Expanding the Role of Software Design

An Honors Thesis (for CS 498)

by

Nathan L. Shadle

Thesis Advisor

Dr. Wayne M. Zage

Wayne M. Zage

Ball State University
Muncie, Indiana

March, 1998

Expected Date of Graduation
December 1998

Table of Contents

Purpose of Thesis

This essay discusses the current views of design in the field of software engineering, and why I, as a former major in the architecture/environmental design field, feel that this view is quite limited. This essay points out some of the drawbacks which stem from these limited views, and how these drawbacks are, quite possibly, detrimental to the quality of software products. Finally, this essay suggests a revision to the traditional waterfall model of the software lifecycle based on the drawbacks of the industry's views of design.

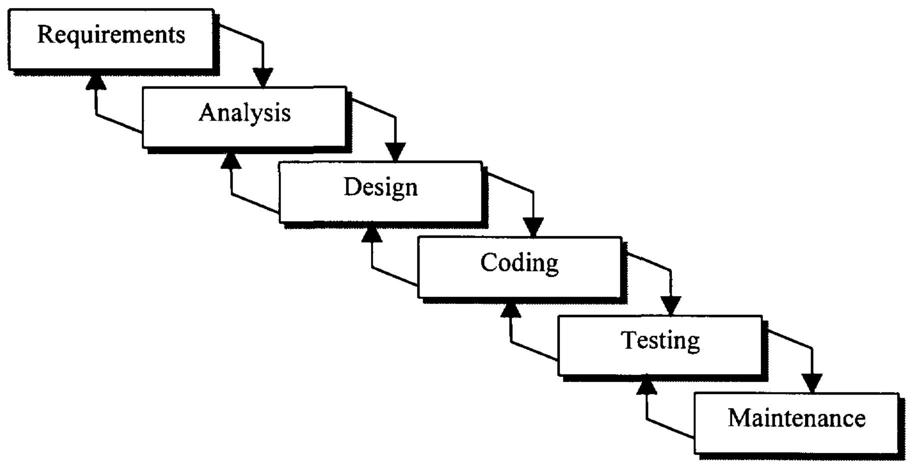
Introduction

E.S. Taylor defines design as "the process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization." The primary definition of design in Webster's Contemporary American Dictionary (1979, p. 195) states simply, "to conceive; invent." The latter interpretation suggests, much more forcefully, that design implies creativity, innovation, and imagination, quite contrary to Taylor's implication that design is only a blueprint of how to build something.

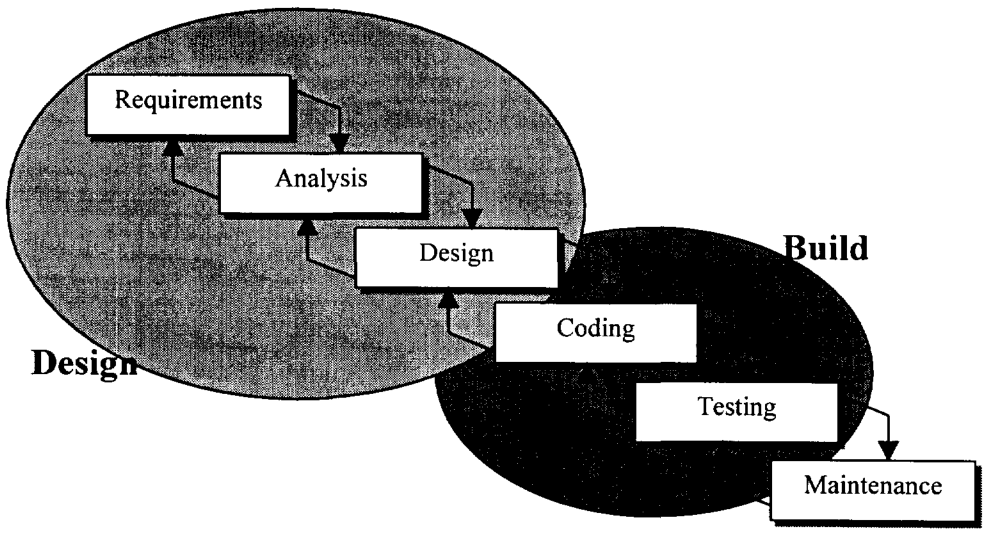
Design is a paramount focus in a number of disciplines, with software engineering being no exception. In that discipline, design is seen as converting the requirements of a problem into a plan for implementing those requirements electronically (Page-Jones 2). While that is certainly true, and perhaps the largest factor in developing software, design is more. The purpose of this essay is twofold: first, and most importantly, to point out that the role of design in software engineering reaches far beyond the confines of the "design" phase of the software lifecycle, and should be expanded accordingly; and consequently, that the phase of the software lifecycle traditionally entitled "design" should be renamed to reflect its actual task.

