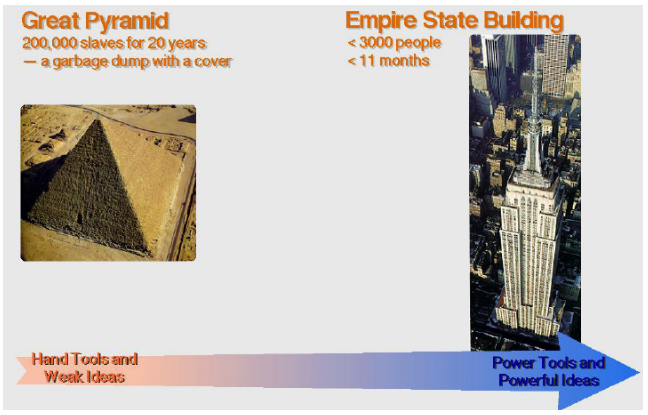
Is “Software Engineering” an Oxymoron?

By Alan Kay

http://web.archive.org/web/20030407181600/www.opencroquet.org/downloads/Croquet0.1.pdf

Real Software Engineering is still in the future. There is nothing in current SE that is like the construction of the Empire State building in less than a year by less than 3000 people: they used powerful ideas and power tools that we don’t yet have in software development. If software does “engineering” at all, it is too often at the same level as the ancient Egyptians before the invention of the arch (literally before the making of arches: architecture), who made large structures with hundreds of thousands of slaves toiling for decades to pile stone upon stone: they used weak ideas and weak tools, pretty much like most software development today.


Great Pyramid
200,000 slaves for 20 years
— a garbage dump with a cover
Empire State Building
< 3000 people
< 11 months
Hand Tools and Weak Ideas
Power Tools and Powerful Ideas

Is there anything worthwhile and powerful between the ancient Egyptians and the Empire State Building?

The real question is whether there exists a practice in between the two —- stronger than just piling up messes —- that can eventually lead us to real modern engineering processes for software.

One of the ways to characterize the current dilemma is that every project we do, even those with seemingly similar goals has a much larger learning curve than it should. This is partly because we don’t yet know what we really need to know about software. But as Butler Lampson has pointed out, this is also partly because Moore’s Law gives us a qualitatively different environment with new and larger requirements every few years, so that projects with similar goals are quite different.

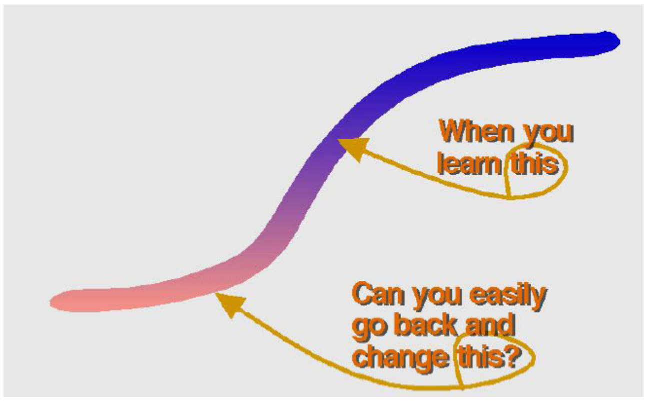
The *Fram oil filter* principle

Very often during a project we’ll find out something that we wished we’d known when designing the project. Today, much of the key to more rapid construction and higher success is what we do with this new knowledge. In the ”Egyptian” SW systems, we pretty much have to start over if we find a basic new way to deal with the foundation of our system. This is so much work that it is generally not done.

Something quite similar obtains in the ominous fact that 85% of the total cost of a SW system is incurred after it has been successfully installed. The bulk of the cost comes from the changes needed from new requirements, late bugs, and so forth. What seemed expedient and cost saving early on, winds up costing exponentially more when the actual larger life cycle picture is taken into account.

Late binding

Until real software engineering is developed, the next best practice is to develop with a dynamic system that has extreme late binding in all aspects. The first system to really do this in an important way was LISP, and many of its great ideas were used in the invention of Squeak's ancestor Smalltalk -- the first dynamic completely object-oriented development and operating environment -- in the early 70s at Xerox PARC. Squeak goes much further in its approach to latebinding.



