CHAPTER

How Failures Come to Be

1

Your program fails. How can this be? The answer is that the programmer creates a defect in the code. When the code is executed, the defect causes an infection in the program state, which later becomes visible as a failure. To find the defect, one must reason backward, starting with the failure. This chapter defines the essential concepts when talking about debugging, and hints at the techniques discussed subsequently—hopefully whetting your appetite for the remainder of this book.

1.1 MY PROGRAM DOES NOT WORK!

Oops! Your program fails. Now what? This is a common situation that interrupts our routine and requires immediate attention. Because the program mostly worked until now, we assume that something external has crept into our machine—something that is natural and unavoidable; something we are not responsible for—namely, a bug.

If you are a user, you have probably already learned to live with bugs. You may even think that bugs are unavoidable when it comes to software. As a programmer, though, you know that bugs do not creep out of mother nature into our programs. (See Bug Story 1 for an exception.) Rather, bugs are inherent parts of the programs we produce. At the beginning of any bug story stands a human who produces the program in question.

The following is a small program I once produced. The `sample` program is a very simple sorting tool. Given a list of numbers as command-line arguments, `sample` prints them as a sorted list on the standard output (`$` is the command-line prompt).

```
$ ./sample 9 7 8
Output: 7 8 9
$ _
```

Unfortunately, `sample` does not always work properly, as demonstrated by the following failure:

```
$ ./sample 11 14
Output: 0 11
$ _
```

BUG STORY 1

The First Bug

We do not know when the first *defect* in a program was introduced. What we know, though, is when the first actual *bug* was found. It may have been in search of plant food, or a warm place to lay its eggs, or both. Now it wandered around in this humming, warm machine that constantly clicked and rattled. But suddenly, it got stuck between the metal contacts of a relay—actually, one of 13,000 high-performance relays commissioned for this particular machine. The current killed it instantly—and its remains caused the machine to fail.

This first actual bug was a moth, retrieved by a technician from the Harvard Mark II machine on September 9, 1947. The moth got taped into the logbook, with the comment "1545 Relay #70 Panel F (moth) in relay. First actual case of bug being found." The moth thus became the living proof that computer problems could indeed be caused by actual bugs.

Although the `sample` output is sorted and contains the right number of items, some original arguments are missing and replaced by bogus numbers. Here, `14` is missing and replaced by `0`. (Actual bogus numbers and behavior on your system may vary.) From the `sample` failure, we can deduce that `sample` has a bug (or, more precisely, a *defect*). This brings us to the *key question* of this chapter:

HOW DOES A DEFECT CAUSE A FAILURE, AND HOW CAN WE FIX IT?

1.2 FROM DEFECTS TO FAILURES

In general, a failure such as that in the `sample` program comes about in the four stages discussed in the following.

1. *The programmer creates a defect.* A defect is a piece of the code that can cause an infection. Because the defect is part of the code, and because every code is initially written by a programmer, the defect is technically created by the programmer. If the programmer creates a defect, does that mean the programmer was at fault? Not necessarily. Consider the following:
   - The original requirements did not foresee future changes. Think about the Y2K problem, for instance.
   - A program behavior may become classified as a "failure" only when the user sees it for the first time.
   - In a modular program, a failure may happen because of incompatible interfaces of two modules.
   - In a distributed program, a failure may be the result of some unpredictable interaction of several components.

BUG STORY 2

F-16 Problems

A programmer who works for General Dynamics in Ft. Worth, TX, wrote some of the code for the F-16, and he has reported some neato-whiz-bang bug/feature they keep finding in the F-16.

- Because the F-16 is a fly-by-wire aircraft, the computer keeps the pilot from doing dumb things to himself. So if the pilot jerks hard over on the joystick, the computer will instruct the flight surfaces to make a nice and easy 4- or 5-G flip. But the plane can withstand a much higher flip than that. So when they were "flying" the F-16 in simulation over the equator, the computer got confused and instantly flipped the plane over, killing the pilot [in simulation]. And since it can fly forever upside down, it would do so until it ran out of fuel.

The remaining bugs were actually found while flying, rather than in simulation.

- One of the first things the Air Force test pilots tried on an early F-16 was to tell the computer to raise the landing gear while standing still on the runway. Guess what happened? Scratch one F-16. [...]
- The computer system onboard has a weapons management system that will attempt to keep the plane flying level by dispersing weapons and empty fuel tanks in a balanced fashion. So, if you ask to drop a bomb the computer will figure out whether to drop a port or starboard bomb in order to keep the load even. One of the early problems with that was the fact that you could flip the plane over and the computer would gladly let you drop a bomb or fuel tank. It would drop, dent the wing, and then roll off.

In such settings, deciding on who is to blame is a political, not a technical, question. Nobody made a mistake, and yet a failure occurred. (See Bug Story 2 for more on such failures.)

2. *The defect causes an infection.* The program is executed, and with it the defect. The defect now creates an *infection*—that is, after execution of the defect, the program state differs from what the programmer intended.

   A defect in the code does not necessarily cause an infection. The defective code must be executed, and it must be executed under such conditions that the infection actually occurs.

3. *The infection propagates.* Most functions result in errors when fed with erroneous input. As the remaining program execution accesses the state, it generates further infections that can spread into later program states. An infection need not, however, propagate continuously. It may be overwritten, masked, or corrected by some later program action.

4. *The infection causes a failure.* A failure is an externally observable error in the program behavior. It is caused by an infection in the program state.

The program execution process is sketched in Figure 1.1. Each program state consists of the values of the program variables, as well as the current execution position (formally, the *program counter*). Each state determines subsequent states, up to the final state (at the bottom in the figure), in which we can observe the failure (indicated by the ✗).

Not every defect results in an infection, and not every infection results in a failure. Thus, having no failures does not imply having no defects. This is the curse of testing, as pointed out by Dijkstra. Testing can only show the presence of defects, but never their absence.

In the case of `sample`, though, we have actually experienced a failure. In hindsight, every failure is thus caused by some infection, and every infection is caused by some earlier infection, originating at the defect. This cause–effect chain from defect to failure is called an *infection chain.*

The issue of debugging is thus to identify the infection chain, to find its root cause (the defect), and to remove the defect such that the failure no longer occurs. This is what we shall do with the `sample` program.

