Another DAY Another BUG

We asked our readers which tools they use to squash bugs. Here's what they said.

As part of this issue on programmer tools, we at *Queue* decided to conduct an informal Web poll on the topic of debugging. We asked you to tell us about the tools that you use and how you use them. We also collected stories about those hard-to-track-down bugs that sometimes make us think of taking up another profession.

We were anticipating a wide range of tools, perhaps even hoping for some exciting homegrown tools that would be useful to other developers. Interestingly, we found that most people used fairly straightforward features of the standard debuggers for their development environment. Even more intriguingly, the most common debugging technique continues to be the manual insertion of tracing statements into code. Perhaps the design-

ers of debuggers should take note.

Without further ado, here are the submissions that piqued our interest.

—Terry Coatta, *Queue* Advisory Board Member

DANIEL W. CROMPTON, INFRASTRUCTURE MANAGER

Favorite debugging tool: gdb

What that tool is particularly good at: Thread debugging and cross-platform debugging

What that tool is NOT good at: Visual debugger, Windows/X-based. Don't like xxgdb debugger (not nice).

Debugging philosophy: The solution is inside, if you can't find the bug, then explain what's happening to somebody—anybody—and the answer will come to you as you explain. Otherwise, write out the piece of code in longhand and compare results.

Worst bug: A race condition that only occurred when debugging was turned on.

NASH E. FOSTER, SENIOR SECURITY DEVELOPER

Favorite debugging tool: `printf()` for C/System.out and `println()` for Java

What that tool is particularly good at: There's nothing that a runtime debugger can tell you that a well-placed print statement can't. And they pretty much work exactly right on every system that the program runs. Plus, if you've built your application correctly, when users have problems you can give them a way to debug it on their own. This almost always saves time for both of you.

What that tool is NOT good at: Unfortunately, you have to recompile and rerun to add new print statements. Debugging by print-out requires attention—if you quit

writing `printf()` statements into your code, you will stop getting feedback from it.

Debugging philosophy: Print (or log) early and often.

A good second principle might be: Declare your intentions. If a user sees a message in a logfile that says `:Pointer <foo>: value is null`. He won't know what to think. But, if the user sees `Tree reference: tree is null; expected a populated b-tree, here!`, there's something to go on. Why is the b-tree there? Why is it empty? Maybe the user just forgot to send it input? If there are several messages relating to the same phenomenon, the user can usually debug the problem and tell your developers how to fix it.

Worst bug: Generally, bugs aren't hard to find. If my code gets so complicated that I can't find my bugs, then the real bug is my design. That's when I begin planning for a rewrite.

Another DAY Another BUG

That said, the bugs I spend the most time on always seem to relate to concurrent processing (“Why is this weird deadlock condition occurring?”) or to performance (“Why is this routine so much slower than I had expected?”).

These bugs seem to be difficult because they occur when there are differences between the system’s behavior and the developers’ expectations, that is, “weird” deadlock is just deadlock that you never anticipated.

Another good example: When J2EE/EJB 2.0 first came out, we were completely flummoxed as to why some methods would take 10 to 20 seconds to execute. There didn’t seem to be anything wrong with our algorithms. Eventually we started watching the network interfaces between the application server and the database. Each time a method ran, it would transfer 10 to 20Mb worth of data from the database! That was certainly an “Aha!” moment. It was also when we decided that J2EE might not be very useful after all. When you need to get a list of a hundred things from the database, you want to execute a single query, not one per item.

BOB KRYGER, SYSTEM ADMINISTRATOR

Favorite debugging tool: truss

What that tool is particularly good at: Showing you what an app is actually doing, or looking for, on a Solaris box. Especially good for apps for which you do not have the source code.

What that tool is NOT good at: I’d like to be able to see library calls as well as system calls. Linux has tools that do this: strace and ltrace.

Debugging philosophy: What philosophy? There are bugs, and there are going to be bugs. Developers will not get them all, so we have to deal with them as they appear. Often we don’t have the ability to do anything about them, so then we kludge a workaround.

TILAK MITRA,
INFORMATION TECHNOLOGY ARCHITECT

Favorite debugging tool: Websphere Studio Application Developer

What that tool is particularly good at: The best IDE

for building J2EE-based enterprise applications. The process of creating and deploying an enterprise app is simplified to the greatest extent. Also, it’s awesome for creating Web services from existing java files.

What that tool is NOT good at: Creating container-managed relationships (CMRs). Sometimes duplicates the foreign keys, so someone must manually change them.

Debugging philosophy: Test using JUnit test scripts.

Debug by putting the server in debug mode and step through the suspected path from where the problem might stem while looking at the variable values.

Worst bug: Trying to add EJB QLs to an existing container-managed persistence (CMP) by the name “Group.” We couldn’t add the EJB QL, but solved this in the end by changing the name of the CMP. *Group* is a reserved word in SQL, hence a CMP with the same name was very problematic.

MUD (ROBERT MUDRY)

Favorite debugging tool: Sabre-C and Perl built-in debugger

What that tool is particularly good at: When using C, I prefer Sabre-C. It is a C interpreter that lets you mix object models with code running via the interpreter. We were building large executables with link times running 40-minutes wall clock.

