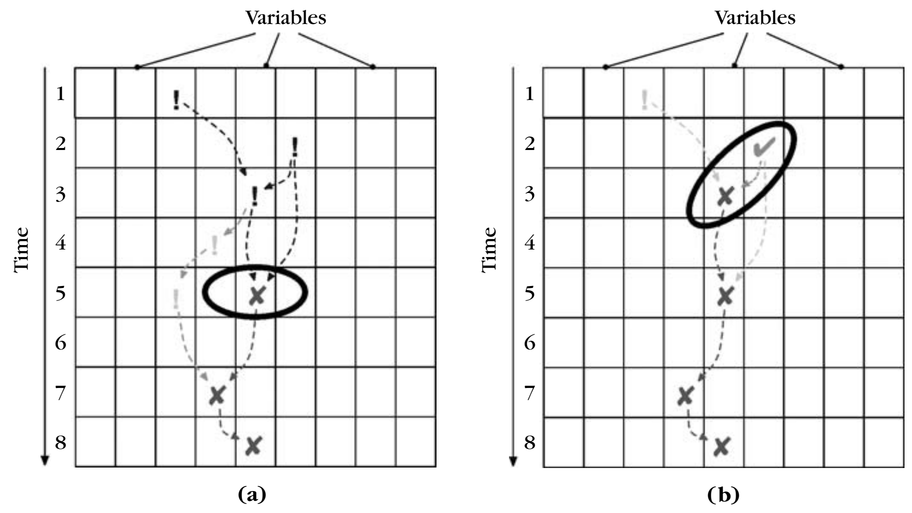

FIGURE 15.2

Narrowing down a defect: (a) anomalies and (b) cause transition.

Although these techniques can help us to focus on specific origins, we still do not know which technique to choose. Starting with those techniques that are most likely to find the defect, the following is our list:

Focus on infections. If you already know that some origin is faulty—from a failing assertion or via observation—focus on this one first and check whether the infection causes the failure. Typically, though, we do not know which of the origins is infected, and thus have nothing to focus on. Therefore, our priority goes to available *automated* techniques (following).

Focus on causes. If delta debugging or any other form of experimentation has highlighted some state or input as a failure cause, focus on these causes and check whether they are infected.

Focus on anomalies. Otherwise, of all possible origins, those that are associated with anomalies are more likely to contain errors. Focus on these and check whether they are infected and cause the failure.

Focus on code smells. Otherwise, if you have determined code smells in your program (see Section 7.5 in Chapter 7), *and* if one of these code smells is a possible origin, first focus on the code smell and check whether it causes infection and/or failure.

Focus on dependences. Otherwise, anything that is not in the backward slice of the infected state cannot possibly have caused the infection. Of all possible origins, check the backward slice for infections, starting with the closest statements.

"Cannot possibly" in fact means "cannot legally." Your program may well find a way to break the rules and use undefined behavior, as discussed in Section 7.6 in Chapter 7. This can be prevented by system assertions (see Section 10.8 in Chapter 10) or checking for code smells (see Section 7.5 in Chapter 7).

These rules constitute the "focus on likely origins" step in the TRAFFIC strategy from List 1.1. Each potential origin must then be verified whether it is infected or not, and we repeat the process for the infected origin.

Fortunately, we need not identify every single bit of the infection chain, as we are only interested in its origin. Therefore, we make larger *gaps*—for instance, toward the *boundaries* of functions or packages. These are places where communication is restricted (typically, to function arguments), which makes it easier to assess whether the state is sane or not.

If we find that some state is sane, we need not consider earlier states. Instead, we search forward for the moment in time the infection takes place. Eventually, we will find some piece of code where the state is initially sane but is infected after execution. This is the place where the infection originates—that is, the actual defect.

15.3 VALIDATING THE DEFECT

In the focusing rules in Section 15.2, I have constantly emphasized that whenever we focus on a potentially erroneous origin we must also check whether it actually causes the failure. Why is that so? This is simple: Finding an error is not enough; we must also show that the error causes the failure. When tracing back the infection chain, we must show at each step that:

- The origin *is infected*—that is, that the variable value is incorrect or otherwise unexpected.
- The origin *causes the infection chain*—that is, that changing the variable value makes the failure (and the remaining infections) no longer occur.