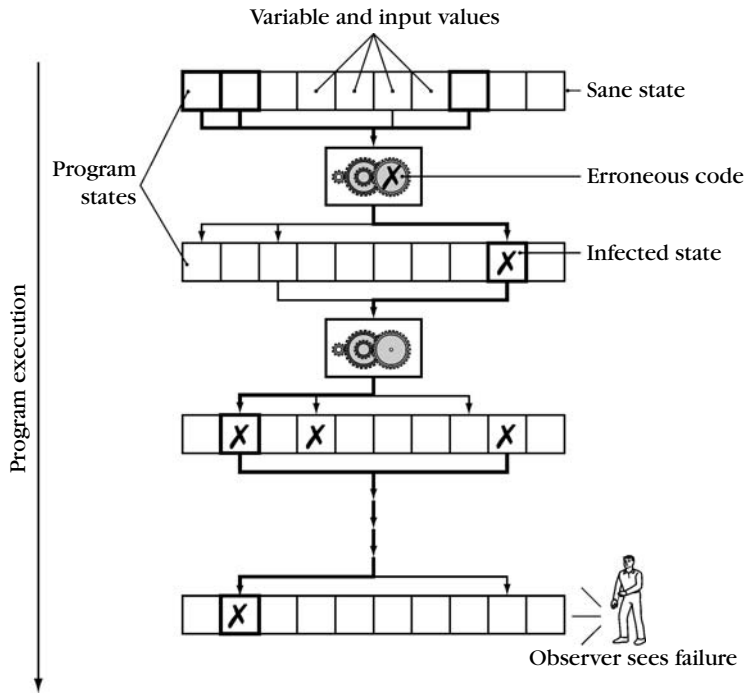

FIGURE 1.1

A program execution as a succession of states. Each state determines the following states, and where from defect to failure errors propagate to form an infection chain.

1.3 LOST IN TIME AND SPACE

In general, debugging of a program such as `sample` can be decomposed into seven steps (List 1.1), of which the initial letters form the word TRAFFIC.

1. Track the problem in the database.
2. Reproduce the failure.
3. Automate and simplify the test case.
4. Find possible infection origins.
5. Focus on the most likely origins.
6. Isolate the infection chain.
7. Correct the defect.

Of these steps, *tracking the problem* in a problem database is mere bookkeeping (see also Chapter 2) and *reproducing the problem* is not that difficult for deterministic programs such as `sample`. It can be difficult for nondeterministic programs and long-running programs, though, which is why Chapter 4 discusses the issues involved in reproducing failures.

Automating the test case is also rather straightforward, and results in automatic simplification (see also Chapter 5). The last step, *correcting the defect*, is usually simple once you have understood how the defect causes the failure (see Chapter 15).

The final three steps—from finding the infection origins to isolating the infection chain—are the steps concerned with *understanding how the failure came to be*.

LIST 1.1: Seven Steps in Debugging (TRAFFIC)

Track the problem in the database.

Reproduce the failure.

Automate and simplify the test case.

Find possible infection origins.

Focus on the most likely origins:

- *Known infections*
- *Causes* in state, code, and input
- *Anomalies*
- *Code smells*

Isolate the infection chain.

Correct the defect.

This task requires by far the most time, as well as other resources. Understanding how the failure came to be is what the rest of this section, and the other chapters of this book, are about.

Why is understanding the failure so difficult? Considering Figure 1.1, all one need do to find the defect is isolate the transition from a *sane* state (i.e., noninfected, as intended) to an *infected* state. This is a search in *space* (as we have to find out which part of the state is infected) as well as in *time* (as we have to find out when the infection takes place).

However, examination of space and time are enormous tasks for even the simplest programs. Each state consists of dozens, thousands, or even millions of variables. For example, Figure 1.2 shows a visualization of the program state of the GNU compiler (GCC) while compiling a program. The program state consists of about 44,000 individual variables, each with a distinct value, and about 42,000 references between variables. (Chapter 14 discusses how to obtain and use such program states in debugging.)

Not only is a single state quite large, a program execution consists of thousands, millions, or even billions of such states. Space and time thus form a wide area in which only two points are well known (Figure 1.3): initially, the entire state is sane (✔), and eventually some part of the state is infected (✘). Within the area spanned by space and time, the aim of debugging is to locate the defect—a single transition from sane (✔) to infected (✘) that eventually causes the failure (Figure 1.4).

Thinking about the dimensions of space and time, this may seem like searching for a needle in an endless row of haystacks—and indeed, the fact is that debugging

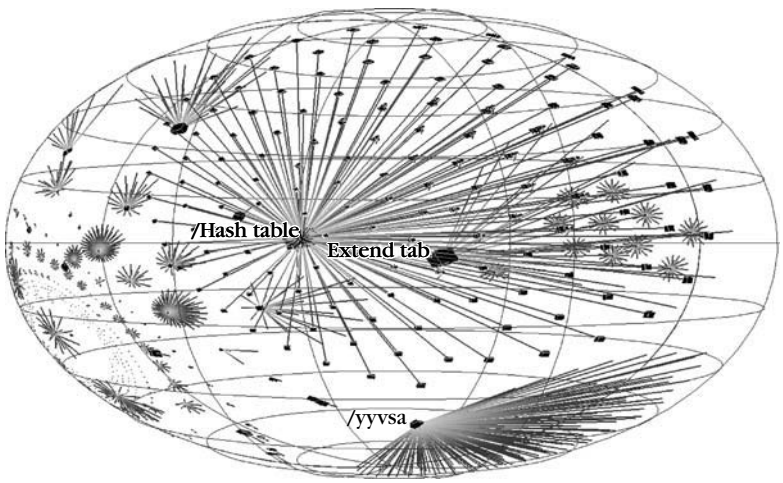

FIGURE 1.2

A program state of the GNU compiler. The state consists of 44,000 individual variables (shown as vertices) and about 42,000 references between variables (shown as edges).