Simplification of Software Lifecycle Model

Often times, procedures for creating software are very closely related to the procedures followed in other areas. Take architecture for example: like software, "Architecture is normally conceived (designed) and realized (built) in response to an existing set of conditions" (Ching 10). This procedure closely reflects the method used by software engineers to develop software systems. In software engineering, the traditional waterfall method of the software lifecycle describes the typical approach to generating software to solve the problems of a client:



Based on Ching's statement, one can infer that the process of creating architecture can be simplified into two basic steps: a design phase; and an implementation phase. Using Ching's two-step scheme as a guide, the process of software development can likewise be simplified:



How well does Ching's two-step method apply to the field of software engineering? The answer to that lies in the discipline's interpretation of *design*.

Software Industry's Current View of Design

Currently, the software industry holds the collective view that "the design model is the equivalent of an architect's plans for a house" (Pressman 345). Coupled with Taylor's aforementioned definition of design, it can be generalized that the structured design phase in the software lifecycle, and design in general, is little more than a detailed mapping of how to proceed in building the software. Dolores Zage takes a slightly broader view, claiming that design is the "abstraction of how we are going to build it," but the principal remains the same: the *blueprint* is the primary goal of the design phase of software.

To further understand the software industry's notion of design, consider again the traditional model of the software lifecycle. The majority of designing that is done on any software system is done in a narrow phase of the lifecycle. "[D]esign is simply the bridge between the analysis of a problem and the implementation of the solution to that problem" (Page-Jones 2). This bridge analogy leads us to believe that design is simply a transition, a set of rigid heuristics into which we can stick an array of requirements and out pops a blueprint. In fact, the name "structured design" alone points out that there is a definite formula, or framework that produces the desired solutions. "Structured design is a disciplined approach to computer systems design" (Page-Jones 2).

Roger Pressman believes that "software design is an iterative process through which requirements are translated into a 'blueprint' for constructing the software"

(343). This statement attacks the idea that the design phase of the lifecycle is a rigid "bridge" by inferring that structured design has flexibility through its iterations, but just barely. Pressman does nothing to discount the fact that design is focused almost entirely in this single step of the lifecycle, straying rarely outside those boundaries. In fact, he reinforces the overwhelming belief that design is little more than a means of blueprinting the solution, regardless of the flexibility of the process. "The designer's goal is to produce a model or representation of an entity that will later be built" (Pressman 341).

Finally, the software industry sees design primarily as a tool to create efficient systems. "Structured design produces systems that are easy to understand, ... and efficient to operate" (Page-Jones 3). Since blueprinting is the focus of design, it leads logically to understand that the major goal of this blueprinting is to generate functional, efficient systems, and to do so quickly and cheaply. "In short, structured design produces inexpensive systems that work" (Page-Jones 3).

Limitations of the Current View of Design

Now that the software industry's current perception of design has been examined, we can answer the question posed previously: Does Ching's two-step method apply to the field of software engineering? Judging by the concentration of design activities into its narrow phase of the lifecycle, the answer is a resounding *no*. As a result, there are a number of outstanding drawbacks to the software industry's perception of design.

One of the biggest, and perhaps most obvious, problems with the software industry's view of design is the misinterpretation of the function of design. Far to often, the role of design in a software system is considered nothing more than creating a blueprint of the way in which the system shall be built. This is immensely contradictory to the act of *designing*. The term *design* has been far too often abused, particularly when comparing it to a blueprint. A blueprint is not a design, but tells how to implement a design. A design is "a gestalt of all the factors that inspire an effective change and/or response to a set of issues" (Segedy). Design is all-encompassing.

