Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Debuggers and Logging Frameworks

Diomidis Spinellis

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.

— Maurice Wilkes, 1949

The testing, diagnostic, and repair equipment of many professions is horrendously expensive. Think of logic analyzers, CAT scanners, and dry docks. For us, the cost of debuggers and logging frameworks is minimal; some are even free. All we need to become productive is to invest some time and effort into learning how to use these tools in the most efficient and effective ways.

Assuming that the bug-finding systems I discussed in my last column (Mar./Apr. 2006) have given our program's code a clean bill of health, our next alternatives for productively pinpointing errors that have crept into our code are debuggers or logging instrumentation. These tools help us locate a bug and then verify our hypotheses on what's going wrong. As you might expect, we'll need to adopt an appropriate strategy and master the corresponding techniques to get the best out of these tools.

Debugging strategies

The most efficient debugging strategy is to go bottom-up: we start from the symptom and look for the cause. The symptom might be a memory access violation (for example, the dereferencing of a NULL pointer), an endless loop, or an uncaught exception. A debugger will typically let us take a snapshot of the program at the point where the symptom occurred. From that snapshot, we can examine the program's *stack frame*: the sequence of function or method invocations that led to the problem code's execution. At the very least, this will give us an accurate picture of our program's runtime behavior. We can also examine the values of variables at each level of the stack frame to really understand what made our program go belly-up.

Unfortunately, we can't always adopt a bottom-up strategy. This happens when we can't precisely tie the bug's symptom to a debugger event. Our program might cause a problem in another application, or a variable's contents could be wrong for reasons we can't explain. In such cases, top-down is the name of the game. Debuggers let us walk through the code, stepping over or into functions and methods. When we debug top-down, we initially step over bodies of code we consider irrelevant, narrowing our search as we near the problem's manifestation. This strategy requires patience and persistence. Often we step over a crucial function and find ourselves repeating the search aiming to step into the function the next time around. This process, while tiring, will sooner or later produce results.

If we don't trust the compiler or we can't access the program's source code, we might have to debug it at the assembly code level. What I've found over the years is that assembly code is a lot less intimidating than it appears. Even if we don't know the processor's architecture, with a few educated guesses and a bit of luck we can often decipher the instructions needed to pinpoint the problem.

0740-7459/06/$20.00 © 2006 IEEE

Debugging techniques

Stack frame printouts and stepping commands are basic, indispensable debugging tools, but more powerful commands can help us deal with stickier problems.

Code and data breakpoints

A *code breakpoint* lets us stop the program's execution at a specific line. We can use these to expedite a top-down bug search by placing a breakpoint before the location where we think the problem lies. This breakpoint acts as a bookmark for us to return to and examine the program's operation in more detail.

Less known, but no less valuable, are *data breakpoints*—also known as watchpoints. Many modern processors provide hardware support that will interrupt a program's execution when the code accesses the contents of specified memory locations. Data breakpoints leverage this support, letting us specify that the program's execution will stop when its code reads or writes a variable, an array, or an object. However, debuggers that implement such commands without hardware support slow down the program's execution to a crawl, rendering this command almost useless (Java tool builders take note!).

Live, postmortem, and remote debugging

Although the typical setup involves starting the misbehaving program under a debugger, other debugging options can also help us escape a tight corner.

Consider nonreproducible bugs (also known as Heisenbugs because they make a program appear as if it's operating under the spell of Heisenberg's uncertainty principle). We can usually pinpoint these by debugging a program after it's crashed. On typical Unix systems, crashed programs will leave behind an image of their memory—the *core dump*. By running a debugger on this core dump we get a snapshot of the program's state at the time of the crash. Windows, on the other hand, offers us the possibility to launch a debugger immediately after a program crashes. In both situations, we can look at the crash's location and examine the values the variables had at the time. If the program hasn't crashed but is acting strangely, we can attach a debugger to the process and examine its operation from that point on using the debugger's commands.

Another class of applications that are difficult to debug are those with an interface that's incompatible with the debugger's. Embedded systems, operating system kernels, games, and applications with a cranky GUI fall in this category. Here, the solution is *remote debugging*. We run the process under a debugger but interact with the debugger's interface on another system, connected through the network or a serial interface. This leaves the target system almost undisturbed but lets us issue debugging commands and view their output from our debugging console.

The logging controversy

Instructions in the program's code that generate logging and debugging messages let us inspect a program's behavior without a debugger. Some believe that only people who don't know how to use a debugger use logging statements. This might be true in some cases, but logging statements also offer several advantages over a debugger session. In fact, the two approaches are complementary.

First, a logging statement's location and output are program specific. So, we can place it permanently at a strategic location, and it will output exactly the data we require. A debugger, as a general purpose tool, requires us to follow the program's control flow and manually unravel complex data structures.

Moreover, the work we invest in a debugging session has only ephemeral benefits. Even if we save our setup for printing a complex data structure in a debugger script file, it still wouldn't be visible or easily accessible to other people maintaining the code. I have yet to encounter a project that distributes debugger scripts with its source code. On the other hand, because logging statements are permanent, we can invest more effort in them than we could justify for a fleeting debugging session. We can format their output so that it will increase our understanding of the program's operation—and our debugging productivity.

Finally, logging statements are inherently filterable. Many logging environments, such as the Unix `syslog` library, Java's `util.logging` framework, and the `log4j` Apache logging services (http://logging.apache.org), offer facilities for identifying a given log message's importance and domain. More impressively, Apple's OS X logging facility stores log results in a database and lets us run sophisticated queries on them. We can thus filter messages at runtime to see exactly those that interest us. Of course, we reap these benefits only when we correctly use an organized logging framework, not simple `println` statements.

As you can see, our toolbag is full of useful debugging tools. Being an expert user of a debugger and a logging framework is a sign of professional maturity. So, the next time you encounter a bug, select the appropriate tool and squash it!

Being an expert user of a debugger and a logging framework is a sign of professional maturity.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.