THE SOFTWARE PARADIGM

B Warboys

Rapporteur: Dr Paul Watson

The Software Paradigm *

Brian Warboys
Department of Computer Science
University of Manchester M13 9PL

September 11, 1995

Abstract

Note from the briefing for this talk.

"Fundamental since software is the *defining technology* and prescribes the extent of automation designed into a system."

The nature of modern computer systems is such that a single paradigm is an insufficient model. The paper attempts to show that what is required is for projects to create a design *framework* which tolerates change (and failure) in the design process. Within this *framework* the appropriate paradigm should be selected for appropriate design components.

1 Introduction

Two case studies will be used throughout to illustrate and illuminate the issues. They are arbitrarily selected from numerous possiblities. They happened to be on my desk when I wrote this paper.

The first is based on the report (Computer 1993) on the series of fatal accidents which occurred between June 1985 and January 1987 from fatal doses of radiation from the Therac-25 computerised radio therapy machine. References to this case study are labelled '**Therac.***n*.'

The second on the report of the inquiry into the chaos caused in the London Ambulance Service when the new Computer Aided Despatch System went operational on the night of 26 October 1992. References to this case study are labelled '**LAS.***n*'.

My point in using these case studies is not to resort to "scare mongering" but to illustrate the difficulties with practical examples. Software Engineering is not an academic exercise. The detailed attention to semantics cannot be ignored. If it is, it can have disastrous consequences for practical systems.

William Blake (1794) "He who would do good to another must do it in minute particulars General Good is the plea of the scoundrel, hypocrite and flatterer For Art and Science cannot exist but in minutely organised particulars".

LAS.1 ... *on 4 November 1992 the system did fail. This was caused by a minor programming error that caused the system to 'crash'. The automatic change over to the back up system had not been adequately tested, thus the whole system was brought down.*

Therac.1 *In the code, the Class3 variable is incremented by one in each pass through Set-Up Test. Since the Class3 variable is 1 byte, it can only contain a maximum value of 255 decimal. Thus, on every 256th pass through ... the variable overflows and has a zero value. The overexposure occurred when the operator hit the 'set' button at the precise moment that Class3 rolled over to zero.... AECL described the technical 'fix' implemented for this software flaw as simple.*

*A paper prepared for the MOD DSAC seminar on 'Practical Limits of Automation in CIS' London November 1994
Published in The ICL Technical Journal *Ingenuity* May 1995

2 On the nature of Software

Software is "soft"; it is generic, formal and extremely malleable. Its application as a *process tool* has social, psychological and management effects. We are effectively moving away from software programmes as calculators towards viewing software systems as *partners*. This is the basic meaning of the expression "Programming in the Large" introduced to describe the practise of building large systems containing many bought-in components. We might better describe modern approaches where the emphasis is on "programming the business" as "Programming in the Huge".

However the conventional issues with software still remain. It is thus, at a technology level, a precise formulation whilst being, at a human system level, the means of producing a very imprecise *guidance* tool. Real world performance issues are an obvious example:

LAS.2 ... *the computer system itself did not fail in a technical sense. Response times did on occasions become unacceptable, but overall the system did what it had been designed to do.*

Therac.2 *They determined that data-entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace (as is natural for someone who has repeated the procedure a large number of times), the overdose occurred.*

The conflicts between its implementation *needs* and its ultimate system usage represent one of the difficulties. Another the challenge of controlling the conflict between software, being on the one hand *malleable* and one the other *manageable* as a development.

An astonishing fact is that, in spite of decades of evidence to the contrary, "management" and software designers are still eternally optimistic about software development.

LAS.3 *The Chief Executive's report also states 'there is no evidence to suggest that the full system software, when commissioned, will not prove reliable'.*

Therac.3 *Given the complex nature of this software and the basic multitasking design, it is difficult to understand how any part of the code could be labelled 'straightforward' or how confidence could be achieved that 'no execution paths' exist for particular types of software behavior.*

This situation is exaggerated by the trend towards *end-user* programming of software applications. Here the "programmer" is thinking much more about programming the business rather than programming the machine and, unfortunately our research over the last 25 years has tended to focus on the problems of the latter.

Djikstra remarked, as long ago as 1973 at the IBM Newcastle Conference on the Teaching of Computer Science [1], that "Is Computer Science nearing its completion? Is computing practise settling down in a way beyond recovery? Or are, as a result of current circumstances, university professors tired and discouraged".

Already signs that the wider problem is perhaps too difficult to solve? So what does an exploration of "The Software Paradigm" reveal about the limits on the application of this most malleable of technologies?

