20
Software Components: Only The Giants Survive[1]

Butler W. Lampson

Abstract

For many years programmers have dreamed of building systems from a library of reusable software components together with a little new code. The closest we've come is Unix commands connected by pipes. This paper discusses the fundamental reasons why software components of this kind have not worked in the past and are unlikely to work in the future. Then it explains how the dream has come true in spite of this failure, and why most people haven't noticed.

Introduction

People have been complaining about the "software crisis" at least since the early 1960's. The famous NATO software engineering conference in 1968 brought the issue into focus, and introduced the term "software engineering". Many people predicted that software development would grind to a halt because of our inability to handle the increasing complexity; of course this has not happened.

What is often overlooked is that the software crisis will always be with us (so that it shouldn't be called a "crisis"). There are three reasons for this:

- As computing hardware becomes 100 times more powerful every decade (because of Moore's law), new applications become feasible, and they require new software. In other branches of engineering the pace of change is much slower.

---

1 This paper is based on a keynote address given at the 21st International Conference on Software Engineering, 1999. It was written for a symposium in honor of Roger Needham, February 2003, and published in *Computer Systems: Theory, Technology, and Applications*, K. Sparck-Jones and A. Herbert (editors), Springer, 2004, pp 137-146.

- Although it's difficult to handle complexity in software, it's much easier to handle it there than elsewhere in a system. A good engineer therefore moves as much complexity as possible into software.
- External forces such as physical laws impose few limits on the application of computers. Usually the only limit is our inability to write the programs. Without a theory of software complexity, the only way to find this limit is trial and error, so we are bound to over-reach fairly often. "A man's reach should exceed his grasp, or what's a heaven for."—Browning.

At the 1968 NATO conference, Doug McIlroy proposed that a library of software components would make programming much easier [7]. Since then, many people have advocated and worked on this idea; often it's called "reusable software", though this term has other meanings as well. Most recently, the PITAC report [9] proposed a major research initiative in software components. This paper explains why these ideas won't work.

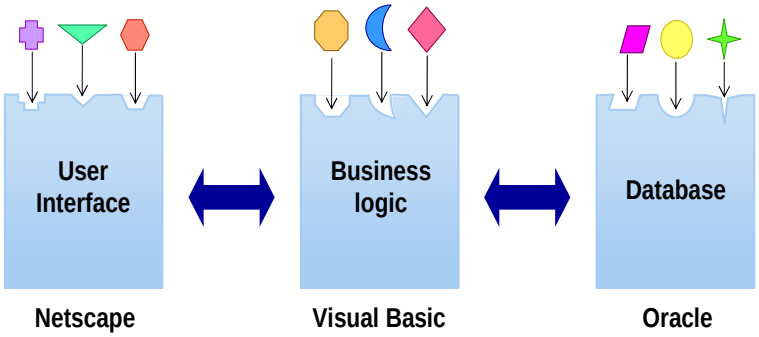


Figure 1: A typical business application

How much progress has there been in software in the last 40 years? Either a little or a lot: the answer depends on what kind of software you mean.

A little, if you are writing a self-contained program from scratch, or modifying an existing self-contained program. The things that help the most are type-safe languages such as Pascal and Java, and modules with clean interfaces [8]; both have been around for 30 years. Program analysis tools help with modifications, and they have been improving steadily [3].

A lot, if you are doing a typical business computing application. You build your application on top of a few very large components: an operating system (Linux or Windows), a browser (Netscape or Internet Explorer), a relational database and transaction processor (DB2, Oracle, or SQL Server), and a rapid application development system (Visual Basic or Java); see Figure 1. You use

only a small fraction of the features of each component, and your program consumes 10 or 100 times the hardware resources of a fully custom program, but you write 10% or 1% of the code you would have written 30 years ago. Certain kinds of domain-specific programs are also dramatically easier. If a spreadsheet, SQL, Matlab, Mathematica, or HTML is a good match for your problem, again you can write your program 10 or 100 times more easily.

The component library: Dream and reality

McIlroy's idea was a large library of tested, documented components. To build your system, you take down a couple of dozen components from the shelves and glue them together with a modest amount of your own code.

