

DIVISION OF COMPUTER SCIENCE

COGNITIVE AND ORGANISATIONAL ASPECTS OF DESIGN

To be presented at The Safety Critical Systems Symposium '94, Birmingham, February 1994

**Martin Loomes
Donald Ridley
Diana Kornbrot**

Technical Report No.171

December 1993

Cognitive and Organisational Aspects of Design

Martin Loomes, Donald Ridley and Diana Kornbrot
The University of Hertfordshire, Hatfield, UK

Abstract

Research into the software design process currently centres on a particular model of the software design which is based on a number of assumptions that are rarely tested, and have little theoretical grounding. This paper attempts to highlight some of these assumptions and to suggest ways in which they might be limiting current research activity. It identifies the life-cycle as the core of the existing paradigm, and introduces an alternative model that may be more fruitful for the discussion of cognitive and organisational aspects of the design process.

Background

The risk of failure has always been recognised as an inherent part of design, particularly when new technologies are being applied to novel problem domains [1]. This failure may manifest itself as visually unattractive buildings that ruin a city centre, stylish chairs that are impossible to sit in, or railway systems that grind to a halt whenever leaves fall onto the track. Systems that involve software are also designed, of course, so it should come as no surprise that they too are liable to fail. What is surprising, however, is the myth that seems to be developing around software design that we can find ways of avoiding these failures. One possible cause of this is the belief that because software is not a "real physical entity" there is no excuse for systems failing due to software errors. Whilst we can accept that metal fatigue may cause an aircraft to crash, there seems to be the suggestion that failure due to faulty software is more easily avoided. The assumption seems to be that we can construct software that is "correct", if we find and apply the "correct" methods for the task. We seem reluctant to recognise that the cause of system failure is not metal fatigue or software errors *per se*, but the designer's decision to use metal or code that is prone to fail: it is the designer that causes the failure, not the materials. Rather than seeing failure and errors as things that exist, but can be avoided with the right methodology, we can view them as things that the designer brings about, and

ask what behaviour causes this. If we understood better why designers make mistakes we might be able to suggest ways they can adjust their behaviour to minimise errors, or contain their impact on the process as a whole.

It has been suggested that primitive societies are reluctant to allow designers to experiment with new approaches, as they live the sort of hand to mouth existence where any unforeseen cost might be catastrophic. As a result

"... primitive societies are very conservative. Tribal customs prescribe exactly how everything shall be done, on pain of God's displeasure. An inventor is likely to be liquidated as a dangerous deviatorist." [2]

Although we cannot describe our western civilisation as primitive, it might be suggested that we are now so dependent on our designed systems, and they are so complex and all-embracing, that we are demonstrating similar attitudes. The consequences of failure are catastrophic not because of the poverty of the environment in which they occur, but because of the richness of the role the systems play in our lives. This poses very real problems for software designers: the systems they are being called upon to design are often very complex, and need innovative approaches, but the environment in which they are working is reluctant to allow them the freedom to take the necessary risks. System failures that do occur usually serve to reinforce the intuition that more control, rather than less, is required. Moreover, the usual prejudice that control implies rules, and centralisation, and cannot be an emergent property of the design process itself, is usually applied, so that responsibility is removed from the designers to higher agencies such as departments, companies, authorities and governments. When failures occur, however, as they surely will, they are usually seen as local to a project, rather than a product of the culture within which the design is taking place. The fixes that are applied are local, *ad hoc*, solutions to an immediate problem, and rarely percolate through to inform the wider society in which design is occurring. Software engineers are constantly being told that they should not simply fix bugs at the code level, but should address the higher levels of design too: this does not usually include addressing the problems with the methodology and tools which led to the problem! They may debug the artefact, but not the process.

