



Software Components

Gerard J. Holzmann

AT THE 1968 Conference on Software Engineering, mathematician and software engineer Doug McIlroy, alarmed by the sorry state of software development, made a strong pitch for the industrial production of software components.¹ Software systems, like bridges, houses, and cars, are built from parts. McIlroy noted that it didn't make much sense for every organization and developer to keep having to reinvent what's basically a common set of core components for software design. McIlroy envisioned an industry that could provide programmers a selection of mass-produced software parts, differing in accuracy, performance, and cost, to fit a broad range of possible applications.

An inspiration for McIlroy's presentation was the way in which the electronics industry had evolved. Electronics were commonly designed as sets of circuit boards populated with standardized components. There were, and still are, catalogs of resistors, capacitors, diodes, and transistors, with each item documented and marked with the intended range of use. For instance, resistors are marked with a standard color code that indicates their nominal value and the percentage by which their actual value can differ from that nominal value. The user can then make the tradeoff between paying somewhat more for greater precision or less when the highest level of

accuracy isn't required. Why couldn't the same thing work in software?

Bricks and Bolts

Resistors, capacitors, and transistors can all be mass produced, like nuts and bolts and bricks, because they're typically used in large quantities. Every device built does, of course, have to use its own copy of all the components it uses, although all those copies are expected to have been produced to the same standards. Just about all circuit board components are standard. Rarely will a circuit designer have to develop an entirely new type of component that's not available in any catalog.

The situation is different in software. The developer needs only one encoding of each standard function, even if the software application that contains it is sold by the millions. The way to achieve accuracy in component design is also different. For instance, to get highly accurate resistors, we can measure the resistance of millions of items and select the ones that deviate by no more than the desired amount to achieve any level of accuracy. If we want to achieve greater accuracy for a software function, the best method is probably not to have large numbers of developers design a version and then select the best one, although in practice it does sometimes seem to work that way. It can be time consuming to rigorously prove the correctness of a software

component, but once that work is done, the result should hold for every copy that's used later.

Another difference is that the large majority of the code written for a new application is usually unique to that application. In one of the larger programs I've maintained for a couple of decades, less than 15 percent of the functions originate in standard libraries. The remaining 85 percent are special-purpose functions unique to the application, starting with the main function itself. I suspect that the same is true for most software applications.

Libraries

The mass production in McIlroy's proposal referred to the creation of a larger-than-usual range of specialized features, target execution platforms, and environments that could be created for each basic routine. A good case can be made for this, but there doesn't appear to be a market for this type of software component industry. Pretty much all languages come freely with extensive libraries that encode everything from the most standard to the most exotic types of applications. Today, almost no one would consider writing his or her own library of trigonometric functions or regular-expression pattern matching.

Can we trust the reliability of all those libraries? Is the standard implementation of a sine routine in

Java identical to the one that's available for C or Go? Who exactly takes responsibility for the accuracy of these functions, and how can you get bugs repaired quickly? After all, if you're not a paying customer, you no longer get to decide what's a bug and what's a feature, so you might just have to live with whatever the anonymous provider of the library decides. Unless, of course, you do decide to build everything from scratch.

Modules and Subsystems

Something else that might have changed since 1968 is the notion of what a software component actually is. To a carmaker, a component is the entire entertainment subsystem, the navigation module, the airbag unit, or the engine control unit. Today, no carmaker designs, builds, and programs its own versions of these modules: it purchases them from contractors that specialize in building them and that are responsible for fixing them if they're inadequate. Each of these modules typically comes with substantial amounts of embedded software to bring the hardware to life and give it its desired functionality. If there's a problem, your favorite repair place will not go hunting for any bugs, locate that burned-out transistor, or even try to upload new software into one of these modules. The repair shop will simply replace the entire module. Problem solved. In all these cases, a component is a subsystem, software included.

In the last 50 years, an industry has also been created for the development of specialized software tools. So, here we might need to consider specialized tools such as Photoshop or TurboTax to be identifiable software "components." Who

would try to develop his or her own image-editing software, text editor, logic-model checker, or static-source-code-analysis tool today? Well, okay, that would be people like me, but we know we're a very small minority.

Software Tools

The notion of a software tool for solving problems was exemplified best in the design of Unix. If you're used to developing code on a Unix or Unix-like system, you'll have come to rely on standard tools such as `make`, `grep`, `sed`, `awk`, `sort`, `diff`, and `tr`. Each of these specialized tools can be used as a component part for solving larger problems. Each tool aims to solve one specific problem as efficiently as possible. And each tool is designed to support a standard I/O format, so that the output from one tool can be fed into any of the others.

We needn't be surprised that the design of Unix, spearheaded by Ken Thompson, originated in the group at Bell Labs that was created by Doug McIlroy. In this group, which I was fortunate to be part of, Doug McIlroy was a gentle force of inspiration behind a lot of the research that came out in subsequent years. As is well known, McIlroy also contributed the key concept of a Unix pipe as a simple notation to connect the standard output of one tool to the standard input of another. The pipe was the glue that was needed to let us build larger software systems from software components.

