Divided by Division

Is there a “best used by” date for software?

George Neville-Neil

Dear KV,

Do you know of any rule of thumb for how often a piece of software should need maintenance? I’m not thinking about bug fixes, since bugs are there from the moment the code is written, but about the constant refactoring that seems to go on in code. Sometimes I feel as if programmers use refactoring as a way of keeping their jobs, rather than offering any real improvement. Is there a “best used by” date for software?

Fresh Code

Dear Fresh,

I definitely like the idea of software coming with a freshness mark like you see on milk cartons. As with other perishable products, software does seem to go bad after a while, and I am often reminded of the stench of spoiled milk when I open certain source files in my editor. No wonder so many programmers grimace whenever they go to fix bugs.

I think that a better analogy for software is that of infrastructure. Anyone who has lived long enough to see new infrastructure built, neglected, and then repaired should understand this analogy straight away. Consider the highways built in the United States in the 1950s. When these roads were first built they were considered a marvel, helping commuters get into and out of large cities. Everyone loves something new, and the building of this infrastructure was heralded with a good deal of fanfare, speeches, etc., that you associate with large projects.

Once completed, however, the process of neglect sets in—cost cutting, slow repairs, ignoring major design flaws until bits of the roadways fall down. Finally, the highway is so poorly maintained that it’s a menace, and then, unless you get lucky and an earthquake destroys the hideous thing, you come to the usual engineering decision: repair or rebuild.

The difference with software is that if code is used in the same way, day in and day out, and never extended or changed—other than fixing previously existing bugs—it should not wear out. Not wearing out depends on a few things—especially that hardware does not advance. A working system delivered in 1980—on, say, a classic minicomputer such as the VAX—should, if the same hardware is present, work the same today as it did when it was built.

The problems of software maintenance arise because things change. While the original libraries used to build a system don’t wear out in any physical sense, the code that they interact with changes over time as idiots (oops, I meant to say marketers) demand new features and as the speed and complexity of hardware advances. Efforts at portability are noble and often worthwhile, but there is simply no way that a piece of code that ran on a 1-MIPS CISC (complex instruction set computing) computer is going to run—without significant retesting and changes—on a modern processor with modern peripherals. Operating systems and device drivers can go only so far to hide the underlying changes from applications.

While I have seen plenty of navel-gazing exercises masquerading as refactoring, there comes a time in the life of all software when the design decisions that it expresses must be reexamined. There is no hard and fast limit for this. If the code was a “prototype”—you know, code that management swore up and down they would never use, and then did—it’s going to go bad sooner rather than later.

Programs that were written in a more reasonable style and without ridiculous schedules imposed from above maintain their freshness longer.

I consider my own code to have a “best by date” of one year from when I complete the project. If I haven’t looked at some code in a year, I’ve probably forgotten how it worked, anyway. In some cases, I’ve actively worked with my local bartender to forget as much of some code as possible.

KV

Dear KV,

I’ve been upgrading some Python 2 code to Python 3 and ran across the following change in the language. It used to be that division (/) of two integers resulted in an integer, but to get that functionality in Python 3, I need to use //. There is still a /, but that’s different. Why would anyone in their right mind have two similar operations that are that closely coded? Don’t they know this will lead to errors?

Divided by Division

Dear Divided,

Python is not the first—and I am quite sure it will not be the last—language to use visually similar keywords to mean different things. Consider C and C++, where bitwise and logical operations use very similar images to mean totally different operations: | for the bitwise or operation and || for the logical, for example. I also recently discovered this change in Python 3, and my coworkers discovered it just after I did, as I don’t drop f-bombs, but instead launch them, violently.

The problem of not having visually distinctive images in programming goes back to the problem, alluded to by another *ACM Queue* columnist (Poul-Henning Kamp in “Sir, please step away from the ASR-33!” October 2010; <http://queue.acm.org/detail.cfm?id=1871406>), of the character set we use to create our languages. Language designers have only the following images to work with when they’re looking for something to represent a shortcut to an operation:

```
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

Many of the characters already have well-established meanings outside of programming, such as the arithmetic operations +, −, *, and /, and the language designer who decides to change their meanings should be treated to a one-way ticket to the bottom of the nearest river or lake.

It’s certainly possible to forgo shortcuts and to make everything a function such as

```
(plus a b)
```

for functional syntax, or create a large list of reserved words as in

```
a equals b plus c
```

FIGURE 1

An APL Keyboard

for Algol-like languages. The fact is, as programmers, we like compact syntax and would balk at using something as bulky as the example I’ve just given.

Another alternative is to throw away ASCII encoding and move to something richer in which we can have more distinct images to which we can attach semantic meanings. The problem then arises of how the hell to type in that code. Modern computer keyboards are meant to allow programmers to type ASCII. Ask Japanese programmers whether they use a Japanese keyboard or an American one, and nine out of ten will tell you an American one. They choose the U.S. version because the “programmer keys,” the ones that represent the glyphs in the list above, are in the easiest-to-use places. Extending our character set to allow for complex glyphs will slow the process of entering new code, and we all know that typing speed is the biggest indicator of code quality. Many years ago there was a language called APL that required a special keyboard. That language is mostly dead; look at the keyboard in figure 1 to find out why.

That brings us to where we are now with `/` meaning one thing and `//` meaning another. I am quite sure many bugs will result from this conflation of images, and I’m sure they’re going to occur when the person working on the code has just been awakened from a deep sleep by a panicked call. In the light of day, it’s easy to tell `/` from `//`, but in the dim light of reawakening, it’s not so easy.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor’s degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.