Postmodern Software Development

HUGH ROBINSON[1,3], PAT HALL[1], FIONA HOVENDEN[1] AND JANET RACHEL[2]

[1]*Computing Department, Open University, Walton Hall, Milton Keynes, MK7 6AA, UK*
[2]*Department of Innovation Studies, University of East London, London, UK*
Email: H.M.Robinson@open.ac.uk, P.A.V.Hall@open.ac.uk, F.M.Hovenden@open.ac.uk and *J.Rachel@east-london.ac.uk*

We discuss the 'software crisis' as a social and cultural phenomenon, arguing that it can be viewed as (one more) manifestation of postmodernism. We illustrate our argument with a range of examples taken from software engineering, demonstrating software engineering's roots in (and commitment to) modernism and the nature of its *fin de siècle* predicament. We argue that current attempts within software engineering to respond to the software crisis have not been adequate and that a new, more humble, approach to software development is required.

Received October 2, 1996; revised September 30, 1998

1. INTRODUCTION

In this paper we look at software development and its crises as a cultural phenomenon, and consider the relationship of these 'crises' to that general cultural movement termed 'postmodernism'. We explore the modernist roots of software development, the ways in which this influence permits certain aspects of software development and inhibits others, and the ways in which the notion of 'crisis' is deployed to maintain a need for salvation. Finally, we consider ways in which software development has broken out of the modernist mould and whether postmodernism has anything to offer this process. We suggest that software development has to become a more locally negotiated phenomenon, embracing a holistic picture of local constraints and resources, and thereby becoming a more responsive technology. Such a reconsideration of the nature of software development implies a change to the software engineering curriculum to incorporate a consideration of the epistemological and socio-historic ideas that we outline below.

2. SOFTWARE, WEREWOLVES AND CRISES

It is one of the canons of software engineering that there are problems with the development of software. On this much there has been widespread agreement since early in the history of the development of software. These difficulties were focused on over 25 years ago at the historic NATO workshop on Software Engineering [1] in 1968, and later discussed in Brooks's [2] *The Mythical Man-Month*. The term 'software crisis' was coined to embrace all supposed ailments afflicting the production of software and this software crisis was seen to be a dominant problem to be addressed by the profession—it was like a werewolf that needed slaying with a silver bullet [3, 4].

Of course, there have been many attempts to slay the werewolf or, at least, to discover the talismanic sign that would ward off its evil afflictions. The history of software development is, in the large part, a history of salvationary devices that have come (full of a high promise of deliverance and redemption) and gone (like yet another lover who has failed to match up to expectations). At various times, the salvationary device has been structured programming, relational databases, logic programming, knowledge-based systems, CASE technology, formal methods, object-oriented technology and so on. These salvationary devices have been the subject of grand passions and have attracted financial support from national and transnational structures. There has been, for example, the United Kingdom Alvey programme of the 1970s and 1980s and the succession of Esprit rounds funded by the European Community. Similar initiatives have taken place in the USA and in Japan.

Yet each salvationary device has failed, for the werewolf still stalks the landscapes of software. It is a sign of the times that a serious but popular magazine—*Scientific American*—can run an article by a staff writer entitled *Software's chronic crisis* [5], cataloguing failures that range from Denver's baggage handling system to the Clementine satellite. The lead page of the article has the following paragraph:

> To veteran software developers, the Denver debacle is notable only for its visibility. Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled. The average software development project over-shoots its schedule by half; larger projects do worse. And some three quarters of all large systems are 'operating failures' that either do not function as intended or are not used at all. [5, pp. 72–73]

Indeed, in the popular imagination, software has a very real *fin de siècle* predicament, with the perceived inability to

[3]Corresponding author.

handle the millennium date change and consequent fears of apocalyptic chaos.

Paradoxically, software engineering is both the victim and the perpetrator in this crisis. Ever since the coining of the term 'software engineering' [1], software engineering has constituted itself as being in crisis. A crisis demands attention and resources. It sets agendas that help define the discipline that deems itself in crisis. However, some incredulity must be exercised about this notion of permanent crisis,[4] as suggested by the study of Glass [6].

What lies behind this perceived failure? Was it that the technical solutions were not appropriate, that technically the software failed? An answer that is sometimes produced—by software engineers—is that the technical success need not be questioned (and that it makes no sense to question the technical success). Rather, the failure is seen to lie with the methods and tools which have been used to construct the software: it is these methods and tools which have not proved acceptable to human users and therein lies the cause for failure. Such an answer is not uncommon and has considerable appeal to certain constituencies within software engineering (human–computer interaction, for example).

However, such an analysis of failure leaves the business of software development *per se* untouched and seeks rectification in matters which have been predefined as somehow separate from software engineering. That is, such an analysis seeks to (re)define the technical as that which does not fail. We would like to re-establish the constitutive relationship between subject and method. We now offer an alternative analysis of software development and software engineering, emphasizing their essential *modernist* nature, with the predicaments that inescapably arise from that. This paper then goes on to present a postmodern[5] account of/for software engineering.

3. SOFTWARE ENGINEERING AS MODERNISM

Modernism is a set of epistemological positions, epitomized by the 'Enlightenment's ideals of perfection and the notion, inspired by modern science, of the infinite progress of knowledge and an infinite advance toward social and moral betterment' [8], where the progress of science seemed to be revealing a universal truth about the world. That is, the triumph of reason over religion, myth, craft, etc. is modernism writ large. A classic exposition of this position is given by Galileo [9].

> SALVIATI. If what we are discussing were a point of law or of the humanities, in which neither true nor false exists, one might trust in subtlety of mind and readiness of tongue and in the greater experience of the writers, and expect him who excelled in those things to make his reasoning most plausible, and one might judge it to be the best. But in the natural sciences, whose conclusions are true and necessary and have nothing to do with human will, one must take care not to place oneself in the defence of error; . . .

A summary is given by Hamilton [10], with the following ten characteristics:

1. Reason—primacy of reason and rationality as ways of organizing knowledge . . .
2. Empiricism— . . . all thought and knowledge . . . is based on empirical facts . . .
3. Science— . . . key to expanding all human knowledge . . .
4. Universalism— . . . reason and science could be applied to any and every situation . . . produces general laws . . .
5. Progress— . . . condition of human being could be improved . . . ever-increasing
6. Individualism—starting point for all knowledge and action
7. Toleration— . . . all human beings are essentially the same . . .
8. Freedom—an opposition to feudal and traditional constraints . . .
9. Uniformity of human nature— . . . the principal characteristics of human nature were always and everywhere the same.
10. Secularism— . . . secular knowledge free of religious orthodoxies . . .

Modernism is also about 'conquest—the imperial regulation of land, the discipline of the soul, and the creation of truth' [11]. Modernism lays the world bare, stripped of myth and mystery, as a rational enterprise, ordered and reliable. Knowledge of this rational world is unproblematic and can be gained by the methods of that defining paradigm of rational knowledge—science: 'Everything that can be thought at all can be thought clearly. Everything that can be put into words can be put clearly.' [12]. Modernism is linked with the idea of progress, of history as the progression of human society from an uncivilized, barbaric state to a civilized, cultivated state that is typified by the ideals of Western liberal democracy. These strong and exclusive themes that run through modernism—rationality, control, progress, etc.—have been seen as defining features by postmodernist thinkers and have been termed metanarratives or grand theories [13].

