‘It’s Engineering Jim … but not as we know it’

Software Engineering - solution to the software crisis, or part of the problem?

Antony Bryant
Professor of Informatics
Leeds Metropolitan University
The Grange, Beckett Park
Leeds LS6 3QS; UK
+44 113 283 7422
a.bryant@lmu.ac.uk

ABSTRACT
This paper considers the impact and role of the ‘engineering’ metaphor, and argues that it is time to reconsider its impact on software development practice.

Keywords
Engineering metaphor, software development practice

1 INTRODUCTION

> … *a new subdiscipline, software engineering, has arisen. The development of a large piece of software is perceived as an engineering task, to be approached with the same care as the construction of a skyscraper, for example, and with the same attention to cost, reliability, and maintainability of the final product. … Even with such an engineering discipline in place, the software-development process is expensive and time-consuming.* (An extract from the entry on *software engineering*, a subsection of the entry on Computer Science: Software, in the CDROM version of the Encyclopaedia Britannica)

As we move into the fourth decade of ‘software engineering’ - taking 1968 as the benchmark - we are faced with the pervasive dilemma that no-one seems satisfied with the meaning and understanding of the term itself. This is not merely a matter of semantics, but has organizational and financial implications. It can affect the allocation of research funding, and the resourcing of teaching and equipment. The community of software engineers may be happy with the term, but they must be aware that those outside this enclave fail to understand its meaning and ramifications: And I suspect that most outsiders are either unaware that software engineering exists, or are puzzled by the term itself.

ICSE 2000, Limerick, Ireland

This ambiguity has ramifications in establishing the relationship of the ‘sub-discipline’ to computer science, information systems and other areas of study; and therefore affects issues such as training, curriculum, career development, funding, professionalism (particularly in terms of visibility, certification and recognition), and practice, in addition to any specific knowledge claims as a distinctive discipline.

The idea that software has to be ‘engineered’ evokes an image of rigour, care and assurance. In the 1980s professional qualifications and university courses emerged with ‘software engineering’ as the key component of their title and content; and such courses seemed to offer a more pragmatic understanding of computer technology than did the more traditional computer science courses. In practice, however, it soon became clear that the situation was not quite as simple. The term was often misunderstood, and some university courses found it difficult to recruit to such courses as many potential applicants were misled by the term itself. Although several standard student texts appeared and have been continually updated and revised, there is still an air of unease around the term and its associated terminology. In the late 1990s there is perhaps a consensus around the term itself within the academic and research community, but as a specific course title its popularity and use is declining.[1]

Discussion threads on the electronic list from the (UK-based) *Conference of Professors and Heads of Computing* (CPHC) illustrate this all too cogently. For various reasons the CPHC have had to produce details of what is involved in research into and the study of computing, informatics, software engineering and so on. In most instances an uneasy consensus has been reached, but all too often with

[1] These observations are derived almost entirely from experience in the UK, and so may seem strange in other countries where software engineering may well be a thriving subject area. The purpose of international gatherings such as ICSE is in part to ensure that our limited areas of experience are remedied by interchange with others from different locations.

the nagging suspicion that this agreement is only understood by the senior academics concerned, and will be misinterpreted by those outside this restricted domain: In particular by the wider engineering community, funding organizations, practitioners, potential students, and commercial developers.[2]

Whatever the basis for understanding the term *software engineering* itself, software developers continue to be faced with the dilemma that they seem to wish to mimic engineers, and lay a claim for the status of an engineering discipline; but commercial demands and consumer desires do not support this. In what follows I shall argue that in part this is because the term itself carries with it a set of mixed cognitive implications that contribute to the intellectual quandary surrounding software development.

2 SOFTWARE ENGINEERING DEFINED & CRITIQUED

'Standard texts' such as those from the USA and UK by McDermid, Pressman and Sommerville [16, 19, 22] embody the consensus view of software engineering. Sommerville defines software engineering as being 'concerned with theories, methods and tools which are needed to develop software for ... computers'. Software engineering differs from other forms of engineering since it 'is not constrained by materials governed by physical laws or by manufacturing processes' (p4).

Pressman quotes Fritz Bauer from the 1968 NATO conference as follows - software engineering is the 'establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines' (quoted p23). Pressman stresses the need for 'engineering discipline in software development'. McDermid points out that '[I]f software engineering is to become a true discipline it too must have an appropriate foundation in science and mathematics' (p1).