Late binding allows ideas learned late in project development to be reformulated into the project with exponentially less effort than traditional early binding systems (C, C++, Java, etc.)

One key idea is to keep the system running while testing and especially while making changes. Even major changes should be incremental and take no more than a fraction of a second to effect. Various forms of "undo" need to be supplied to allow graceful recovery from dangerous changes, etc.

Another key idea is to have a generalized storage allocator and garbage collector that is not only efficient in real-time (so that animations and dynamic media of many kinds can be played while the gc is collecting), but that allows reshaping of objects to be done safely. For example, you have been working on your project for more than a year, and many important things have been built. Can you add several instance variables to a key class that has hundreds of thousands of working instances and have these reshaped on-the-fly without crashing the system? In Squeak this is done all the time, because its storage allocation system was designed for late-bound reshaping, as well as good realtime performance.

Modeling and models of all ideas

A radical key idea is to make the development system have a model of itself so that it can be extended as new ideas occur during development. This means that there are not only classes for workaday tools, such as UI widgets, but there are also classes that represent the metastructure of the system itself. How are instances themselves made? What is a variable really? Can we easily install a very different notion of inheritance? Can we decide that prototypes are going to work better for us than classes, and move to a prototype-based design?

An extension of modeling basic building blocks is to also model and simulate larger facilities. For example, the Squeak Virtual Machine specification is not represented as a paper document (it's not even clear that a 1/2 page paper specification could be made without error just by desk checking). Instead, the Squeak VM specification is represented as a working simulation model within itself that can be run and debugged. New facilities, such as adding FM musical synthesis are not created by writing in C or other lower level languages, but by modeling the new processes in Squeak itself as a module for the VM specification. This is debugged at the high Squeak level (in fact, the VM simulator is still fast enough to run 3 or 4 separate real-time voices). Then a mathematical translator is brought to bear which can make a guaranteed translation from the high level model to low-level code that will run on the two dozen or so platforms Squeak runs on. Further changes are implemented in the VM simulator and become part of the "living spec", then translated as described above, etc.

Fewer features, more *meta*

If we really knew how to write software, then the current (wrong) assumption that we can define all needed features ahead of time in our development system would be sufficient. But since we don't really know how to do it yet, what we need is just the opposite of a feature-laden design-heavy development system: we need one "without features", or perhaps a better way to say this is that we need a system whose features are "mostly meta". Unfortunately, most software today is written in feature-laden languages that don't cover the space, and the needed extensions are not done: partially from ignorance and partially from difficulty.

For example, C++ is actually set up for truly expert programmers, but is generally used by non-experts in the field. E.g. "new" can be overridden and a really great storage allocation system can be written (by a really great programmer). Similarly other features intended as templates for experts can be overridden. It is a source of frustration to Bjarne Stroustroup, the inventor of C++, that most programmers don't take advantage of the template nature of C++, but instead use it as a given. This usually leads to very large, slow, and fragile structures which require an enormous amount of maintenance. This was not what he intended!

Squeak takes a different route by supplying both a deeper meta system and more worked out initial facilities (such as a really good real-time incremental garbage collector and many media classes, etc.). In many cases, this allows very quick development of key systems. However, it is also the case that when deep extensions are needed, many programmers will simply not take advantage of the many mechanisms for change provided. The lack of metaskills hurts in both languages.

Part of the real problem of today's software is that most programmers have learned to program only a little, at the surface level, and usually in an early bound system. They have learned very little (or nothing) about how to metaprogram —- even though metaprogramming is one of the main keys today for getting out of software's current mess.

What are the tradeoffs of learning "calculus"?

Part of the key to learning Squeak is not just to learn the equivalent mechanisms and syntax for workaday programs, but to learn how to take new ideas and reformulate whatever is necessary to empower the system with them. An analogy that works pretty well is to view Squeak as a new kind of calculus. The calculus is not the same as simply a better algebra or geometry, because it actually brings with it new and vastly more powerful ways of thinking about mathematical relationships. Most people will not get powerful in calculus in a month, but many can get a very good handle on some of the powers in a year if they work at it.

