DIVISION OF COMPUTER SCIENCE

CODING OR COMPREHENSION?
THE ESSENCE OF SOFTWARE SYSTEM DESIGN

(To appear in the Proceedings of the International AMSE Conference
"Systems Analysis, Control & Design" SYS'93, London , September 1993)

Martin Loomes
Carol Britton
Paul Taylor

Technical Report No.162

September 1993

Coding or comprehension? - The essence of software system design.

Martin Loomes, Carol Britton and Paul Taylor
Division of Computer science,
University of Hertfordshire,
College Lane,
Hatfield, AL10 9AB
email: comqmjl@herts.ac.uk

Abstract

Throughout the disciplines of engineering, modelling and simulation are generally considered as important components of any modern designer's tool kit. They allow the powerful mechanisms of abstraction, decomposition and formalisation to be brought to bear in tangible ways, creating platforms for the analysis, discussion and communication of ideas. Many traditional engineering design disciplines have developed well-integrated, and highly successful procedures for using these tools, with the result that there is no need for individual designers to reflect on the status of the tools or on their use. One reason for this is that the models and simulations come from domains that are clearly separable from that of the artefact being designed. There is little danger, for example, of the civil engineer confusing the plaster model of the dam, or the partial differential equations capturing stress in the structure with the dam itself.

The designer of software, however, is faced with a rather more complex scenario. The artefact to be constructed is, itself, an abstract representation of some perceived reality, and it is often not clear how modelling and simulation relate to this abstract domain. Is the artefact itself a model and, if so, what is it exactly that is being modelled? Is a software simulation of the finished system a model of the system, and if so, is the system a model of itself? Questions such as these may seem contrived and esoteric, and most software designers can function quite satisfactorily without ever discussing them - indeed many designers feel distinctly uncomfortable when such issues are raised. It is important, however, that such issues are directly addressed by those individuals who are proposing methods and CASE tools as frameworks for system design. Underlying assumptions about the role of modelling and simulation in the system development process play a central part in all methods and tools; it is important that such assumptions are explicitly recognised, both by the designer of the tool and by the system developer who is to use it. If the views of the tool designer on such fundamental issues are not fully understood and shared by practitioners, effective use of the tool may be seriously impaired.

This paper discusses the role of modelling in software design, and puts forward the view that software engineers use models, not in the same way as traditional engineers, but more like scientists. The latter construct models to give substance to a theoretical understanding of some domain that does not yet possess a clearly defined structure. The model and theory are developed hand in hand, and can often be understood only in conjunction. We consider two important implications of this view of the software development process. The first of these is that it is not obvious what comprises the artefact: the theory or the model. A further implication is that the traditional system life cycle may well be placing the emphasis of the development process in the wrong place, by stressing the central role of the lines of code being developed, rather than the theoretical comprehension of the system.

Keywords: Modelling, theory, software design

Introduction

Why should we bother to question the manner in which designers set about the task of producing software systems? Surely the Software Development Life Cycle is sufficiently well established that little more of value can be added? Wouldn't we be better employed devising better tools and techniques for empowering the designer to engage in the process more effectively, rather than wasting time on philosophical questions?

Reactions such as these are, sadly, all too common amongst software engineers when they are asked deep questions about the design process. This paper is predicated upon the belief that such reactions are not acceptable in modern engineering practice. Alexander forcibly made the point three decades ago that the modern designer must adopt a self-conscious attitude towards design, and accept the "loss of innocence" that this implies: the modern designer cannot simply work within an established style or school and abrogate responsibility to the establishment, or argue for Divine inspiration, and pass responsibility to a Higher Being [1]. The arguments that this should apply to software engineering have been rehearsed elsewhere [2]. Recent trends within software engineering towards encapsulating many aspects of the process within tools should lead to a deliberate increase in research to place our understanding of the design process on a well-founded basis, in case we encapsulate all the weaknesses of our current understanding in complex, monolithic systems that will render change even more difficult in the future, and also make it impossible for the engineer to accept responsibility for a design task. The difficulties that are perceived in bringing about this change of attitudes may be, in part, attributable to the fact that Computer Science is still a relatively immature, and the vast industry that has sprung up around it often seems to be producing highly sophisticated products with rather primitive technologies. As Sprague de Camp has observed [3]:

> "Primitive peoples live a hand to mouth existence ... Therefore they can less well afford to risk experiment than more advanced people. ...
> As a result, primitive societies are very conservative. Tribal customs prescribe exactly how everything shall be done, on pain of the God's displeasure. An inventor is likely to be liquidated as a dangerous deviationalist"

The tribal customs may be termed "methodologies" and "CASE tools", and the God in question may be corporate management, but the end result is the same: a short-term view of the enterprise.
Holistic research into the design process has largely been ignored in recent years in favour of extensive micro-studies of aspects of the problem. The weakness of this approach is that research findings are only useful within the context of the design process as currently understood. To use the Kuhnian term, research is within the currently accepted paradigm [4]: the conventional life cycle.

