Toward a Methodology of Rational System Design

Andre Vellino
Nortel Networks
Advanced Software and Networking Technology
vellino@nortelnetworks.com

1.0 Introduction

In this paper I defend the view that there are rational methodological criteria for preferring one system design[1] over another and that they are substantially the same as those of the scientific method. The parallels between the notion of a rational system design and the notion of a rational scientific theory are not coincidental: they reflect a fundamental similarity between the invention of designs that meet artificial specifications and the creation of theories that conforms to "specifications" given by nature.

The analogy between the creation of system designs and the invention of scientific theories exists both in the "psychology of discovery" and in the "logic of justification." Just as there are no recipes for producing good scientific discoveries, there are also no recipes for generating good designs. Yet there are rational criteria for preferring some designs over others and these criteria are by and large the same as those that are used for preferring one scientific theory over another: simplicity, coherence, adequacy in meeting requirements and adaptability to changes.

Like scientific theories, system designs also have a history and evolve within a social and intellectual context. This means that designs should be treated as evolving over time (rather than piecemeal), and hence that the criteria used for comparing designs should be as sensitive to the historical environment within which they arise as are the criteria in the scientific method. What matters, then, is not so much the relative merits of designs at any point in time, but rather, how successful they are as instances of historically evolving design paradigms.

One of the conclusions I want to draw from this parallel is a pedagogical one. Contrary to the received views, system design is an innovative and creative discipline that cannot be taught in the way that, for example, engineering is taught. The talent to innovate cannot be commanded at will.

The first section in the body of this paper surveys some of the main positions and traditional definitions of design in the literature. The next section explores in greater detail some of the rational criteria for evaluating designs that fall out of the parallel with the methodology of science. The last section concludes with some suggestions about the art and practice of system design.

1. While "system design" in this paper refers primarily to the design computer software systems, many of the conclusions drawn below are applicable to other domains in which design is a principle activity.

2.0 Definitions of Design

When we speak about design we use phrases like “the design of X for Y” (e.g. “the design of a software system for controlling a robot”, “the design of a service for telcos” and “the design of a software system to control a switch”). So whatever a “design” is, in the abstract, it is always a design *of* something *with respect to* something. In other words a design is always created in a domain for an end (or purpose). Three general definitions of design that acknowledge this fact are:

> “Design is the adaptation of means to ends” Oxford English Dictionary.
>
> “Design is the art of defining a system to meet a set of constraints” (J.W.J. Williams) [Will92].
>
> “Design is concerned with devising artifacts to attain goals” (H. Simon) [Sim69]

There are two salient features of these three definitions that deserve attention. The first is that all these three recognize design to be a creative, artistic activity. Designers *invent* systems that serve a function. The second is that what they invent is constrained both by the function to be served and by what constructions are possible. These three general definitions have the virtue that they encompass the usual sense of the word “design”, while not precluding other activities that pass as design.

So the art of system design is a creative mixture of technical know-how and imagination whose purpose is to satisfy a set of requirements or constraints. This mixture inevitably embodies implicit assumptions and presuppositions that are not necessarily apparent in the implementation of the design itself, but which are nevertheless crucially important in the evolution of the design over time since they determine how changes are made to the system and at what point a system re-design is necessary. The role of these assumptions in the evolution of designs is discussed in more detail in section 3.

2.1 Layers of Design

The conventional view is that there are several levels (or layers) of design activity. The constraints that designs address range from user requirements, upwards compatibility, reliability, efficiency, cost of development, ease of use, conformity to standards, etc. At that level, the constraints on the design may be very informal and loose. As [Gui90] puts it concerning software design specifically (p. 279)

> Many organizations divide the development of large software systems in the following phases: requirements analysis; system design (high-level and detailed); implementation (coding, debugging, unit testing); and system testing, and maintenance. During the high-level system design, a designer has to transform an incomplete and ambiguous specification of the requirements into a high-level system design, expressed in a formal or semi-formal notation.