Let's briefly examine why both of these steps are necessary.

15.3.1 Does the Error Cause the Failure?

Why do we have to show causality for an infection origin? The first reason is that if we find an origin that is infected but does not cause the failure we are being put on the wrong track. We risk a *post hoc ergo propter hoc* ("after this, therefore because of this") fallacy, as discussed in Section 12.2 in Chapter 12. As an example of being put on the wrong track, reconsider the example from Section 12.3 in Chapter 12.

```
a = compute_value();
printf("a = %d\n", a);
```

Because the program outputs `a = 0`, we assume that `compute_value()` produces an infection. However, we have not shown that `a` being zero causes the program to output `a = 0`. Indeed, if we change `a` to `1` the program still outputs `a = 0`. Therefore, we know that `a` does *not* cause the output.

As we found in Section 12.3 in Chapter 12, the `printf()` format is wrong. The program outputs `a = 0` for most values of `a`. Without verifying the cause, we might have gone for a long search to determine why `a` could possibly have become zero.

Being put on the wrong track is especially dangerous when dealing with "suspicious" origins—variables where we cannot fully tell whether their values are correct or not. Before following such a scent, you should ensure that the origin actually causes the error—for instance, by replacing its value with a nonsuspicious one and checking whether the failure no longer occurs.

15.3.2 Is the Cause Really an Error?

The previous section discussed errors that are not failure causes. Let's now turn to another source of problems: failure causes that are not errors.

Breaking the infection chain for a particular failure is easy. You simply check for the infected value and fix it for the run at hand. The issue, though, is to break the cause–effect chain in such a way that we prevent *as many failures as possible*. In short, we want our fix to actually *correct* the program.

The following is an instance of a fix that makes a failure no longer occur, but nonetheless fails to correct the program. A loop adds up the balance for a specific account.

```
balance[account] = 0.0;
for (int position = 0; position < numPositions; position++)
{
    balance[account] += deposit[position];
}
```

It turns out that the sum for account 123 is wrong, and thus we "fix" it by including:

```
if (account == 123)
    balance[123] += 45.67;
```

Likewise, for some reason, some people do not get their savings bonus:

```
if (account == 890 && balance[account] >= 0)
    balance[account] *= 1.05;
```

These "fixes" are wrong because they do not correct the program. *They fix the symptom rather than the cause*. The origin of the infections may well be in the original claim amounts, which must be investigated.

The following is a less blatant example. Consider once more the `sample` program from Example 1.1. Assume I have no real clue why the program fails. As I always have trouble with loop boundaries, I suspect the number of loop iterations is off by one. Thus, I replace the `for` loop (in line 16)

```
for (i = h; i < size; i++)
```

with

```
for (i = h; i < size − 1; i++).
```

Does this help? Yes, it does:

```
$ sample 11 14
Output: 11 14
$ _
```

This clearly proves that the loop header caused the failure. I may have no clue why it was wrong, but at least the program now works. Did I really correct the program? I don't know. What we have here is a case of ignorant surgery.

Such a "fix" is even worse than the one described earlier. I have changed the program to make it work, but I actually have no clue how it works. The actual defect that still lurks in the code is likely to produce similar failures in the future. Worse, with my "fix" I have introduced a new defect that will manifest itself as soon as some other part of the program invokes the "fixed" `shell_sort()` function.

The "technique" of twisting and fiddling with the code until the failure miraculously goes away is also known as *debugging into existence*. We change the code although we have not fully understood how the failure came to be. Such a "technique" may eventually help in fixing the failure at hand, but it is so likely to induce new defects (or simply leave defects in the code) that it is best avoided.

The Devil's Guide to Debugging (List 15.1) lists more techniques to be avoided. Have fun.