When considering simply the structure of a software system, namely its blueprint or structure chart, design guidelines generally apply. This is perhaps the reason why the task of creating this structure chart was termed "structured design." In the big picture, however, the structure chart, like certain schematics of a home, are not as important to the user; they are used behind the scenes. The structure chart simply maps out the code, and neither concerns nor is visible to the user of the system. It's analogy in the construction world would not be simply the blueprint, because the user *would* be concerned with this, and the result would in fact be visible to the homeowner. Perhaps a better analogy would be the schematic diagrams for the mapping of wires, pipes, load bearing walls, etc. All of these are tremendously important, just as code is, but like code, they are something the user *uses*, but does not see or worry about. What the user does concern herself or himself with, however, are such things as interface, function, and interrelationships, but *not* the interface and interrelationships between modules on a structure chart. That is the logical equivalent of how two wires, pipes, or load bearing walls communicate with each other, share responsibility, etc. The homeowner concerns himself or herself with the interfaces and relationships between these about as much as the software user does with that of the modules. Instead, the predominate concerns of users and homeowners alike are the relations and interfaces between humans and that which is to be built. Therefore, while certain general guidelines of design are appropriate in the structured design phase of the software lifecycle, the limited role of design can cause structured design to fall short of meeting the needs of the entire system. In the "design phase," the software industry often fails to see design as it applies to the whole picture and instead focuses on a far too narrow spectrum. This severely inhibits the ability to *design* for the gestalt.

Likewise, it can even be seen that the important goals of the "design" phase of the software lifecycle are quite opposite to those of pure design. While "good" structure charts strive for low coupling, in order to pass the least information between modules, and high cohesion, so that a module performs only a single task, environmental designs follow the inverse. Using the analogy that modules on a structure chart correspond to rooms of a building (which is gross misrepresentation of the broad picture, or gestalt), the exact opposite of the traits which make up a "good" structure chart define a "good" design. Cohesion is absolutely abhorred to those sitting in a waiting room, for example. That kind of "single-mindedness" of a room is mind numbing, for the human psyche seeks stimulation and excitement. Similarly, uniformity between neighboring rooms should be maximized to promote strong relationships, and although, like software engineering, there are tradeoffs between coupling and cohesion, high coupling (and hence strong relationship) does not necessarily imply low cohesion. Two rooms may follow a motif without boring the viewer silly. This notion is similarly tied to the software "design" concept of "black boxes". A "black box" is a module of which nothing is known of its internals, only its inputs, outputs, and what it does. In our simplified building example, this idea is ludicrous since it implies that a kitchen can be placed next to a dining room without worrying about how they work. So long as they do what is expected, never mind that the kitchen may have a neo-modern theme while the dining room has a classical motif. These are but a few examples of why the narrow focus of the "design" phase of the software lifecycle falls quite short of solid design practices which center around the big picture.

To further the analogy between the construction industry and software development, one must consider the actual role of those people involved. The architect, who is responsible for designing an urban space or building, takes into account Segedy's gestalt. Webster's Contemporary American Dictionary (1979, p.301) declares gestalt to be "a unified configuration having properties that cannot be derived from its parts." A true designer considers all relationships and responses that a design elicits to develop the best design. The role of software's "design" is quite different. The "design" phase is much too focused to consider the gestalt, concerning itself instead with functional needs, as well as blueprinting. Segedy agrees. In fact, more *designing* is done in the analysis phase of the software lifecycle than in the design phase. For example, evaluating users (for things such as computer literacy) in order to tailor the software to their needs, considering hardware constraints on which the system must fit, determining interfaces and relationships between neighboring systems, are a few of the tasks of the analysis phase. Software engineering as a whole tends more towards the realm of development than design. Development, especially as it applies to software, "is a much more rational and linear process that can often incorporate aesthetic (in its broadest definition) features and functions, but tends toward less of a whole package" (Segedy). Ignoring the whole package can often be costly.

Once the architect has created this broad, inclusive design, often times it is the responsibility of a draftsman to plot the design on paper, generating a blueprint. It is not the traits of the architect that the software "design" phase is most closely likened to. Most of the tasks which are done in structured design argue heavily that it is most closely related to the role of the draftsman.