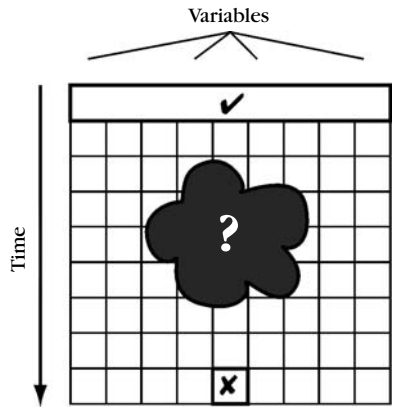

FIGURE 1.3

Debugging as search in space and time. Initially, the program state is sane (✔), eventually, it is infected (✘). The aim of debugging is to find out where this infection originated.

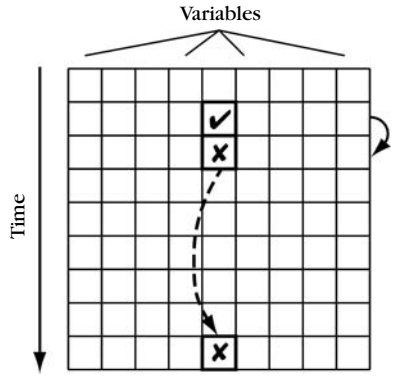

FIGURE 1.4

The defect that is searched. A defect manifests itself as a transition from sane state (✔) to infected state (✘), where an erroneous statement causes the initial infection.

is largely a search problem. This search is driven by the following two major principles:

1. *Separate sane from infected.* If a state is infected, it may be part of the infection propagating from defect to failure. If a state is sane, there is no infection to propagate.
2. *Separate relevant from irrelevant.* A variable value is the result of a limited number of earlier variable values. Thus, only some part of the earlier state may be relevant to the failure.

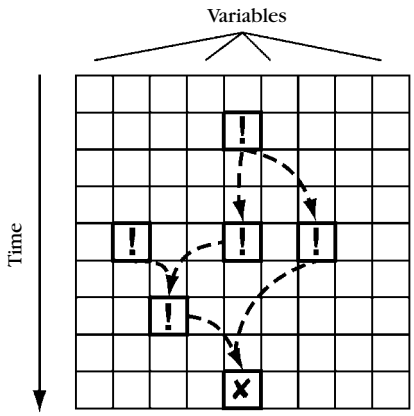

FIGURE 1.5

Deducing value origins. By analyzing the program code, we can find out that an infected variable value (✗) can have originated only from a small number of earlier variables (!).

Figure 1.5 illustrates this latter technique. The failure, to reiterate, can only have been caused by a small number of other variables in earlier states (denoted using the exclamation point, !), the values of which in turn can only have come from other earlier variables. One says that subsequent, variable values *depend on* earlier values. This results in a series of *dependences* from the failure back to earlier variable values. To locate the defect, it suffices to examine these values only—as other values could not have possibly caused the failure—and separate these values into *sane* and *infected.* If we find an infected value, we must find and fix the defect that causes it. Typically, this is the same defect that causes the original failure.

Why is it that a variable value can be caused only by a small number of earlier variables? Good programming style dictates division of the state into *units* such that the information flow between these units is minimized. Typically, your programming language provides a means of structuring the state, just as it helps you to structure the program code. However, whether you divide the state into functions, modules, objects, packages, or components, the principle is the same: a divided state is much easier to conquer.

1.4 FROM FAILURES TO FIXES

Let's put our knowledge about states and dependences into practice, following the TRAFFIC steps (List 1.1).

1.4.1 Track the Problem

The first step in debugging is to *track the problem*—that is, to file a problem report such that the defect will not go by unnoticed. In our case, we have already observed

the failure symptom: the output of `sample`, when invoked with arguments `11` and `14`, contains a zero.

```
$ ./sample 11 14
Output: 0 11
$ _
```

An actual problem report would include this invocation as an instruction on how to reproduce the problem (see Chapter 2).

1.4.2 Reproduce the Failure

In case of the `sample` program, reproducing the failure is easy. All you need do is reinvoke `sample`, as shown previously. In other cases, though, reproducing may require control over all possible input sources (techniques are described in Chapter 4).

1.4.3 Automate and Simplify the Test Case

If `sample` were a more complex program, we would have to think about how to automate the failure (in that we want to reproduce the failure automatically) and how to simplify its input such that we obtain a minimal test case. In the case of `sample`, though, this is not necessary (for more complex programs, Chapter 5 covers the details).

1.4.4 Find Possible Infection Origins

Where does the zero in the output come from? This is the fourth step in the TRAFFIC steps: We must *find possible infection origins.* To find possible origins, we need the actual C source code of `sample`, shown in Example 1.1. We quickly see that the program consists of two functions: `shell_sort()` (which implements the shell sort algorithm) and `main`, which realizes a simple test driver around `shell_sort()`. The `main` function:

- Allocates an array `a[]` (line 32).
- Copies the command-line arguments into `a[]` (lines 33–34).
- Sorts `a[]` by invoking `shell_sort()` (line 36).
- Prints the content of `a[]` (lines 38–41).

By matching the output to the appropriate code, we find that the `0` printed by `sample` is the value of the variable `a[0]`, the first element of the array `a[]`. This is the infection we observe: at line 39 in `sample.c`, variable `a[0]` is obviously zero.

Where does the zero in `a[0]` come from? Working our way backward from line 40, we find in line 36 the call `shell_sort(a, argc)`, where the array `a[]` is passed by reference. This function might well be the point at which `a[0]` was assigned the infected value.

