SQUEAK: *OBJECT-ORIENTED DESIGN WITH MULTIMEDIA APPLICATIONS*

Mark Guzdial

GVU Center and EduTech Institute
College of Computing
Georgia Institute of Technology

Prentice Hall
Upper Saddle River, NJ 07458

Foreword

Software: Art, Engineering, Mathematics, or Science?

by Alan C. Kay

A 500-foot-high Egyptian pyramid took hundreds of thousands of workers several decades to construct. They piled up material brick on brick and then finished the outside with a smooth layer of limestone. By contrast, the 1,000-foot-high Empire State Building was constructed from scratch in fewer than 11 months by fewer than 3,000 workers. Quite a bit of today's software and their construction processes resemble the Egyptian pyramid in the difficulty it took to build them; I would dare say that no one currently knows how to organize 3000 programmers to make a major piece of software from scratch in less than 11 months.

I interpret this to mean that "software engineering" is still an oxymoron (like "airline food," "university parking," or even "computer science"). Still, what we do today is rather like the design and construction of buildings before architecture—literally, the "tech-ing" of "arches"—in that we can occasionally make something that functions, even if it does resemble a jumble of materials. Software engineering is a kind of ancient engineering, an ad hoc cookbook of recipes that have somewhat worked in the past.

Today, science (a concern with what is real) is mixed with mathematics (a concern with what is true), which is mixed with engineering (a concern with how something can be made). Each worker in any of these fields also partly works in the other two. Each field has a different temperament associated with it: mathematicians tend to be idealists, scientists realists, and engineers pragmatists. And each field finds itself temporarily adopting a borrowed temperament when it uses the other areas to aid advances in its own.

Now, what is computing? It seems to be a kind of mathematics (in that the machine is a kind of inference engine that works out the consequences of relationships) coupled with a kind of engineering (in that rather large language representations usually have to be constructed in order to express anything really interesting).

But where is science in all this? In our normal use of the term, we think of being presented with a universe, which is not necessarily connected with any of our hopes or beliefs, science is a special way of getting at how this universe seems to work. In modern times, we especially like to express the way we think the universe works in terms of mathematical models that seem to have enough correspondences with reality to allow both discussion and prediction. We tend to think of science as being analytic.

By contrast, computering seems to be much more synthetic, in that we start with rules and compute a kind of "reality". In this light, can there be a "computer science"?

I think the answer is yes, and it lies in an analogy to the construction of the physical world. Historically it has not been possible to compute, from first principles involving fundamental particles, whether a large building or bridge would collapse or stay up. The approach has been to make large structures as sound as possible and to study them as though they were part of the universe given to us to understand. This has led to a new kind of scientific engineering that is not oxymoronic, and to a vastly improved set of techniques for building large, reliable structures.

I think that the following is what needs to be done to finally create software engineering: We need to do more building of important software structures, and we need to do it in a form that allows analysis, learning, and reformulating of the design and fabrication from what has just been learned.

There seems to be a bit of a chicken-and-egg problem here. If we don't really have an engineering discipline, then won't it be very difficult to make big constructions that are also understandable enough to learn from? And won't the mess we've made be too difficult to reformulate to give us a chance to understand whether our new findings really have value?

I believe that the secret weapon that can be used to make progress here is extreme late binding. But of what? Of as many things in our development system as possible.

One can make a good argument on the thesis that most of the advances in both hardware and software design have been facilitated through the introduction of new late-binding mechanisms. Going way back in hardware, we can think of index and relocation registers, memory-management units, etc. In software, we went from absolute instruction locations and formats to symbolic assemblers, to subroutines, to relocatable code, to hardware-independent data-structure formats, to garbage collection, to the many late-binding advantages of objects, including classes and instances, message sending, encapsulation, polymorphism, and metaprogramming.

In Squeak, you have in your hands one of the most late-bound, yet practical, programming systems ever created. It is also an artifact that is wide, broad, and deep enough to permit real scientific study and the creation of new theories, new mathematics, and new engineering constructions. In fact, Squeak is primed to be the engine of its own replacement. Since every mechanism that Squeak uses in its own construction is in plain view and is changeable by any programmer, it can be understood and played with to no end. "Extreme play" could very easily result in the creation of a system better than Squeak, very different from Squeak, or both.

We not only give permission for you to do this, we urge you to try! Why? Because our field is still a long way from a reasonable state, and we cannot allow bad de facto standards (mostly controlled by vendors) to hold back progress. You are used to learning a programming system as a language with certain features, with the goal of using the features to make things. Squeak is very good at this and has many features (too many actually!) with which one can build things. But the best way to approach the learning you are about to do is to consider Squeak a metalanguage that can build other languages. Besides learning how to make things with the existing features, try to learn how the features themselves were invented and made. All the code is visible, and much of it has explanations of how it works. Here, the system is the curriculum. Even the online version of this book will have a hard time keeping track of an ever-changing and improving system, so it is best to learn how to find out from the system itself what it does. Then try to add new deep features of your own. Eventually, you will form a point of view of your own about better ways to program. Squeak will allow you to add these, or even to replace all of its current features with new ones that you have invented. Some of these ideas will be good enough to advance the art and the engineering and the science and the math of programming.

Then you will have used Squeak in all the ways we intended. At some point, a much better system than Squeak will be created, and nothing could make us happier—especially if you can do it while we're still around to enjoy the new ideas!