So we have a common view that software development needs a rigorous basis, and this is to be found in engineering, and indirectly in science and mathematics. I suspect that few readers will find anything exceptional in such statements. We are all too well aware that software development often proceeds with little or no constraint, and so the application of an immediately appealing, and intuitively obvious archetype or exemplar needs little or no justification. But what do all these writers mean when they talk of 'engineering'? Are they suggesting that software development can benefit from simply mimicking engineering? McDermid stresses the need for a foundation in science and mathematics. Pressman, quoting Bauer, talks simply of 'engineering principles', which might incorporate theoretical principles - and so include science and mathematics - but which might also hint at principles related to engineering *practise*. Sommerville includes 'theories, methods and tools' which seems to cover both the theoretical and practical aspects; but he also makes the point that software engineering is distinct from other forms of engineering because of the nature of software itself.

The basic motivation for use of the term 'software engineering' remains the same as it was in the 1960s, but the scope and focus have certainly developed in the last 30 years. These developments have been summarized by Wasserman [23] who points to seven key changes that have impacted on software engineering practice.

- Criticality in the time-to-market for commercial products
- Changing economics of computing - lower hardware costs, greater development costs
- Emergence of powerful desk top computing
- Extensive local and wide area networks - especially the web as publishing and distribution mechanism
- OO Technology - growing in availability and acceptance
- WIMPS and GUIs
- Unpredictability of waterfall model of software development (*inapplicability* seems a more accurate term for Wasserman's argument)

Wasserman argues that any one of these developments would have had a significant impact on the software development process; but taken together they have resulted in a major transformational force. Software engineering in the 1990s and beyond cannot be an extension of what it was in the 1970s. Current software development has to make a quantum leap from its origins. In so doing, software engineering theory and practice have to focus around the fundamental principles of an 'effective discipline'. Wasserman offers a list of eight fundamental facets of software engineering 'that form the basis for an effective discipline'. These are as follows:-

- Abstraction - a key aspect for focusing on relevant detail - 'fundamental technique for understanding and solving problems'
- Analysis & Design Methods and Notations - for communication, a basis for checking and reuse - 'basic tools for communication in an engineering discipline'
- User Interface Prototyping - for requirements determination, assurance, feasibility - the 'most effective way to elicit user requirements'
- Modularity and Software Architecture - reflecting principles of 'good design', including decomposition, different perspectives - this plays a 'major role in determining system quality and maintainability'

[2] As this paper was being prepared the CPHC discussion list was developing a thread around the topic of the *scope of software engineering* - a topic that has been evoked by a recent benchmarking exercise with ramifications for funding and assessment.

- Lifecycle and Process - with a matching of process to different goals and contexts - 'having some defined and manageable process for software development is much better than not having one at all'
- Reuse - including more than just code; and covering aspects such as responsibility for any reused component - 'an essential part of any software development process'
- Metrics - improvements … cannot be calculated without an effective metrics effort'
- Tools and Integrated Environments - 'should provide comprehensive and integrated support for the development process' (p24)

Given the contested nature of the term itself, particularly by non-software engineers (ambiguity intended), we might ask in what ways would progress in terms of any or all of these eight result in the firm foundation of a truly engineering-based discipline of software development? On the other hand we might take the views of those such as Parnas [18] who argues for a distinction between engineering and science: With the basis for software engineering being found in computer science, in much the same way as physics provides the disciplinary basis for electrical engineering. Whether we side with Wasserman's more self-contained approach, or Parnas' foundational model, we still need to pose a further series of questions:

- Are we pursuing the right goal when striving for engineering status, however understood?
- Are we trying to attain it in an appropriate manner?
- Assuming that the idea of software engineering was appropriate in the 1960s, to what extent has this become a misplaced assumption?

A glance through key journals, particularly *IEEE Software*, indicates that the philosophical basis of software engineering has remained a live issue since the 1960s. This is not in any way symptomatic of a critical deficiency in software engineering. On the contrary it should be seen as evidence of an emerging and vibrant discipline that has perhaps yet to establish itself fully; and where there is opportunity to influence its nature and scope. What does need to be demonstrated, however, is the existence of certain problematic issues that arise from the (often unquestioned) bases of software *engineering* itself: In particular the ways in which the terminology and imagery in current use may influence our understanding of the domain and its associated practices.

