13.7 The Art of Debugging

Software testing is an action that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is an action that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging. The

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

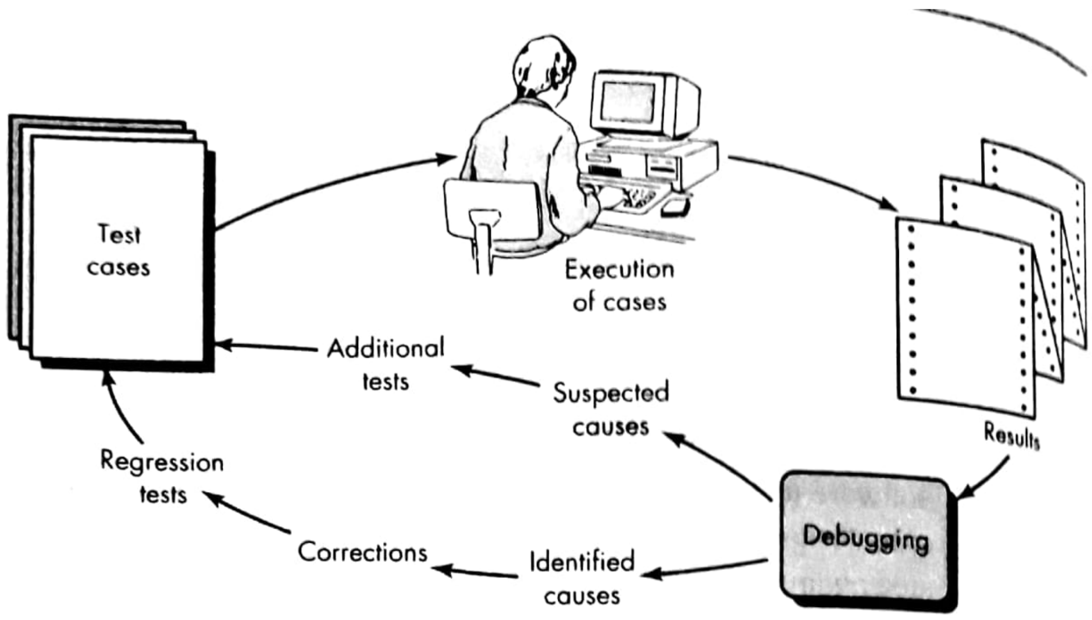
Maurice Wilkes, discovers debugging, 1949

13.7.1 The Debugging Process

Debugging is not testing but always occurs as a consequence of testing.[3] Referring to Figure 13.7, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. Debugging attempts to match symptom with cause, thereby leading to error correction.

Debugging will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the person perform-

FIGURE 13.7
The debugging process



ing debugging may suspect a cause, design one or more test cases to help validate that suspicion, and work toward error correction in an iterative fashion.

Why is debugging so difficult? In all likelihood, human psychology (see the next section) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

? Why is debugging so difficult?

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 11) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors [CHE90].

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces a software developer to fix one error while at the same time introducing two more.

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"

Brian Kernighan

13.7.2 Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it, and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.

Commenting on the human aspects of debugging, Shneiderman [SHN80] states:

> Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected.

Although it may be difficult to "learn" debugging, a number of approaches to the problem can be proposed. We examine these in the next section.

13.7.3 Debugging Strategies

Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition, and luck. Bradley [BRA85] describes the debugging approach in this way:

Debugging is a straightforward application of the scientific method that has been developed over 2,500 years. The basis of debugging is to locate the problem's source [the cause] by binary partitioning, through working hypotheses that predict new values to be examined.

Take a simple non-software example: A lamp in my house does not work. If nothing in the house works, the cause must be in the main circuit breaker or outside. I look around to see whether the neighborhood is blacked out. I plug the suspect lamp into a working socket and a working appliance into the suspect circuit. So goes the alternation of hypothesis and test.

In general, three debugging strategies have been proposed [MYE79]: (1) brute force, (2) backtracking, and (3) cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

"The first step in fixing a broken program is getting it to fail repeatably (on the simplest example possible)."
T. Duff

ADVICE

Set a time limit, say, one hour, on the time you spend trying to debug a problem on your own. After that, get help!

Debugging tactics. The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of *binary partitioning*. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised, and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Automated debugging. Each of these debugging approaches can be supplemented with debugging tools that provide semi-automated support for the software engineer as debugging strategies are attempted. Hailpern and Santhanam [HAI02] summarize the state of these tools when they note, ". . . many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation." One area that has caught the imagination of the industry is the visualization of the necessary underlying programming constructs as a means to analyze a program [BAE97]. A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TOOLS

Debugging

Objective: These tools provide automated assistance for those who must debug software problems. The intent is to provide insight that may be difficult to obtain if approaching the debugging process manually.

Mechanics: Most debugging tools are programming language and environment specific.

Representative Tools[4]

Jprobe ThreadAnalyzer, developed by Sitraka (www.sitraka.com), helps in the evaluation of thread problems—deadlocks, stalls, and race conditions that can pose serious hazards to application performance in Java apps.

C++Test, developed by Parasoft (www.parasoft.com), is a unit testing tool that supports a full range of tests on C and C++ code. Debugging features assist in the diagnosis of errors that are found.

CodeMedic, developed by NewPlanet Software (www.newplanetsoftware.com/medic/), provides a graphical interface for the standard UNIX debugger, *gdb,* and implements its most important features. *gdb* currently supports C/C++, Java, PalmOS, various embedded systems, assembly language, FORTRAN, and Modula-2.

BugCollector Pro, developed by Nesbitt Software Corp. (www.nesbitt.com/), implements a multiuser database that assists a software team in keeping track of reported bugs and other maintenance requests and managing debugging workflow.

GNATS, a freeware application (www.gnu.org/software/gnats/), is a set of tools for tracking bug reports.

The people factor. Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.[5] A final maxim for debugging might be: When all else fails, get help!

---

rather a sampling of tools in this category