The second problem with the software industry's view of design is its absolute dependency upon the rigid heuristics which help transform requirements into blueprints. Even the contradictory name "structured design" shouts about step-by-step methods. Pressman calls software design a "multi step process" (31) before defining each of those steps. He even adds later that "...software design is the first of three *technical* [my italics] activities – design, code generation, and testing..." (342). Design is a highly abstract and flexible endeavor, and should be treated as such. It is a proven fact that such inflexible methods *obviously* stifle creativity and imagination. Children are a prime example. Before toddlers are immersed in a rigid school structure, their imagination's are absolutely without bound. Once schools clutch them, and children are trained how things *have* to be, however, creativity has been shown to degrade substantially. It is not explicitly *taught*, but children with limitless imaginations learn from friends and teachers that, for example, trees can't be drawn upside down, and glue isn't for playing with. Why not? That child may have been on the verge of discovering the handiest use for glue ever. Some of the grandest and most ingenious ideas have come to people in the weirdest of places, at the strangest of times, and it is stifling to then try to cram that innovation into a rigid structure, rather than re-define the entire process to fit the idea. No landscape architect or urban planner attacks different problems the same way, in fact the differences are shocking. One architect may immerse himself in a Native American reservation for weeks in order to gather the essence for his design of a Native American museum, while another may spend time finding his own food and sleeping in a teepee on the plains for inspiration. That's not something that can be done following a set of heuristics.

Yet another, and perhaps the greatest, problem with the software industry's notion of design is its tendency to believe that the primary goals of design are efficiency and function. This raises a great debate among environmental designers, particularly Ching, who declares that design is "more than satisfying the purely functional requirements" (10). Segedy's gestalt ideas reinforce that a good design inspires a response to *all* factors, not simply functional ones. "Design implies a purposeful act that combines technical and functional needs with more subjective needs (beauty, inspiration, behavior modification, etc.)" (Segedy). These subjective needs are every bit as important in software development as in any other discipline. Far too many systems have been utterly abandoned due purely to poor interface and human-computer relationship factors. Even Karl Marx would jump on board when he says in the Communist Manifesto: "Modern industry has converted the little workshop of the patriarchal master into the great factory of the industrial capitalist." In a quote that holds as much water today as at the time of its conception, Marx proclaims that producing as efficiently and inexpensively as possible at the expense of the quality of the subjective needs is detrimental to the quality of the overall product.

While structured design performs masterfully in ensuring that all requirements are met, design should be so much more. Pressman states that design "begins by representing the totality of the thing to be built (e.g. a three-dimensional rendering of the house)…" (345), yet even this is so grossly oversimplified. Design encompasses so much more, and it is apparent that Pressman has overlooked many of the factors which represent the "totality" of the object. Using his house analogy, a three-dimensional rendering only scratches the surface of the factors which face good designers. "A good designer always considers the context as an integral part of the design challenge" (Segedy), and although this is only one of many factors, Pressman has left it out entirely. Factors such as the position of the house on the lot, external trees, and the motif of the geographical location are but a few of the factors which competent designers must consider, but once again, the software industry's preoccupation with the function of the thing to be built has crippled its ability to design for the gestalt. It is probably that Pressman will argue that the analysis phase of the software lifecycle covered the context of the software system, yet it is insufficient. The system must be *designed* to not only communicate with neighboring systems, but to relate as completely as possible.

> No program exists on its own. Certainly it must interact with the operating system, but the elegance of the *design* [my italics] comes in the program's ability to seamlessly interact with not only the application environment (actually many different

> environments), but also with other programs that the user might bring to the context. i.e. can a spreadsheet program create communication tools like graphic generators which can then easily translate into a report and/or presentation package. (Segedy)

These are factors that would not normally appear, but ones which a true designer must consider. Analysis only identifies the existence of these factors, but until they have become a part of the design, the designer has failed to design for the gestalt.

