Editor: Gerard J. Holzmann
NASA/JPL
gholzmann@acm.org

Code Inflation

Gerard J. Holzmann

MOST PEOPLE DON'T get too excited about software. To them, software applications are like cars: inconspicuous when they work, and merely annoying when they don't. Clearly, cars have been getting bigger and safer over the years, but what about software? It sometimes seems as if it has just gotten bigger, not safer. Why?

Software tends to grow over time, whether or not there's a need for it.

If you compare the state of today's software development tools with those used in, say, the '60s, you of course see many signs of improvement. Compilers are faster and better, we have powerful new integrated program development environments, and there are many effective static-source-code-analysis and logic-model-checking tools that help us catch bugs. This would have made a fabulous difference if our software applications still looked like they did in the '60s. But they don't.

Many of my NASA colleagues are astronomers or cosmologists. To explain how rapidly things are changing in software development, I've often been tempted to make an analogy with their field. One of the first things you learn in cosmology is the theory of inflation. The details don't matter too much here, but in a nutshell, this theory postulates that the universe started expanding exponentially fast in the first few moments after the Big Bang and continues to expand. The parallel with software development is easily made.

The First Law

Software too can grow exponentially fast, especially after an initial prototype is created. For example, each Mars lander that NASA launched in the past four decades used more code than all the missions before it combined. We can see the same effect in just about every other application domain. Software tends to grow over time, whether or not a rational need for it exists. We can call this the "first law of software development."

The history of the true command in Unix and Unix-based systems provides a remarkable example of this phenomenon. Shell scripts often employ this simple command to enable or disable code fragments or to build unconditional while loops—for instance, to perform a sequence of random tests:

```
while true
do ./test `rand`
done
```

The /bin/true and /bin/false commands first appeared in January 1979 in the seventh edition of the Unix distribution from Bell Labs. They were defined as tiny command scripts:

```
$ ls –l /bin/true /bin/false
-rwxr-xr-x 1 root root 0 Jan 10 1979 /bin/true
-rwxr-xr-x 1 root root 7 Jan 10 1979 /bin/false
```

Yes, true was actually defined fully with an empty file. How did it work?

Because true contained nothing to execute, it always completed successfully, returning the success value of zero to the user. The false command contained seven characters (including the line feed at the end), to return a nonzero value, which signified failure:

```
$ cat /bin/false
exit 1
```

This implementation would seem to leave nothing left to desire, but that would contradict the first law of software development.

In the first commercial version of Unix from 1982, marketed as System III, the implementation of false changed from exit 1 to exit 255, for unclear reasons, but taking up two more bytes. Then, in a version created for the PDP-11 microcomputer in 1983, the implementation of true grew to 18 bytes, and the empty file now contained a comment:

```
@(#)true.sh 1.2
```

In a 1984 version of Unix, things started heating up, and true grew to 276 bytes. The contents were now a boilerplate AT&T copyright notice claiming intellectual ownership of the otherwise still empty file.

A 2010 Solaris distribution further upped the ante by replacing the shell script with a 1,123-byte C source program consisting of a main procedure that called the function _exit(0). The C program for false similarly had main call _exit(255). Both programs also contained a hefty new copyright notice. If I compile these programs on my system today, the executables tap in at 8,377 bytes each.

We're not done yet. The executable for the most recent version of true on my Ubuntu system is no fewer than 22,896 bytes:

```
$ ls –l /bin/true /bin/false
-rwxr-xr-x 1 root root 22896 Nov 19 2012 /bin/true
-rwxr-xr-x 1 root root 22896 Nov 19 2012 /bin/false
```

The source code for this command has grown to 2,367 bytes and includes four header files, one of which is itself 16 Kbytes of text. That's quite a change from the zero bytes in 1979, and all that without any significant difference in functionality.