It is our contention that software development and software engineering are very strongly rooted in modernism. For example, structured development methods can be seen to have antecedents in the Taylor/Ford models of early modernist industrial production.[6] The future-directedness of the software industry, and the search for the next

[4] We are reminded of the fable of the shepherd boy who cried 'Wolf!' and the attention which that brought, but only on the first few occasions. Crying 'crisis' is rather like crying '(were)wolf', except—it seems—you can never cry (were)wolf too often in software engineering.

[5] There is no single spelling of the name of the movement: it may be 'postmodern', 'post-Modern' or 'Post-Modern'. Jencks [7] gives a brief account of the variations, commenting that the various spellings 'differ slightly, like the changing aliases of a criminal desperately wanting to be captured'.

[6] Murray [14] gives the four basic principles of Fordism as: standardized products; repeated tasks having potential for automation; unautomated tasks

'silver bullet', emulate the modernist commitment to progressiveness and a break with history. We show further examples by referring to the writing of some of the leading figures in software development. We should emphasize that our choice is in no way intended to be invidious: the choice was almost at random, coloured by our own technical interests and by the highly articulate way in which the writings represent ideas.

Let us start with consideration of a paper by one of the most influential academic leaders of software engineering, C. A. R. Hoare of the Programming Research Group at Oxford University. In a paper [15] *Programming: Sorcery or Science?* published in the leading journal *IEEE Software*, he looks forward to a move from the Dark Ages into the Enlightenment. He states that 'the programmer of today shares many attributes with the craftsmen of yesterday' through apprenticeship and experiential learning, and 'shares many attributes with the high priest' with software manuals taking the role of sacred works. Hoare then considers 'the rise of engineering' and, moving from third person to first person, looks forward: 'we would like to claim that computer programming has transcended its origins as a craft, has avoided the temptation to form itself into a priesthood, and can now be regarded as a fully fledged engineering profession' [15, p. 7]. He looks forward to a continuing flow of standard text books which 'promises to transform the arcane and error-prone craft of computer programming to meet the highest standards of a modern engineering profession' [15, p. 8]. The process of software development would be characterized by its use of mathematics, producing 'a complete, unambiguous, and provably consistent specification for the entire end product' [15, p. 8] and 'mathematical proof that if each of the components meets its specification, then when all the components are assembled, the overall product will meet the overall specification agreed to by the client' [15, p. 9] so that 'we hope to eliminate the so-called 'system integration' phase of many current projects, in which bugs are painfully detected and laboriously removed from the interfaces between the components' [15, p. 10]. In conclusion, Hoare states,

> And I believe that in our branch of engineering, above all others, the academic ideals of rigor and elegance will pay the highest dividends in practical terms of reducing costs, increasing performance, and in directing the great sources of computational power on the surface of a silicon chip to the use and convenience of man.

This is the proclamation of an ideal worthy of the best of the Enlightenment!

At the time of Hoare writing this article the practice of software development was already well established in looking to conventional engineering for its salvation, through the adoption of quality systems and engineering standards such as those characterized by the UK standard BS5750.

Hoare, of course, was not alone in his modernist advocacy of mathematical rigour and elegance. During the 1980s, for example, work on database systems (and, to a lesser extent, information systems generally) became dominated by the notion of a formal conceptual schema that would capture and represent the essence of (what was regarded as) some muddled and contextual 'real-world' situation, in a manner directly inspired by the logical atomism of the early Wittgenstein ([16] gives a detailed account). It was seen as quite inappropriate to use natural language to describe this conceptual schema: '. . . natural language fails the criteria. While rich and contextual, suiting the real world, it allows ambiguity and inaccuracy. It cannot be subjected to rigid rules of mechanistic logic' [17]. Using these 'rigid rules of mechanistic logic' would allow the true nature of reality to be represented, rather than the seeming world of appearances, as Vermeir [18] claims: 'conceptual information analysis or conceptualisation . . . is concerned with the problem of precisely and unambiguously representing the deep semantic structure of a given situation ("Universe of Discourse") in some formal system'. Rationality and logic were seen to be all-embracing, to be the metanarrative by which the world could be articulated and quantified, as evinced by Steel's [19] choice of title for a paper at a prestigious conference on database semantics: *A minimal conceptual schema for life, the universe and everything*.

Let us now continue through the modern era of software development with an extended quotation from Lehman [20], writing under the intriguing title *Uncertainty in computer applications and its control through the engineering of software*. In this quotation we have italicized a number of terms to emphasize the prescriptive, deterministic and mathematical nature of the view of software development being offered. Lehman gives an account of the software engineering agenda as:

> Recognition of the need to define and follow a *disciplined* process is perhaps the most important advance in system development of recent years. To achieve it one first needs to develop a *process model* (Lehman and Stenning, 1985; Potts, 1981; Wileden and Dowson, 1986; Dowson, 1987) to define a *systematic* and *coherent* path from formulation of an application concept via realisation of a usable system to its subsequent evolution. Process models may be generic or specific. Whether they can reasonably be considered *algorithmic* is a matter of some controversy (Osterweil, 1987; Lehman, 1987b). Models are developed by first *identifying the technical and management activities required*, the extent of information capture and storage and the interfaces, relationships and dependencies between all these. Together they provide the *structure and composition* of the basic

analysed using work study methods, to enable the easy training of workers and their easy replacement; and production lines with the work moving to the workers.

> process. Given this, one then selects or develops methods to *execute technical development* activities. The introduction of defined and disciplined methods permits the application of *computer based development tools*. These provide mechanised support for individual activities and their systematic control. If appropriately conceived, the *totality of methods and tools provides support for all* aspects and stages of system evolution. True overall effectiveness will, however, be achieved only if data representation, methods and tools can and are integrated to provide *full and coherent lifetime support*. Even then, a process is only as good as is the *rigour* of its application.
>
> This is why techniques and tools to *facilitate and control planning and management* of a group and its activities, the **project**, *must* be included when planning and implementing integrated lifetime development support. Equally, one requires both to support management of the emerging software and system **product** during development, its subsequent release to users and its evolution. Such product related function is exemplified by the need for management of component variants and versions, system configuration and fault fixing.

Elsewhere he writes of the need for training and education:

> Technical personnel not educated in the new technology ab initio *must learn that they cannot be allowed to follow a development process of their own choosing. They must accept constraints in the letter and the spirit; faithfully and reliably follow documentation and quality assurance standards and procedures*. To do this, they must understand and have true appreciation of the new technology and the reasoning that underlies it. For *successful exploitation* understanding and *acceptance* is, indeed, *necessary at all levels, management* and operational.

All of this, we would submit, fits completely within the modernist agenda, of the application of science to solve human problems.