3 Software Crisis or Software Affliction

Dasgupta in his book on "Design Theory and Computer Science" [2] introduces the notion that we are suffering not from a *crisis* but from an *affliction*. Software Engineers have been talking about a "Software Crisis" virtually since the term "Software Engineering" was coined at the Nato Conferences of 1968/1969 [3]. Crisis is defined as a "turning point". In particular the turning point of a desease when it becomes clear whether the patient will live or die. Yet we have had a crisis for nearly 30 years so the term is misplaced. What we really have is a *chronic affliction*, that is something which lasts for a very long time.

Some of the causes of this affliction lie with:

- The attention to detail that is required
- The conflicts between a malleable and manageable technology
- The evolutionary nature of the software design process
- The evolutionary nature of software itself
- The differing emphasis on the development of products versus the support for human processes

4 On paradigms

A paradigm is usually defined as a pattern or a model. The term is associated closely with the work of Kuhn which led to the formulation of the notion of Kuhnian Paradigms [4]. He identified them as having two related aspects.

- A disciplinary matrix - essentially a network of beliefs, techniques and theorems. They have three main properties:
  - Symbolic generalisations: General formal assertions that are later taken for granted and employed without question. (e.g Ohm's Law).
  - Model beliefs: A commitment to a belief in a model to which the relevant domain conforms. (e.g Bohr-Rutherford model of the atom).
  - Values: e.g scientists believe that prediction should be quantative rather than qualitative.
- Exemplars: Shared examples that illustrate the properties of the paradigm.

5 A superficial review of the most influential Software Paradigms

In the beginning there was:

5.1 The Algorithmic Paradigm (AP)

Based on the notion that software design problems are well-structured. It is characterised as the execution of a domain-specific algorithm, which given a set of requirements, generates a design satisfying them in a finite number of steps. This is classical design automation. It views computing as an algorithmic style and is essentially language-based. Typical examples are mathematical models of airflows, CFD problems and language processors. Algorithms are problem solving systems that do not have explicit access to external knowledge. Rather the knowledge is contained in the algorithm itself. More precisely, control strategy and knowledge about the task domain are intertwined and together "define" the algorithm.

Significantly the regularity of the this paradigm does guarantee the avoidance of errors, only too prevalent, which arise when an 'ad-hoc' approach is taken to well-structured problem domains.

LAS.4 ... *an example of such problems was the failure to identify every 53rd vehicle of the fleet.*

It was soon clear that the class of software which was producable this way was, although important, very limited.

So then:

5.2 The Analysis-Synthesis-Evaluation Paradigm (ASE)

Essentially, given a set of requirements, the software design process principally involves:

- Analysis of requirements
- One or more stages of synthesis principally driven by decomposition
- Evaluation

The best known example is the *Waterfall* Model of the Software Lifecycle. It derives from the desire to make software development "scientific"; essentially to define an *Engineering Method.* At the Nato conference in 1968 [3] Ross observed that "The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. ...The projects that are called successful, have met their specifications. But these specifications were, in the main, based upon the designers ignorance before they started the job."

LAS.5 *It should be noted that the SO quotation for the CAD development was only 35000 pounds - a clear indication that they had almost certainly underestimated the complexity of the requirement..... the bid...was some 700K pounds cheaper....*

The Ross quote was, for the first time to my knowledge (note it was 1968 though!) that it had been suggested that software development was a *learning* process. That software *evolves.*

Therein lies the fallacy with this paradigm. Essentially Software Design:

- Problems are incomplete
- Requirements may be inconsistent
- Acquires a "life of its own". Second order requirements arising from the design process itself.

This leads us to the concept of "Bounded Rationality" expounded by Simon in 1976 [5]. In essence he pointed out that in such systems there are constraints on the cognitive and information processing capabilities of the decision making agent which means that the agent is not independent in a way which could possibly lead to a normal rational process.

LAS.6 *It should be said that in an ideal world it would be difficult to fault the concept of the design. It was ambitious but, if it could be achieved, there is little doubt that major efficiency gains could be made.*

Thus the ASE paradigm leading to the classical design approach of decomposition is not widely applicable to software development. Decomposition essentially identifies possible choices of components considered independently. This produces unbounded problems as the components interact and generate second order requirements.

The ASE paradigm leads the designer to assume that his problem is well-structured. The characteristics of such systems are that they are empirical, that is their solutions, possible transitions are observable. This tendency is exaggerated by modern quality procedures which emphasise the need for formal inspections at the end of each development phase resulting in a "freezing" of the outputs of that stage.

The problem with modern "guidance" systems as distinct from simple tools is that they are essentially non-deterministic. There are all manner of second-order effects. The most significant being that the use of these systems changes our behaviour and hence our requirements of the system.

This rapid and inevitable evolution means that the ASE paradigm is too rigid for all of our needs although useful for some aspects of the system.