Responsible, Self-conscious Design

Alexander has suggested that a crucial component of modern design is the realisation that the designer must be self-conscious, and be prepared to discuss and change the process of design as well as the artefacts produced [3]. This is because modern design is, in general, not carried out to solve problems of immediate concern to the designer, but on behalf of others. Moreover, it is usually carried out by teams of designers rather than by individuals. Thus communication of ideas is crucial to the process. Primitive man, however, can proceed "unconscious of the fact that among his faculties there is one which allows him to refashion nature

according to his desires" [4]. He can apply the tried and trusted methods of his ancestors to the everyday problems he encounters, and, should they fail him, he can deviate slightly and gain immediate feedback as to whether or not the deviation is helping or hindering the process of solving the problem. The computer hobbyist, of course, can use very similar approaches to "hack" solutions to problems on a home computer. Many computer scientists would have it that this is a "wrong" approach to software design: in fact, of course, it is simply a technique that does not scale up beyond the single-designer, immediate-feedback, paradigm. One of the consequences of accepting that software design must be self-conscious is that the designer must accept the loss of innocence that this entails [5]. Alexander cites two reactions from designers who are unwilling to accept this loss of innocence, particularly when faced with complex tasks that they feel unable to handle: the refuge in genius and the refuge in style.

Refuge in genius might be viewed as acceptance of the idea that there is a Muse of design who provides the inspiration behind the process: this is an escape from the loss of innocence because clearly the designer cannot be held to blame if the Muse gets it wrong! Although few people would admit to holding this view, there are certainly many who see intuition or inspiration as central to the design process, and would argue that attempts to be scientific in design are dangerous because they cause designers to inhibit their intuitions (to close their minds to the Muse).

The refuge in style identified by Alexander is the attempt to carry out design within a particular style or school. The designer who adopts an "art nouveau" or "Georgian" style, for example, and creates a monstrosity, can seek refuge amongst like-minded designers, and argue that it was the style that failed, not the designer. This is a far more worrying trend amongst software designers! Top-down design, stepwise refinement, formal methods, object oriented design, and many more, can all be seen as styles that designers might work within and try to escape the loss of innocence. If they choose to work within a style, of course, they are not escaping at all, but should be accepting responsibility for the choice of style as part of the design process. Unfortunately, education often legitimises this escape route by introducing "methodologies" in such a way as to suggest that they are useful without careful consideration of the task for which they are being selected.

If we put together the apparent reluctance of the software design community to take risks when developing systems, and the use of refuge in style to escape the loss of innocence, we have a potentially disastrous situation. Industry wants designers to use "tried and tested" methods and designers are often only too happy to abrogate responsibility for the choice of method and work within a house style. Moreover, to make the designer more "efficient" the style is often buried deep in the workings of a CASE tool, so that the designer does not even have to be aware of the processes involved. Unfortunately, the tried and tested methods we have are only useful for

technologies and problem domains which are well understood. Although there are still many design tasks to be carried out in these areas, the real challenges are in areas where there are no such aids.

We are currently deeply wedded to a culture in which the true nature of design is buried beneath the quest for the methods we need to avoid failure. Companies that have recognised the severe limitations of such methods for most real problems dare not say so too explicitly, for the software house that advertises with the slogan

"We don't use any particular method or tool"

will certainly find itself upstaged by the company who uses OOD and the most trendy CASE tools. Individual designers who constantly question a house style may well find themselves promoted to "design consultant" positions, where they can be isolated as deviationists if necessary, or put in charge of "special projects", but will rarely be made project managers for mainstream developments. We would argue that while the quest for such methods dominates research, and their application dominates practice, we will never significantly improve the ways in which we design software systems. We may well "polish the shiny bits", making aspects of design we choose to see rather more acceptable, but we will not tackle the messy problems that lie beneath the surface.

Technocentrism

This sort of global criticism is nihilistic, of course, unless we also take the bold step of putting forward some alternative ways of proceeding. We have denied ourselves the easy route of simply proposing another methodology, or encapsulating existing ones into more powerful tools, and so we need something rather more radical. The avenue we wish to explore is to see what insights can be gained by rejecting the notion of a development life-cycle as the dominant feature of the paradigm in which we work. Rejecting the life-cycle as the foundation upon which we want to build effectively eliminates most existing methodologies at a stroke, for most of these are prescribed routes and notations for navigating around a life-cycle of some form.