But as Guindon demonstrates, the actual design process is more complex than the above picture suggests. The “layers of design” are typically intermingled: not only do the components of a design take on the higher levels of the design as constraints, but what is possible at a more detailed level of design constrains what classes of system designs can be considered. For example, the requirements analysis phase in the design cycle is not done in the void: it usually presupposes something about what kind of designs are possible for the problem area. In some cases the complete set of requirements for a problem might be impossible for *any* design to meet (i.e. the requirements might be inconsistent in a subtle way) but they are not usually because the requirements themselves are specified with, if not a specific design in mind, at least a general class of possible designs.

Furthermore, when a design is decomposed from a “top level” into components, the design of the components themselves depends on how the top level design was sliced up. If you take the design of a computer to be an assembly of the design of hardware components (the CPU, the Bus, the memory management system etc.) you will probably end up with a very different beast than if you subdivide the problem into its functional components (input/output system, operating system, user-interface). In each case the requirements will be different, as will the constraints on the sub designs.

The “top-down” decomposition view of design also fails to recognize that system components typically have structural similarities. These similarities are well captured by the notion of “hierarchy” in object-oriented design environments and makes system decomposition possible along a different axis than “degree of generality”.

The different layers of design are also often dispersed over time. High-level architectures are not immutable constructions and often change with time as lower layers are specified and implemented more fully. As the higher levels change, the lower levels may not be re-implemented to be consistent with the new higher level design.

2.2 Design Constraints

No matter how one “layers” designs, the process of specifying a design consists, in part, of discovering what the constraints *are* on the design. Typically, the problem to which the design is intended to be an answer, is unknown (at least partially) and a large part of the design activity is uncovering what constraints the design is supposed to meet. What this means in practice is that the design ‘cycle’ goes something like this: a first design sketch D_0 is proposed (which assumes that the system has the constraints C_0) and the design is then criticized by users (for example in design reviews) which reveals either some missing or some false assumptions about C_0. C_0 is then revised to C_1 for which a better design D_1 is devised etc.

This cycle of constraint discovery is complicated by the fact that even though some design constraints are known, they may not be under the designer's control (e.g. delivery date, functionality, development costs) whereas others are (e.g. degree of compatibility with other products, implementation details). Thus, the design problem may be solved not only by *satisfying* design constraints but also by *relaxing* some constraints in order to satisfy other constraints. Indeed there are typically many possible designs that meet some but not all constraints and that

the choice between one design and another requires an assessment of the relative importance of the constraints and the degree to which they can be balanced one against another.

It is fair to say that all designs share some global constraints. Indeed, depending on the industry, some constraints are regulated by law (e.g. safety, durability, environmental friendliness, availability, cost to customer etc.). But more importantly, any (realistic) design must be *implementable*. Of course it needn't be implementable *here and now* (it might be implementable by someone else in the future), but it is obviously constrained by the laws of physics, at the very least.

Generally, the degree to which a specific part of a design is “implementable” (within a given time-frame) influences the system design as a whole, sometimes in subtle ways. For example, in [Gui90b] Guindon explores the degrees of abstraction that designers would think about in the design of a software solution to the Lift Control Problem and found that, for ill-defined design problems there is a substantial amount of interleaving between abstract and concrete thinking (i.e. the interleaving of top-down and bottom-up design). The point here is that the different stages in the design process are not independent of one another and cannot be isolated from one another.

The interrelatedness of these stages in the design process suggests another picture of design which is more “holistic” than the traditional view would have us believe. Not only does the system architecture determine what gets implemented, but what can be implemented (i.e. what implementation tools and technologies exist for system building) determine what systems are possible. Indeed even the constraints that define the parameters of a design may change over time (for example, market demand for a product line might change or the availability or price of certain commodities used in manufacturing phase might change) so that it is important to be able to re-examine the inter-connections among the design constraints. In extreme circumstances, such changes to design constraints may require that existing designs be thrown away and replaced by entirely different ones.

