

How is a piece of software like a building? Toward general design theory and methods.

**Mark D Gross; Professor and Director, Design Machine Group
Department of Architecture; University of Washington; Seattle WA 98102
<http://faculty.washington.edu/mdgross>
<http://dmg.caup.washington.edu>**

Buildings, like software, come in different sizes and levels of complexity. Some buildings, like houses, are small and simple; others, like hospitals, are large, very complicated, and serve a multitude of users and uses. Buildings, like software, often have extended lifetimes - - one expects a large, expensive, building to last at least 30 years. (Software has had an unexpected longevity: Consider Y2K, or air traffic control systems.) Throughout its extended lifetime a building will encounter a variety of changing uses, many of which may well not have been anticipated by the original designers. Some components of a building may change, or be renewed, every few years. For example, retail stores, or office tenants, may move in and out and these spaces are renovated on a short cycle. Some components are replaced on a longer cycle: Technical systems become obsolete, requiring changes, for example in the communications infrastructure, or in control systems for heating, ventilating, and air-conditioning. Other parts of a building remain substantially the same throughout its lifetime. Some components, such as technical systems, are only experienced by a small, though important, subset of users - - who are responsible for maintaining the building. Other parts of the building - - such as public areas - - are experienced by everyone. Some components of a design (for example, common construction details) may be re-used from one building to the next; yet each building must respond to specific variations in site and architectural program. The client, (who orders and pays for a building) is a different actor from the building's users; and therefore may have a quite different agenda: as with software, a building's usability is sometimes sacrificed for other priorities, resulting in user frustration, and higher operating costs. As with software, in extreme cases, serious health and safety concerns can arise from poor design.

In the 1960s an early 1970s as part of a larger movement in design theory and methods (e.g., Simon, Rittel, Asimow, Cross, Jones), in recognizing the increasing complexity and need to address concerns such as more diverse users, manufacturability, life cycle maintenance, and environmental and societal contexts, architects and urban planners began to look seriously at ways of organizing and structuring the design process. Some of these approaches have been embraced subsequently by software designers and HCI practitioners, such as Alexander's notion of a pattern language, or user needs assessment and usability studies. The latter echo the design methods developed and practiced by architects—such as the practice of “programming by design” (designing the activities that a building will support) and post-occupancy evaluation of buildings. Important lessons could be learned from the earlier exploration of design methods in architecture and planning, although the software engineering community in its eagerness to embrace such design methods has largely ignored these lessons. For example, in 1960s and 1970s “participatory design” was a widely heralded as a means to ensure that designs would adequately reflect the needs of users, but in practice it was often difficult to engage users productively in the design phases, and in any case only the first group of users who would inhabit the building at the outset could be represented. On a larger, more institutional scale, the democratic successes of participatory design, arguably led to entirely adequate but mediocre buildings.

A run of initial activity was sparked by key theoretical pieces such as Simon's “Science of Design” and Rittel's “Dilemmas of a General Theory of Planning” (the notion of “wicked

problems”). The 1960s and 1970s saw the development of practical design methods such as that of Pahl and Beitz in mechanical engineering design, or Habraken’s systematic design of housing, which recognized the inherent hierarchy of structure and control in buildings, taking advantage of this hierarchy to allow for greater variability and flexibility in designs - - and metrics for assessing these qualities - - without sacrificing cost or buildability. However, interest in systematic design methods, and a general effort to place these methods on a more solid theoretical footing seemed to diminish during the 1980s and 1990s. Paradoxically during this time, increasingly larger numbers of people began to use software in daily life, and software systems grew increasingly large and complex. Software engineers and interaction designers realized that the artifacts they produce may well last decades and they became aware of the need for a more systematic science (and practice) of design. Nevertheless, efforts to reintroduce a systematic and theoretically – grounded “science of design” in the production of software and human – computer interaction have so far been somewhat ad-hoc and directed at quite specific technologies (such as the design of Web pages or graphical – user interfaces). It is not that these are unimportant in and of themselves, but so far they have largely failed to lead to more fundamental principles that can outlast these technologies.

What is needed therefore, is a reexamination of the conditions and requirements of the science of design as articulated in the earlier literature and elaborated later - - theoretically, by researchers such as Schön (“The Reflective Practitioner”; “The Design Studio”); and pragmatically in a diverse array of efforts in various engineering disciplines (for example, reflected in Gero’s AI in Design conferences). This reexamination will on one hand be general —attempting to identify fundamentals of a science of design - - on the other hand, efforts rooted in specific design domains (such as software engineering, or architecture, or mechanical engineering) will yield highly specific systems and these may serve as exemplars for more general efforts. One might, for example, imagine a computationally expressed language for design, in which the operations, structure, and interpretation, directly support the activities of designing—much as a language such as Mathematica supports the operation, structures, and interpretation, of mathematical work. A broader societal impact would be new and more effective ways of teaching and learning design, based on a richer understanding of design’s s general principles.

My background bridges computer science, human – computer interaction, and architectural design. I was drawn to artificial intelligence and computer programming in the early 1970s, and studied architectural design, which led me to investigate systematic methods of design through the vehicle of computation. I worked in Negroponte’s Architecture Machine Group where I learned the importance of human – computer interactions, and Papert’s Logo Group, which emphasized end-user programming as a means of intellectual exploration. My doctoral dissertation, in 1986, titled “Design as Exploring Constraints” articulated the application of constraint-based programming as a basis for more intelligent and end-user programmable computer-aided design systems. My research since then has focused on computational means and methods to support design. While at the University of Colorado (1990 to 1999) I worked with members of Fischer’s Center for Lifelong Learning and Design. At the University of Washington I direct an interdisciplinary research group, building software prototypes to explore the intersection of design, computation, and the built environment. In 2002 I gave a keynote address to the Japanese Software Engineering Association’s annual meeting in Matsue, Japan .. titled “Design, Computation, and the Interface”