4. . . . AND ITS MODERNIST BAGGAGE

A key component within this commitment by software engineering to modernism is that of instrumental rationality: software engineering is viewed as a rational artefact that is deployed by professional experts to solve discrete problems in institutional settings. This instrumental rationality is unquestionable (that is, it is part of an underlying belief) because it is the product of the metanarrative. For example, as we have observed elsewhere [21], software engineering embodies a view of the world as being composed of unitary problems, each capable of rational solution via the application of the technology. Whilst accepting that these problems are an articulation of the genuine needs of real people in a world suffused with the full complexity and richness of human life, software engineering nevertheless chooses a rubric with terms such as 'problem domain' or 'user requirements' to describe where it begins its activities. Software engineering constructs the world as being composed of problems, with an existence that is distinct from the methods and tools of software engineering. These problems are conceived of as being consistent, precise and capable of specification—the requirements of the user. These requirements are objective and separate from the methods and tools that will be applied to them, as will be the solution—the final, delivered system that is a product of the application of the methods and tools.

Such a grand theory as instrumental rationality is not without baggage that becomes part of both software engineering and the institutional settings in which it is deployed. Instrumental rationality *qua* software engineering, demands that people be viewed in a particular way. Software engineers are conceived as the professional expert whilst the consumers of the product of the engineers' labour are conceived as the users (or, occasionally, as the client, although this term would more properly be reserved for the management purchasing software artefacts for use by its staff). The roles and relationships that come with this expert/user distinction are not those of equals: the professional expert enjoys an epistemologically privileged position in comparison to that of the user (as in the 'deep semantic structure' of the Vermeir quotation, above, for example). Consequently, the user's experience and knowledge of their world is devalued and denied. Such a view of the world places an emphasis on apparently enduring and objective truths, and discounts local context, circumstance and expertise. Thus, when faced with any perceived mismatch between the technology and the users' circumstances, there is an impetus to deny the authenticity of users' knowledge, re-affirm the appropriateness of the technology and confirm the perceived mismatch as being more apparent than real. That is, if only the technology were more appropriately understood, applied and presented, then the perceived mismatch—any real problems—would disappear!

This also indicates the reverse aspect of the modernist commitment to universalism and tolerance. Where there is a commitment to both rationality and universalism there can only be one acceptable rationality. To maintain this position, modernism needs to monopolize notions of rationality—what is not modern, therefore, is irrational. Such universalism requires uniformity—'all men are equal', as long as everyone behaves like a certain kind of man, she or he will be regarded as an equal. This is a social contract which requires the adoption of a certain kind of identity. To be concerned with gender, racial or class differences, for example, is to be un-modern. Where users, or even members of the development community, disagree with the 'rational' solution, this is not seen as an aspect of contextually situated knowledge to be taken into account, but as an 'irrational' response, which can therefore be ignored.

5. THE BREAKDOWN OF SOFTWARE'S MODERNISM

We contend that software engineering exhibits symptoms of breakdown, with an acceptance amongst leading advocates that it has failed to deliver the promises of high rationality. We have already cited examples of such an acceptance (as reported by Gibbs [5], for instance) and the trend away from using structured methods may be seen as further example. It is also interesting to note, in this context, Hoare's reported [22] admission that the use of mathematical proof has been overstated as the touchstone by which competent software may be produced. Coupled with this lack of confidence in success is a preoccupation with Messianic approaches and cults that offer the promise of concluding the project of modernism within software engineering. Despite a recognition within the profession that, probably, a software engineer's stone does not exist [3] there continues to be the belief that such a salvationary device is within software engineering's grasp [4].

As before, we support our contention by looking at key writings. The view that software development should be solely underpinned by mathematics, depends on a view of mathematics and science as being unproblematic. That is, they are the very epitome of a detached, dispassionate and rational activity. This view of mathematics and science would be challenged by many current philosophers and sociologists of mathematics and science. For example, studies of mathematicians reveal that the concept of an agreed proof is sometimes illusory: mathematicians make mistakes (or, at least, believe that other mathematicians make mistakes). For example, empirical studies by MacKenzie [23] have shown that: 'Although mathematical proof is being sought precisely because of the certainty it is ordinarily held to grant, constructing proofs of computer systems correctness again turns out to be no simple "application" of mathematics. It involves negotiating what proof consists of.' This supports the contentions made in DeMillo *et al*. [24]. Science, it can be argued [25], proceeds as much by constructing facts as by discovering them and there can be bitter dispute over what are the 'facts' that cannot be resolved by appeal to observation.

A deep concern about the use of proof in the development of software was raised by Fetzer [26], in the premier computing journal, the *Communications of the ACM*. Fetzer was a philosopher, spending a postgraduate fellowship in a computing department. His paper was confrontational in its title—*Program verification: the very idea*—and proceeded to attack the idea of formal proof of correctness from various angles. Regarding the notion that the act of proof itself is a social process, he asked:

> . . . if program verification, like mathematical validation, could only occur as the result of a fallible social process, if it could occur at all, then what would distinguish programming procedures from other expert activities, such as judges deciding cases at law and referees reviewing articles for journals? [26, p. 1048]

Further, he distinguished deductive methods from inductive methods,

> For results in logic and mathematics fall within the domain of deductive methodology and require demonstrations. Lawful and causal claims fall within the domain of empirical inquiries and require inductive warrants. [26, p. 1050]

and argued that the verification of operational programs was more like applied mathematics:

> Algorithms, rather than programs, thus appear to be the appropriate candidates for analogies with pure mathematics, while programs bear comparison with applied mathematics. Propositions in applied mathematics, unlike those in pure mathematics, run the risk of observational and experimental disconfirmation. [26, p. 1057]

leading to the conclusion that

> As Einstein remarked, insofar as the laws of mathematics refer to reality, they are not certain; and insofar as they are certain, they do not refer to reality. [26, p. 1060]

> The operational performance of these complex systems should never be taken for granted and cannot be guaranteed. [26, p. 1062]

In other words, certainty about the products of software development was not achievable, at least not by reference to mathematics. Proof within a mathematical system does not necessarily entail proof within the 'lifeworld' system that the mathematical system is modelling.

This attack on the rationalist stance of computing was echoed in another leading journal—the *Computer Journal*—again by an outsider making a temporary sojourn within computing. In 1986 Leith considered the other end of the connection between formal computational systems and the world, between the system and the application domain, critically appraising the use of logic programming in the formalization of law in a paper [27] also confrontationally titled—*Fundamental errors in legal logic programming*. He expressed his concerns as follows:

> I wish to criticise the use of logic programming in the law, not from the computer science perspective but from the position outside the usual domains of computing. For most of the claims made for logic programming are not made in terms of computer science goals (e.g. computational efficiency, efficient usage of memory, etc.) but arise from the relationship of logic programming to the extra computational world—for example the legal world.

and then went on to carefully consider the treatment of the British Nationality Act within logic programming by Kowalski and colleagues at Imperial College, concluding

> I would argue that the problems of logic programmers are the result of a false epistemology: they see the world in terms of a computational model and fail to stand outside that model. Thus, whenever they attempt to apply their model to the real world, they will always fail. For the world is not a logical world.

Both papers have very similar concerns about the way formal mathematical systems are used to describe phenomena in the world. These formal systems must be connected to that world at some point and this connection demonstrates their (non-formal) validity. It is at these points of connection that problems may occur, for this is where the implications of the epistemologies of the system and the world connect. The system's success depends upon whether these epistemologies collide or co-operate.