One hundred years ago structural engineers did not have to learn the powerful form of calculus called Tensor Calculus, a very beautiful new mathematics that allows distortions, stresses and strains of many different materials (including seemingly empty space) to be analyzed and simulated. But, by 50 years ago, all structural engineers had to get fluent in this new math to be certified as a real engineer. Many of the most powerful and different ideas in Squeak are like those in Tensor Calculus (though easier, we think). They are not part of current practice in spite of their power, but will be not too far in the future.

Most of current practice today was invented in the 60s

It is worth noting the slow pace of assimilation and acceptance in the larger world. C++ was made by doing to C roughly what was done to Algol in 1965 to make Simula. Java is very similar. The mouse and hyper-linking were invented in the early sixties. HTML is a markup language like SCRIBE of the late 60s. XML is a more generalized notation, but just does notationally what LISP did in the 60s. Linux is basically Unix, which dates from 1970, yet is now the "hottest thing" for many programmers. Overlapping window UIs are one of the few ideas from the seventies that has been adopted today. But most of the systems ideas that programmers use today are from the data- and server-centric world of the 60s.

The lag of adoption seems to be about 30 years for the larger world of programming, especially in business. However, the form and practice of Squeak as we know it today was developed in the mid70s and has not yet had its 30-year lag play out. Peer-peer computing was also developed by the ARPA/PARC community around the same period and is just being experimented with in limited ways. Building systems that work the way the Internet works is not yet part of current practice, etc. (This general observation about the "30 year lag" could be facetious, but it seems amazingly true to history so far.)

Perhaps the most important fact about the new kind of software development as exemplified by Squeak is that a small group of computer scientists have gained quite a bit of experience over the last 25+ years. Thus these are no longer the philosophical speculations of the late sixties, but instead are now a body of knowledge and examples about a wide variety of dynamic computing structures. This makes these new ways much more "real" for those who pride themselves on being practical. For example, a paradigmatic system such as generalized desktop publishing can be examined in both worlds, and some answers can be given about why the late bound one in Squeak is more than 20 times smaller, took even less fractional person-time to create, and yet still runs faster than human nervous systems at highly acceptable speeds. This will start to motivate and eventually require more programmers to learn how to think and make in these "new" ways.

What is it like to learn the new ideas?

How complex does such a flexible facility have to be? It certainly doesn't have to be complex syntactically. The familiar English word order of Subject Verb Object (often with prepositions and sometimes with an implied subject) is quite sufficient. In object terms, the Subject is the receiver of the message, and the rest of the sentence is the message (we'll mark the "verb" by darkening it). This allows readable forms such as:

```
3
'this is some text'
pen up
3 + 4
3 * ( 4 + 5 )
car forward by 5
{1 2 3 4 5 6 7} collect [ n | n odd ]
Repeat (1 to 100 by 2) do [ ******* ]
```

And, just as important, these simple conventions allow us to make up new readable forms as we find needs for them.

This means that a user only has to think of one syntax-semantic relationship.

Receiver message means *the meaning is in the receiver*

thus objects are thought of and used just like peer-peer servers on a network. (This is not a coincidence and the origin of this idea dates back to the development of the ARPAnet -> Internet in the late sixties).

Thus what we are doing with dynamic object-oriented programming is to design and build a system made of intercommunicating objects. If our object system allows the interiors of all objects to be themselves made from objects, then we have learned all of the structural knowledge needed.

The interiors can be portrayed as views of the behavioral properties of the object, some of which are more dynamic than others. These behaviors can be grouped into “roles” (sometimes “perspectives” is a good term to use). One role could be “the object as a visible graphical entity”. Another role could be “the object as a carrier/container of other objects”. A third role could be “the object as a basic object”. There might be roles that are idiosyncratic to the object’s use in some system. For example, the object could have a special role as a slider widget in a UI.

This is why children and nonprogrammers find this style of programming so particularly easy to learn: it uses a familiar syntax, and has a familiar view of the world, as objects that interact with each other. The objects are made from roles that themselves are collections of relevant behaviors, themselves in terms of objects.

