

Toward a Science of Design

Joel Moses

Institute Professor, Professor of Computer Science and Engineering Systems

MIT

September 2003

Herb Simon's wonderful volume, "*Sciences of the Artificial*," contains a chapter entitled "The Science of Design," the name of this workshop. I would actually like to comment on his chapter in the book entitled "The Architecture of Complexity." The chapter is based on a paper Herb wrote over forty years ago. I have read it several times over the years, and continue to wonder why we have not made more progress toward a science of design or the architecture of complex systems, especially software systems. Part of the answer, I believe, is implicit in Simon's writings in AI, cognitive science, economics as well as computer science. He assumes a problem solving methodology that is, in my opinion, only partially successful. Herb argues that hierarchies are important for design, something with which I completely agree. I think he assumes that the hierarchies are tree-structured, but a tree structure is not the only hierarchical architecture that is useful in problem solving and design. The Catholic Church is structured as a hierarchy. Whatever we may think about its recent or past practices, I would argue that the reason the Church has lasted as long as it has is partly due to the fact that the hierarchy is a layered structure and not a pure tree-structure. The College of Cardinals is a relatively flat layer, as are synods of bishops. This structure gives the organization much flexibility, which it has used to advantage over two millennia, albeit changing slowly in most time periods. Much of software engineering relies on tree-structured hierarchies and its divide-and-conquer methodology for problem decomposition. Lip service, I believe, is paid to alternative approaches, and we pay a price for it in that our systems tend to become overly complex, and cannot be modified with relative ease after some period of operation and the changes that ensue.

Pure mathematicians, such as algebraists and algebraic geometers, rely on layers of abstractions as a problem solving architecture. You create a tower of abstractions in which the problem can be represented, and you often solve the problem by specializing it to lower layers in the tower until you get to a layer in which it is easy to solve the specialized problem. Do so enough times to be able to lift the solution to the original layer and you are done. A key is that layers, such as the integers or the polynomials in one variable, are flat sets. In contrast, pure tree structures do not permit horizontal interconnections. If you violate the tree structured interconnection rule, you will in short order greatly increase the complexity of the system and make it harder to modify the system further. Network architectures are flat, but they are usually not hierarchical. If the methodology described above is so common in pure math, certainly in the past century, why is it not more widely used in design-oriented fields? One answer is that we do use it. Note the number of Turing Awards that were given to individuals who created a new layer of abstraction, such as a programming language or a computer architecture. One might also argue that middleware relies on a layered architecture. But why don't we commonly use layering as a design methodology? I think the reason is quite deep in our culture. Herb Simon fitted in very well with our culture in emphasizing pragmatism and reductionism. He would not have fit in quite so well in Germany and Japan that tend more to idealism and abstraction. That is part of my answer about what it would take to have a science of design. That is, we need to overcome our cultural hang-ups. Many argue that abstraction comes at too high a price in performance. That is sometimes true, but we tend to overemphasize it to avoid having to use standards implicit in a layered approach. People also argue that finding the right abstractions is too difficult. I believe that we do not try

hard enough to do so, but I also grant that there will be times when even a good try will fail to discover the one or two abstractions that will decompose the problem into layers. Some people argue that the software world is full of legacy systems that were not designed with layering in mind. I agree, but wish to point out that one reason it is difficult to modify and especially combine legacy systems is that they were not designed with layering in mind.

One price we pay for relying so much on tree-structured or network-based methodologies is related to complexity. Complexity has been defined in dozens of ways. Many define complexity in a dynamic sense. Complex systems, in this view, possess behavior that is difficult to understand and predict. Unfortunately, it does not take much to create a set of procedures whose behavior is difficult to understand or predict. Others emphasize the complexity of a system's interfaces. Those who emphasize ease-of-use fall into this category. Finally, some emphasize the complexity of a system's structure or architecture. Herb Simon was wise in discussing complexity in his original chapter in essentially a structural or Kolmogorov sense – the complexity of a system is the size of its shortest description. I think he would have agreed that (structural) complexity in itself is not the issue that concerns most people. Some systems need to be quite complex due to their great functionality. What bothers people is, I believe, that an overly complex system is hard to modify readily. I call the system property that permits one to modify a system readily for large classes of changes its **flexibility**. What people are bothered by, I believe, is a poor trade-off between complexity and flexibility. That is, a system that has been changed a number of times (where the changes each appear to be relatively minor) becomes so complex that it is difficult to change it further. I claim that the trade-off between complexity and flexibility is quite poor for tree-structured hierarchies, which many of us implicitly think is the only type of hierarchy. The relationship is better for layered or networked systems, but the latter systems, as already noted, are not hierarchical. Thus a key relationship of interest to software architects ought to be between architectures, complexity and flexibility.

More generally, I classify system properties and characteristics in the following three categories:

Standard properties: function, performance, cost

Non-standard properties: flexibility, scalability, robustness, safety, ...

System and context characteristics: complexity, architecture, uncertainty, emergence

The non-standard properties become of increasing importance in a long-term view of a system. System characteristics, such as complexity, are rarely goals in themselves. Rather they affect our ability to design software having the properties we want, whether they are standard or non-standard ones. Our ability to achieve a science of design, in particular software design, will depend on the extent to which we can determine the relationships and trade-offs between members in each of the three categories, and then design software that achieves a good balance among our multiple goals and the ease of achieving and maintaining them.

Joel Moses was the leader of the group that designed the Macsyma system, the first large-scale system for computer algebra. This experience as well as his earlier research in AI led to the views expressed here. He was also head of the EECS department, dean of engineering and provost at MIT.