At around the same time as these papers were published, a number of significant failures of computing systems began to emerge. For some time Peter Neuman had been running a column 'Risks' in *Software Engineering Notes*, to which people posted reports of entertaining nonsenses like systems issuing invoices for $0.00 and insisting that the corresponding payment for $0.00 occurred in order to avoid legal action. Computing professionals realized that computing systems could be in error, with disastrous consequences, and professional committees began to look at these 'Safety Critical' systems. In 1986 concern emerged that the failure of the Therac-25 radiation treatment equipment in the US was attributable to software. The Institute of Radiologists in the UK wrote to the British Computer Society and concern, already active there, became more focused.

The Therac failure led to several deaths. It provides a useful case study because, although no official accident investigation was ever carried out [28], a detailed report was compiled from several sources by Leveson and Turner [29]. There were a number of programming errors that led to the system failing, one in particular occurred when operators became proficient, worked rapidly and made an error in data entry that they had to redo. The errors were made by an engineer who had never been formally trained in the production of software, but the system was not designed to cope with such a use case. Leveson and Turner [29] concluded that the problem was larger than a single cause:

> Most previous accounts of the Therac-25 accidents blamed them on a software error and stopped there. This is not very useful and, in fact, can be misleading and dangerous. If we are to prevent such accidents in the future, we must dig deeper. Most accidents involving complex technology are caused by a combination of organisational, managerial, technical, and, sometimes, sociological or political factors. Preventing accidents requires paying attention to all the root causes, not just the precipitating event in a particular circumstance [29, p. 41]

This conclusion on the multiplicity of co-factors at work in software-based system failures[7] indicates that developing software is about more than writing bug-free code. It is an amalgam of various analytical, design, implementation, predictive and managerial activities, embedded in dynamic social systems, replete with already developed sites of co-operation and conflict.

6. POSTMODERN SOFTWARE DEVELOPMENT PRACTICE

We have seen that the modernist narrative of software engineering is breaking down and that, quite appropriately, there is strenuous activity to repair the breakdown from within the modernist narrative. However, some people have attempted to break away from the modernist tradition, to develop software in different ways, ways which might be termed 'postmodernist'.

Postmodernism can be thought of as a reaction to modernism—characterized by the triumph of reason, for example in 'the French Enlightenment's ideals of perfection and the notion, inspired by modern science, of the infinite progress of knowledge and an infinite advance toward social and moral betterment' [8]—a reaction in which the old confidence has disappeared (Huyssen [30] writes of 'the Exhaustion of the Modernist Movement'). Hassan [31] has characterized this by 11 definiens, listed here without the elaboration given by him: indeterminacies, fragmentation, decanonization, self-less-ness and depth-less-ness, unpresentable and unrepresentable, irony, hybridization, carnivalization, performance and participation, constructionism and immanence. The overall theme of Hassan's [31] essay was that of pluralism: pluralism is seen as important in legitimating other voices within the postmodern world, feminists and gays and ethnic minorities, for example, by explicitly dispersing the authority of the authorial voice.

Lyotard has defined the postmodern as 'incredulity towards metanarratives' [13], which might be regarded as suspicion towards universal explanations or themes. Jameson [32] articulates this breakdown of metanarratives as 'the transformation of reality into images' and 'the fragmentation of time into a series of perpetual presents' [32, p. 125]. Modernism is a finite and limited project where, as Smart [11] writes:

> the pursuit of unshakeable foundations for analytic truth constitutes a fruitless project, one that will continue to remain incomplete in so far as the metaphysical presuppositions at the heart of Western philosophy are themselves problematic.

and, as characterized by Bauman [33], it is

> an inherently and irreparably controversial project, torn apart right through its heart by incompatible,

[7] As Leveson [28] indicates, this is not just a problem that faces the software industry. Similar problems occur in the more traditional engineering aspects of the aeronautics and nuclear industries, for example. However, software is our business, and our concern, here.

> demands/promises/hopes of autonomy and order, emancipation and normativity, freedom and discipline.

So philosophers studying modernism predict its demise, where the grand narratives and unitary notions of epistemological authority begin to break down.

We have seen in the previous section a need for incredulity towards the metanarratives of software engineering, the use of mathematics and methodologies. We now consider themes and practices within software development which might be termed postmodern. Importantly, the account we give below reflects our interests and preoccupations and should not be taken as any exclusive prescription for what is postmodern—other authors have written in a way which broadly echoes and articulates our concerns (see the collection of Floyd *et al.* [34] or Bjerknes *et al.* [35], for example).

6.1. Hard and soft

Computing and software development have arisen out of diverse disciplines, from mathematics and electrical engineering, to cognitive psychology and management science. This diversity gives rise to a variety of approaches and positions within software development, but such variety is often hidden by the modernist move to create universal explanations. A universalizable solution, for example, requires a universalizable problem definition.

We argue that the hard/soft division is one such universalizing move, where 'hard' is the preferred form of problem definition and solution, and 'soft' is the problematic. The distinction between hard and soft is often taken to mean the difference between the definite information that is regarded as the stuff of the software-based system, and the uncertain, 'woolly' information that is regarded as the stuff of the people-based application domain. However, all software development is application-driven, to some degree or other, and therefore has to take account of contextual influences.

Hard systems are those that are intended to have precise function, either in the solving of clearly articulated scientific problems or through being embedded into larger engineering artefacts like aircraft or nuclear power stations, or as systems software. It is failures of these hard systems that we focused on above but, as the quotations indicate, proponents have been vigorous in defence of their beliefs.

By contrast, soft systems approaches ([36], for example) are concerned with the social situation of the system and have particularly been associated with applications where there has been a strong commercial imperative. The functions of the system are typically determined by some surrounding human system and that leads to greater uncertainty. They are often called 'information systems', emphasizing the human view of the data stored in the system. Problems with information systems may be seen in terms of 'resistance', rather than failure—as argued by Symons [37]. For various reasons a system may not be acceptable to an organization and its members. Symons advocates an 'interactionist approach' in which both technical and human aspects are taken into account, reporting that this approach 'assumes no common purpose in IS development, recognising that participants have different motives and interests' [37, p. 186]. Various other paradigms of software development have been suggested ([38], for example) all of which could be applicable, and which is applicable will depend upon factors such as national and company culture and the nature of the system being developed. Many approaches require the reconciliation of the needs of multiple stakeholders in the system, introducing a form of pluralism into software development.

6.2. Consultative software development

The move towards forms of pluralism in development can be illustrated by what is often regarded as the problem of requirements. This can be stated as follows. To create a software product we need a specific statement of what that product is intended to do. Determining the requirements of the situation, or the requirements of the end users, is regarded as the way to get this specific statement. Where the development community come up with the requirements themselves, there is a strong likelihood that the lack of domain knowledge will mean that the software will fail to address aspects of the application situation, and that the software will be unacceptable to users. However, it is also often the case that users cannot appreciate what can be done with the technology and may have either unrealistic expectations about the proposed system or, alternatively, request something that is technologically uninteresting to produce. Apart from these possible conflicts, there are also the problems of dynamism. Many domains change all the time. The more sophisticated the software intervention—for example, decision support, as opposed to payroll system—the more dynamic the situation is likely to be. It is also the case that introducing a new software-based system into any situation is likely to change it, so that the requirements that were apparent before the introduction of the software may be changed by that introduction. At the same time, however, no tractable software project can proceed without some definite starting statement of intent. And, in the interests of producing a software artefact within a specified time frame, requirements cannot be continually changed. Software development is constituted as a goal-directed activity and as such cannot deal with too much movement of the goal posts.

