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

Bug Busters

Diomidis Spinellis

Although only a few may originate a policy, we are all able to judge it. — Pericles of Athens

One way to deal with bugs is to avoid them entirely. For example, we could hire only the best software engineers and meticulously review every specification, design, or code element before touching a computer. However, this approach would be wasteful because we'd be underutilizing the many automated tools and techniques that can catch bugs for us. As Pericles recognized, creating a bug-free artifact is a lot more difficult than locating errors in it. Consequently, although humans can seldom cast large-scale bug-free code from scratch, successful bug-finding tools abound.

Most tools for eliminating bugs work by tightening the specifications of what we build; an industrial engineer might similarly seek to reduce variability by manufacturing to tighter tolerances. At the program code level, tighter specifications will affect the operations allowed on various data types, our program's behavior, and our code's style. Furthermore, we can use many different approaches to verify that our code is on track: the programming language, its compiler, specialized tools, libraries, and embedded tests are our most obvious friends here.

Languages

Modern programming languages do a great job in restricting many risky code constructs and expressions. First of all, structured languages (anything better than assembly language and old-style Fortran) prohibit, or at least impede, many programming tricks that can easily lead to unmaintainable spaghetti code. Even C, with its support for `goto` and `longjmp` (giving us ample rope to hang ourselves) doesn't allow arbitrary jumps across different functions. Also, once using a structured language makes us properly indent our code, we're also forced to split it into separate functions or methods: we'd be mad to try to write code with more than a handful of indentation levels. This splitting eliminates bugs by promoting attributes such as encapsulation and testability.

Additionally, languages can often enforce correct behavior on our code. In Java, if a method can throw an exception, methods that call it will have to catch it or declare that they may also throw that exception; in C# we can ensure that resources we acquire will be properly disposed of via the `using` construct.

More importantly, languages with strong typing rules can detect numerous problems at compile time as data-type errors (adding apples to oranges). Obviously, errors we catch at compile time won't appear when the program runs: this is an effective way to eliminate bugs. For example, the introduction of generics into Java 1.5 lets us specify that a list container will house only strings; our program won't compile if we attempt to store a value of a different type in it. In earlier versions of Java where the list contained values of the type `Object`—the least common denominator of all Java types—the error would manifest at runtime as a bug when we attempted to cast an element retrieved from the list into a string.

Compiler tricks

Even when the programming language lets us write unsafe code, we can often ask the compiler to verify it for us. Most compilers will generate warnings when encountering questionable code constructs; we can save ourselves from embarrassing bugs by actually paying attention to these. However, many of us, when we're working under a pressing deadline, will ignore compiler warnings. We can deal with this problem by using another commonly supported compiler option that treats warnings as errors: the code won't compile until we deal with all warnings.

We can also often help the compiler generate better warnings for us. Consider, for example, C's notoriously error-prone `printf`- and `scanf`-like functions. These functions require us to match the types specified in a format string with the supplied arguments. If we get this correspondence wrong, our program could crash, print garbage, or, worse, open itself to a stack-smashing attack. Some compilers will verify format arguments for the C library functions, but we often add our own functions with similar behavior, which the compiler can't check. For these cases, the GNU C compiler provides the `_attribute_((format()))` extension. We tag our own function declarations with the appropriate attribute, and the compiler will check the arguments for us. Along the same line, Microsoft's attributed program extensions to C++ and corresponding Windows API header annotations can catch dangerous buffer overflows.

Specialized tools

Another way to eliminate bugs is to pass our code through one or more tools that will explicitly check it for problems. The progenitor of this tool family is Lint, a tool Stephen Johnson wrote in the '70s to check C code for nonportable code and error-prone or wasteful constructs. For example, Lint will flag the construct `if (b = 0)` as an error, complaining of an assignment in a conditional context; we probably intended to write `if (b == 0)`. Nowadays we can find commercial and open source lint-like tools for many commonly used languages. Some examples include CheckStyle, ESC/Java2, FindBugs, JLint, Lint4J, and PMD (covering Java); FxCop and devAdvantage (covering C#); and PC-lint and Coverity (covering C or C++). Other tools specialize in locating security vulnerabilities—a class of bugs that stand out for their potentially devastating consequences. Tools in this category include Flaw-finder, ITS4, Splint (secure programming Lint), and RATS (rough auditing tool for security).

Specialized tools can cover a lot more than what we could realistically expect a compiler to warn us about. For example, many tools will report violations of coding style guidelines, such as indentation and naming conventions. Furthermore, some tools are extensible: we can add rules particular to our own project (calls to `launchMisile` must be preceded by a call to `openHatch`), and we can precisely specify the rules that our project will follow. Integrating a code-checking tool into our build process, configuring its verification envelope, and extending it for our project should be an important part of our development process. In some projects, a clean pass from the code-checking tools is a (sometimes enforced) prerequisite for checking code into the version control system.

Code

Finally, we can delegate bug busting to code. Many libraries come with hooks or specialized builds that can catch questionable argument values, resource leaks, and wrong ordering of function calls. As a prime example, consider the C language dynamic memory-allocation functions—a potent source of both bugs and research papers that describe versions of the library that can catch them. We can catch many of these bugs by using the valgrind tool, by loading the `watchmalloc.so` library (under Solaris), or by setting the `MALLOC_CHECK` or `MALLOC_OPTIONS` environment variables (under GNU/Linux distributions and FreeBSD).

When writing our own code, we've even more options at our disposal. We can sprinkle our code with assertions, expressing preconditions, postconditions, and invariants. Any violation of them will trigger a runtime error and help us pin down a possibly difficult-to-locate bug. At a higher level, we can instrument our classes with unit tests, using the JUnit testing framework or the equivalent for our environment. When churning out code, unit tests will identify many early bugs; later on, when we focus on maintenance, unit tests will ring a bell when we introduce new bugs.

Bugs many be a fact of life, but they're not inevitable. We have some powerful tools to find them before they mess with our programs, and the good news is that these tools get better every year. Go out and use them!

Many libraries come with specialized builds or hooks that can catch questionable argument values, resource leaks, and wrong ordering of function calls.

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.