LIST 15.1: The Devil's Guide to Debugging

Find the defect by guessing. This includes:

- Scatter debugging statements throughout the program.
- Try changing code until something works.
- Don't back up old versions of the code.
- Don't bother understanding what the program should do.

Don't waste time understanding the problem. Most problems are trivial, anyway.

Use the most obvious fix. Just fix what you see:

```
x = compute(y);
// compute() doesn't work for y == 17, so fix it
if (y == 17)
    x = 25.15;
```

Why bother going all the way through `compute()`?

Source: McConnell (1993).

15.3.3 Think Before You Code

Does one really need to verify causality for every step in the infection chain? Not if you have a clear understanding of how the failure came to be. That is, you should have understood the infection chain to a point such that your *hypothesis* about the problem cause becomes a *theory*—a theory that allows you to exactly *predict*:

- How your change to the code will break the infection chain.
- How this will make the failure (as well as similar failures) no longer occur.

One way to ensure you have a theory is to have your fix *reviewed* by someone else before applying it. If you can clearly explain how your fix will work, you have a good theory.

Of course, your prediction about how our change will correct the program had better come true. Otherwise, you will know that you have made a huge mistake. If it comes true, though, and the failure is gone, your change *retrospectively validates causality*. Fixing the defect made the failure no longer occur, and therefore the original defect caused the failure.

15.4 CORRECTING THE DEFECT

Assume you have fully understood the infection chain and prepared a correction for the problem. Before you apply the correction, be sure to save the original code—for instance, using the version control system. Then, you actually *correct the code.*

Correcting the code can be a great moment. You have reproduced the failure, observed the execution, carefully tracked back the infection chain, and gained complete understanding of what was going on. All of this has prepared you for this very moment—the actual correcting of the code. (And there was much rejoicing.)

Unfortunately, all great moments are futile. As soon as you have applied your correction, you must take care of four problems, as follows.

15.4.1 Does the Failure No Longer Occur?

After correcting the code, you must ensure that the correction makes the failure no longer occur. First, this retrospectively validates causality (Section 15.3.3). Second, it makes sure we actually solved the problem.

Ensuring that the correction was successful is easy to determine: If the original problem (see Chapter 4) no longer occurs with the changed code, the correction was successful. (If you feel like a hero the moment the failure is gone, you have not been systematic enough. You should be confident about the success of your correction, but the problem no longer occurring should give you just the last bit of confirmation you needed.) If the program still fails after your correction has been applied, though, there *is still a defect that must be fixed*.

- It may well be that a failure is caused by multiple defects, and that removing the first defect causes the second defect to become active.
- However, there is also a chance that the code you fixed was not a defect at all, and that your understanding of the infection chain was wrong. To exclude this possibility, work through your earlier observations and experiments, as noted in the debugging logbook (see Section 6.5 in Chapter 6). Check whether your conclusions are valid, and whether other conclusions are possible.

  Being wrong about a correction should:

  - Leave you astonished.
  - Cause self-doubt, personal reevaluation, and deep soul searching.
  - Happen rarely.

  If you conclude that the defect might be elsewhere, bring back the code to its original state before continuing. This way, your earlier observations will not be invalidated by the code change.

15.4.2 Did the Correction Introduce New Problems?

After correcting the code, you must ensure that the correction did not introduce new problems. This, of course, is a much more difficult issue—especially because many corrections introduce new problems (List 15.2). Practices that are most useful include the following.

- Having corrections *peer reviewed*, as mandated by the problem life cycle (see Chapter 2). A *software change control board* (SCCB) can organize this.
- Having a *regression test* ready that detects unwanted behavior changes. This is another reason to introduce automated tests (see Chapter 3).

LIST 15.2: Facts on Fixes

- In the ECLIPSE and MOZILLA projects, about 30 to 40 percent of all changes are fixes (Śliwerski et al., 2005).
- Fixes are typically two to three times smaller than other changes (Mockus and Votta, 2000).
- Fixes are more likely to induce failures than other changes (Mockus and Weiss, 2000).
- Only 4 percent of one-line changes introduce new errors in the code (Purushothaman and Perry, 2004).
- A module that is one year older than another module has 30 percent fewer errors (Graves et al., 2000).
- Newly written code is 2.5 times as defect prone as old code (Ostrand and Weyuker, 2002). (All figures apply to the systems considered in the case studies.)