Furthermore, the evolution of a design depends on the environment in which it lives (sometimes even the social and political context) and the task of designing may involve changing the environment itself. This last point is well put by Winograd and Flores’ in [WinFl 86], where they say (p.177-178):

> We are concerned with what happens when new devices are [designed] and with how possibilities for innovation arise. There is a circularity here: the world determines what we can do and what we do determines our world. The creation of a new device [...] can create new ways of being that previously did not exist and a framework for actions that would not have previously made sense. As an example, systematic bookkeeping techniques did not just make it easier to keep track of the financial details of business as it existed. New ways of doing business... became possible, and the entire financial activity of society evolved in accord with the structure of the new domain.

In short, designing systems has far reaching consequences that cannot be isolated from the context in which they are created.

2.3 Capturing Design

However one defines design--all levels of design--there is a need to “capture” it so that it can be managed, modified, re-engineered and re-implemented. The mechanism of “design capture” must take into account the fact that designs are dynamic entities: they always have a *history* and are continually *evolving*. Design decisions are made that trade-off various design attributes, e.g. costs vs. benefits, performance vs. reliability, integrity vs. compatibility and these choices should be recorded for latter consideration when the design requires revision.

Since a design always has a history, it also has an explanation, in principle: design decisions are made based on trade-offs that are intended to optimize a balance of constraints and these decisions can be recorded. But one design decision often entails a whole collection of further constraints on the designs for subsystems. And as we saw in the previous section, the design decisions may be made at one level of design and have repercussions at many other levels so it is critical to be able to re-examine the inter-connections among the design constraints.

Thus a record of design decisions does not just mean a trace of who decided what when. It also needs to contain the background of assumptions against which the design decisions were made, as well as the dependency tree between the components of the design from which one can infer the impact of any given design decision.

2.3.1 Implementation

One form of design capture is, of course, the implementation itself. There is no doubt that the implementation of a design is crucially important. It is after all, the design's ultimate concrete realization.

Regrettably, though, implementations are often the *only* tangible consequences of the design. This is unfortunate because system designs cannot be reverse-engineered from their implementation even in principle. One reason is that the process of deriving an implementation from a design is akin to a “compilation” process and thus suffers necessarily from a loss of general structural information while gaining in unnecessary (from the design point of view) detailed information. Not only are there many possible implementations of a given design, there are also many possible designs that lead to the same (or similar) implementation.

A contrasting point of view of the design activity (in the software domain specifically) offered by DeMarco in [DeMarc82] suggests that the process of compiling a design to an implementation shouldn’t entail any loss of information. His argument is that there ought to be a one to one map between the design and its implementation, the acid test of whether an implementation is based on a proper design being whether the design can be reconstructed from its implementation. He says:

> A software design must [...] be thought of as a rigorous blueprint for construction... The design must dictate one and only one possibility for construction down to the level of the module...Design is the determination of what modules and what intermodular interfaces shall be implemented to fulfill the specified requirement.

So is DeMarco right to think that if reverse engineering can't be done it is because these systems were not properly designed in the first place, or is it an inherent characteristic of the relation between designs and their implementation? My hunch is that what DeMarco refers to as "design" is what might more properly be called "detailed design" rather than "system design". A detailed design is then more an "implementation plan" than an architectural framework. Detailed design is a step between system design and implementation, just as a high-level language computer program is a step between an algorithm and executable object code.

2.3.2 Documentation

Even implementations, though, have to be measured against something and documents largely written in a (technical) natural language, perhaps a reference manual, are still the most common form of design documentation. But this conventional way of documenting designs assumes that the changes in the implementation are followed by documentation. Versions of documents are kept up to date with respect to versions of implementations rather than used as specifications for the implementation. Ideally, the changes in the implementation of a design should be motivated by changes in the specification provided by the documentation.

Another model, favoured by some CASE tools is to generate both documentation and implementation (or at least partial implementations of components) from the specification. The idea here is to make the design specification the object from which everything is "compiled"; i.e. to express a design in a formalism that is both understandable by users and precise enough to produce executable code.