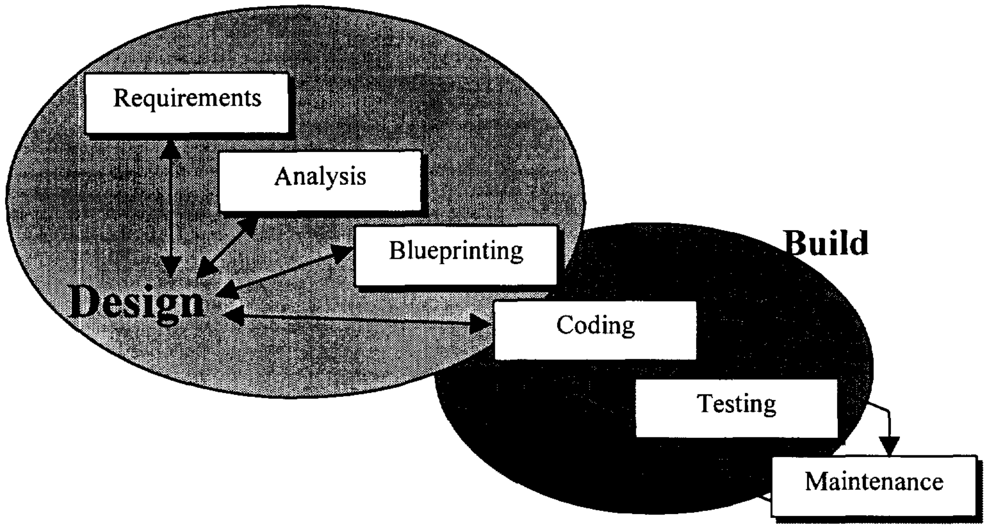
Even the "utilitarian concerns of function and use can be relatively short-lived…these primary elements of form and space comprise the timeless and fundamental vocabulary of the architectural designer" (Ching 6). As this statement explains, interfaces are often the enduring factor in architecture, much more so than functional concerns. The same can be said of software systems. Perhaps the biggest example of this is the Windows95 operating system. One large reason for the popularity of this product was its innovative use of form and space to create the interface between user and machine. As compared with previous versions of Windows, space was completely reinvented. Now the user had free reign over the desktop, to create and change her background in such a way that every user found different meaning. A different response came with different users, and in fact the same users could find new meanings each time they interacted. These are absolutely fundamental principle of design. Of course function was important, and like many software systems, is paramount, but what separates this product from many command line operating systems that have identical functionality? The ability to continually stimulate users through the unlimited possibilities of desktop wallpaper and icon configuration, as well as the numerous different ways to carry out a single task (such as copying, moving files), helped to rocket Windows95 to the dominating stature it holds today.

Often times, software developers invent an interesting, enthralling design, only to have it ousted to increase the efficiency or function of the system. Who is to say that function should always be the primary goal? Imagine an urban landscape which separates two buildings. The only limit to the design of an interesting landscape is the imagination of the designers. The most obvious, and perhaps most ordinary and boring, solution is to specify a path between the buildings, and perhaps to decorate it lavishly. Who requires, however, that a path connects the two buildings directly? Function does. Function and efficiency dictate that the shortest path between two points is a straight line. A true designer would take into account the gestalt experience, and might find that a large obstacle or a detoured roundabout would make the urban space much more enjoyable, dynamic, and exciting, and thus more widely used. Perhaps the minor climb over an obstacle, or the mystery and surprise of a roundabout would elicit a fresh and innovating feeling to viewers. Occupants might be much happier with the stimulus and intrigue than with simple function. Function is often paramount in software development, but what if all urban spaces or buildings abandoned interesting spaces for function? Designs concerned entirely with function and practicality are often not the best designs, as believed by Le Corbusier: "My house is practical. I thank you as I might thank railway engineers or the telephone service. You have not touched my heart."

The final problem with the software industry's notion of design is its narrow scope. Design should not be limited to a particular step of the software lifecycle. Instead, it should flourish during its entire duration, and should begin as soon as the first requirement is uttered. Currently, containing design to one specific phase constrains the developers, ties them down. By the time the design step rolls around, functional characteristics have already been etched into the minds of the developers, and it is often exceedingly difficult to force the design to fit the other factors, such as interface concerns. Instead, other less imaginative interfaces would be used since they already fit the design.