Moreover, we suggest that concentrating on code as the end-product of software design may be placing emphasis in the wrong place, allowing the technical aspects of the problem to determine the paradigm within which design will take place. Papert uses the term "technocentrism" to capture the idea that we refer all questions to technology [6]. Adopting a life-cycle model epitomises the way in which technocentrism is the norm in software design. It suggests that the life-cycle is an emergent property of some naturally occurring phenomena. The quest for more accurate models appears to be interpreted by many as the search for a better understanding of this essential property. In fact, as Papert notes, technology does not have to be considered in this way: we can, if we choose, recognise that we control the technology. Rather than asking questions such as "how does inheritance work in

object oriented design" we can ask "how do we want to use the notion of inheritance when we design systems". If we want to consider not only the technical problems, but also wider issues such as how software designers might be made more effective, how users should be integrated into the design task, what organisational structures should we put in place that will have a beneficial effect on the task, and what cognitive processes are involved in design, we must adopt less technocentric views of the process. If we do not adopt such views we sacrifice control over how the questions are posed, and we often have to accept that some questions cannot be expressed at all. We should note that the term "safety-critical systems" itself invites a technocentric view, for it is often taken as the classification of systems that are safety-critical, rather than systems which we are making safety-critical by the way we intend using them.

It is important to note that we are not advocating replacing one model of the process with another: rather that we should be encouraging researchers to choose new ways of viewing the design process to supplement and stimulate the current models.

The Theory-Building View

One model that deserves more detailed analysis is the Theory Building Model. Here we do not refer questions to the technical product of the design task, but rather we anchor discussion in the production of theories, which have the desired system as a model. Exactly what comprises a "theory" in this context, of course, is an essential part of the exploration, and not something that can be given a glib answer here. We should note, however, that blind acceptance of the dated view that theories are implicit in nature, and the scientist's task is to discover them, will lead us straight back to technocentrism: rather we will assume that theories are the attempts of mankind to impose some order onto phenomena. They are thus designed artefacts in their own right.

This idea is not new. It has been proposed by at least two other sources. Burstall and Goguen, for example, suggested that we can put theories together to make specifications, where the theories are captured in a suitable logical system [7]. Naur, on the other hand, suggested that programming can be viewed as theory building, where the theories are in the minds of the designers [8]. It is interesting to note, however, that these two ideas are rarely seen cited in the same literature. The former is now seen as part of "formal methods" (although it originally appeared in the AI literature), whilst the latter is more closely associated with the "softer" aspects of system design. The current paradigm within which Software Engineering research is being carried out makes it very difficult to reconcile these two views. Formal methods and intuition are seen as opposite poles in some implicit construct. We would argue that they are both essential tools to be used by a software designer, but that we need to explore how they can be reconciled.

Cognitive and Organisational Processes

Consideration of the theory-building view raises some very interesting, and important, questions. In this section we will outline some of these, and suggest possible avenues of research that might be used in the quest for answers.

If primary outcomes of the software design process are theories we should consider the relationship between software design and science, which also has theories as a major goal. A "scientific approach" to software design has been advocated in many places before, but usually the benefits claimed for such an approach seem undeliverable. The approach has been presented as a way of ensuring "correctness". In fact modern science has given up all claims to be discovering correct theories, recognising that all theories, if scientific, are potentially refutable. The scientific approaches suggested for software design certainly can deliver "correctness", when the term is given a severely limited meaning that divorces software systems from the real world, but most designers recognise that this avoids the really interesting issues and areas where failures usually occur. Unfortunately, this has caused many designers to reject any discussions of science as relevant to the design process, as they assume that similar simplifying assumptions will be made. We would argue that a genuine attempt to relate scientific practice to software engineering practice will provide some powerful insights that might lead to real improvements in the design process. This can be achieved by examining, and rationalising, the ways in which scientists work in practice: an endeavour which has been carried out by philosophers of science for many centuries. Viewing design projects in the context of Kuhnian paradigms [9], for example, provides a way of discussing the interaction between projects, something that cannot be achieved in most models of the software design process. Extending Kuhn's ideas to those of Lakatos [10] allows us to recognise hard cores of the design process that serve to form paradigms: areas that will rarely be challenged by those working in specific problem domains.