EXAMPLE 1.1: The `sample` program sorts given numbers—that is, mostly

```
 1  /* sample.c -- Sample C program to be debugged */
 2
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5
 6  static void shell_sort(int a[], int size)
 7  {
 8      int i, j;
 9      int h = 1;
10
11      do {
12          h = h * 3 + 1;
13      } while (h <= size);
14      do {
15          h /= 3;
16          for (i = h; i < size; i++)
17          {
18              int v = a[i];
19              for (j = i; j >= h && a[j - h] > v; j -= h)
20                  a[j] = a[j - h];
21              if (i != j)
22                  a[j] = v;
23          }
24      } while (h != 1);
25  }
26
27  int main(int argc, char *argv[])
28  {
29      int *a;
30      int i;
31
32      a = (int *)malloc((argc - 1) * sizeof(int));
33      for (i = 0; i < argc - 1; i++)
34          a[i] = atoi(argv[i + 1]);
35
36      shell_sort(a, argc);
37
38      printf("Output: ");
39      for (i = 0; i < argc - 1; i++)
40          printf("%d ", a[i]);
41      printf("N");
42
43      free(a);
44
45      return 0;
46  }
```

Unfortunately, `shell_sort()` in lines 6–25 is quite obscure. We cannot trace back the value of `a[0]` to a specific origin simply by *deduction* from the program code. Instead, we have to *observe* what actually happens in the failing run.

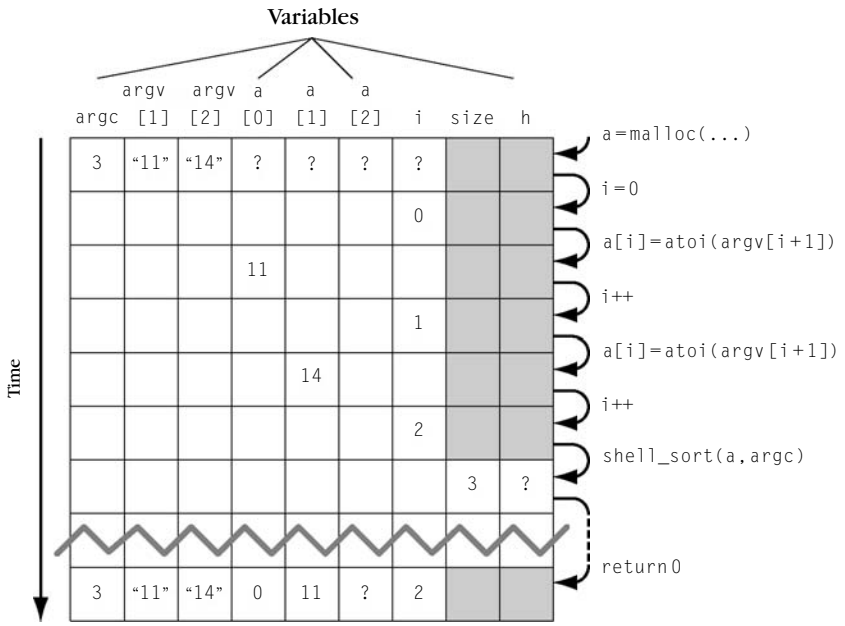

FIGURE 1.6

Observing the `sample` run. Using observation tools, we can observe the program state as it progresses through time.

In principle, we can observe anything about the `sample` run, as sketched in Figure 1.6. We can even "execute" it on paper. However, this approach does not scale. We must focus on specific parts of the state or on specific moments in time. Relying on our earlier deduction on the origins of `a[0]`, we focus on the execution of `shell_sort()`.

- We can easily find out that `shell_sort()` does not access any nonlocal variables. Whatever comes out of `shell_sort()` is determined by its input. If we observe the arguments at the invocation of `shell_sort()`, two things can happen:
  1. The arguments at the invocation of `shell_sort`(✓) are sane (i.e., are just as intended). In this case, the infection must take place *during* the execution of `shell_sort()`, as sketched in Figure 1.7.
  2. The arguments are already infected. In this case, the infection must have taken place *before* the execution of `shell_sort()`.

To find out how `shell_sort()` was actually invoked, we need a means of observing the state during execution. In this introductory chapter, we use the simplest of all observation techniques: We insert output statements in the code that *log* specific variables and their values when executed. For instance, we could insert the following code in line 10 to have the values of the parameters `a[]` and `size` logged on the standard error channel whenever `shell_sort()` is invoked.

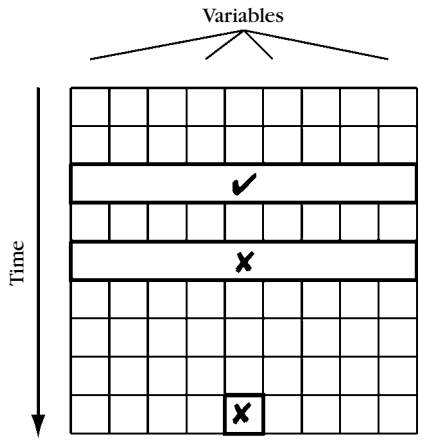

FIGURE 1.7

Observing a transition from sane to infected. If we know that an earlier state is sane (✔) and a later state is infected (✘), we can narrow down our search to isolate the transition between these two states.

```
fprintf(stderr, "At shell_sort");
for (i = 0; i < size; i++)
    fprintf(stderr, "a[%d] = %d\n", i, a[i]);
fprintf(stderr, "size = %d\n", size);
```

1.4.5 Focus on the Most Likely Origins

After inserting the code and restarting `sample` with the arguments `11` and `14`, you will find that at `shell_sort()` the values of the parameters are as follows.

```
a[0] = 11
a[1] = 14
a[2] = 0
size = 3
```

We see that `shell_sort()` is invoked with *three* elements—that is, the array `a[]` to be sorted is `[11, 14, 0]`. This state is infected—that is, `a[]` should contain only two elements. As discussed previously, an infected state is likely to cause failures, and this particular state may well be the cause of our failure. Our hypothesis is that `shell_sort()` properly sorts the three elements of `a[]` in place to `[0, 11, 14]`. Later on, though, only the first two elements of `a[]` will be printed, resulting in the failure output.