What that tool is NOT good at: It didn’t help much with the classic C-developer nightmare of finding memory leaks.

Debugging philosophy: Mine has not changed much in 30 years. Insert a breakpoint and execute until it traps. Examine process state and external variables. Back in the dark ages of dedicated machine time (DEC PDP-11 during the 1970s), we would insert a *spin* instruction into the executable to accomplish the same thing.

Worst bug: Encountered a bug that only existed on the IBM AIX version of a slightly hacked version of Bill Joy’s diff program. IBM had changed the behavior of a pointer returned on error from one of the built-in functions. Took three weeks—what a pain. The best part was trying to understand Bill Joy’s code. He could write the tightest, most obtuse C that I’ve ever seen. In one *sort* function, there were no comments except a terse two-word phrase: “CACM 273,” which referred to an issue of the *Communications of the ACM*, in which he had found this very efficient but insanely complex sort algorithm.

DAVID NAUGLER,
ASSOCIATE PROFESSOR OF COMPUTER SCIENCE

Favorite debugging tool: TextPad

What that tool is particularly good at: Textpad is a versatile programmer's ASCII editor that is especially useful for those working in multiple languages. Syntax files are available for many languages.

What that tool is NOT good at: Textpad is not for large projects. It's just for small programs and program pieces.

Debugging philosophy: Find the smallest, simplest program that exhibits the same problem.

Worst bug: C++ code that I thought was allowable would not work—it turned out to be C++ valid code. (I believe that Version 5 was the current one at the time.) Metrowerks CodeWarrior turned out to have a bug (counterexample to my “Don't blame the compiler” rule!).

CHRIS PULSIFER, CONSULTANT

Favorite debugging tool: Visual Studio

What that tool is particularly good at: On-the-fly rebuilds, automatically locating and stepping into library source

What that tool is NOT good at: Debugging the screen-manipulation code because of repeated repaints of debugged program.

Debugging philosophy: Identify the event/action that is the root cause, then step through the handling of that event/action.

Worst bug: An incorrect cast/realloc in a library that should have been `sizeof(foo)` instead of `sizeof(bar)`. The structures were close enough in size that the program did not start to crash until three years after deployment, when the count of resources being managed had slowly increased over time. The problem was solved after three long days of debugging.

SANJIV SAHAI, CONSULTANT

Favorite debugging tool: VC .Net Debugger

What that tool is particularly good at: Breakpoint at hit count

Debugging philosophy: Narrow down area to the problem point.

Worst bug: Missing packets. The request packet response was coming before the wait for response began.

ALEX SHUM, STAFF SOFTWARE ENGINEER

Favorite debugging tool: Rational Purify

What that tool is particularly good at: Identifying potential memory leaks and handling leaks in conjunction with C/C++. Also pointing out misuse of pointers (dangling and null ones) with object code assertions.

What that tool is NOT good at: (1) Lacks common tracing functionalities, (2) also lacks Visual debugging

functionalities; and (3) isn't customizable.

Debugging philosophy: The best debugging tool is the tracing/logging routines you write that are used by your team. Third-party tools are only good at identifying a very niche subset of errors. Peer reviews and the cross-reading of each other's work are the best debugging tools.

Worst bug: A resource allocation problem in conjunction with out-of-file handles/descriptors. Was eventually resolved by running Purify, and we then issued an exclusive lock on certain contending resources to avoid race conditions.

JOHN TOBIAS, SOFTWARE ENGINEER

Favorite debugging tool: The Debugger and gdb

What that tool is particularly good at: Finding all those things that other debuggers don't, such as memory watch-points with logical equations for triggering stop or simply logging.

What that tool is NOT good at: Understanding prediction branches and going from C/C++ source to optimized assembly, especially unrolled loops and vector processing.

Debugging philosophy: Understand the hardware, and code simply and clearly from the beginning.

Worst bug: Spent five weeks figuring out why an interrupt would freeze the mouse and discovered that eight phases in the clock were driving the application-specific integrated circuit (ASIC), only one of which would allow correct transfer from streaming to metadata.

GREG WILSON, SENIOR SOFTWARE ENGINEER

Favorite debugging tool: Visual Studio

What that tool is particularly good at: The Visual C++ 6.0 debugger is hands-down the best I have ever worked with, especially with multithreaded code. It's fast, it's accurate, it handles dynamic linkage well, and it's very well-integrated with the rest of the IDE.

What that tool is NOT good at: Debugging non-Windows code; debugging templated C++ (but then, no debugger on the market today handles templates well).

Debugging philosophy: Find the simplest test case that reproduces the problem and try to make your code fail as early as possible.

Worst bug: An intermittent synchronization error in a chess program that ran on several hundred processors in the late 1980s. The program deadlocked every couple of days. I never did find it, which is one of the reasons I got out of massively parallel computing. ☹

LOVE IT, HATE IT? LET US KNOW:

queue-ed@acm.org or www.acmqueue.org/forums