One major problem with the modernist approaches in the development of software was the belief that requirements could be fully specified, up front. This can be seen in the strictly sequential phased development of structured methods, such as SSADM [39]. This accords with beliefs that there is *a* rational solution, expressible in an unambiguous requirements document, that transcends developer/user conflicts or domain dynamism. It is based on an older model of science (pre-'New Physics', for example), which holds that there exists a unique reality which is there for us to discover and this discovery could be made by anybody working anywhere. Lyotard

parodies this as 'mephistophilean functionalism of sciences and technologies' and claims that '. . . there is no reality unless testified by a census between partners over a certain knowledge and certain commitments' [13]. There is now strong evidence to support the importance of social processes in the construction of scientific theory (for example, [40]) and a retreat from the idea of a unique reality waiting to be discovered. These movements in science and technology studies leave us with a spectrum of ontological and epistemological positions. These range from the single, discoverable, knowable reality of early Enlightenment science and early Modernist computing, to the mild relativism that states there are many experienced realities, but only one (at any time) which works for specific purposes ([41], for example), to the extreme relativism that states that there are no knowable realities, and that we are always dealing with illusion and surface appearance (see [42], for example).

As modernists, we have to choose a position towards the 'single reality' end of the spectrum, and argue for its truth above all others; as postmodernists we can play with all of them. The hard systems approach tends more towards the 'single reality' end of the spectrum, but there has been increasing recognition of the problems with this. One line of response in computing has been a pursuit of pluralism, to 'triangulate' the requirements by determining a number of perspectives or viewpoints of the requirements (for example, [43]). In the modernist version of this, the various viewpoints would be reconciled to produce a single consistent statement of requirements (for example, Mullery [44]), but others, like Nuseibeh *et al.* [43], would not necessarily require this.

Another line of response is to recognize that the requirements are implicit in the world and the way to proceed is to allow these to emerge as the development of the system proceeds. Brooks in his 'No Silver Bullet' paper [3] attributed the problem primarily to 'complexity' and, while he argues that no single cure exists, is keen on the use of 'great designers'. Curtis *et al.* [45] come up with a similar idea, pointing out how important the lead designer (software architect) is, as well as the problem of communication implicit in Lehman's uncertainty. One response to the problems has been to build prototypes, an approach discussed by Brooks [3]. But the use of prototypes is subtle.

Following a prototyping approach (for example, [46]) is one way of allowing requirements to emerge. The general idea is to build a prototype focusing on part of the perceived need and use this to focus discussions with the intended beneficiary(ies). This leads to changes and a new prototype, and so the prototype may develop through many cycles into the system that is eventually put into service. In practice the use of prototypes may be formalized and adopted by modernist methodologies, perhaps given confident names like 'Rapid Application Development', effortlessly transmuted and reified by the acronym 'RAD'.

However, whilst each of these solutions has moved away from the 'single reality' end of the spectrum, they still carry a specific part of the modernist heritage. This can be seen in the determination of what constitutes relevant knowledge. To explore this we need to ask where the locus of power is situated—who determines which of the emergent requirements are the important ones, for example?

The notion that there is a single right answer that anyone (correctly trained) will be able to discover has been replaced by the notion that particular people, deploying particular 'expert' knowledge, will be able to determine the most appropriate answer (putting our faith in 'great designers', for example). Even the apparently pluralist positions have to be examined closely in practice to determine what respective weighting is given to which views. There may be ways in which the human user, although considered important in the approach, can become submerged within a modernist hijack. For example, prototyping is important in managing the expectations of both developers and users, but can also become a technical lever which helps to 'configure the user' [47]. Furthermore, there is the danger of 'tokenism'.

6.3. Consultative development and the perils of configuring the user

One of the more visible fields in which the role and influence of modernism can be seen is in architecture. However, there is a strong postmodern movement within architecture (see [48], for example). One way in which this movement has manifested in architecture has been consultation (for example, [49]). The concomitant of this in computing is called participation.

In 'participative development' user representatives participate in the software development process, typically as members of the development team. This arose out of socio-technical approaches to computing, championed by Mumford [50]. Clement and Van den Besselaar [51] survey these, arguing that:

> The focus of participatory design (PD) is not only the improvement of the information system, but also the empowerment of workers so they can codetermine the development of the information system and of their workplace.

They conclude:

> The experiences from the projects reported here offer some encouragement and guidance for further development of PD. The basic tenets of PD are seen to work in a variety of settings. Researchers report that users have become better informed about the nature of information technology and more self-confident in taking initiative with it. Several of the computer systems that have resulted appear to function well from the user's perspective and are still in operation. Systems development approaches specifically suited to supporting PD activities are also gaining acceptance. However, PD is still characterized by isolated projects with few signs

> that it leads to self-sustaining processes within work settings. While in part this reflects short-term project aims, the reasons for this appear mainly to do with organizational inertia and resistance. The main challenge now for PD is to deal effectively with the political and ideological aspects of the broader organizational contexts on which PD initiatives depend for their long-term survival. The dilemma remains that without organizational reform in the direction of greater democratisation at all levels, the knowledge and commitment that PD can stimulate in users will ultimately reinforce patterns that limit the growth of their capabilities and thus undermine further initiative. The projects evaluated here suggest that an increased and positive role for management PD would be useful. A careful involvement with management, without abandoning the desirability of an independent perspective, could open up important possibilities for PD. Only by giving participation the meaning of full engagement in vital organizational affairs is the process likely to flourish.

As identified by Clement and Van den Besselaar, the main factor preventing the self-sustaining uptake of PD is the 'inertia' of organizational culture. If there are political inequities in an organization then PD will not work, it will become a form of tokenism and be recognized as such, losing credibility. There are many subtle ways in which some participants may not be able to express their views and therefore not participate fully [21]. To expect equality of input, and, more particularly, equality of representation in the final product, would be naive. But what is important is that the inequities that do exist are not structural. By this we mean that the suggestions or votes that carry the day, or the views represented in the final product, do not always come from the same subgroup within the PD team, for example, managers or designers. If the opportunity to contest any suggestion is not available to all participants then PD can be said to be tokenistic. If the locus of power lies primarily with the development, then participation can be said to have been bestowed. This, we submit, is oxymoronic. Such a situation assumes a *given* value differential between the technical expertise of developers and the domain expertise of users. This assumption, in itself, can be seen as modernist. It is associated with the modernist concern to delineate a progressive elite, with all the hierarchical power structures, claims to dominant knowledge and value differentiated expertise that this implies.