1.4.6 Isolate the Origin of the Infection

According to our earlier reasoning, the infection must have occurred *before* the invocation of `shell_sort()`. Obviously, the parameter `size` is wrong. We can trace

back its origin to the point at which `shell_sort()` is invoked: In line 36, we find the invocation

```
shell_sort(a, argc);
```

and find that the `size` parameter gets its value from the `argc` variable. However, `argc` is not the number of elements in `a[]`. It is the number of arguments to the `sample` program, including the name `sample` itself (`argc` is always *one more* than the number of elements in `a`). Thus, the following is our speculation about what is happening in our program:

1. The array `a[]` is allocated and initialized with the correct number of elements (`2`).
2. `shell_sort()` is invoked such that the `size` parameter is `3` instead of `2` (the state is infected).
3. `size` being `3` causes `shell_sort()` to access `a[]` beyond the allocated space (namely, at `a[2]`).
4. The uninitialized memory at `a[2]` happens to be `0`.
5. During the sort, `a[2]` is eventually swapped with `a[0]`, thus setting `a[0]` to zero (the infection has spread to `a[0]`).
6. Thus, the zero value of `a[0]` is printed, causing the failure.

You may wonder why `sample` actually worked when being invoked with the arguments `9 7 8`. The defect was the same, and it caused the same infection. However, as `a[3]` in this case turned out to be larger than `9` it did not get swapped with another array element. At the return of `shell_sort()` the infection was gone, and thus the defect never showed up as a failure.

1.4.7 Correct the Defect

So far, we are still *speculating* about the failure cause. To deliver the final proof, we have to correct the defect. If the failure no longer occurs, we know that the defect caused the failure.

In addition to prohibiting the failure in question we want to prohibit as many failures as possible. In our case, we achieve this by replacing line 36,

```
shell_sort(a, argc);
```

with the correct invocation

```
shell_sort(a, argc - 1);
```

Repeating the test with the fixed program, as follows, shows that the original failure no longer occurs.

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

This resolves the `sample` problem.

1.5 AUTOMATED DEBUGGING TECHNIQUES

Essentially, we have solved the `sample` problem *manually*—that is, without using any specific tools. In principle, all debugging problems can be solved manually—by deduction from the source code and observation of what is going on in a program. (Purists might even argue that deduction alone suffices to prove a program correct, removing the need to fix defects.)

In practice, though, it is unwise to rely on manual debugging alone, as the computer can relieve you of most boring and tedious tasks. In particular, the `sample` program discussed earlier can be debugged almost automatically. Figure 1.8 depicts the automated debugging techniques discussed in the following.

Simplified input. Chapter 5 introduces *delta debugging*—a technique that automatically narrows down the difference between a passing and a failing run. Applied to program input, delta debugging returns a *simplified input* wherein each part contributes to the failure.

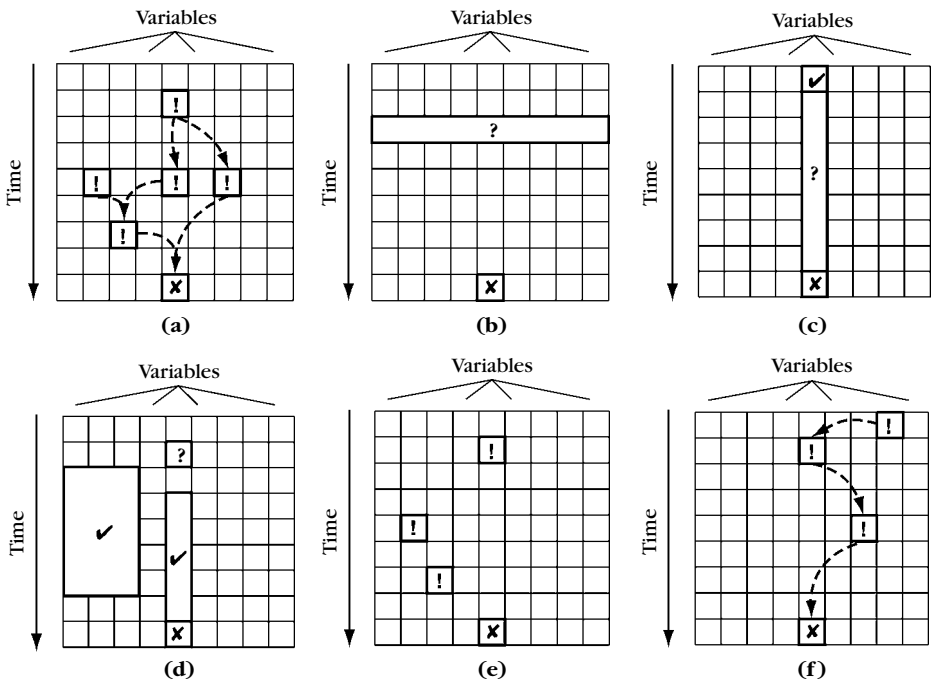

FIGURE 1.8

Some automated debugging techniques: (a) program slice, (b) observing state, (c) watching state, (d) asserting an invariant, (e) anomalies, and (f) cause–effect chain.

Applied to the failing `sample` run, delta debugging determines that each of the arguments `11` and `14` is relevant. The failure no longer occurs if `sample` is being called with one argument only.

Program slices. Chapter 7 explores basic deduction methods—that is, deducing from the (abstract) program code what can and cannot happen in the (concrete) program run. The most important technique is *slicing*—separating the part of a program or program run relevant to the failure. In Figure 1.8(a), we can see that only a fraction of the state actually could have caused the failure. Applied to `sample`, a program slice could determine that `a[0]` got the zero value because of the values of `a[2]` and `size`, which is already a good hint.

Observing state. Chapter 8 discusses *observation techniques*, especially using *debuggers.* A debugger is a tool that can make a program stop under specific conditions, which allows a programmer to observe the entire state (see Figure 1.8b). This allows us to tell the sane program state from the infected state. Using a debugger on `sample`, we would be able to observe the values of `a[]` and `size` at any moment in time without changing or recompiling the program.

Watching state. Another important feature of debuggers, also discussed in Chapter 8, is that they allow us to *watch* small parts of the state to determine if they change during execution. As sketched in Figure 1.8(c), this allows us to identify the precise moment at which a variable becomes infected. Using a debugger on `sample`, we would be able to watch the value of `a[0]` to catch the precise statement that assigns the zero from `a[2]`.

