Reliable Code

Code Inflation

Gerard J. Holzmann

Jet Propulsion Laboratory, California Institute of Technology

Most people don't get too excited about software. To them software applications are like cars: they are 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, but not safer. So why is that?

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

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

The First Law

Software too can grow exponentially fast, especially after an initial prototype is first created. To give a simple example, each new Mars lander that NASA has launched to the red planet in the last four decades used more code than all the missions before it combined. We can see the same effect in just about every other domain of application. Software tends to grow with the passage of time, whether or not there is a rational need for it. We can call this the *"First Law of Software Development."*

We can find a remarkable example of this phenomenon if we look at the history of a command named **`true`** in Unix® and Unix-based systems. This simple command is often used in shell-scripts to enable or disable fragments of code, or to build unconditional while loops, for instance as follows to perform a sequence of random tests:

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

The commands **`/bin/true`** and **`/bin/false`** first appeared in January 1979 in the 7th Edition of the Unix® distribution from Bell Labs. They were defined as very 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, the **`true`** command was actually defined fully with an empty file. How did it work?

Since the **true** command contained nothing to execute, it always completed successfully, returning the success value *zero* to the user. The **false** command contained seven characters (including the line-feed at the end), to successfully return a non-zero value, which signifies failure.

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

It would seem that this implementation leaves nothing left to desire, but that would contradict the first law of software development that we just discussed.

In the first commercial version of Unix® from 1982, marketed under the name 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 PDP11 from 1983, the implementation of **true** grows to 18 bytes, and the empty file now contains a comment:

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

In a version of Unix® from 1984 things start heating up, and **true** has grown to 276 bytes. The contents are now a boilerplate copyright notice from AT&T claiming intellectual ownership of the otherwise still empty file.

A Solaris distribution from 2010 ups the ante still further by replacing the shell script with a C source program of 1,123 bytes, that consists of a main procedure that calls the function **_exit(0)**. The C program for the command **false** similarly has main call **_exit(255)**. Both programs also contain a hefty copyright notice. If I compile these programs on my system today, the executables tap in at 8,377 bytes each.

And we're not done yet. The executable for the most recent version of **true** on my Ubuntu system is no less 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 now also grown to 2,367 bytes, and includes four headerfiles, one of which is itself 16 Kbytes of text. That is quite a change from the zero bytes from 1979, and all that without any significant difference in functionality. And if you're still on the fence with this: no, the **true** command really does not need a **–version** option to explain which version of the truth the command currently represents, and nor does it need a **–help** option, whose only purpose seems to be to explain the unneeded **–version** option. And just in case someone was thinking about this: the **true** and **false** commands also don't need an option that can invert the result, or one that would allow it to send its 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® tool-box just yet, but we do seem to have gotten close.

Table 1 shows how both the source code size and the size of this command have grown over the years. Figure 1 shows the growth of the executable program as a graph. Note that the y-axis is a log-scale so that the early numbers are not completely drowned out by the later ones.

Table 1 Growth of the Source and Executable of the Unix **`true`** *command.*

Year	Source	Executable
1979	0	0
1983	18	18
1984	276	276
2010	1123	8377
2012	2367	22896

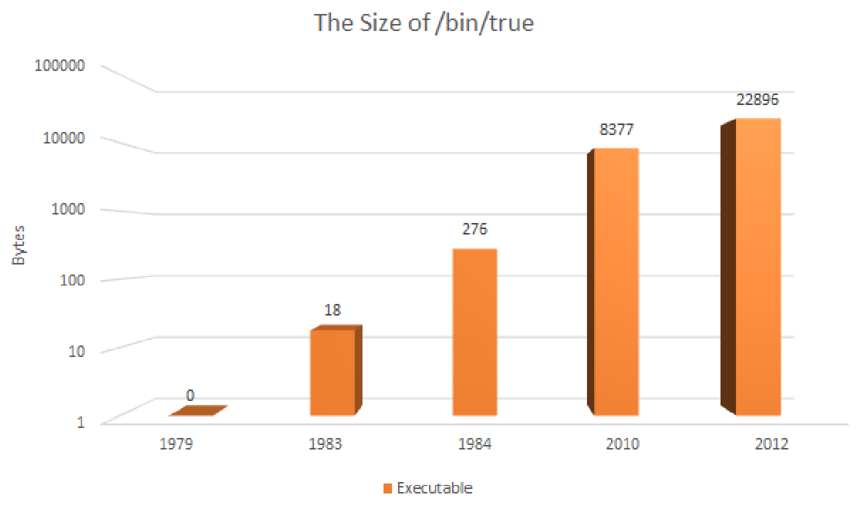

Figure 1 The size of /bin/true, measured in bytes

Just like in the theory of inflation, the implementation of **`/bin/true`** increased infinitely fast in the first few years since it was created (since, like the universe, it started at a size of zero). Okay, we're not talking 10^{-32} seconds, but we're moving more at humanly achievable speeds here. Once we got to a non-zero size, the expansion continued steadily, producing an increase in size of over three orders of magnitude since 1983.[2,3]

The best part of all this is perhaps that the copies of **`true`** and **`false`** that are available in the **`/bin`** directory of your system are no longer the ones that are actually executed when you use these commands in a shell script. Most command shells today define these two commands as builtins, and bypass the externally defined versions completely. 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`**."