3 ACCIDENTS, ESSENCES AND METAPHORS

We have all been influenced by F P Brooks' classic paper 'No Silver Bullet' [5]; the paradigmatic exemplar of the dilemmas of software engineering. Using the Aristotelian categories of 'accidents' and 'essences', Brooks distinguishes between the characteristics of software that are problematic and ineluctable, and those that are contingent - historical *accidents* - that beset software development. He offers several examples of these accidents, and points out that they can and usually will be overcome. This might be seen to imply that the number and range of such contingent factors will simply reduce over time. Yet, although not making any specific mention of the possibility, Brooks does not rule out that new historical accidents may arise.

The conclusions of Brooks' argument are that we have a range of activities concerned with the production of the most complex of human artefacts, where many of the usual possible conceptual and practical support mechanisms are unavailable. We cannot build scale models, we cannot demonstrate key aspects of the processes involved; and once software is produced, it is unlikely that it can be stabilized. It will constantly be changed to yield to demands for it to be altered to adapt to changing requirements and contexts. No wonder that software developers apply terminology borrowed from elsewhere, or create new terms closely related to those from other contexts. This inevitably results in software development being described and defined in terms of metaphors borrowed from other disciplines. In particular the imagery of engineering is used, and so we have aspects of software development concerned with maintenance, prototyping, construction, specification, design and so on.

Brooks himself expressed some uneasiness with this imagery in later - and perhaps less noticed - sections of his *No Silver Bullet* paper. He noted the impact that the construction metaphor had on him when he first came across it in 1958. 'The metaphor shift was powerful, and accurate. Today … we freely use other elements of the metaphor, such as *specifications*, *assembly of components*, and *scaffolding*.' (p18) Writing in the mid-1980s, however, he argued that, although useful and powerful, this construction metaphor had outlived its usefulness, and needed to be replaced by an image of software development more akin to *growth* than construction. It is almost as if the term 'software engineering' has itself become a newly emergent historical accident.

This feature of Brooks' argument needs careful analysis and understanding. He uses the term 'metaphor *shift'* when noting the impact of the building metaphor; implying that it replaced an existing metaphor. More than that, he takes it as axiomatic that metaphors operate with regard to software development. The particular shift he locates in the 1950s is one from *writing* software to *building* software: And these metaphors have a powerful impact on people's practices and cognition. The terms used are not simply words on a page, they have significant effects: As Brooks asserts when pointing out that perhaps the *construction* metaphor should be replaced by one concerned with *growth and nurture* - 'I have seen dramatic results since I began urging this technique (the growing of software using incremental development) on my project builders in my Software

Engineering Laboratory class.' (p18) He continues in a similar vein, mentioning heightened morale, jumps in enthusiasm, redoubled efforts and so on. He concludes that 'teams can *grow* much more complex entities in four months than they can *build*'.

Brooks seems clear in his mind that metaphors are powerful and crucial for software developers. He underlines the positive impact of the *construction* metaphor in the 1950s, and the development of software engineering is closely related to this. But if Brooks is correct in assessing the impact of the new metaphor around *growth*, does it mean that the older ones are now operating as constraints on software development practices? Is the term software engineering itself now an obstacle? Is it possible that the engineering metaphor has become part of the 'software crisis' rather than a solution for it?

From Brooks we can understand that there will always be a 'software crisis'; the cause of many of the problems lies with the nature of software itself. But that is not to say that other, less intrinsic factors, are not important: And these can be confronted and eradicated if they arise from historical contingencies or *accidents*. Perhaps the term *software engineering* itself may now be one of these constraints, particularly resulting from the ramifications of the *engineering metaphor* from which it is derived.

Software development is a practice justifiably and inevitably founded on metaphor. The term *software* is itself a metaphor[3]. But now perhaps the engineering metaphor has served its purpose, and needs to be superceded or fundamentally revised.

4 METAPHORS

The activities surrounding software and software development are usually defined in terms of the 'engineering metaphor'.[4] The metaphor extends to use of terms such as construction, development, maintenance, prototyping, and so on. This terminology and imagery can be simultaneously intuitively obvious and accompanied by numerous caveats. Thus it is common for software developers to talk about *maintaining* software, but they are usually well aware that this use of the term is peculiar in this context, and only partially resonates with the mainstream engineering use: Similarly terms such as prototyping, specification, construction and so on.