Proposed Revision to Traditional Waterfall Model

The best time for design is in the earliest possible stages of the lifecycle, immediately after system requirements are set forth; when minds are freshest and most imaginative. Though the entire scope of the problem is not fully comprehended, this can work to the designer's advantage. Similar to the children who are fathoms more creative before the structure of school sets in, designers are most innovative and creative prior to the firm grasp of structure. Based on this drawback of the software industry's perception of design, broadening of the scope of design would contribute immensely to software quality. Thus, a revised model of the software lifecycle originates:



This revised model, with more emphasis placed on the designing of software, holds many advantages over the traditional waterfall model. Problems which were shown to stem from the software industry's view of design, namely limiting design to the "design" phase, the dependency on heuristics, the tendency of design to be little more than blueprinting, and the overwhelming focus on efficiency and function, have been righted in this model. Since creativity and innovation decrease with time, it is important to implement design practices as early as possible in the lifecycle. As soon as the first mention of requirements are passed along to the developers, it is important to begin *designing*. Only in that way will creativity and imagination be maximized.

Also, unlike the traditional one, the revised model requires that, once the requirements have been determined, every step be evaluated as to how it changes the overall design. Once an analysis is made, for example, it is important to incorporate it immediately into the design. Far too often, software developers make excellent analyses, but are unable to work them into their design because they have waited so long that it had to be sacrificed for other factors. A prime example of this would be an analysis which insists that the system be user friendly, since the users have very limited computer experience. Yet the final system may be the exact opposite of user friendly if the blueprinting of the system were done before any interfacing was even considered, and the interface was forced to fit the structure of the system. The revised model insists that these analyses be incorporated into the design immediately.

Another advantage of the new model is its ability to near identicalness to the "structured design" phase of the traditional model, though that step has now been more appropriately named. The "Blueprinting" phase still incorporates all of the heuristics and methods of the "structured design" step, including the mapping of data flow to a structure chart and improving coupling and cohesion of modules. In fact, there is almost no difference between the two steps, except that the big picture has been identified prior to arrival at the "Blueprinting" step, and only details will be worked out, where previously such things as interfaces and relationships were handled in this step. Similarly, the "Blueprinting" step can now be used to focus on those things which developers have always yearned to focus on in this step: function. Since the gestalt has already been considered, efficiency and function can be nearly the entire focus of the "Blueprinting" phase.

Conclusion

So, in conclusion, there are a number of limitations to the software industry's current view of design, which leads to drawbacks in the quality of software in general. Unfortunately, at the present time, the model used to create software cannot be generalized to fit Ching's two-step model of creating. This is due primarily to the view of design in the software community. A narrow scope of design, too much emphasis on efficiency and function, a dependency on heuristics, and a misinterpretation of blueprinting as design, all lead to software which focuses less on the gestalt, the primary goal of a good design. "Elements and systems should be interrelated, interdependent, and mutually reinforcing to form an integrated whole" (Ching 11).

It is not the purpose of this essay to imply that software engineers lack creativity, cleverness, or the ability to design in the true sense of the word, but instead show that software engineering, like countless other disciplines, could profit enormously by expanding the role of design in the current process, by reaching beyond the limitations of the current methods. Only through a revised model can software engineers break free of the limitations of *development* to truly *design* software. "[S]oftware *development* is concerned primarily with function and responding to known and anticipated issues and needs, while software *design* should allow and inspire the user to become an integral part of the programming process – to take the lines of code and create new opportunities that will change with every use and user" (Segedy).

Works Cited

Ching, Francis D.K. Architecture: Form, Space & Order. New York: Van Nostrand Reinhold, 1979.

Page-Jones, Meilir. The Practical Guide to Structured Systems Design. Englewood Cliffs, NJ: Yourdon Press, 1988.

Pressman, Roger S. Software Engineering: A Practitioner's Approach. 4th ed. New York: The McGraw-Hill Companies, Inc., 1997.

Segedy, James A., Associate Professor of Urban Planning, Ball State University. E-mail interview, 27 Jan 1998.

Taylor, E.S. "Information System Design Methodology." Software Design Techniques, 4th ed., P. Freeman and A. Wasserman, eds., IEEE Computer Society press, 1983, p. 43.

Zage, Delores, Assistant Professor of Computer Science, Ball State University. CS498 Class Lecture, 21 Jan 1998.