The Perpetual Crisis

The rapidly increasing size and complexity of software applications was discussed at length at the 1968 Conference on Software Engineering, famously leading to the first recorded

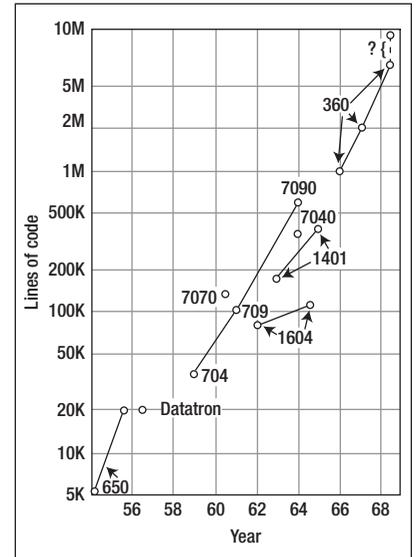


FIGURE 1. The growth in software requirements.¹ The growth continues, although the pace has slowed.

use of the term “software crisis.”¹ As noted in the discussion transcripts,

*Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death.*¹

You probably would have believed me if I said that quote was from last year instead of 50 years ago.

An example of a large software system that was discussed at the 1968 conference was the OS for the IBM System/360. A chart (see Figure 1) illustrated the trend. The chart showed the size of IBM OS/360 increasing from around 1 million to around 7 million lines of assembly code, or the equivalent of about 1.4 million lines of C code today. The chart also illustrated the general trend of software size increasing by about a factor of 100 from 1958 to

ABOUT THE AUTHOR



GERARD J. HOLZMANN works on developing stronger methods for the design and analysis of safety-critical software as a consultant and researcher at Nimble Research. Contact him at gholzmann@acm.org.

1968. Luckily, that pace of growth didn't continue, or we would be seeing software applications with tens of billions of lines of code.

But clearly, the growth trend hasn't stopped, and we still feel we're nearing the point at which we'll lose all intellectual control over the software systems we're routinely creating. Of course, a well-designed large system isn't a single homogenous blob of intertwined code. It consists of many parts with, hopefully, well-designed interfaces and a limited number of functional dependencies so that all these components can be designed and checked independently. Anyhow, we can always hope.

Speed

One thing that has changed dramatically since 1968 is the speed and size of the machines we can use to execute our code. Even a humble Raspberry Pi C executes about 2,000 times faster than the fastest IBM System/360 model, the Model 75, did in 1965. For that matter, that Raspberry Pi is also more than 10 times faster than a Cray-1 supercomputer from 1975.²⁻⁴

Looking on the bright side again, these phenomenal gains in speed not only make our code run faster but also make it possible to analyze our code more thoroughly than ever

before. In early compilers, for instance, many checks for consistency and correctness that could in principle have been included were left out to avoid slowing down compilation too much. Even so, in the '60s and '70s you could often literally take a coffee or lunch break before a larger application would finish compiling. Today's compilers can execute fast enough that they can routinely perform far more sophisticated types of checks of our code, without anyone noticing a difference in performance. And we now also have a range of dedicated static-source-code-analysis tools for even more rigorous types of checks. These tools are good enough that they've pretty much become standard in industrial software development.

Indeed, many application domains have a well-defined process for building larger software systems from reusable modules with well-defined interfaces. For instance, if you build spacecraft, the mission of the system generally differs from one vehicle to the next, but all the missions need certain specific types of functionality. Standard software modules needed on every interplanetary mission include the code for navigation (getting from Florida to Mars without too many course corrections), telemetry (getting mission

data back to the ground controllers), resource arbitration (don't start driving your rover when it's busy drilling into a rock), and data handling (compression, packetization, and so on).

In the long run, it pays to develop generic software components for all these functions as robustly as possible, and to put them in the arsenal of software components on which mission designers can rely. With time and experience, these modules can be expected to get better and better, and handle more and more cases.

It should go without saying that this approach to the development of software components is well worth using, even if your job is not to fly the occasional mission to Mars. 

References

1. *Software Engineering*, P. Naur and B. Randell, eds., Scientific Affairs Division, NATO, 1968; homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.
2. L. Poulsen, "IBM 360/370/3090/390 Model Numbers"; www.beagle-ears.com/lars/engineer/comphist/model360.htm.
3. J. Armstrong, tweet, 17 Jan. 2018; twitter.com/joeerl/status/953711344783691777.
4. R. Longbottom, "Roy Longbottom's Raspberry Pi, Pi 2 and Pi 3 Benchmarks"; [www.roylongbottom.org.uk/Raspberry percent20Pi percent20Benchmarks.htm](http://www.roylongbottom.org.uk/Raspberry%20Pi%20Benchmarks.htm).

myCS

Read your subscriptions through the myCS publications portal at

<http://mycs.computer.org>