Assertions. When observing a program state, the programmer must still *compare* the observed values with the intended values—a process that takes time and is error prone. Chapter 10 introduces *assertions*, which are used to delegate this comparison process to the computer. The programmer specifies the expected values and has the computer check them at runtime—especially at the beginning and ending of functions (i.e., pre- and postconditions). Such assertions typically include *invariants* over data structures that hold during the entire run.

If all assertions pass, this means that the state is just as expected. When used to check invariants, as sketched in Figure 1.8(d), assertions can mark large parts of the state as "sane," allowing the programmer to focus on the other parts.

One specific instance of assertions is *memory assertion*, checking whether memory is accessed in a legal way. Applied to `sample`, tools exist that can easily identify that `a[2]` is accessed without being allocated or initialized. These tools are also discussed in Chapter 10.

Anomalies. In general, we can assume that a program works well most of the time. If a program fails nonetheless, we can use our knowledge about the passing runs and focus on the *differences* between the passing runs and the failing run. Such differences point out *anomalies*, as sketched in Figure 1.8(e). Detecting anomalies requires techniques designed to *compare* program runs. It also requires techniques for creating *abstractions* over multiple runs. Chapter 11 discusses these techniques.

Applied to `sample`, we can, for instance, compare the *coverage* of the two runs `sample 11` (passing) and `sample 11 14` (failing). It turns out that the statements where `a[j]` is assigned a value are executed only in the failing run, but not in the passing run. Thus, if we are looking for a zero value in `a[0]` these two lines might be a good starting point.

Cause–effect chains. Chapter 14 applies delta debugging to *program states*, thus identifying in each state which particular variable(s) caused the failure. This results in a *cause–effect chain*, as sketched in Figure 1.8(f). Although causes are not necessarily errors, they help to narrow down the relevant elements of a failure.

Delta debugging on states is also the basis of the ASKIGOR automated debugging server. Its diagnosis, shown in Figure 1.9, summarizes how the failure came to be: variable `argc` was 3; hence, `a[2]` was zero; thus, `sample` failed.

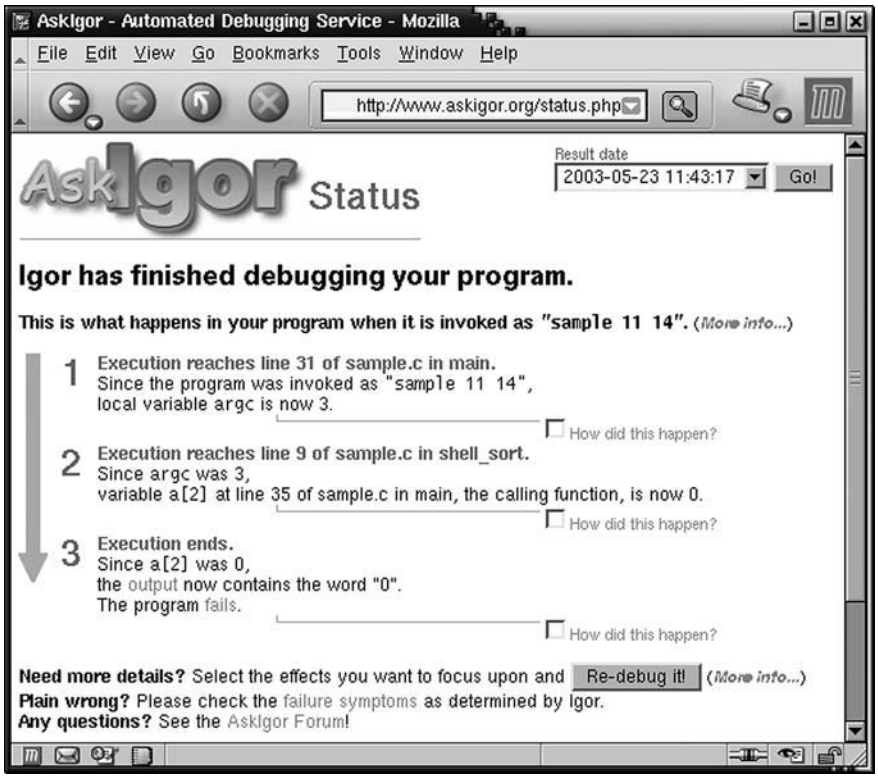

FIGURE 1.9

The ASKIGOR debugging server with a diagnosis for `sample`. Given an executable, a failing invocation, and a passing invocation, ASKIGOR automatically produces a diagnosis consisting of a cause–effect chain from invocation to failure.

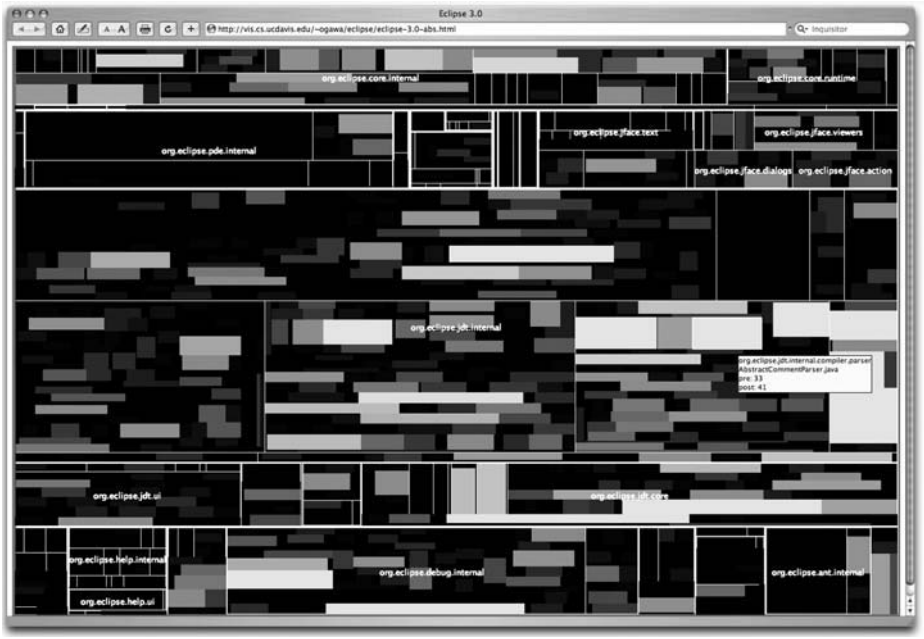

FIGURE 1.10

Defect distribution in ECLIPSE . Each rectangle stands for a package; the brighter the rectangle, the more defects were discovered (and fixed) in the package after release.