3.0 A Theory of Rational System Design

Design decisions that evaluate the trade-offs between different conflicting constraints, including the decision to scrap the original design and go back to the drawing board, can be made *rationally.* In the following section, I would like propose an analogy between the process of scientific discovery and the process of designing that sheds light on the rationality of design decision making. The analogy is, I believe, not a coincidence: the two activities are not just similar, they are in essence the same.

At first blush, Freeman [Free83],appears to hold a view similar to the one I want to suggest. He argues that to improve design

> we [must] have a solid scientific understanding of design on which to build a viable (software) engineering design methodology. Essentially, we must have a body of teachable knowledge about design. It should be intellectually rigorous, analytic, formalized where appropriate and based on empirical studies where possible.

The implication here is that (i) the art of design can be understood scientifically and (ii) that this art itself can be codified as a methodical discipline, i.e. a science.

But this view reflects the common misunderstanding that scientific knowledge, the crowning achievement of 18th century rationality, can be produced by a method or recipe. This view considers that knowledge is obtained by theory-neutral observations and experiments whose essential attributes are generalized and synthesized in the theory with the use of mathematical tools.

It is tempting to believe a similar story about design: a system design "emerges" from an analysis of the requirements and constraint specification. All that's needed are tools and models that formalize the requirements and constraint specifications.[1]

But the act of designing a system is similar to the invention of a scientific theory to the extent that both activities are essentially *creative* and *artistic*. The view that the art of design could be made into a scientific and rigorous discipline by introducing *formality* into the process is misguided. A rigorous, formal discipline, one that is well grounded in mathematics has the advantage that it has provable properties and quantifiable attributes: formal methods do have a role to play in the art of design. But using them doesn't automatically make your discipline scientific. A word processor may help an author write a novel but it doesn't make him or her a better writer.

To "adopt the scientific method" and adapt it to the design process assumes that the "scientific method" is a mechanical procedure for churning out scientific answers to problems and that generating good designs involves a similar procedure. This view is misleading of both science and design as the following section attempts to demonstrate.

3.1 Scientific Method

The parallels that exist between the activity of design and the process of scientific discovery are useful for understanding design (some of these parallels are also drawn in [Glyn85]). Although science is, to quote Simon, "concerned with how things are" and design is "concerned with how things ought to be", it appears that creating designs to satisfy constraints is essentially the same as inventing scientific theories that explain phenomena in the physical world.

In the world of science, what constrains a scientist's theories for describing the world is the experimental data that her theories have to explain. In the world of design what constrains the designer's architecture are the requirements, specification and implementation of the design. In both fields the discovery of good (designs/theories) that adequately satisfy the constraints is a creative activity that cannot be mechanized or reduced to a formula. In both fields there are techniques, tools, established practices, that the adept (scientist/designer) can use to achieve his end, but in neither case does the process of creation have a *method*. A mechanical method for producing designs (theories) from requirements (empirical evidence) would then generate something substantially new and hitherto unknown from parts that are already known and established.

1. This misrepresentation of what counts as scientific is rampant in popular culture. This is illustrated by an article in Business Week (Jan. 1992) entitled "Turning the black art of programming into a science". The gist of this article is that it is sufficient to use the SEI maturity model, which describes process improvement and quantitative monitoring methods, to improve software quality. Methodical techniques in process and the use of quantitative tools thus mutates a "black art" into a "science".

Both science and designs more subtle and more complex than a mechanistic account would have us believe. The aim of science (design) is to improve our knowledge (designs) and get "closer to the truth" ("optimal designs"). By recent analyses of scientific discovery [Kuh62] [Lak70] and progress, the process of getting "closer to the truth" often takes place by a radical shift in paradigm or "world view".

For example, in pre-Copernican astronomy, the predominant paradigm was that the earth was at the center of the universe and the rest had to be explained. The geo-centric theory explained some phenomena well and others were more difficult to explain and required "kludges" to the theory. Real progress was made when a radical change in paradigm took place and the Sun was placed at the center of the universe. The challenge then was to explain observable phenomena in those terms.