Look at the screen of your computer and realize (perhaps with a shock) that there is really only one kind of object that you are dealing with: one that is represented by a costume consisting of colored regions of space, that can handle user actions, interact with other similar objects, carry similar objects, and may have a few special roles in the “theatrical presentation” of which it is a part. This is all there is from the end user’s perspective. Then why is software so complex? Part of the answer is that normal human nervous systems are attracted to differences, not similarities, and it is similarities that create small descriptions for seemingly large complex systems. Differences often lead to lots of needless special cases that cause increasingly more trouble as time marches on.

“Similarities” are what an algebra does in mathematics: it unifies many seemingly different relationships under a single rubric which represents all. When dynamic object systems were invented in the 60s it was realized that they could have algebraic properties, and it is the algebraic nature of Squeak, combined with the uniform self-contained “objectness”, that accounts for quite a bit of its ability to do a lot with a little. The possibilities of algebra are not at all obvious in the normal old-style early-bound work of work-a-day programming. This is mostly yet to come for most programmers.

Children are also able to deal with parallelism more easily than long strings of sequential logic, so they readily take to writing parallel event-driven systems that deal with interactions dynamically. This leads to small simple programs.

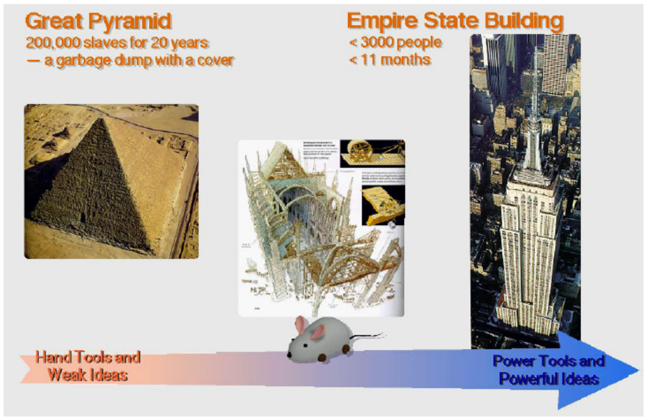
On the other hand, most programmers learn their craft by starting with “algorithms and data structures” courses that emphasize just the opposite: sequential logic and nonobjects. This is a kind of simple “godlike” control of weak passive materials whose methods don’t scale. I think it is very hard for many programmers to later take the “nongod” view of negotiation and strategy from the object’s point of view that real systems designs require.

My friend and colleague Dave Reed (whom I think of as the “/” in TCP/IP, because he was instrumental in creating the distinct roles for these two protocols that make the Internet work) likes to point out that there was a certain odd humbleness in the ARPA, ONR and PARC creators of most of today's computer technologies. They certainly had the hubris to work on really large problems — such as a worldwide network that could grow by 10 orders of magnitude without having to be stopped or breaking, personal computing for all users, dynamic object oriented system building, a local area net like the Ethernet that is essentially foolproof and dirt cheap, etc. But it was also true that most of these inventors didn't feel that they were smart enough to solve any of these problems directly with brute force (as "gods", in other words). Instead they looked for ways to finesse the problems: strategies before tactics, embracing the fact of errors, instead of trying to eliminate them, etc.

Architecture Dominates Materials

In short, you can make something as simple as a clock and fix it when it breaks. But large systems have to be negotiated with, both when they are grown, and when they are changed. This is more biological in nature, and large systems act more like ecologies than like simple gear meshing mechanisms.

So the trick with something as simple and powerful as Squeak is to learn how to "think systems" as deeply as possible, and to "think humble" as often as possible. The former is a good perspective for getting started. The latter is a good perspective for dealing with human inadequacies and the need for "metastrategic" processes rather than trying to scale fragile sequential algorithmic paradigms where they were never meant to go, and where they can't go.



One powerful architectural principle, such as the vaulted arch -- or dynamic late bound objects in software -- can provide a boost into a much higher leverage practices for designing and building large complex structures while modern scientific engineering is still being developed