Learning from mistakes. After discussing how to fix the defect in Chapter 15, Chapter 16 shows how to aggregate earlier problem reports and fixes to find out where the most defects occur in the program. Figure 1.10, for instance, shows the distribution of defects across the ECLIPSE project: Each package is depicted by a rectangle; the larger the rectangle, the larger the class, and the brighter the rectangle, the higher the *defect density*—that is, the higher the number of defects that were fixed per 1,000 lines of code in the package.

Defect distributions as in ECLIPSE can guide the debugging process toward likely defect sources. They can also help in predicting where the next defects will be, such that quality assurance efforts are directed toward those components that deserve most of the attention.

All of these techniques can be combined to locate the defect systematically—and with a substantial degree of automation. In addition to these chapters focusing on concrete techniques, the following chapters focus on *prerequisites*:

- Tracking failures (Chapter 2)
- Running tests automatically (Chapter 3)
- Reproducing failures (Chapter 4)

- Combining various reasoning techniques (Chapter 6)
- Finding failure causes systematically (Chapter 12)

The first of these prerequisites is that a problem exists. Consequently, the next chapter starts with tracking and reproducing failures.

1.6 BUGS, FAULTS, OR DEFECTS?

Before we conclude this chapter, let's examine our vocabulary. As illustrated at the beginning of this chapter, the word *bug* suggests something humans can touch and remove—and are probably not responsible for. This is already one reason to avoid the word *bug*. Another reason is its lack of precision. Applied to programs, a bug can mean:

- An incorrect *program code* ("This line is buggy").
- An incorrect *program state* ("This pointer, being null, is a bug").
- An incorrect *program execution* ("The program crashes; this is a bug").

This ambiguity of the term *bug* is unfortunate, as it confuses causes with symptoms: *The bug in the code caused a bug in the state, which caused a bug in the execution—and when we saw the bug we tracked the bug, and finally found and fixed the bug.* The remainder of this book uses the following, more precise, terms:

- *Defect:* An incorrect program code (a bug in the code).
- *Infection:* An incorrect program state (a bug in the state).
- *Failure:* An observable incorrect program behavior (a bug in the behavior).

The wording of the previous example thus becomes clearer: *The defect caused an infection, which caused a failure—and when we saw the failure we tracked the infection, and finally found and fixed the defect.*

The industry uses several synonyms of these terms. The IEEE standards define the term *fault* as *defect* is defined here. They also define *bug* as a fault, thus making it a synonym of *defect*—and *debugging* thus becomes the activity of removing defects in the software.

The terms *error* and *fault* are frequently used as a synonym of *infection*, but also for mistakes made by the programmer. Failures are also called *issues* or *problems*. In this book, we use *problem* as a general term for a questionable property of the program run. A problem becomes a *failure* as soon as it is considered incorrect.

Some defects cannot be attributed to a specific location in the software, but rather to its overall design or architecture. I call such defects *flaws.* In general, flaws are bad news, because they suggest major changes involved in fixing the problem.

So much for our family of bugs and related terms. Actually, your choice of one term over another shows two things. First, your wording shows how seriously you take the quality of your work. As Humphrey (1999) points out, the term *bug* "has

an unfortunate connotation of merely being an annoyance; something you could swat away or ignore with minor discomfort." He suggests *defect* instead, and this book follows his advice. Likewise, the word *failure* is to be taken more seriously than *issue*. (If you find a *flaw*, you should be truly alarmed.)

Second, your choice shows whether you want to attribute failures to individuals. Whereas *bugs* seem to creep into the software as if they had a life of their own, *errors* and *faults* are clearly results of human action. These terms were coined by Edsger W. Dijkstra as pejorative alternatives to *bug* in order to increase the programmers' sense of responsibility. After all, who wants to create a fault? However, if a program does not behave as intended this may not be the effect of a human mistake (as discussed in Section 1.2). In fact, even a program that is *correct* with respect to its specification can still produce *surprises.* This is why I use the terms *defect* and *infection* instead of the guilt-ridden *faults* and *errors.* All of these definitions (and more) can be found in the Glossary.

1.7 CONCEPTS

In general, a failure comes about in the following three stages (see also List 1.2):

- The programmer creates a *defect* in the program code (also known as *bug* or *fault*).
- The defect causes an *infection* in the program state.
- The infection causes a *failure*—an externally observable error.

LIST 1.2: Facts on Debugging

- Barron (2002) states that *roughly 22% of PCs and 25% of notebooks break down every year*, compared to 9% of VCRs, 7% of big-screen TVs, 7% of clothes dryers, and 8% of refrigerators.
- According to a U.S. federal study conducted by RTI (2002), software bugs are costing the U.S. economy an estimated *$59.5 billion each year.*
- Beizer (1990) reports that of the labor expended to develop a working program, *50% is typically spent on testing and debugging activities.*
- According to Hailpern and Santhanam (2002), validation activities (debugging, testing, and verification) can easily range from *50% to 75% of the total development cost.*
- Gould (1975) reports that out of a group of experienced programmers the three programmers best at debugging were able to find defects in about 30% the time and made only 40% as many errors as the three worst.
- In RTI (2002), developers estimate that improvements in testing and debugging could *reduce the cost of software bugs by a third*, or $22.5 billion.

How To

To debug a program, proceed in seven steps (TRAFFIC):

1. *Track:* Create an entry in the problem database (see Chapter 2).
2. *Reproduce:* Reproduce the failure (see Chapter 4).
3. *Automate:* Automate and simplify the test case (see Chapters 3 and 5).
4. *Find origins:* Follow back the dependences from the failure to possible infection origins (see Chapters 7 and 9).
5. *Focus:* If there are multiple possible origins, first examine the following:
   - *Known infections*, as determined by assertions (see Chapter 10) and observation (see Chapter 8)
   - *Causes* in state, code, and input (see Chapters 13 and 14)
   - *Anomalies* (see Chapter 11)
   - *Code smells* (see Chapter 7)
   - *Earlier defect sources* (see Chapter 16)

   Prefer automated techniques where possible.
6. *Isolate:* Use scientific method (see Chapter 6) to isolate the origin of the infection. Continue isolating origins transitively until you have an infection chain from defect to failure.
7. *Correct:* Remove the defect, breaking the infection chain (see Chapter 15). Verify the success of your fix.