It might be thought that this imagery is now so ingrained in the discipline that it can hardly be controversial. Students, teachers and practitioners all *know* that the language used is not a direct translation from mainstream engineering; so why raise it as an issue? Can the engineering metaphor really have a profound impact on software development practices? Is this a common characteristic of metaphor or is it specific to software development?

To those unfamiliar with recent work on the topic, metaphor might be thought of as a restricted linguistic device, usually (consciously) employed for its dramatic effects. But this is an inadequate view, and severely underestimates the role that metaphor plays in our lives. Fowler's Modern English Usage [10] notes that 'our vocabulary is largely built on metaphors; we use them, though perhaps not consciously, whenever we speak or write'. More recently Lakoff and his colleagues have been instrumental in establishing the field of metaphor as a key site for cognitive and social issues, neatly evoked by the title of Lakoff and Johnson's book *Metaphors we live by*. This work not only underlines the ubiquitous nature of metaphors, but stresses their role and power in our thought processes. In Fowler, they are pervasive, but essentially passive. Recent work now challenges this view, seeing metaphors as playing an *active* role in thought and cognition. In particular, metaphors are now seen as a crucial aspect in the spread and understanding of new ideas and concepts.

Metaphors operate on two distinct subjects; 'primary' and 'secondary', mapping terms from the latter on to the former.[5] For *software engineering*, the primary subject is *software development*, the secondary is *engineering*. Max Black stresses that 'the secondary subject is to be regarded as a system rather than an individual thing' (p27). In other words, the term *engineering* is not used in isolation; but evokes a whole range of concepts and terms involved in the engineering frame of reference. The metaphorical utterance then works by 'projecting upon the primary subject a set of associated implications ... that are predicable of the secondary subject' (p28).

To illustrate this we can consider the metaphorical aspects of the phrase *software development is an engineering discipline*. The relationship between the primary and secondary subjects operates in the following manner (using Black's words drawn from a different example);

- 'the presence of the primary subject incites the hearer to select some of the secondary subject's properties; and that
- invites him (sic!) to construct a parallel implication-complex that can fit the primary subject; and

[3] I am grateful to one of the anonymous reviewers for pointing this out.

[4] For the purposes of this discussion, metaphor will be taken to mean a 'figure of speech in which a name or descriptive term is transferred to some object to which it is not properly applicable' - Shorter OED, 1973.

[5] This overview of metaphor is based largely on Max Black's essay [4]. Many of the other ideas introduced later derive from other contributors to the collection edited by Ortony [17].

- reciprocally induces parallel changes in the secondary subject' (p28)

This view of metaphor depends on *selection* by the hearer/reader of aspects derived from the secondary subject - i.e. our own (collective) understanding of what is involved in *an engineering discipline*. This then leads to changes in our conception of the secondary subject, as well as affecting our understanding of the primary one. If we state the engineering metaphor in terms of 'software development is an engineering discipline', then according to Black we select aspects of engineering to fit software development, and also reinterpret engineering as a result of applying its properties to software development. (This may partly explain why non-software engineers never seem to get to grips with what software engineers mean by engineering.)

The *engineering metaphor* then leads us to understand that engineering is a discipline, with all that is implied in terms of rigour, knowledge claims, expertise and so on: And so software development can be considered in a similar fashion. The metaphor then develops in terms of identity, extension, similarity, analogy, and metaphorical coupling (linking a series of metaphors).

The engineering metaphor in the context of software development implies a model of software development practice, and a whole host of associated concepts. This model might take the form of seeing software development *actually* as an engineering discipline; as *like* an engineering discipline; or as *bearing some similarity to* an engineering discipline. Disagreements about the definition of the term *software engineering* itself reflect these sorts of distinction, although some originate in different ideas about the nature of *engineering*. The key point, however, is that the resulting models impact upon the software engineering community's conceptualization of both the domains of software development and engineering. If Black's argument is correct, software engineers will understand engineering differently in the light of the use of the metaphor in the realm of software development. (This would make an interesting research project, and could now be extended to other domains from which we have developed metaphors - e.g. architecture.)