By analogy, there are design architecture "paradigms" that are central and unquestioned and in whose terms the detailed design must be formulated. For example, until recently the dominant paradigm in telephony was the concept of a *call*. A call connects two endpoints and for most purposes (e.g. POTS) it is sufficient and adequate. However, there are design goals, such as the design of multi-way calling features, for which this paradigm is clumsy and a different paradigm, the view that the primary object is the *terminal* (the end-points out of which calls are constructed) is preferable. What makes one paradigm preferable over another is the simplicity with which designs are built from them. In this case the view that calls are constructed out of terminals doesn't limit the concept of a call and one terminal can be on several calls at once or have some calls on hold or queued or whatever. Whereas the view that a call has two endpoint makes it difficult to construct the notion of a multiway call: in that view each call at a given endpoint must refer explicitly to every other call and know about the states of the other calls.

Similarly, the design paradigm for a digital key system might conceive of telephones as (essentially) communicating computers. This paradigm contrasts significantly with the paradigm of an intelligent central-office controller to which dumb terminals are attached. The question of how to design a specific feature for a digital key system has to be couched in those terms: the paradigm is a given and the design must be formulated in those terms. Indeed designs that are produced within this paradigm have direct observational consequences that are almost accidental. For example, a digital key system which operates by broadcast messaging might easily have the consequence that a terminal maintains its identity regardless of which line to which it is connected. On the other hand the implementation of a similar feature in a traditional call processing paradigm might require convoluted and difficult feats of engineering.

It is instructive to see why the Copernican theory of astronomy was ultimately a progressive shift in paradigm and constitutes a rational improvement in knowledge. According to [Kuh62], one of the problems with the geo-centric theory was its complexity: to explain the behavior of planets, as observed from the earth, one needed to concoct circles within circles within circles to account for their behavior. Furthermore, the geo-centric approach provided no novel predictions. Given the observation of a new phenomenon, the theory could be modified to accommodate it, but it never predicted new phenomena. The helio-centric theory, on the other hand, was *simple*: simple things (planets) moved in simple circles about the sun. The theory could account for the phenomena (e.g. the retrograde motion of Mars as seen from the earth) and moreover, it predicted new phenomena (eclipses, stellar parallax).

The parallel with system design should be apparent here too. Different design paradigms structure the world in different ways and have differing degrees of simplicity and coherence. In *The Mythical Man-Month*, Frederic Brooks [Bro75] calls this property of designs "conceptual integrity" and contends that

> ...conceptual integrity is the most important consideration in system design. (p.42)

Furthermore, Brooks continues,

> Conceptual integrity [...] dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds. (p.44)

So conceptual integrity, whether in science or design is highly desirable. Why? Because in both cases the conceptual coherence comes from a framework within which all the pieces fit. It provides *predictability.* Given the essential principles that underlay a coherent system design then it's a straightforward matter to predict how a component *ought* to work.

Conceptual integrity (simplicity) is related to another important principle in design formulated by Malaya [Maya79] quoted in [Free83] is the *principle of totality:*

> All design requirements are always interrelated and must always be treated as such throughout a design task.

In short, the coherence of a design contributes to its simplicity and simplicity, in turn contributes to intelligibility.

3.2 Design Evolution

If we look at successful designs we find that how they *evolve* is critical: their adaptability to changes in the constraints that they must meet is an essential ingredient of their success. A design whose complexity is unmanageable, whose processes are too bureaucratic, whose evolution is slow and clumsy will be unsuccessful in the long run. On the other hand, designs that are conceptually simple coherent, intelligible and predictable have obvious advantages which, everything else being equal, should give them a competitive advantage.

Yet, companies continue to be plagued by the design complexity of systems that have evolved over decades and whose relationship to the initial design is tenuous at best. If simplicity and intelligibility are competitively advantageous why then do systems grow to unmanageable size and loose their design integrity?