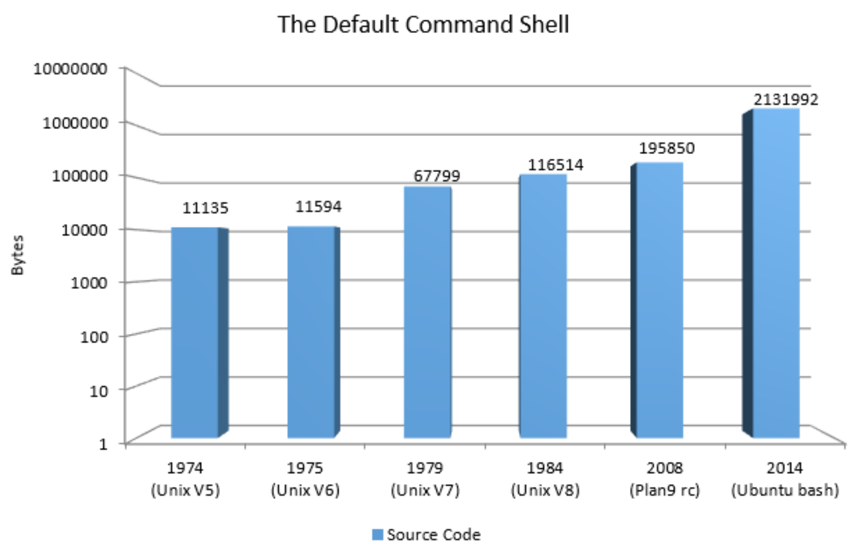

Figure 2 The size of the source code for default command shell on Unix and Unix-like systems, measured in bytes.

If this type of code inflation can happen to code that is this trivial, and in some ways even redundant, what happens with code that is actually useful? We 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 over the years, again using a log-scale for the y-axis. From about 11K bytes in 5th Edition Unix in 1974, to 2.1M bytes for bash forty years later: an increase of 191 times. You can pick almost any other software application, from any domain at all, and you will see the same effect.

cat –v

In the early days of Unix® development a deliberate attempt was made to reduce the number of command-line options of standard applications. The thinking then was that if additional command-line options were needed, this likely meant that the original code for an application wasn't thought out carefully enough. In 1983, Rob Pike gave an often quoted presentation on this topic at the USENIX Summer Conference, titled "*Unix Style, or cat –v Considered Harmful*."[1] At that time Rob noticed with some dismay that the number of options to the original **`cat`** command had increased from zero to *four*. It didn't help. If you check on your system today, you'll see that the number of options to this same basic command has meanwhile reached *twelve*, with *seven* additional options that can be used as aliases to the others.

So, why does software grow with time? The answer seems to be: because it can. When memory sizes were measured in Kilobytes, it was simply not possible to write a program that consumed more than a fraction of that amount. With memory sizes now reaching into the Gigabytes, there does not seem to be any incentive to pay attention to the size of a program, and so we don't.

Does it matter? Clearly it does not matter much for the implementation of **`true`** or **`false`**, other than that we may want to object on philosophical grounds. But for code that matters it may well make a difference. This brings us to the second and the third law of software development. The second law of software development says that *all non-trivial code has defects*, and the third law says that *the probability of non-trivial 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 and complex code almost always contains ominous fragments of 'Dark Code' that nobody fully understands and that have no discernable purpose, but that are somehow needed for the application to function as intended. You do not want to touch the dark code in an application, so you tend to work around it.

The reverse of dark code also exists. There can be functionality in an application that is hard to trace back to actual code: the application is somehow able to do things that nobody programmed it to do. If we want to push our analogy with cosmology a little further we could say that such code has 'Dark Energy.' It provides unexplained functionality that does not seem to originate in the code itself. As an example of code with this mysterious type of dark energy, you can try to find where in the current 2.1 Million bytes of Ubuntu source code for the **`bash`** shell the built-in commands **`true`** and **`false`** are processed. It is harder than you might think.

There is one important difference between astronomy or cosmology and software development. The difference is that 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 cannot do much about the expansion of the universe, other than to study it. But in software development we can, at least in principle, resist the temptation to continue to grow the size of applications just because we can, even when there is no real need for it.

Your Turn

So, if you've read this far, let's resolve to simplify the next version of our code instead of just adding more features to it. See if you can make the next release of your application smaller than the one before it. To get started, if you work on a Linux system, take a stand and remove the gargantuan modern version of **`/bin/true`** and replace it with the original empty executable file. Similarly, toss that new-fangled version of **`/bin/false`** and replace it with the single line "**`exit 1`**" that works just as well. You'll feel better, and you'll save yourself some disk-space. As 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.*"

References

1. `http://harmful.cat-v.org/cat-v/`
2. 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`
3. An online archive of many early Unix® source code distributions can be found at: `http://minnie.tuhs.org/cgi-bin/utree.pl`
4. Antoine de Saint Exupéry was a French writer who lived from 1900-1944.

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.