Of all debugging activities, locating the defect (the find–focus–isolate loop in TRAFFIC) is by far the most time consuming.

Correcting a defect is usually simple, unless it involves a major redesign (in which case we call the defect a *flaw*).

Not every defect results in an infection, and not every infection results in a failure. Yet, every failure can be traced back to some infection, which again can be traced back to a defect.

1.8 TOOLS

Toward your own experimentation with techniques, the "Tools" section within chapters provides references where tools mentioned in the text are publicly available. Typically, the text provides a URL where the tool is available—often as an open-source download. If the tool is not publicly available, the reference describing it will be listed in the "Further Reading" section. You may want to ask the authors whether they will make their tool available to you.

As this chapter is an introduction, references to tools will come in the later chapters. However, note that Clint Jeffery's *Algorithmic and Automatic Debugging Home Page*—a Web page that collects links to debugging techniques and tools, and which will give you the latest and greatest in debugging, is available at *http://www2.cs.uidaho.edu/~jeffery/aadebug.html.*

1.9 FURTHER READING

To avoid breaking up the flow of the main text, references to related work are collected in a section at the end of each chapter. This first "Further Reading" section describes papers, books, and other resources relevant to the material covered in this chapter.

The story about the "first bug" was reported by Hopper (1981). Apparently, Hopper believed that this "first bug" coined the term *bug* for computer programs ("From then on, when anything went wrong with a computer, we said it had bugs in it"). However, as Shapiro (1994) points out, *bug* was already a common "shop" term in Edison's time (1878) for unexpected systems faults. The carryover to computers (certainly complex systems) is almost unavoidable.

Dijkstra's quote that testing can only show the presence of bugs stems from 1972. In 1982, Dijkstra was also among the first to criticize the word *bug* and suggest *error* instead. In 1989, he made clear that this wording would put "the blame where it belongs, viz., with the programmer who made the error."

The origin of the word *bug* is clarified by Beizer (1999). In 2000, he suggested dropping *fault* and *error* due to their pejorative aspect. The terms *bug* and *defect* are compared by Humphrey (1999). The term *infection*, as well as the idea of an infection propagating from defect to failure, were proposed by Voas (1992).

Finally, in that this chapter serves as introduction to the book, we will now look into other material that serves as an introduction to debugging.

The Soul of a New Machine, by Kidder (1981), tracks a team of engineers at Data General working on an innovative new computer. This is not a technical book but a well-orchestrated hymn to the man behind the machine. It describes the challenges and strains of debugging. It the Pulitzer prize winner for nonfiction.

Showstopper!, by Zachary (1994), describes how Microsoft created Windows NT. Not too technical, it focuses on the people and their processes. Not too surprising, eventually finishing the product becomes a struggle with always resurfacing "showstopper" bugs.

Zen and the Art of Motorcycle Maintenance, by Pirsig (1974), despite its title, neither contains many facts on Zen nor many facts on motorcycle maintenance. It is an inquiry on what is good and what is not, in a clear engineer's language, digging into the depths of philosophy—still a cult book today. The section on how a mechanic fixes a motorcycle is a must-read.

Code Complete, by McConnell (1993), is a practical handbook on how to construct software. "Practical" means "pragmatic" and "easily understandable." McConnell goes straight at the topic and tells you how to code and how not to code—and how to debug, of course.

The Practice of Programming, by Kernighan and Pike (1999), describes best practices that help make individual programmers more effective and productive. Although just 250 pages long, barrels of wisdom and experience have been distilled in this book.

Figure 1.10, visualizing the distribution of defect density in ECLIPSE was created by Michael Ogawa (2007), inspired by Martin Wattenberg's "Map of the Market" (1998). The actual data was mined from the ECLIPSE version and bug databases.

Bug Story 2, about the F-16 problems, was posted in *Risks* digest (vol. 3, issue 44), August 1986.

EXERCISES

1.1 Relate the following statements to the terms *defect*, *infection*, *propagation*, and *failure*. Discuss how they (possibly) came to be, and how they (possibly) relate to the output.

- A program throws a null pointer exception.
- A print statement `printf("Helo World")` has a typo.
- A constant $\pi = 31.4$ is declared, but all but one test case pass.
- Variable `z` has the value `15`.
- A bug is removed by fixing three files.
- A clock shows Greenwich mean time rather than the local time zone.

1.2 Compile `sample` on your system. (You can download the source from *http://www.whyprogramsfail.com/*.) When compiling, enable all possible warning options you can find.

1.3 Test `sample` on your system. Do the failures occur as described here? If not, can you find a test case that triggers a failure?

1.4 Each of the following actions effectively fixes the `sample` program. Which are the advantages and disadvantages of these actions?

(a) Insert a statement `argc = argc - 1` at the top of `main`, and replace all later occurrences of `argc - 1` by `argc`.

(b) Change the loop in `shell_sort()` such that it ends at `size - 1` instead of `size`.

(c) Introduce a variable `size = argc - 1` at the top of `main`, and replace all later occurrences of `argc - 1` by `size`. Change the `shell_sort()` invocation to `shell_sort(a, size)`.

(d) Insert a statement `size = size - 1` at the top of `shell_sort()`.

1.5 "If we can prove a program is correct, we have no need for testing or debugging." Argue for and against this assertion. Use at least three arguments in either case.

1.6 Perform a Web search for as many occurrences of *bug*, *defect*, and *fault* you can find via the following:
(a) On the entire Internet.
(b) On the Web pages of your preferred software vendor.
(c) In computer-related newsgroups.

You are in a little maze of twisty passages, all different.

– Will Crowther
Adventure game (1972)