The outstanding success of this model is the Unix commands designed to be connected by pipes: `cat`, `sort`, `sed`, and their friends [6]. There are quite a few of these, and you can do a lot by putting them together with a small amount of glue, usually written in the shell language. McIlroy [1] gives a striking example. It works because the components have a very simple interface (a character stream, perhaps parsed into lines or words) and because most of them were written by a single tightly-knit group. Not many components have been added by others.

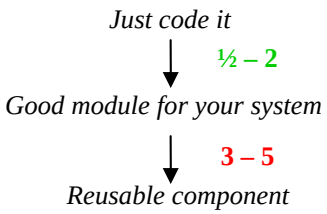
Another apparent success is the PC hardware industry. PC's are built from (hardware) components: processor and chipset, DRAM SIMM, hard disk, monitor, graphics card and driver, etc. Manufacturers really do slap these components together to make systems. Reality is uglier than appearance, though. Only a few components really work well, the ones that can be tested adequately by running Windows on them for a few days. Others cause lots of problems, as anyone knows who has tried to build a PC. And Microsoft is responsible for the integrity of the PC ecosystem.

For the most part, component libraries have been a failure, in spite of much talk and a number of attempts. There are three major reasons for this:

- There's no business model.
- It costs a client too much to understand and use a component.
- Components have conflicting world views.

No business model

Design is expensive, and reusable designs are very expensive. It costs between ½ and 2 times as much to build a module with a clean interface that is well-designed for your system as to just write some code, depending on how lucky you are. But a reusable component costs 3 to 5 times as much as a



good module. The extra money pays for:

- Generality: A reusable module must meet the needs of a fairly wide range of 'foreign' clients, not just of people working on the same project. Figuring out what those needs are is hard, and designing an implementation that can meet them efficiently enough is often hard as well.
- Simplicity: Foreign clients must be able to understand the interface to a module fairly easily, or it's no use to them. If it only needs to work in a single system, a complicated interface is all right, because the client has much more context.
- Customization: To make the module general enough, it probably must be customizable, either with some well-chosen parameters or with some kind of programmability, which often takes the form of a special-purpose programming language.
- Testing: Foreign clients have higher expectations for the quality of a module, and they use it in more different ways. The generality and customization must be tested as well.
- Documentation: Foreign clients need more documentation, since they can't come over to your office.
- Stability: Foreign clients are not tied to the release cycle of a system. For them, a module's behaviour must remain unchanged (or upward compatible) for years, probably for the lifetime of their system.

Regardless of whether a reusable component is a good investment, it's nearly impossible to fund this kind of development. It's not necessary for building today's system, and there's no assurance that it will pay off.

It's also very difficult to market such components:

- There are many of them, so each one gets lost in the crowd.
- Each client needs a number of them, so they can't be very expensive.
- Each one is rather specialized, so it's hard to find potential customers.

Cost to understand

To use a component, the client must understand its behaviour. This is not just the functional specification, but also the resource consumption, the exceptions it raises, its customization facilities, its bugs, and what workarounds to use when it doesn't behave as expected or desired. One measure of this cost is the ratio of the size of a complete specification (which of course seldom exists) to the size of the code. For a modest-sized component, this ratio is usually surprisingly large.

Furthermore, because the written spec is almost always quite inadequate, there is uncertainty about the cost to discover the things that aren't in the spec, and about the cost to deal with the surprises that turn up. If the module has been around for a while and has many satisfied users, these risks are of course smaller, but it's difficult to reach this happy state.

The client's alternative is to recode the module. Usually this is more predictable, and problems that turn up can often be handled by changing the module rather than by working around them. This is probably feasible if the module is built as part of the same project, but impossible if it's a reusable component.

Conflicting world views
The interface to a component embodies a view of the world: data types, resource consumption, memory allocation, exception handling, etc. If you take 10 components off the shelf, you are putting 10 world views together, and the result will be a mess. No one is responsible for design integrity, and only the poor client is responsible for the whole thing working together. There can easily be n^2 interactions among n components.