If you're still on the fence with this: no, true really doesn't need a –version option to explain which version of the truth the command currently represents. Nor does it need a –help option, whose only purpose seems to be to explain the unneeded –version option. And just in case you were thinking about this: true and false also don't need an option that can invert the result, or one that would let these commands send their result by email to a party of your choice. Some have joked that all software applications continue to grow until they can read and send email. This hasn't happened with the two simplest commands in the Unix toolbox just yet, but we seem to have gotten close.

Table 1 shows how the source code and executable code for true have grown. Figure 1 graphs the executable program's growth. The *y*-axis is a log scale so that

TABLE 1 The growth of the source code and executable code of the Unix true command.

Year	Source code size (LOC)	Executable size (LOC)
1979	0	0
1983	18	18
1984	276	276
2010	1,123	8,377
2012	2,367	22,896

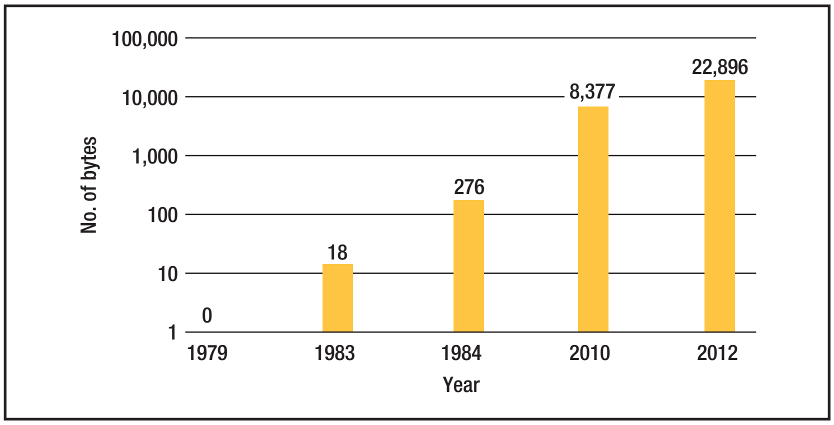

FIGURE 1. The size of /bin/true over time. The *y*-axis is a log scale so that the early numbers aren't completely drowned out by the later ones.

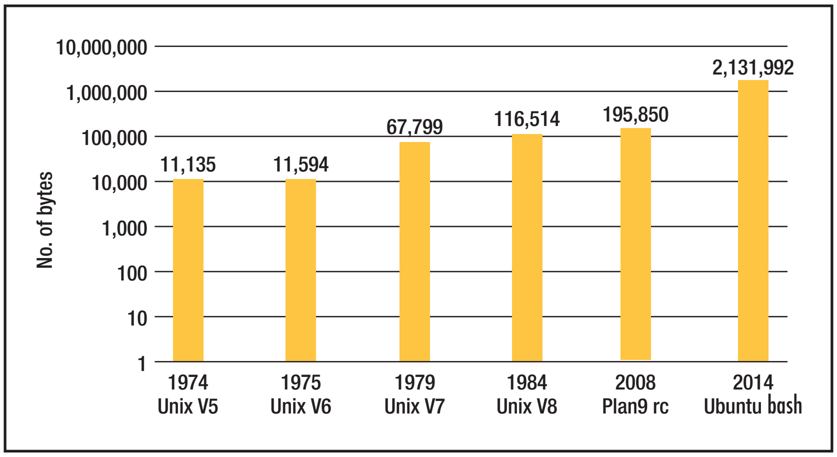

FIGURE 2. The source code size over time for the default command shell on Unix and Unix-like systems. From approximately 11 Kbytes in fifth-edition Unix in 1974 to 2.1 Mbytes for the bash shell 40 years later is an increase of 191 times.

the early numbers aren't completely drowned out by the later ones.