It should be stressed that there is no single, 'correct' model of software development that emanates from the engineering metaphor, although with time a consensus may emerge. At present there are a range of interpretations, and these different interpretations evoke a variety of models of what constitutes software development. We need to recognize the nature and characteristics of these models; uncovering their ramifications, similarities and distinctions to ensure that they are appropriate and relevant. Aspects which are widely accepted may be just as interesting as objects for analysis as those where there is contention: The former may be seen as indicative of the general consensus (e.g. software engineering, process models); the latter as areas of uncertainty and dispute (e.g. artificial intelligence, formal methods).

We have to understand that the use of generally accepted terms is not accidental. The terminology surrounding the engineering metaphor plays a *cognitive* and *constitutive* role for software development.[6] As representatives of the community of software developers, we need to clarify the impact of this metaphor on our practice, and ascertain if alternative or additional metaphors might be appropriate to complement or remedy some of the deficiencies of the engineering one. We might also wonder why other metaphors did not gain the currency of the engineering one – a point to which we shall return in the concluding section.

5 METAPHORS AND COGNITION

Brooks' argument about a *metaphor shift* implies that the construction metaphor (a specialized case of the engineering one) did not simply emerge in a context where none had previously existed. There was no *literal* understanding of software development prior to the emergence of the construction metaphor; the preceding dominant metaphor was concerned with *writing* software. There was never a mythic golden age when the term was understood directly and without the benefit of this linguistic trope in some form or other.

This puts Brooks together with those who argue that all cognition employs metaphor. This is an argument that goes well beyond the scope of this paper, but it certainly seems to be the case that metaphors are indispensable in contexts of novelty and innovation. Black maintains that 'strong metaphors' act as indispensable bases for generating genuine insights about reality. Indeed he contends 'that some metaphors enable us to see aspects of reality that the metaphor's production helps to constitute'.

It is hardly surprising that metaphors have played, and continue to play, such a critical role with regard to software. The tension between its invisibility and intangibility on the one hand, and its complexity on the other, almost demands metaphorical mechanisms and allusions. For a whole variety of aspects associated with computers and software, metaphors are not simply some form of linguistic baggage that obscures reality, they are actually crucial in *constituting* that reality. (Although this is not to deny that sometimes metaphors do indeed obscure

[6] The examples are all drawn from English, and other languages may treat some of the concepts in different fashion. However, for good or ill the language of software development is English - albeit the US variant. I would, however, be interested to hear from readers familiar with non-English software development communities on the ways in which the engineering metaphor applies - or appears inappropriate - in other languages.

rather than illuminate.) Moreover this is not a one-way process. Berman [3] has argued that computers are 'a defining technology [developing] links, metaphorical or otherwise, with a culture's science, philosophy or literature; it is always available to serve as a metaphor, example, model or symbol'.

6 METAPHORS AS GENERATIVE DEVICES

All this should give credence to the relevance of metaphor to software development. Yet it may still be argued that software developers clearly understand the inadequacies and limitations of existing software engineering terminology. Surely all this linguistic baggage is mere adornment, and has no substantive impact on the practices and processes of software development itself? Brooks' point about the 'power' of the construction metaphor suggests otherwise, as does much recent work on the power and role of metaphor. Linguistic devices do have an impact, and it is critical for practitioners to recognize how these tropes operate; and where necessary, remedy or clarify the terminology. This sort of effort may lead to the generation of innovative metaphors, that in turn lead to a reorientation in a similar fashion to that described by Brooks.

One argument exemplifying the cognitive power of metaphors - both constraining and liberating - can be found in the work of Donald Schön [21]who views metaphors as generative; explicitly undermining the idea that a metaphor is a 'kind of anomaly of language, one which must be dispelled in order to clear the path for a general theory of reference or meaning'. He proposes that metaphor be viewed as 'central to the task of accounting for our perspectives on the world' (p137). Metaphorical utterances such as 'software development is an engineering discipline', are symptoms of a process of '*carrying over* of frames or perspectives from one domain of experience to another'.