Good things that aren't reusable components

People often ask "What about Corba and COM; aren't they successful?" Perhaps they are, but they are ways to run components, not components themselves. They play the role of a linker and a calling convention for distributed computing.

The "components" that you can get for Visual Basic, Java, Microsoft Office, and browsers are not reusable components either. You can use a couple of them in your system, but if you use 10 of them things will fall apart, because they are not sufficiently robust or well-isolated. If you don't believe this, try it for yourself.

Nor is a module with a clean interface a reusable component, for all the reasons discussed above. A clean interface is a very good thing, and it's certainly necessary for a reusable component, but it's not sufficient.

Platforms

The last section shows why a public library of software components is not possible. Some less ambitious things have worked, however. Most of them are variations on the idea of a *platform*, which is a collection of components on top of which many people can build programs, usually application programs. Windows, Linux, Java, DB2, Microsoft Office, OpenGL, the IMSL numerical library, and PC hardware are examples of platforms. So, on a smaller scale, are the Unix shell and text processing commands discussed in the introduction.

The essential property of a platform is that someone takes responsibility for its coherence and stability. Often this is a vendor, motivated by the fact that having lots of application expands the market for the platform. It can also be a community, as in the case of Linux or OpenGL, in which component builders

are motivated by status in the community or by the fact that they are also clients. A platform needs a shared context that everyone understands and a common world view that everyone accepts; this means that its community must include both the component builders and many of the clients. A shared context is much easier when the domain is narrow and there's a clean mathematical model, as with graphics or numerical libraries.

Sometimes people try to build lots of components on a common and hospitable platform, such as Visual Basic or Java. This can work if the components come from (or pass through) a single source that takes responsibility for their coherence. Otherwise the problems of too little generality, cost to understand, and conflicting world views make it impossible to use more than two or three of them in a system.

Big components

As we saw in the introduction, big components like browsers and database systems do work, even though a library cannot. They are five million lines of code and up, so huge that you only use three or four of them: Linux or Windows, Netscape or Internet Explorer, Oracle or DB2, Visual Basic or Java. How do they overcome the problems with component libraries?

Business model: There's a market for such big things. Lots of people need each one, there are only a few of them, and the client only has to buy a couple of them, so marketing is feasible. Building your own, on the other hand, is not feasible, even if you only use 1% of the features: 1% of 20 million lines is still 200,000 lines of code to write, and that's a low estimate of the amount of code for 1% of the features.

Cost to understand: The specification may be large and complicated, but it is much smaller than the code. Because the market is large, vendors can afford to invest in documentation; in fact, every such component has a mini-industry of books about it. They can also afford to invest in customization: operating systems have applications and scripting languages, browsers have scripts, Java, plug-ins, and dynamic HTML, and database systems have SQL.

Conflicting world views: If you use three of them, there are only three pairwise interactions, and only two if they are layered. The vendor provides design integrity inside each big component.

In fact, big components, along with transaction processing, spreadsheets, SQL, and HTML, are one of the great successes of software in the last 20 years.

People often complain about big components because they are wasteful. A business application built on a browser and a database system can easily con-

sume 100 times the resources of one that is carefully tailored to the job at hand. This is not waste, however, but good engineering. There are plenty of hardware resources; what's in short supply are programmers and time to market, and customers care much more about flexibility and total cost of ownership than about raw hardware costs.

Another way to look at this is that today's PC is about 10,000 times bigger and faster than the Xerox Alto [10], which it otherwise closely resembles. It certainly doesn't do 10,000 times as much, or do it 10,000 times faster. Where did the cycles go? Most of them went into delivering lots of features quickly, which means that you can't have first-class design everywhere. Software developers trade hardware resources for time to market. A lot of them also went into integration (for example, universal character sets and typography, drag and drop, embedding spreadsheets in text documents) and into compatibility with lots of hardware and with lots of old systems. And a factor of 10 did go into faster responses to the user's actions.

What else could work?

If components can't help us much to build software, what can? Two approaches are promising: declarative programming, and specifications with teeth.