Do not attempt to fix multiple defects at the same time. Multiple fixes can interfere with one another and create failures that look like the original one. Check each correction individually.

15.4.3 Was the Same Mistake Made Elsewhere?

The defect you have just corrected may have been caused by a particular *mistake*, which may have resulted in other similar defects. Check for possible defects that may be caused by the same mistake.

The following is a C example. The programmer copies a character string from a static constant `t[]` to a memory-allocated area `s`, using `malloc(`*n*`)` to allocate *n* characters, `strlen(`*t*`)` to determine the length of a string *t*, and `strcpy(`*s*, *t*`)` to copy a string from *t* to *s*.

```
char t[] = "Hello, world!";
char *s = malloc(strlen(t));
strcpy(s, t);
```

What's wrong with this code? In C, character strings are NUL-terminated. A five-character string such as `Hello` actually requires an additional NUL character in memory. The previous code, though, does not take the NUL character into account and allocates one character too few. The corrected code should read:

```
char t[] = "Hello, world!";
char *s = malloc(strlen(t) + 1);
strcpy(s, t);
```

The programmer may have made the same mistake elsewhere, which is why it is useful to check for further occurrences of `strlen()` and `malloc()`. This is also an opportunity to *refactor* the code and prevent similar mistakes. For instance, the previous idiom is so common that one might want to use the dedicated function

```
char t[] = "Hello, world!";
char *s = strdup(t);
```

where `strdup(`*s*`)` allocates the amount of required memory—using `malloc (strlen(`*s*`) + 1)` or similar—and copies the string using `strcpy()`. By the way, `strdup()` can also handle the case that `malloc()` returns `NULL`.

15.4.4 Did I Do My Homework?

Depending on your problem life cycle (see Chapter 2), you may need to assign a *resolution* (such as FIXED) as to the problem. You also may need to integrate your fix into the production code, leaving an appropriate log message for version control.

Finally, you may wish to think about how to avoid similar mistakes in the future. We will come to this in Chapter 16.

15.5 WORKAROUNDS

In some cases, locating a defect is not difficult, but correcting the defect is. The following are some reasons this might happen:

- *Unable to change.* The program in question cannot be changed—for instance, because it is supplied by a third party and its source code is not available.
- *Risks.* The correction induces huge risks—for instance, because it implies large changes throughout the system.
- *Flaw.* The problem is not in the code, but in the overall design—that is, the system design must undergo a major revision.

In such situations, one may need to use a *workaround* rather than a correction—that is, the defect remains, but one takes care that it does not cause a failure.

Such a workaround can take care to detect and handle situations that would make the defect cause a failure. It can also take place after the defect has been executed, correcting any undesired behavior.

A workaround is not a permanent solution, and is typically specific to the situation at hand. Workarounds thus tend to reintroduce the failure again after a change has taken place. Therefore, in implementing a workaround it is important to keep the problem open (in the tracking system, for instance) so as to later implement a proper solution.

Of course, if there were a better solution available immediately one would use that instead of a workaround. But at least a workaround solves the problem—for now. In practice, customers often find themselves living with workarounds for long periods of time. List 15.3 outlines a few.

LIST 15.3: Some Common Workarounds

Spam filters are a workaround for solving a flaw in the email system. Anyone can forge arbitrary messages and conceal his or her true identity. The proper solution would be to redesign the entire email system, even incurring all associated costs and risks.

Virus scanners are a workaround to what is a flaw of some operating systems. By default, every user has administrator rights, and thus any downloaded program can gain complete control over the machine. The proper solution would be to assign users limited rights and ask them for proper authorization before attempting to change the system. Unfortunately, too many regular programs (and their installation routines) assume administrator rights, and thus the fundamental problem is not easy to change.

