

Toward a Science of Software Design

Nancy G. Leveson

Design can be described as the *process* of partitioning a problem and its solution into significant pieces. The design *problem* then is how to partition or decompose a system into parts, each with a lower complexity than the system as a whole, while minimizing interaction between the parts, such that the parts together solve the problem. In most systems, the design problem is complicated by *emergent properties*, i.e., properties that arise from the interactions among the pieces and cannot be reduced to properties within the individual components. Decomposition is not the simple process it has at times been considered to be.

Note also that the goal of design is to solve a problem. Without understanding that problem, (i.e., trying to divorce form from function), the design process becomes irrelevant. The design of a system cannot be considered separate from the goals, values (criteria for acceptability), and the potential use of the system. The priority for the acceptability criteria (desired properties) is externally determined and the job of the designer(s) is to satisfy as many of these criteria as possible while making tradeoffs according to the required priorities.

In the early 80s, it seemed like our field was on the way, or at least had taken the first steps, to establishing a science of design for software. Such a science must include general design *principles* applicable to all design, design *concepts* that reflect these principles and include criteria for deciding when and how to apply the concepts, *methods* for implementing the concepts, and *criteria* for evaluating the success of the effort.

By 1980, some initial software design principles had been identified, for example, separation of concerns, abstraction (particularly hierarchical abstraction), simplicity, and restricted visibility (locality of information). The overall goal behind these principles was stated to be mastering complexity through intellectual manageability and it was thought that the key to intellectual manageability was the structure of the software itself. Note the emphasis in this early search for design principles on the human mind and the cognitive capabilities of the designer. This emphasis seems to have gotten lost along the way and replaced with a focus on the design and its features without regard to the strengths and limitations of human cognition.

General principles are implemented through design *concepts*. Again in the seventies, various design concepts were identified involving approaches to decomposition such as program families, virtual machines, information hiding, modularity, distinctions between programming in the small (designing modules) and programming in the large (designing the overall software structure), restricted control structures, ways to minimize connectivity, and abstract data types. Criteria were proposed for evaluating implementations of the principles in the design concepts, such as coupling and cohesion and various types of complexity metrics.

In the 80s, for some reason, we got off track in developing a general science of software design and started all going down one narrow path. Software engineering became focused on design methods and eventually on only one design method. Design *methods* are sets of guidelines, heuristics, and procedures or steps to take in designing a system. These usually offer a notation to express the result of the design process. The goal is to provide a systematic means for organizing the design process and its products.

Design methods were proposed based on functional decomposition using data flow, data structures, control flow, and objects, but as time passed almost all attention became focused on decomposition using objects. OO design was argued or assumed to be the best and only approach to all software design, regardless of the type of system being designed or its required properties and without much regard for complexity, intellectual manageability, or many of the principles derived in the 70's such as minimizing connectivity by increasing functional cohesion and reducing functional coupling.

Pursuing object orientation became a goal in itself and claims were made about its potential, almost magical, qualities but little was done to establish the validity of these claims. Teachers and textbooks taught only this one approach to design. Ridiculous statements were made, e.g., that one should first create the OO design and then the system requirements can be determined from the design.

The 90s saw some limited research on software architectures and some software structuring approaches such as patterns, frameworks, kits, etc. What was surprising about some of the latter was its anti-scientific nature—the leaders even argued against the scientific method and against quantification (e.g., Coplien, “Idioms and Patterns as Architectural Literature” and Kerth and Cunningham, Using Patterns to Improve Our Architectural Vision, IEEE Software, January 1997). Rather than a science, software design took on a “new age” aspect with talk of a “quality without a name” and design strategies that are “ethereal” and “difficult to talk about.” Kerth and Cunningham describe software design using objects as a creative experience and an artistic endeavor rather than engineering and something that can be pursued scientifically: “Before you have experienced it, no words will help you understand; after you experience it, words don’t exist to explain it.” Patterns were collected but not evaluated and little effort was devoted to determine the conditions under which particular patterns should be used in terms of the desirable system properties they might provide.

For the most part, these approaches ignored the very different characteristics of the application areas for software design and the conflicts and tradeoffs required for achieving various types of desired system qualities. Instead, they narrowly focused on a set of predefined design steps and on the act of developing the design itself. In a keynote at one of the OOPSLA conferences, the architect Christopher Alexander (upon whose concepts the patterns movement was supposedly based) chided the software community on missing the point of what he had proposed—he argued it was not possible to divorce design from context and that design patterns and objects had to be evaluated with respect to the goals, values, and culture of the system within which the design was embedded. It does not appear that anyone heard him. Alexander’s ideas evolved from a strong mathematical foundation, systems theory (including an understanding of emergent properties), and an appreciation of the cognitive and intellectual aspects of design. These components will be needed if we truly want to create a science of design, along with a scientific approach to evaluation of our hypotheses.

It is not clear why we got off track in our progress toward establishing a science of software design. There were probably a lot of factors. One unique aspect of software design, noted by David Budgen, is the extent to which commercial interests have dominated the codifying of design methods. The most widely known design methods and tools have been developed and marketed largely by consultants and commercial organizations. While this clearly indicates the desire and need of industry for design skills, it does not create an objective forum for considering the evaluation of the design methods being promulgated. The result was that almost no scientific evaluation has occurred, even in the universities. Instead, sales trumped science and training trumped education.

Another factor may be the tremendous growth in the use of software and the number of people (often minimally trained) producing it. The fastest and easiest way to train a large number of people is to teach them one method, i.e., a set of rules and steps to blindly follow. This influx of people into programming created pressures to treat software design as a trade—teaching people to follow rules, like an electrician or a plumber. While it is difficult and time consuming to teach more than design methods (and in particular one method), our education needs to get beyond the belief that designing software is largely a matter of following a set of pre-defined activities and instead focus on design principles and concepts and the way to select the most appropriate methods for a particular problem. Curtis has noted that great designers are not necessarily great programmers—perhaps we need to separate these activities.

In summary, the goal of an NSF initiative to return to creating a science of software design is overdue. Such a science must: include understanding the many aspects of complexity; go beyond simple hierarchical decomposition and create design principles and concepts based on systems theory, including the principle of emergence; and explore such topics as the relationship between human cognitive limits (intellectual manageability) and software design, the relationship between software design and the types of systems being created, the relationship between design and achieving the various desired system attributes (sometimes called non-functional requirements), and criteria for evaluating designs. In this pursuit, we will have to put the scientific method and careful experimentation up front—without repeatable experimentation, there can be no science. Handwaving, proof by vociferous argument, and claims of having the “right” or “best” design method must be ignored and even discriminated against. We need to take the pursuit of knowledge about software design back from the hucksters and establish a research agenda based on the scientific method.