Declarative programming

"Declarative program" is not a precise concept, but the idea is that the program is close to the specification, perhaps even the same. For example, in a simple spreadsheet the program is just the formulas; if there is no higher structure, the formulas express the user's intent as simply as possible. Of course, if the user's intent was "a capital gains worksheet with data from my brokerage account", the raw spreadsheet has a lot of extra detail. On the other hand, when equipped with suitable templates Excel can come fairly close to that intent.

Other examples of declarative programming are the query language of SQL, a parser generator like YACC, a system for symbolic mathematics like Mathematica, and a stub generator for remote procedure call. What they have in common is that what you have to tell the system is closer to your intent than an ordinary program. This makes programming faster and more reliable. It also opens up opportunities for analysis and optimization; parallel implementations of SQL are a good example of this.

Specifications with teeth

Specifications are useful as documentation, but they have the same problem as all documentation: they are often wrong. A spec is more valuable if it has teeth, that is, if you can count on its description of the program's behaviour. Such specs are much more likely to pass Parnas' coffee-stain test: the value of a spec

is proportional to the number of coffee-stains on the implementers' copies. A type declaration is an example of a spec with teeth.

Teeth mean tools: the computer must check that the spec is satisfied. There are two kinds of teeth: statically checked, and dynamically enforced by encapsulation. A type-safe language, for example, usually is mostly statically checked, but has dynamic checking of some casts. Static checks are better if you can get them, since they guarantee that the program won't crash in Peoria. We are slowly learning how to check more things statically.

Encapsulation takes many forms. The simplest and most familiar is the sandboxing provided by operating system processes or Java security permissions. Much more powerful is the automatic concurrency, crash recovery, and load balancing that a transaction monitor provides for simple sequential application programs [5]. Another example is the automatic Byzantine fault-tolerance that a replicated state machine can provide for any deterministic program [4].

Conclusion

A general library of software components has been a long-standing dream, but it's unlikely to work, because there's no business model for it, it costs the client too much to understand a component, and components have conflicting world views. In spite of this discouraging conclusion, very large components do work very well, because they have lots of clients and you use only three of them.

Two other approaches can make software easier to write: declarative programming, and specifications with teeth. The latter guarantee something about the behaviour of a module. The enforcement can be done statically, as with a type checker, or dynamically, as with transaction processing.

References

1. BENTLEY, J., KNUTH, D., AND MCILROY, M.D., 'A literate program,' *Comm. ACM*, vol. 29, no. 6, June 1986, pp. 471–483.
2. BROOKS, F., 'No silver bullet,' *IEEE Computer*, vol. 20, no. 4, April 1987, pp. 10–19. Reprinted in Brooks, The mythical man-month, 2nd ed., Addison-Wesley, 1995.
3. BUSH, W., PINCUS, J., AND SIELAFF, D., 'A static analyzer for finding dynamic programming errors,' *Software—Practice and Experience*, vol. 30, no. 7, June 2000, pp. 775–802.
4. CASTRO, M. AND LISKOV, B., 'Practical Byzantine fault tolerance and proactive recovery,' *ACM Trans. Computer Systems* vol. 20, no. 4, October 2002, pp. 398–461.
5. GRAY, J. AND REUTER, A., Transaction processing, Morgan Kaufman, 1993.
6. KERNIGHAN, B. AND PIKE, R., The Unix programming environment, Prentice-Hall, 1984.
7. MCILROY, M.D., 'Mass produced software components,' In P. Naur and B. Randell, eds., *Software engineering*, Report on a conference sponsored by the NATO Science

Committee, Garmisch, Germany, October 1968, Scientific Affairs Division, NATO, Brussels, 1969, pp. 138-155.
http://www.cs.dartmouth.edu/~doug/components.txt

8. PARNAS, D., On the criteria to be used in decomposing systems into modules. *Comm. ACM*, vol. 15, no. 12, December 1971, pp. 1053–1058.
9. President's Information Technology Advisory Committee, information technology research: investing in our future.
http://www.ccic.gov/ac/report/
10. THACKER, C., Personal distributed computing; The Alto and Ethernet hardware, in A history of personal workstations, A. Goldberg, ed., Addison-Wesley, 1988, pp. 267–290.