One major result of this is the phenomenon we can call 'configuring the users'. In Woolgar's original formulation [47] this phrase indicates both the search for the 'right' user—the ideal user implicit in the minds of the developers—but also the subtle working upon the users, whereby the user comes to believe that the system being delivered by the developers is the system s/he really wants. The result of this work is a satisfied user—'the configured user is a happy user'. Whilst acknowledging that decisions have to be made, expectations managed and requirements settled, and also acknowledging that many developers would see the advantage of the 'configured user', there is a major problem. This is the problem that always results when one group is structurally validated over another and it is the problem of responsibility. In the modernist world of identifiable experts, dominant knowledge and hierarchical power structures, the responsibility always lies with those who have power. The rest are infantilized. However, developers often (and increasingly?) feel that they are being cast as the 'bad guys', duping innocent users into accepting systems they do not want. The only way to avoid this binary division is for users to accept some responsibility for the choices that are made. However, responsibility cannot be assumed where there is no power. In order to inculcate mutual responsibility, power needs to be shared.

6.4. Trust and risk

The paper by Lehman [20], quoted extensively above, comes from the hard systems camp. After a highly abstract analysis he concludes 'The outcome of software system operation in the real world is inherently uncertain with the precise area of uncertainty also unknowable.' [20, p. 12]. The rest of his paper is spent arguing even more vigorously for modernist software engineering methods, though his conclusions are cautious:

> . . . uncertainty in the detailed properties of software and its behaviour when executing and, therefore, in computer application is inescapable. This fact is a challenge to society in general; . . . Uncertainty will always be there. It is the responsibility of prime movers to ensure that society is not unnecessarily exposed through thoughtless application. To avoid this, they and society at large must be informed of the threat. . . . Rigorous enforcement of advanced software technology, systematic application of disciplined methods and mechanization can make a fundamental contribution to this end. Their widespread, if not universal, adoption must be accepted as an urgent societal priority [20, pp. 24–25].

As we saw above, Lehman, Brooks and others end by putting their trust in the dissemination of information and in the action of prime movers. In other words, in communication and in people. However, as we also saw in the previous section, if unchallenged, the influence of the hierarchical structure in modernism entails a reliance upon an elite, developing and disseminating information, whilst responsibility is implicitly placed in the hands of the 'great designers'. But if risk is to be fully appreciated, by society at large, and responsibility to be shared between developers and users, is this hierarchical model useful? Can responsibility be taken without an understanding of risk or an acceptance of uncertainty?

For high integrity and safety critical system we must be very confident that we can trust ourselves and our society to the proposed system. To be able to do this it would seem desirable that we can guarantee that the system will work perfectly or, at least, that we give a figure for the probability of error. To be able to achieve this, it could be argued that technology must not follow science and the arts into the postmodern and that technology should choose to use those parts of science for which a rational modernist agenda is appropriate. This may mean choosing technologies for which all the appropriate theories have been worked out—as is seen in recent moves to only build safety critical systems using 'safe' subsets of languages like Ada, for example.

But we have seen that this is problematic. We need to consider other options. Of course computing systems are not the only place where complex decisions are required—the whole ecological movement, for example, is concerned with this. Giddens [52] has examined this in depth, seeing many of the dilemmas of modernism as arising from the growth in complexity, where the complexity arises not from the uncertainties in the external physical world, but from the world of people and their science and technology.

The UK Health and Safety Executive works to a principle of As Low As Reasonably Possible (ALARP) for risks, accepting that not all risks can be measured quantitatively or with certainty—the important thing is that the risk is judged to be low and that processes are in place that are driving down the risk [53]. However, this is entirely dependent upon our being able to ascertain what the risks are in the first place. Leveson's [28] assertion that the failures of software-based systems arise from a multiplicity of co-factors indicates that modernist analyses, which look for single and universalizable causes (or individual saviours) are not helpful. The full complexity of the system and its situation need to be taken into account and to do this we need multiple voices and multiple approaches.

This foregrounds the importance of people in the development process, but not just as a bridge into user requirements as envisaged by the participatory development agenda. The hopes placed in the lead designer and software architect by Brooks [3] and Curtis *et al.* [45] also depend too much on the single hero and the single voice. Playing more than lip service to pluralism in development means encouraging teams that flourish across the divisions of manager, user and developer.

Is eradicating risk modern or postmodern? Would it be more postmodern to live with risk—to accept uncertainty? This would alleviate the problems cited in Leveson [28] about the dangers of dumbing down the user (for example, [28, pp. 22–23]) or the complacency engendered by having a 'good system' (the case of Chernobyl [28, p. 647]). Does minimizing risk lead us to act more rashly? Is this what leads to the kind of user complacency about system failure that was a tragic feature of the Therac-25 cases? The eradication of risk and uncertainty—or the belief that this should be possible—are, we argue, modernist notions, that elide a belief in the exploitation of 'nature's regularities' (see [41]) with a belief in dominion and control.

7. SOFTWARE DEVELOPMENT AND POSTMODERNISM—WHAT IS THE RELATIONSHIP?

We are now past the point (late 1980s and early 1990s) where postmodernism may have been regarded as an obligatory passage point. However, this passing means that many of the notions of postmodernism are now part of our culture. In this sense we may argue that software development needs to take account of ideas which are already influencing work in the field. This is an argument about making explicit those influences which may currently be implicit. The other argument for making such notions conscious is that we can then explicitly ask about the usefulness that postmodernism might hold for software development.

Postmodernism is a descriptive theory, enabling us to comment on what is observed and to relate observations from disparate parts of human activity. It does not claim to have the predictive power to tell us what will happen or the prescriptive power to tell us what to do. Nevertheless, we can look at the social trends that postmodernism highlights and comment on those things that have happened in the development of software as instances of the same phenomena.

We have seen that the development methods of software have been grounded in modernism and the Enlightenment, but that this has led to conflicts and contradictions, and a notion of 'software crisis' as the modernist metanarrative has broken down. These problems are by no means limited to software development and have been repeated across high technology and architecture, for example. We have also seen a number of 'standard' responses to the crisis of modernity and how these also appear in software development. We have considered, for example, arguments for greater rigour and control, but also the acceptance that much of this involves placing our trust in 'great designers'. We have looked at Leveson's argument that it is a mistake to attribute software failures to single causes. We have also considered the democratization of development, with such approaches as participative development. Each of these cases manifests a different relationship to postmodern thought (although none of the authors cited have, or maybe would, make this connection with their work). For example, placing our trust in the communication of information and the competencies of designers to mobilize the rigour required to ensure safe systems may be seen as an opening out of the traditional formalist positions, which never addressed contextual concerns. However, advocating that trust be placed in the competencies of 'great' individuals negates the notion of a shared responsibility, supports the concentration, rather than the distribution of power, and sets up a 'fall-guy' to blame if things go wrong, thus allowing the power structure to remain unchallenged. Arguing against the 'single cause' explanation of software failure, however, does challenge such power structures and considers the broad web of responsibility that even software intended as the product of a single great designer may have. It insists that all software products are more properly considered 'systems'

in order to make this clear. Participative development falls somewhere between these two positions. It is an approach that can only replicate the power distributions already present in an organization, and so its ability to employ postmodern strategies is dependent upon what values and beliefs have already colonized the context. Choosers and users of software and systems are still regarded as very separate from the developers in the participative design approach. As we have seen above, lip service can be paid to participative design in ways that make it appear that views are being canvassed when, in fact, resistance is being annexed. In an earlier paper [21] we identified the ways in which various stakeholders in a participative design situation had been co-opted or otherwise invalidated from making any counter-suggestions to those intended by the project manager. The user representatives, for example, had both been saved from redundancy by being offered jobs using the new system once it was up and running. Given the primacy attributed to technical knowledge, users can be intimidated about making their views known and developers can use this situation to convince users that they really want what the developer wants to supply them with—thus 'configuring' them. The reason that this can be the case is that there is still an implicit value differential between the views of the choosers and users and those of the developers. Until that divide can be undone users will always be vulnerable to the possibility of being configured and until developers realize that the reconstruction of the social space that occurs when a system is introduced or changed is as much our responsibility as the technical success, then we will only be doing part of our job. In any development situation we need to identify stakeholders in such a way that the given or assumed variations in importance between them are not accepted at face value. We also need to challenge the assumptions that govern notions of relevant and irrelevant knowledge so that users, choosers and developers pool their resources rather than competing for dominance.