This leads us to question the source of such paradigms and hard cores. Some of these are explicit, being axiomatic in the problem domain or in the methodology that the designers have been told to adopt. Many more are implicit, however, being hidden in the tools used for the job, the education of the designer, or the culture the designer is working within. Examining such issues is a difficult task, and it is easy to see why technocentrism is often the preferred route, as we only have to ask "hard" technical questions, rather than these "soft" questions which we don't expect to have satisfactory answers. It is also easy to see why many institutions and companies would prefer technocentric answers: it is much easier to accept that you need to buy a new tool or impose a new methodology, than to risk asking why your company organisation is such that the designers are making a mess of so many projects. Never-the-less, if we want to do more than "polish the shiny bits" we believe that

such difficult questions need asking. There are approaches to the systematic investigation of these sort of questions that may be fruitful, typically those drawn from occupational and organisational psychology. The cultural audit [11], for example, may provide a way of carrying out the type of holistic study of a design team that is necessary to start exploring the human causes of error. No doubt it will need tailoring in order to extract the right sort of information, but early studies suggest this might be feasible [12].

Once we have taken the bold step of realising that software design is a human process, rather than a purely technical one, we can also start to ask questions about how individual designers interact and make decisions. The life-cycle model assumes that there is a single specification and a single design, whereas in reality we must assume that each designer will place different interpretations on these, even if document control is sufficiently draconian to ensure only one written form is permitted. Current approaches attempt to impose highly structured notations and diagrammatic forms on the process, so that we limit what can be said. The implicit assumption seems to be that this will reduce the problems of different theories in the heads of individual designers. There appears to have been little or no research into such claims, and little research into the extent to which the imposition of notations reduces the ability of designers to communicate the theory that is being constructed effectively. There has also been very little research into how designers make judgements in the course of a design project. Clearly a good understanding of this is essential if we want to improve the process by suggesting ways in which we might help them to avoid judgements that lead to system failure. Achieving this is a nontrivial task, but to ignore it because it is difficult is to avoid the crux of the problem.

It is important to recognise the interaction between the cognitive and organisational aspects of the problem. We hope that by integrating both aspects in one research programme we can achieve a constructive tension between the two, making hypotheses in one area that we can test in the other.

Conclusions

In this short paper we have attempted to raise the question of the wisdom of centring all research into software system design on a single paradigm, the life-cycle model, which seems to have arisen with little supporting empirical or theoretical evidence. We have also introduced another model that we believe opens up the possibility of exploring a number of research questions which are currently not sufficiently represented in Software Engineering research. In particular, by taking a less technocentric view we are able to discuss the cognitive and organisational aspects of design in a way that makes them central to the human activity of design, rather than relegating them to peripheral issues that hang from the life-cycle of the artefact.

References

- [1] Petroski H. To engineer is human, Macmillan, 1982
- [2] de Camp L S. Ancient engineers, Tandem, 1977
- [3] Alexander C. Notes on the synthesis of form, Harvard University Press, 1964
- [4] Ortega y Gasset J. Thoughts on technology. In: Mitcham C and Mackey R (ed) Philosophy and technology, The Free Press, New York, pp 290-316, 1972
- [5] Loomes M. Selfconscious or unselfconscious software design?, Journal of Information Technology, 5(1): 33-36, March 1990
- [6] Papert S. A critique of technocentrism in thinking about the school of the future, E&L Memo No. 2, Massachusetts Institute of Technology Media Laboratory, September, 1990
- [7] Burstall R M and Goguen J A. Putting theories together to make specifications, In: Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977
- [8] Naur P. Programming as theory building, Microprocessors and Microprocessing, 15, pp 253-261, 1985
- [9] Kuhn T S. The structure of scientific revolutions (second enlarged edition), University of Chicago Press, 1970
- [10] Lakatos I. Mathematics, science and epistemology, Philosophical Papers 2, CUP, 1976
- [11] Fletcher B. The cultural audit, an individual and organisational investigation, CPI, 1989
- [12] Crimes M. The safety culture audit, MSc Thesis, The University of Hertfordshire, 1992.