One reason for the loss of design integrity is that what gets changed over time is the implementation rather than the design. As [Dev91] point out, a system design often begins as a simple, cohesive and well-thought-out design, arrived at by a small group of experienced and knowledgeable designers and architects. After a while, though, the subsystems are implemented by people who don't have knowledge of the system design. According to [Dev91] they

> implement their subsystems in a manner that violates the architectural principles of the large system that they are modifying. The code may work cor-

> rectly and consequently pass system test; however this kind of violation, carried out at various points over a period of time, results in a loss of architecture that gradually erodes the original simplicity of the system. Thus this lack of knowledge among the developers leads to a vicious cycle where the system becomes progressively more complex and thus harder to know.

One solution to this problem, then, is to communicate the designs to the implementor. But another solution is to remove the need for separating the implementation team from the design team by automating the process of implementation. In this solution, what gets reused, reengineered and reimplemented is the design itself rather than the implementation. This implies that designs should be *tangible entities* whose components can be altered and from which new implementations can be built with minimal effort (see Design Practices below).

Whether or not designs are tangible, they all have limitations and the evolution of any particular technology must, inevitably, come to an end. There are limits to growth and complexity beyond which a incremental modification to existing technology is no longer cost-effective. Unless there are regulatory restrictions or monopolies induced by prohibitively high capital investments, the forces of free market economics should determine the time of death for a particular technology: i.e. when competitive technology emerges that provides the same or improved services or products. But designs never specify the conditions under which they are no longer valid, as though this ought to be blatantly obvious. There is no concept of *design obsolescence* and yet there is a need to know when the limits of a design have been reached and when new designs should be sought.

3.3 Design Choices

When choosing among competing designs we would like an answer to the question “what makes one design better than another?”. Similarly, when we trade off different design constraints against one another we would like to know if there are rational criteria for preferring to meet one constraint over another. If the analogy between designing and scientific theorizing holds water, there ought to be some rationality to design choices and to choices among designs.

The view that design choices can be made rationally shouldn’t be confused with the view that there is always an optimal design decision. What makes one choice better than another is not that choice in particular, but rather, that choice in combination with other choices. Since design decisions almost always affect lots of other aspects of a system’s design, it’s rarely the case that we can say “everything else being equal” this choice is better than that one. Whether a decision is good or not depends on whether all the design decisions, taken together, meet the design constraints. The decision to build a car with a powerful engine can be bad if it doesn’t also have powerful brakes and appropriate tires.

Since the quality of a design decision depends on the way it affects the evolution of the design it is usually impossible to tell, *a priori*, whether a design decision will be the right one. It might have been a mistake to bet on digital designs for telephony applications if computer industry had not been able to produce increasingly more powerful and inexpensive hardware.

4.0 Design Practices

What does the foregoing discussion entail about design practices? We mentioned that the process of designing is never done “top-down” or “bottom-up” or linearly in time according to the waterfall model. So does this mean that no design method should be applied at all?

One way to ensure “violations of architectural principles” of a system design are not committed is to force its implementors to comply with “cultural institutions” that oblige them to conform to certain “established practices”. In large organizations, this has the consequence that implementors don’t understand why it is that they have to do what they are doing and contributes to the frustration of their creative abilities.

Another means of enforcing design consistency is to restrict the implementation techniques and design tools so strongly that such violations are not possible. If you want a civil engineer to design a bridge for you, you don’t put her to task with a pencil, ruler and blank piece of paper. What she needs are archetypal bridges that she can adapt to your needs.

Restricting the range of possibilities for a design by bringing some preconceptions to it can be useful, but constraining the range of possible objects that a designer can manipulate is a little like obliging a poet to write sonnets: the choice of words at any given point is limited by the constraints of rhyme and rhythm. The result is not the same as blank verse.

As Thomas Kuhn did with scientific theories, we could perhaps distinguish between “normal” designs and “extraordinary” designs. To produce extraordinary designs system designers should have all the freedom in the world to invent and create and this means the right, if not the obligation to challenge the status quo and to suggest radical alternatives. This approach is imperative when systems have reached the outer limits of their comprehensibility and complexity. For those limits are invariably accompanied by exponentially increasing costs and diminishing returns.