This paper calls into question whether the life cycle paradigm is really capable of supporting such extensive research programmes, and attempts to introduce the outline of a new paradigm that might perform this role.

Scientific Approaches.

It is common for academic disciplines that want to be taken seriously to seek to establish a scientific basis. Software Engineering is no exception to this, and there have been numerous calls for a more scientific approach to software design. What is meant by a scientific approach, however, is not well defined. Gries, for example, calls for the establishment of a science of programming, where the laws of each language are clearly, and formally, stated and the task of programming is underpinned by explicit guidelines[5]. Hoare suggests that programming should become more scientific by accepting the onus of mathematical proof between formal specifications and designed solutions [6]. Neither suggests, however, that software designers should behave more like scientists, simply that formality, theory and proof should be given a more prominent place in the life cycle.

A more radical proposal can be made, however, if we try to explore ways in which software design may be made more like science, rather than ways in which it can be carried out scientifically. On the face of it this may seem absurd, since clearly software development is an engineering activity, leading to the construction of a tangible product. The final artefact, however, is not a physical entity such as a bridge or a new chemical compound, but an information structure. We would argue that an alternative to the life cycle paradigm can be found by considering this information structure as a theory: and disciplines whose final products are theories are more usually termed sciences than engineering.

This approach is not original. Burstall and Goguen, for example, proposed that the task of producing specifications could be seen as one of theory construction, with formality being achieved by developing a suitable algebra of theories [7]. Naur has suggested that the task of programming goes beyond the production of code, and involves the construction of a theory of the problem [8]. These two papers, whilst often cited, are usually seen as representing different spheres in the discipline. The former is seen as part of "formal methods", as the goal was to suggest how the formal concept of a theory presentation could be used to capture system requirements (originally the aim was to capture micro-worlds for AI). The latter is usually seen as part of the softer end of systems development, as Naur suggested that theories are essentially mental entities that cannot be captured and communicated, and hence software design must be a small group activity carried out in a holistic fashion by the theory-builders

If we take these two ideas together, however, we have the embryo of a new paradigm for software system development: the theory-building view. Once we have made this transition we can look to the Philosophy of Science for inspiration for our methodology, and some suggested avenues for exploration are outlined below. Before turning to methodological considerations, however, let us briefly consider what we mean by a "theory".

Theories

The term "theory" is not easily defined, and a detailed discussion of its semantics is beyond the scope of this paper. We can, however, clarify some of the properties that are generally expected of a theory, and discuss how these relate to software design.

Ryle builds a powerful analogy between theories and pathways [9]. To have a theory, according to Ryle, is to be aware of a pathway from one place to another in such a way as to be able to both use it and also explain its whereabouts to others. Thus self-consciousness is a necessary pre-requisite for the theory-building approach: the engineer must realise that a theory is being constructed, and be prepared to discuss this explicitly with others. During the construction of a pathway one must be prepared to stamp up and down the path in order to establish it. Subsidence or rocks in the way may cause deviations to the intended route, or even total abandonment of a chosen route. Existing paths may be incorporated in new ones, and may also be used to aid the builder during construction. Thus our software engineer needs to be a rambler as well as a path builder, able to use existing theories as well as construct new ones. We do not sacrifice the ideals of Gries and Hoare in moving to the theory building view, rather we extend them to require the software engineer to contribute to the theoretical world as well as using it.

One very important property of a theory is that we must admit it might not be true: indeed, we might even expect it to be refuted at some future time. There is no doubt that during the software design process many theories are constructed that turn out not to be true. Note, however, that constructing such theories is not "wrong", or "bad design", but an essential part of the process. Science progresses by putting forward bold and relevant conjectures and attempting to refute them. Good engineering practice, under this view, would involve stating theories clearly and explicitly, so that they may more easily be refuted. This seems rather a long way removed from the attitude of many software designers who see their role as proposing a design and defending it against all-comers. Clearly, there must come a point in the design process when the theory is accepted as sufficiently tested to support the task in

hand, and we proceed with a theory that we still expect to be refuted at some future stage. This is not unscientific, of course, for as Popper has observed [10]:

> "The empirical basis of objective science has nothing absolute about it. Science does not rest on solid bedrock. The bold structure of its theories rises, as it were, above a swamp. It is like a building erected on piles. The piles are driven down from above into the swamp, but not down to any natural or "given" base; and if we stop driving the piles deeper, it is not because we have reached firm ground. We simply stop when we are satisfied that the piles are firm enough to carry the structure, at least for the time being".