Schön wishes to direct attention to areas where there are conflicting frames, demanding 'frame restructuring' - i.e. domains where differences are not explicable or reconcilable by recourse to facts or fixes, but which emanate from conflicting frames for the construction of reality, and where the only recourse is to 'restructuring, coordination, reconciliation, or integration of conflicting frames'. Software engineering is beginning to become the subject of precisely this sort of conflict; and again this should be welcomed and developed. The restructuring process may well result in the emergence of a generative metaphor that has the same beneficial impact on the wider community of software developers as the construction one did on Brooks 40 year ago.

Schön illustrates his argument using the example of a group of researchers seeking to develop a new paintbrush. The new product incorporated artificial bristles instead of natural ones. The brush failed to give the same smooth finish as its bristle counterpart, leaving a surface that was marked by discontinuous application of the paint. The group had observed that natural bristles produced 'split ends', whereas synthetic ones did not; so they split the ends of the synthetic brush, but with no marked improvement. Real progress was made, however, when one of the team remarked that 'a paintbrush is a kind of pump'; and that painting really amounts to using the *spaces* between the bristles as channels through which the paint can flow. This directed their attention to a range of new factors, particularly the curve formed by the bristles of the non-synthetic brush. What in strict terms might be thought of as a mistake - a paintbrush is *not* a pump - results in the generation of new perspectives, explanations and inventions.

This is not a simple process that leads to 'mapping' from one domain (pumps) to the other (painting). The initial response is what Schön refers to as one of 'unarticulated perception'. Only later is the relationship between the domains clarified by interpreting ostensibly different 'tools' as examples of a single category.

7 A SOFTWARE ENGINEERING EXAMPLE

Requirements - Capture, Specification, Engineering?

Thus far I have been concerned to explain what metaphors are, and to stress the influential role they play as an indispensable component of cognition. Taken together, this provides justification for the project of analyzing and partially dismantling or even replacing the engineering metaphor for software development. To add further weight to this I wish to consider the example of requirements determination.

Requirements determination is particularly critical to software development. In numerous surveys it is the *requirements* phase that is seen as the most important and the most difficult. The standard texts usually view this in terms of the documents that are produced during and at the end of this phase - e.g. requirements specification, requirements definition and so on. The current vogue seems to be to use the term *requirements engineering* for the activities that lead to these products, although Sommerville notes that 'the term *engineering* is used rather loosely'.

Although Sommerville hints at some discomfort with the 'engineering' epithet, its use is not too surprizing given that the terminology surrounding requirements for computer-based systems is replete with terms drawn from the engineering metaphor. *Requirements definition*, *requirements specification*, *requirements validation*, and so on may appear unremarkable in the 1990s, but their source is clearly the realm of engineering.

The assumption, often stated as such, is that the requirements process must be systematic, with some element of management and formality. Here the engineering metaphor provides a useful basis for what is widely agreed to be the most critical and disparate part of

software development, and the power of the engineering metaphor seems beneficial in this context: But there are drawbacks. The engineering metaphor itself reverberates beyond the ideas of rigour and formality, with connotations that obscure key features of the process. This is not a problem that is restricted to software engineering; it is far more widespread.

In a remarkable paper Michael Reddy [20] discusses what he terms the *conduit* metaphor - a metaphor deeply embedded in our ideas of communication (and I suspect it is not restricted to English).

Reddy poses the question 'What do speakers of English say when communication fails or goes astray?'. Rather than use his examples, it will be more pertinent for our purposes to consider those that might emerge during a requirements exercise. In such cases it is common to find that clients' and users' views have not been fully understood by the developers; and that developers have not made clear the trade-offs and constraints to the domain experts.

1 The developers should try to get their thoughts across better
2 None of the real issues about users demands came across to me with any clarity
3 We have not given the domain experts a clear idea about the technological constraints

(These are broadly similar to the examples that Reddy himself offers.)

At first sight all these examples appear unremarkable, albeit familiar: And surely they are metaphor-free? Reddy's point, however, is that all of them - and many more of their ilk - are deeply bound to a particular metaphor; and one with enormous impact. They are all based around the idea that information is *transferred* from one point or person to another. Effective communication then resembles friction-free, blockage-free flow of information. Good reception involves extraction and unwrapping.