Just like in the theory of inflation, the implementation of /bin/true increased infinitely fast in the first few years since it was created (because, like the universe, it started at a size of zero). Okay, we're not talking 10^{-32} seconds; we're moving more at humanly achievable speeds here. Once we got to a nonzero size, the expansion continued steadily, with the size increasing more than three orders of magnitude since 1983. (You can find more about the curious history of the /bin/true command at John Chambers blog, http://trillian.mit.edu/~jc/;-)/ATT_Copyright_true.html. An online archive of many early Unix source code distributions is at http://minnie.tuhs.org/cgi-bin/utree.pl.)

The best part of all this is perhaps that the copies of true and false in your system's /bin directory are no longer the ones that actually execute when you use these commands in a shell script. Most command shells today define these two commands as built-ins and bypass the externally defined versions. You can check this with the bash shell, for instance, by typing type true at the command prompt. On most systems, the answer will be true is a shell builtin.

If such code inflation can happen to code that's this trivial, and in some ways even redundant, what happens with code that's actually useful? I already mentioned that later versions of the default command shell on Unix and Unix-like systems picked up additional functionality with the interception of calls to true and false.

Figure 2 shows how the source code for the shell itself, measured in raw bytes, has grown, again using a log scale for the *y*-axis. From approximately 11 Kbytes in fifth-edition Unix in 1974 to 2.1 Mbytes for bash 40 years later is an increase of 191 times. Pick almost any other software application, from any domain, and you'll see the same effect.

cat –v

In the early days of Unix development, an attempt was made to reduce the number of command-line options of all standard applications. The thinking was that if additional command-line options were needed, the original code for an application probably wasn't thought out carefully enough. In 1983 at the Usenix Summer Conference, Rob Pike gave an often-quoted presentation on this topic called "Unix Style, or cat –v Considered Harmful." (For more on the presentation, visit http://harmful.cat-v.org/cat-v.) Rob noticed with some dismay that the number of options for the original cat command had increased from zero to four. That didn't help. If you check your system today, you'll see that the number of options for this same basic command has reached 12, with seven additional options that you can use as aliases to the others.

So, why does software grow? The answer seems to be: because it can. When memory was measured in Kbytes, it simply wasn't possible to write a program that consumed more than a fraction of that amount. With memory sizes now reaching Gbytes, we seem to have no incentive to pay attention to a program's size, so we don't.

Does it matter? Clearly, it doesn't matter much for the implementation of true or false, other than that we might object on philosophical grounds. But for code that matters, it might well make a difference. This brings us to the next two laws of software development: all nontrivial code has defects, and the probability of nontrivial defects increases with code size. The more code you use to solve a problem, the harder it gets for someone else to understand what you did and to maintain your code when you have moved on to write still larger programs.

Dark Code

Large, complex code almost always contains ominous fragments of "dark code." Nobody fully understands this code, and it has no discernable purpose; however, it's somehow needed for the application to function as intended. You don't want to touch it, so you tend to work around it.

The reverse of dark code also exists. An application can have functionality that's hard to trace back to actual code: the application somehow can do things nobody programmed it to do. To push the analogy with cosmology a little further, we could say that such code has "dark energy." It provides unexplained functionality that doesn't seem to originate in the code itself. For example, try to find where in the current 2.1 Mbytes of Ubuntu source code for the `bash` shell the built-in commands `true` and `false` are processed. It's harder than you might think.

Software development has one important difference from astronomy or cosmology. In our universe, we can do more than just watch and theorize: we can actually build our universe in the way we think will perform most reliably. Astronomers can't do much about the expansion of the universe other than study it. But in software development we can, at least in principle, resist the temptation to continue to grow the size of applications when there's no real need for it.

So now it's your turn. Instead of just adding more features to the next version of your code, resolve to simplify it. See if you can make the next release smaller than the last one. To get started, if you work on a Linux system, take a stand and replace the gargantuan modern version of `/bin/true` with the original empty executable file. Similarly, replace that newfangled version of `/bin/false` with the single line `exit 1`, which works just as well. You'll feel better, and you'll save some disk space. As the writer Antoine de Saint Exupéry famously noted, "Perfection is achieved not when there is nothing more to add, but when there is nothing more to remove."

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.