Of course that “normal” design activity may benefit from a particular design method such as “top down” or functional decomposition. For example, it seems clear that special purpose CASE tools (i.e. CASE tools that have a specific semantic content and don’t allow implementors to do anything other than what the CASE tool permits) enforces conformance to the design paradigm and is an appropriate tool to use in some circumstances: circumstances where “normal” designs are required.

The enforcing of design paradigms in special purpose CASE tools has the interesting consequence of narrowing the gap between the implementor and the designer, a view that was advocated in 1983 by Balzer Cheetham and Green in [BCG83], but it is still applicable today. The authors projected that future software development environment would contain an “automated software-assistant” that enables users to become developers by specifying the behaviour of software in high-level specification languages. The assistant would help the user to manage, maintain, document and test the software by maintaining records the activities of the “designer/developer”, how the component parts interact and the rationale behind the design decisions. It would also assist in compiling designs and “record” the choices that the designer takes (e.g. data-structures, algorithms etc.) for future use in “reimplementation”. This, they predicted would make

software systems more maintainable, easier to modify and implied that “off-the-shelf” software for development would be highly customizable. More time would be spent prototyping and software manufacturing will be available to less knowledgeable users.

The automated software-assistant predicted in [BCG83] is already with us to some extent already with general purpose CASE tools such as ObjecTime and Software Through Pictures. These environments provide the designer with prefabricated models of what systems do and what needs to be said about them in order to model their behaviour.

5.0 References

[BCG83] Balzer, R., Cheetham, T. E. Jr., Green, C. “Software Technology in the 1990’s: Using a New Paradigm” *IEEE Computer?* November 1983.

[Bro75] Brooks, F. *The Mythical Man-Month,* Addison Wesley, Reading, Mass, 1975.

[DeMarc82] DeMarco, T., *Controlling Software Projects*, Yourdon Press Computing Series, Prentice-Hall, New Jersey, 1982.]

[Dev91] P. Devanbu, R. Brachman, P. G. Selfridge & B. W. Ballard “LaSSIE: a Knowledge-Based Software Information System” *Communications of the ACM* Vol. 34, No. 5, May 1991.

[Eek91] J. Eekels and N.F.M Roozenburg “A methodological comparison of the structures of scientific research and engineering design: their similarities and differences” *Design Studies* 1991, **12**, No.4.

[Free83] P. Freeman “Fundamentals of Design” IEEE, 1983

[Gui90a] Guindon R. “Designing the Design Process: Exploiting Opportunistic Thoughts” *Human-Computer Interaction* 1990, Volume 5, pp. 305-344

[Gui90b] Guindon R. “Knowledge exploited by experts during software systems design” *Int. J. Man-Machine Studies* 1990, Volume 33, pp. 279-304

[Glyn85] Glynn, S. “Science and Perception as Design”, *Design Studies* 6 (3), 122-126.

[Koes64] Koestler A., *The Act of Creation* Hutchison & Co. 1964, reprinted in Penguin Paperbacks, 1989.

[Kuh62] Kuhn T. *The Structure of Scientific Revolution* Chicago Press, 1962.

[Lak70] Lakatos, I. “Falsification and The Methodology of Scientific Research Programmes” in *Criticism and the growth of knowledge* (Eds. Lakatos & Musgrave), Cambridge University Press, 1970.

[Par86] Parnas, D. and Clements, P., “A Rational Design Process: How and Why to Fake It” *IEEE Transactions of Software Engineering*, Vol. SE-12, No. 2 (February 1986) pp.251-257.

[Sim69] Simon, H. *The sciences of the the artificial*, MIT Press, Cambridge, Mass. 1969.

[Will92] Williams J.W.J. Personal communication.

[WinFl86] Winograd T. and Flores F., *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corp., New Jersey 1986.