The engineers and customers together must be happy that the piles have been driven deep enough to ensure that the theory will support the task in hand, and that the theory will not be refuted when the system is put to use, but this is sufficient. The theory must be fit for purpose, but need not be "correct" in any absolute sense. The engineers must also ensure that the theory is internally consistent, for otherwise it is trivially refuted. This is significant, for if the engineer does not take the scientific approach of actively seeking refutations, an inconsistent theory can also be used to prove that the system has every desirable property!

Another property that most people would expect of a theory is that it should aid understanding. This adds a vital new dimension to the life cycle view, for it adds "explication" to the current "specification"--->"implementation" pairs. Specifications are usually interpreted as expressing what a system should do, and implementations as capturing how it should be done. Theories, however, also capture some notion of explanation: typically by the construction of sets of laws and relationships involving both well-understood (grounded) phenomena and new theoretical terms. This often poses a problem for the designer, for increasing the customers' understanding of the problem frequently leads to subsequent refutation of the theory, as they become better placed to devise more stringent tests for the theory. This is a contractual, rather than technical, problem, however, for the theory building view, and it simply means that clean contractual boundaries need to be established and documented [11]. This is a non-trivial matter, but it is not a weakness of the theory building view, rather it demonstrates that the paradigm is capturing explicitly what software designers already know: estimating how much it will cost to solve a customer's problem is virtually impossible in the software industry. It may mean that project management will want to superimpose a documentation structure onto the theory building activities that reflects a traditional life cycle, of course, but this is perfectly possible.

Methodology

It is important to realise that we are not suggesting a program *is* a theory, rather that the task of writing a program involves the production of a theory, and that the driving force behind the design process is the theory, not the program text. One way of viewing the relationship between specifications, theories and programs is to consider both specifications and programs as models of the theory.

In the early stages of a design, the models we construct will generally be considered as specifications, unless we are adopting a strategy of prototyping, in which case we will construct executable models, or programs, throughout. These models will not usually capture all aspects of the current theory, but will address specific abstractions pertinent to the current sub-task, just as an aircraft designer might construct a model to place in a wind tunnel to test aerodynamics, ignoring safety evaluation procedures or fuel consumption, or a statistical model to test system safety parameters under component failure, ignoring aerodynamics. It is important to understand the proper use of these models, for they are being used to test hypotheses drawn from the theory. Without the theory neither the specification nor the program cannot be tested, for no predictions can be made to allow refutation.

One advantage of the theory building approach is that it is easy to see how the process starts. The traditional life cycle usually begins with requirements capture: but how can we capture the requirements for something before we know what that something is? With theory building, however, we start with a theory, no matter how tentative and abstract, that provides the searchlight for seeking out refutations. The facts that cause our tentative theory to be refuted thus become requirements for the subsequent theory to capture. In general, the more experienced an engineer is in a particular problem domain, the better the initial theories will be. Indeed, a good engineer solving a standard problem may already have sufficient knowledge to construct a suitable theory at the first attempt. There may be no need for "specifications" or "prototypes": the only model constructed will be the program. The attempts at refutation must still be as sincere, of course, and experience will equip the engineer with knowledge of the weaknesses of the theory as well as the strengths. The novice software designer will undoubtedly find the construction of a suitable initial theory a very demanding task, and may well flounder, or try lots of inappropriate experiments, hoping to see some patterns emerge that will suggest suitable laws to simplify the task. The task will be complicated by the theories that the customers and users bring to the process, which will need to be elicited and reconciled.

The central trunk of the design process is thus a series of theories, each of which is "better" than the last in some sense. Models may be constructed at any stage, with the final model (for the time being) comprising a computational model that the customer recognises as a program. The issue of what makes one theory "better" than another is one that has caused debate in the Philosophy of Science for many years. It is usually the case that theories that have been refuted are rejected, although a refuted theory may still have a role to play in a less general domain of application. Thus a theory that does not quite meet the requirements of a design task might be the best the designer can find within the resource constraints imposed: the manual then contains instructions as to how the user navigates around the areas where problems might arise. In science it is also the case that more general theories are preferred to those of limited applicability. This may not be the case in Software Engineering, where selling products that are more powerful than the customer is paying for may not be acceptable commercially.