5.3 The Formal Design Paradigm (FD)

Since decomposition was clearly limited, the alternative of abstraction with subsequent refinement was introduced. Abstraction provides us with a more flexible tool. Further mathematics provides a tool for abstraction and further as Hoare reasoned in 1986 at his inaugural lecture at Oxford [6]:

- Computers are mathematical machines (behaviour is mathematically defined)
- Programs are mathematical expressions (describe precisely what)
- A programming language is a mathematical theory (a formal system for programming)
- Programming is a mathematical activity.

Thus software design becomes a mathematical proposition or theory that solves the problem as represented by the specification of the requirements. Hence, in the FD paradigm, the design process is an activity which exploits the traditional methods of mathematical reasoning. It has been frequently justified by quoting Djikstra's remark that "Testing can show the presence of errors but never their absence".

However like the ASE paradigm it assumes that requirements are known and that essentially the problem is well-structured. It is in essence a refinement of the ASE paradigm.

As applications of software became more strategic so grew the desire for decision-support systems with more "intelligence".

Thus we arrive at:

5.4 The Artificial Intelligence Paradigm (AI)

This derives from the view that software design problems are not well-structured. Thus rather than define a rigid set of requirements the system should absorb domain-specific knowledge to an extent that it can provide a set of possible answers which satisfy the constraints implied by that knowledge. The design process involves:

- A symbolic representation of the problem (the problem space) structured in terms of:
  - Initial problem state
  - Goal or desired problem state
  - All other states are reached or considered in attempting to reach the goal state from the initial state
- Transitions from one state to another are affected by applying one of a finite set of operators
- The result of applying operators is, in effect, a search for the solution through the problem space

The paradigm is explicitly founded on the concept of search, knowledge and heuristics, in that the search is determined by knowledge of the problem domain and by a collation of general heuristics. The rules are partly domain specific and partly domain independent. The more the problem solving system relies on domain specific knowledge the "stronger" is the problem solving method itself. "Expert Systems" are instances of such "strong" methods. In the case of the Algorithmic Paradigm, the algorithmic design styles correspond to the domain independent heuristics of AI. However domain specific knowledge in algoithms are generally fewer in number but of greater scope and granularity than the rule-based type of knowledge seen in AI systems.

In such situations the algorithmic design system converges to a solution with virtually no search of the problem space.

Again an interesting class of problems scan be addressed in this style but the rule-base tends to become unstructured and in very large systems difficult to digest. In addition many components are dealing with well-structured problems and can be easily developed through algorithmic approaches.

This leads us to the obvious conclusion that systems are hybrid in nature and that the appropriate paradigm is one which recognises this fact. Thus:

5.5 The Theory of Evolutionary Design Paradigm (TED)

This is also referred to by Dasgupta [2] as "The Theory of Plausible Design" but I prefer the emphasis on evolution. This takes as its starting point that the software design process is evolutionary. The requirement is to create a design *framework* which accomodates change (and failure) and utilises one of the previous paradigms at the appropriate places. Basically a collection of tentative hypotheses such that:

- One can attempt to provide evidence in their favour to establish the "plausibility" of the design
- Belief in the design's plausibility may have to be revised

TED is based on the view of design as an "empirical scientific" activity in stark contrast to the Formal Paradigm which views design as a "mathematical modelling" activity. Essentially TED maintains that design in mixed paradigm systems can only consist of establishing constraints on the implementation. Collectively these constraints *represent* the design.

The approach means, in practise, that designers are forced to resort to satisfactory rather than optimal designs. Thus recognising both the evolutionary and error-prone nature of software development. It is worth noting that, even given that there will be elements which clearly can be formulated as well-structured problems, their optimal solutions may turn out to be intractable. It is interesting to observe that in both our case studies "perfection" was required and being unachievable led to problems which were major contributing factors.

LAS.7 *However its success would depend on the near 100 percent accuracy and reliability of the technology in its totality. Anything less could result in serious disruption to LAS operations.*

Therac.4 *A common mistake in engineering, in this case and many others, is to put too much confidence in software.*

Under such conditions it is fruitful to view the act of software design as an evolutionary process and the design itself at *any* stage of its development as a *tentative* solution to the problem posed. The adequacy of the design is determined solely according to whether it meets the requirements prevailing at *that* stage of the design process. Thus the design is not only tentative at intermediate stages of its development but also when the designers (or their managers!) see fit to *terminate* or *freeze* the design.

In other words, according to the evolutionary model, a design at any stage of its development (including the *final* stage) is:

- An evolutionary offspring of an earlier design form
- Likely to evolve further in the future

Thus software design problem solving is a special case of the process of scientific discovery and suggests that we recognise in our design management systems:

- The strong link between natural and artificial sciences

- The use of testable hypotheses as a method (use of prototyping and simulation)
- The nature of evolutionary systems and hence the need for an incremental development approach (a working product (or subset) which evolves on a *daily* basis)

This is recognised in the attachment which accompanied our briefings. "An early and modified SHAPE description of CIS shows its complex and multi-disciplinary nature. It reads "CIS is a designed system comprising organisational structure, doctrines, procedures, rules, personnel, data, software, communications and equipment...". It is worth studying the scope of this description since it indicates very clearly the 'socio-technical' nature of CIS and the hybrid nature of the science-base which underpins it."