A major aspect of postmodern thinking is the challenging of privileged (meta)narratives and privileged (meta)narrators. Privileged narratives are those stories within our social world which are deemed more important than any others. In software development there has traditionally been a privileging of technical information and related values over social information. In deconstructing the assumptions we bring to our work we need to question the very notion of a technical/social divide. A major problem with such divides is that different constituencies become associated with each side and often identify themselves by associating positive characteristics with their own side, whilst attributing negative characteristics to the other. For example, users may regard developers as overpaid 'cowboys' who do not appreciate the problems with users' work and offer technical solutions that are not helpful. Developers may regard users as technological naïfs who cannot understand what technology can do for them, but also do not really know what they want. In each case both groups are validating their own expert knowledge at the expense of other expert knowledge, rather than attempting to understand the inputs of both and exploit the benefits of different perspectives.

Once the assumptions of value differentials have been questioned, the automatic *droit de seigneur* of the dominant narrative no longer works. This allows the voices of the disenfranchized to be heard and it allows things to be said that were previously censored as irrelevant. Above all, it fosters communication.

However, for there to be no dominant narrative makes life all the more complex. Everything has to be locally negotiated and predictive ability may be constrained. This automatically makes the deployers of previously dominant knowledge less powerful than they once were and it is very easy for another group, and dominant narrative, to slip into the vacuum as a substitute. This often gives us the same problem in reverse, such as an anti-technology backlash. Under postmodernism the vacuum would not be seen as such, rather the space created would provide room for the previously unheard to be heard. For software developers who thrive on certainty, predictability and control, the concept of variable values, locally negotiated, may seem particularly challenging. How does anything get done? How do we know if it's right? If there are no generic laws laid down will we not be in danger of re-inventing the wheel, creating local systems that cannot communicate beyond their local network or creating limited systems, baroque in their narrow application domains, but not reusable?

However, we have already fallen prey to each of these possibilities: many projects fail to get enough done to be successful; the complexity of current systems ensures that total formal predictive proof that the system is correct is impossible; squabbles over the best way of doing things (languages, tools, methods, etc.) and the competitiveness encouraged in our society facilitates both the re-invention of the wheel and the creation of systems that do not talk to each other. As for limited systems, whilst some application domains are similar and respond well to generic packages, very many are locally varied. To be really useful many off-the-shelf packages have to be customized. Software development is part of a richly varied social world and as such, if it is to respond to that world, we need to accept locally produced variety in software.

Harding [54] uses the term local knowledge systems (LKS) to describe locally produced sets of beliefs, practices and resultant artefacts. For her work this facilitates a symmetry of analysis in studying Western and non-Western science. We feel that it is a useful concept to adopt in software development such that we can identify, for example, users and developers as having two distinct LKS that need to be combined to produce one artefact. Mobilizing this concept to inculcate symmetry enables us to break the positive/negative attributions of divides such as the social/technical.

Another, more explicitly postmodern, strategy we can use is to recognize our own hybridity. This enables us to avoid identifying too strongly with any dominant narrative or group. Multiple narratives will not remain stable, but will change and cross-fertilize. This phenomenon has been noted

in studies of postcolonialism (for example, [55]). This is a very positive outcome of conflicting narratives, Ashcroft *et al.* noting 'hybridity and the power it releases' [55, p. 207]. Bhabha [56] writes of hybridity as a 'third space' where 'we may elude the politics of polarity and emerge as the others of ourselves'. To allow other narratives, to listen to other voices, to distribute power, creates space both for innovation and for responsive technology. It also allows a shared responsibility that can exploit the benefits of difference, instead of requiring that responsibility be held in specific, standardized ways, by specific standardized individuals.

The crucial issue here is one of communication. Good communication and collaboration can only occur when all inputs are deemed equally valid, when all perspectives on a situation are encouraged and this, it is argued here, is only possible when assumed values are deconstructed, multiple narratives are heard and development becomes a multi-partnered enterprise where the partners are open to the hybridization of themselves. The deconstruction undermines the rigidity of thinking that currently characterizes software development, seeing vitality, rather than chaos, in doubt. It encourages a greater fluidity of thinking through claiming that rules, values and perspectives are all negotiable and that which is currently regarded as irrelevant may in fact be crucial. The energy of doubt and the fluidity of negotiation also create space for innovation. The views expressed here are that the postmodern ethos can offer some emancipation to the process of software development. In questioning the accepted rules and values and, crucially, not offering any others in their place, software development has to become a more locally negotiated phenomenon. The rules you follow are the ones that are suggested by the situation at hand. This requires a holistic picture of local constraints and resources, enabling a more responsive technology to be developed.

Conceiving such a future implies a change to the way in which software developers are educated and to current conceptions of what might constitute a software engineering curriculum. We modestly suggest a place in that education and curriculum for a consideration of the epistemological and socio-historic ideas that we have sketched above, for—as we have argued—they are the very stuff of software development.

REFERENCES

[1] Nauer, P. and Randell, B. (eds) (1969) *Software Engineering—Report on a Conference Sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, Brussels.

[2] Brooks, F. P. (1975) *The Mythical Man-Month*. Addison-Wesley, Reading, MA.

[3] Brooks, F. P. (1986) No silver bullet—essence and accidents of software engineering. In Kugler, H.-J. (ed.), *Information Processing 86*, pp. 1069–1076, North-Holland, Oxford.

[4] Cox, B. J. (1990) There is a silver bullet. *Byte*, **15**, 209–218.

[5] Gibbs, W. W. (1994) Software's chronic crisis. *Scientific American*, **271**, September, 72–81.

[6] Glass, R. L. (1994) The software-research crisis. *IEEE Software*, **11**, November, 42–47.

[7] Jencks, C. (1992) The post-modern agenda. In Jencks, C. (ed.), *The Post-Modern Reader*. Academy Editions, London.

[8] Habermas, J. (1992) Modernity: an unfinished project. In Jencks, C. (ed.), *The Post-Modern Reader*. Academy Editions, London.

[9] Galileo, G. (1632) *Dialogue Concerning the Two Chief World Systems—Ptolemaic & Copernican*. University of California Press, Berkeley and Los Angeles, 1967.