A more difficult issue is deciding what to do when a theory is refuted. Clearly it is not acceptable to start again: we must build on what has gone before. Lakatos suggests that research programmes contain a "hard core", that is, a kernel of the theory that will not be refuted except as a last resort. Moreover, this hard core can be carried from project to project, providing continuity for the engineer, and a starting point for each new design task. The scientist employs negative heuristics, avoiding difficult paths, and positive heuristics, driving the research programme in the right direction [12]. Further discussion of these issues are beyond the scope of this brief introduction to the theory building view, but many of these ideas has been developed elsewhere [13].

One important property of any new paradigm that is proposed is that it should be able to explain more than the one it replaces. The life cycle view of the development process is rather limited: indeed, many people would claim that it only represents an idealised structure that many managers would like their engineers to conform to, rather than capturing the design process as actually carried out. This may, of course, still be a valuable enterprise [14]. Let us briefly illustrate that the theory building view is sufficiently powerful to express two of the major approaches to design that have been suggested.

First, note that if we construct computational models from the outset we can carry out rapid prototyping. Each prototype is used to experiment via computational simulations of the theory, leading to refutation and subsequent refinement, or contractual boundaries if the customer is satisfied with the abstraction presented.

If we construct a formal specification it may be a partial theory presentation for the theory. In this case we can test aspects of the programme directly against the specification. We may even establish a suitable morphism between the specification and the programme text, in

which case there is no point in carrying out testing of both the programme and the specification, for no additional refutations can be found. Establishing this morphism is sometimes referred to as a "correctness proof" [15], a somewhat confusing term as the proof says nothing at all about the fitness for purpose of the theory being modelled. It is simply a correspondence proof, allowing a simplification of the task of refutation.

Conclusions

In this brief paper we have put forward the view that Software Engineering should be considering alternative frameworks for the discussion of the software design process. The documentation-driven approach that is captured by the traditional life cycle has achieved a dominance that threatens to stifle the exploration of issues that do not fit comfortably within it. Moreover, it is difficult to see how research from within this dominant paradigm, with the acceptance of all the assumptions this entails, can ever lead to any radical alternatives.

We have put forward the idea that a fruitful avenue for future research into the software design process might be to consider the theoretical understanding of the system being developed as the true end product, rather than the executable code. This theoretical understanding might be termed a theory, and we can then start to explore the design process as a theory building enterprise, drawing upon the Philosophy of Science for insights regarding methodology. No claim is being made that this alternative framework is "correct", simply that it may stimulate areas of research that do not arise naturally through the life cycle.

References

[1] Alexander,C: *Notes on the synthesis of form*, Harvard University Press, 1964.
[2] Loomes,M: *Selfconscious or unselfconcious design*, JIT 5(1):33-36, March 1990.
[3] Sprague de Camp,L: *Ancient Engineers*, Tandem, 1977.
[4] Kuhn,T: *The structure of scientific revolutions*, University of Chicago Press, 1970.
[5] Gries, D: *The science of programming*, Springer-Verlag, 1981.
[6] Hoare, C: *Programming: dsorcery or science?*, IEEE Software: 141-154, April 1984.
[7] Burstall,R & Goguen,J: *Putting theories together to make specifications*, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977.
[8] Naur,P: *Programming as theory buiolding*, Microprocessing and Microprogramming, 15:253-261, 1985.
[9] Ryle,G: *The concept of mind*, Peregrin Books, 1949.
[10] Popper,C: *The logic of scientific discovery*, Hutchinson & Co., 1959.
[11] Cohen, B: *Justification of formal methods for system specification*, Software and Microsystems, 1(65):119-127, August 1982.
[12] Lakatos,I: *Falsification and the methodology of scientific research programmes*, in Lakatos and Musgrave (editors) Criticism and the groth of knowledge, CUP 1970.
[13] Loomes, M: *Software engineering curriculum design*, PhD thesis, University of Surrey, 1991.
[14] Parnas,D & Clements, P: *A rational design process: how and why to fake it*, in LNCS 186, Springer-Verlag, 1985.
[15] Loomes,M: *Logic and correctness proofs*, in Software Engineering Reference Book, Butterworths, 1991.