Date windowing is a workaround to the inability of many legacy systems to deal with four-digit years. The workaround consists of having the systems still keep two-digit years and resolving ambiguity by defining a 100-year window that contains all years in the data. If the 100-year window begins in 1930, for instance, then `27` refers to the year `2027`, whereas `35` means the year `1935`. The genuine solution, of course, would be to adapt the legacy system—but again, this incurs costs and risks.

15.6 CONCEPTS

How To

To isolate the infection chain, transitively work backward along the infection origins.

To find the most likely origins, focus on:

- *Failing assertions* (Chapter 10)
- *Causes* in state, code, and input (Chapters 13 and 14)
- *Anomalies* (Chapter 11)
- *Code smells* (Chapter 7)

Function and package boundaries are good places to check for infection origins.

For each origin, ensure that it is an *infection* as well as a *cause.*

If a correction is too costly or too risky, apply a *workaround* (the defect remains in the program but the failure no longer occurs).

To correct the defect, wait until you can predict:

- How your change to the code will break the infection chain.
- How this will make the failure no longer occur.

To ensure your correction is successful, check whether:

- The correction makes the failure no longer occur.
- The correction does not introduce new problems.
- The mistake leading to the defect has caused other similar defects.

To avoid introducing new problems, useful techniques include:

- Having corrections *peer reviewed*.
- Having a *regression test* ready.

15.7 FURTHER READING

McConnell (1993) is a great read on how to fix programs properly.

EXERCISES

15.1 Sommerville (2001) describes the four stages of the debugging process (Figure 15.3). Develop a more detailed model in which "Locate Error" is expanded into at least six stages.

15.2 Consider the `bigbang` code shown in Example 8.3. Where would you locate the defect and how would you correct it?

15.3 For the `bigbang` code, devise three fixes that make the concrete failure no longer occur, but that do not correct the program—that is, so that minor variations can still reintroduce the failure.

15.4 In addition to the TRAFFIC model, there can be other systematic processes to locate the defect. Sketch two.

15.5 Illustrate, using an example, the difference between "good" and "bad" fixes.

15.6 The following piece of code is supposed to read in a number of elements, and to print their sum.

```
n = read(); // Number of elements
for (int i = 0; i < n; i = i + 1)
    a[i] = read();

// sum up elements in a[0]..a[n - 1]
sum = computeSum(a, n - 1);
print(sum);
```

Unfortunately, this program has a defect. If you read in the numbers

```
2     // n
2     // a[0]
2     // a[1]
```

the program prints `2` as the sum, rather than `4`. It turns out that rather than summing up the elements from `a[0]` to `a[n − 1]`, it computes only the sum of `a[0]` to `a[n − 2]`.

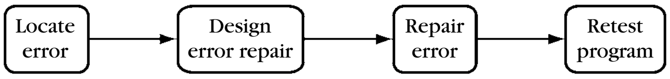

FIGURE 15.3

The debugging process. *Source:* Sommerville (2001).

1. The following are suggestions for fixing the bug. Which one of these actually causes the failure to disappear?
   (a) Replace the `computeSum()` call by the following piece of code.

   ```
   sum = 0;
   for (int i = 0; i < n; i = i + 1)
      sum += a[i];
   ```

   (b) Add the following piece of code after the `computeSum` call.

   ```
   if (n == 2 && a[0] == 2 && a[1] == 2)
       sum = 4;
   ```

   (c) Fix `computeSum()` such that it conforms to its specification.

   (d) Replace the `computeSum(a, n - 1)` call with `computeSum(a, n)` and fix the specification such that it conforms to the actual behavior of `computeSum`.
2. How do these fixes rate in terms of generality (fixing as many failures as possible) and maintainability (preventing as many future failures as possible)? Rank the alternatives, justifying your choices.

15.7 Consider the "fix" to the `sample` program in Section 15.3.2. Is the program actually correct?

Would that I discover truth as easily as I can uncover falsehood.

– CICERO
(44 BC)