In building such software systems we are attempting to apply the style of the software paradigm(s) to real world problems, but recognising at the same time that the problem cannot be transformed into a basic calculation problem.

6 Conclusions

The advantages and shortfalls of the various paradigms outlined above are, I hope, evident. They all contain useful notions and, as the various quotations from the case studies clearly illustrate, they should not be ignored.

However the TED paradigm suggests that we need to construct a design framework for modern systems which allows for the embedding of the appropriate paradigm to meet a specific design component requirement. This framework must also, from the beginning, cope with the problem of the evolution of the system. As systems move from a task-oriented "do this" approach to a process-oriented "achieve this" approach so the need for smooth evolution will grow. As more of the human process is codified so the need for the process to change as it is used will grow.

Further, in practise, the problem is how to apply the disciplines of systems engineering at a real world level. How to include the behaviour of the human user in the system design.

At this level the system needs to be evaluated against three criteria:

- Organisational Adoptability (How easy is it for the organisation to adopt)

  LAS.8 *..the inability of the system to cope easily with certain established working practises (eg the taking of a vehicle different to the one allocated by the system).*

  LAS.9 *..the impact of CAD upon the existing communications infrastructure was never properly and systematically understood.*

- Community Adoptability (How much does the benefit depend on everybody else adopting it)

  LAS.10 *It was recognised that a system such as this would be a 'first' ... (other similar ambulance systems were rejected - my paraphrasing) ... there is no confidence in the system.*

- Agent of Change (How likely is it to facilitate some new approach or make an existing one obsolete)

  LAS.11 *Management were misguided or naive in believing that computer systems in themselves could bring about such changes in human practises.*

  LAS.12 *The changes to CAC operation ... made it extremely difficult for staff to intervene and correct the system.*

Such questions are a clear indication of the hybrid nature of the design task. In such an environment "design by constraint and evolution" is a very attractive option.

References

[1] E.W.Dijkstra "Selected Writings on Computing: A Personal Perspective" Springer-Verlag 1982

[2] S.Dasgupta "Design Theory and Computer Science" Cambridge University Press 1991

[3] P.Naur and B.Randell "Software Engineering: Report on 1968 NATO Conference" Nato 1969

[4] T.S.Kuhn "The Structure of Scientific Revolutions" University of Chicago Press 1972

[5] H.A.Simon "The Sciences of the Artificial" MIT Press 1981

[6] C.A.R.Hoare "The Mathematics of Programming" Inaugural Lecture, University of Oxford, Clarendon Press 1986

DISCUSSION

Rapporteur: Dr Paul Watson

Following Professor Warboys' points about the difficulties of writing Critical software Professor Randall made the point that at the time software is written its criticality may not be known. For example, a medic had been discovered using Excel on-line during operations.

Dr Lesk asked why Professor Warboys hadn't mentioned Prototyping and Testing. He felt that Software has the advantage over hardware in that it is easier to test. Also, time invested in prototyping can reduce surprises when Users get the product.

Professor Warboys replied that Testing and Prototyping were complementary to what he had been discussing. He said that Testing was important for management because the nearer a product gets to a delivery date, the more attention management give to it, and so the testing phase gets a great deal of attention.

Regarding the evolution of requirements as software is developed, Dr Bourgonjon asked if Professor Warboys agreed that it was necessary to establish domains within which change can occur and this was helped by establishing requirements as early as possible.

Professor Warboys agreed that systems can only change in certain dimensions and software should be designed with this in mind.

Mr Ainsworth made the point that one of the problems with developing software was that when it was written, there was only limited knowledge of how it would be used, particularly because products evolve in the marketplace.

Mr Dobson made the point that a consequence of the evolutionary approach to software development espoused by Professor Warboys was that requirements must be able to evolve, but this went against current practise in which the requirements are fixed in a legally binding contract.

Professor Warboys said that this is a problem and there is a danger that the legal situation may force IT developers to adopt methods of working which aren't the best.

Mr Dobson added that if some major IT suppliers are bankrupted as a result of legal action, then the laws may change.

Professor Martin asked whether lawsuits will lead to a trend towards large companies buying in and configuring commodity software, rather than risking writing their own. Professor Warboys felt that there would be increased outsourcing to specialized software companies - for example all telephone software might be written by one company. These companies would insure themselves against lawsuits, and because insurers would only insure those companies with a good track record the overall effect would be an increase in the quality of software.