[10] Hamilton, P. (1992) The enlightenment and the birth of social science. In Hall, S. and Gieben, B. (eds), *Formations of Modernity*. Polity Press, Cambridge.

[11] Smart, B. (1990) Modernity, postmodernity and the present. In B. S. Turner (ed.), *Theories of Modernity and Postmodernity*. Sage, London.

[12] Wittgenstein, L. (1961) *Tractatus Logico-Philosophicus*. Routledge and Kegan Paul, London.

[13] Lyotard, J.-F. (1984) *The Postmodern Condition: a Report on Knowledge*. University of Minneapolis Press, Minneapolis.

[14] Murray, R. (1992) Fordism and post-Fordism. In Jencks, C. (ed.), *The Post-Modern Reader*. Academy Editions, London.

[15] Hoare, C. A. R. (1984) Programming: sorcery or science? *IEEE Software*, **1**, April, 5–16.

[16] Robinson, H. M. (1988) Reading formalities: the influence of logical atomism. *Proc. ISTIP '88*, 28–30 March, Sunningdale.

[17] Maddison, R. N. and Stanczyk, S. K. (1988) Time in information systems and its impact on modelling processes and data. *Inform. Software Technol.*, **30**, 12–22.

[18] Vermeir, D. (1983) Semantic hierarchies and abstractions in conceptual schema. *Inform. Syst.*, **8**, 117–124.

[19] Steel, T. B. (1985) A minimal conceptual schema for life, the universe and everything. In Steel, T. B. and Meersman, R. (eds), *Proc. IFIP WG 2.6 Working Conf. on Database Semantics*, 7–11 January, Hasselt, Belgium. North-Holland.

[20] Lehman, M. M. (1989) Uncertainty in computer applications and its control through the engineering of software. *Software Maintenance: Research and Practice*, **1**, 3–27.

[21] Low, J., Johnson, J., Hall, P., Hovenden, F., Rachel, J., Robinson, H. and Woolgar, S. (1996) Read this and change the way you feel about software engineering. *Inform. Software Technol.*, **38**, 77–87.

[22] Peltu, M. (1995) Safety in numbers. *Computing*, 12 January, p. 34.

[23] MacKenzie, D. (1993) Negotiating arithmetic, constructing proof: the sociology of mathematics and information technology. *Social Studies Sci.*, **23**, 37–65.

[24] DeMillo, R., Lipton, R. and Perlis, A. (1979) Social processes and proofs of theorems and programs. *Commun. ACM*, **22**, 271–280.

[25] Collins, H. M. and Pinch, T. J. (1993) *The Golem: What Everyone Should Know About Science*. Cambridge University Press, Cambridge.

[26] Fetzer, J. H. (1988) Program verification: the very idea. *Commun. ACM*, **31**, 1048–1063.

[27] Leith, P. (1986) Fundamental errors in legal logic programming. *Comput. J.*, **29**, 545–552.

[28] Leveson, N. G. (1995) *Safeware, System Safety and Computers*. Addison-Wesley, Reading, MA.

[29] Leveson, N. G. and Turner, C. S. (1993) An investigation of the Therac-25 accidents. *IEEE Comput.*, **26**, 18–41.

[30] Huyssen, A. (1992) Mapping the post-modern. In Jencks, C. (ed.), *The Post-Modern Reader*. Academy Editions, London.

[31] Hassan, I. (1992) Pluralism in postmodern perspective. In Jencks, C. (ed.), *The Post-Modern Reader*. Academy Editions, London.

[32] Jameson, F. (1983) Postmodernism and consumer society. In Foster, H. (ed.), *Postmodern Culture*. Pluto Press, London.

[33] Bauman, Z. (1995) Savoir: the cul-de-sac of pouvoir. In *The Times Higher Education Supplement*, 12 May.

[34] Floyd, C., Züllinghoven, H., Budde, R. and Keil-Slawik, R. (1992) *Software Development and Reality Construction*. Springer-Verlag, Berlin.

[35] Bjerknes, G., Ehn, P. and Kyng, M. (1987) *Computers and Democracy—a Scandinavian Challenge*. Avebury, Aldershot.

[36] Checkland, P. (1981) *Systems Thinking, Systems Practice*. Wiley, Chichester.

[37] Symons, V. J. (1991) Impacts of information systems: four perspectives. *Inform. Software Technol.*, **33**, 181–190.

[38] Hirschheim, R. and Klein, H. (1989) Four paradigms of information systems development. *Commun. ACM*, **32**, 1199–1216.

[39] Ashworth, C. and Goodland, M. (1990) *SSADM: A Practical Approach*. McGraw-Hill, London.

[40] Latour, B. and Woolgar, S. (1986) *Laboratory Life: The Construction of Scientific Facts* (2nd edn). Princeton University Press, Princeton, NJ.

[41] Harding, S. (1991) *Whose Science? Whose Knowledge?* Open University Press, Milton Keynes.

[42] Baudrillard, J. (1995) *The Gulf War Did Not Take Place*. Indiana University Press, Bloomington, IN.

[43] Nuseibeh, B., Finkelstein, A. and Kramer, J. (1996) Method engineering for multi-perspective software development. *Inform. Software Technol.*, **38**, 267–274.

[44] Mullery, G. (1979) CORE—a method for controlled requirements specification. *Proc. 4th. Int. Conf. on Software Engineering*, Munich, pp. 126–135. IEE.

[45] Curtis, B., Krasner, H., Shen, V. and Iscoe, N. (1987) On building software process models under the lamppost. *Proc. 9th Int. Conf. on Software Engineering*, March, Monterey, CA, pp. 96–103. IEEE Computer Society.

[46] Sommerville, I. (1992) *Software Engineering* (4th edn). Addison-Wesley, Wokingham.

[47] Woolgar, S. (1991) Configuring the user: the case of usability trials. In Law, J. (ed.), *A Sociology of Monsters: Essays on Power, Technology and Domination*. Routledge, London.

[48] Jencks, C. (1993) *Architecture Today*. Academy Editions, London.

[49] Collins, J. (1992) Post-modernism as culmination. In Jencks, C. (ed.), *The Post-Modern Reader*. Academy Editions, London.

[50] Mumford, E. (1983) *Designing Human Systems For New Technology: the ETHICS Method*. Manchester Business School Press, Manchester.

[51] Clement, A. and Van den Besselaar, P. (1993) A retrospective look at PD projects. *Commun. ACM*, **36**, 6, 29–37.

[52] Giddens, A. (1990) *The Consequences of Modernity*. Polity Press, Cambridge.

[53] Rimington, J. D. (1993) Coping with technological risk: a 21st century problem. *The CSE Lecture*, The Royal Academy of Engineering, 22 November.

[54] Harding, S. (1998) *Is Science Multicultural: Postcolonialisms, Feminisms, and Epistemologies (Race, Gender, and Science)*. Indiana University Press, Bloomington, IN.

[55] Ashcroft, W., Griffiths, G. and Tiffin, H. (eds) (1995) *The Post-colonial Studies Reader*. Routledge, London.

[56] Bhabha, H. (1988) The commitment to theory. *New Formations*, **5**, 5–23; reprinted in [55] as Chapter 35, Cultural diversity and cultural differences, pp. 206–209.