Reddy offers numerous examples to support his case before offering what he terms the four categories that constitute the critical features of the conduit metaphor

> (1) 'language functions like a conduit, transferring thoughts bodily from one person to another; (2) in writing and speaking, people insert their thoughts or feelings in the words; (3) words accomplish the transfer by containing the thoughts or feelings and conveying them to others; and (4) in listening or reading, people extract the thoughts and feelings once again from the words.' (p170)

He further points out that one sub-component of this *conduit metaphor* characterizes thoughts and feelings as being 'ejected ... into an external *ideal space*, where they are reified, and take on an independent existence; and from where they may or may not 'find their way back into the heads of living humans'.

The following examples illustrate these features with regard to software development

1. Get those requirements down on paper before we lose them.
2. We've been trying to pin down that idea for ages.
3. There's more than a head-full of issues here.

Reddy is not saying that we have to stop using this metaphor, or that it is erroneous in some fashion; his main concern is to raise awareness of what is a deeply hidden metaphor. Furthermore Reddy offers an alternative, in order to direct attention to the deficiencies and misconceptions fostered by the conduit metaphor: He terms it *the toolmakers paradigm*.

In order to do this justice, and lay the basis for 'frame restructuring', the toolmakers paradigm needs to be described at some length, using Reddy's own illustration. Suppose that there exists a community of people living in a compound shown schematically in figure 1. Each person has their own sector, and no two sectors are alike. The hub of the wheel contains a mechanism for delivering paper messages from one person to another, and this is crucial in people passing on their ideas about how best to survive in terms of building shelters, developing tools, and so on. People cannot visit each other's sectors, nor can they exchange products; only crude blueprints.

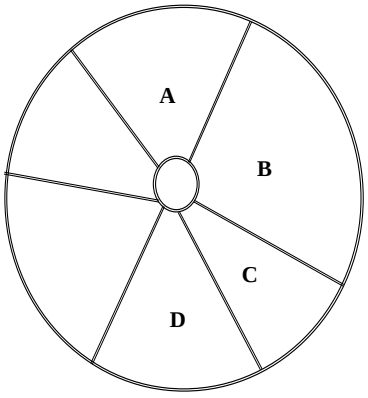

Figure 1

Any individual only knows about the existence of others as an inference from the exchange of pieces of paper, plus other supporting deductions. Reddy calls this the 'postulate of radical subjectivity'.

Suppose that the person living in sector A develops a tool; a rake for clearing away leaves and other debris. She goes to the hub and draws three identical sets of instructions for fashioning this tool, leaving a copy for B, C and D. A's environment has a lot of wood and trees in it, but B's sector is mostly rocky. So B tries to copy the design for A's rake,

using a wooden handle and a stone head: A did not specify the material for the handle or the head (which were both wood) since there seemed no alternative. B completes the 'rake', but finds it unwieldy and heavy; and wonders at the strength of A. He is also somewhat bemused at the tool itself, and guesses that A uses it to clear away small rocks in her sector. He improvises for his own environment, and eventually decides that a better form of the tool will be one that has two prongs to unearth large rocks. This is far more useful for B's sector. B then sketches out his tool design and makes three copies for the others to study.

Persons C and D develop their own tools on the basis of the plans from A and B. Person A makes a tool along the lines suggested by B, but can see no use for it in her own rock-free sector. She also wonders if B has misunderstood her original design, and so produces a more detailed one for circulation. The interchange between A and B goes on for some time, until A comes across two small pebbles in her sector, and she begins to understand that her assumptions about wooden materials and organic debris may not be universally applicable. The result is that A and B develop more accurate inferences about the other's environment by a process of dialogue that gradually makes explicit each other's assumptions. This is an iterative and error-ridden process.

We can now try to understand the requirements phase of software development from the toolmakers perspective. This will highlight the inadequacies of many ideas regarding current practices, all heavily imbued with the conduit metaphor. Software developers understand that communication with users, clients and so on is critical; but this is invariably seen as a flow between domain experts, users, and developers. Difficulties or failures are described in terms of blockages or breakdowns in the channelling of information. The basic conceptual imagery of requirements engineering rests on the conduit metaphor. The information about requirements is passed from 'users' to 'developers'; or in some instances the requirements exist in disembodied form, and have to be captured.

If the limitations of the conduit metaphor can be understood, and alternatives such as the toolmakers paradigm considered, then the issues in requirements *engineering* come to be seen in a different light. The process is not one of passing or capturing information; the aim is for all those involved to work together to achieve a coherent and consistent view of the system context, an objective that is inherently difficult and requires collaborative input from all participants.

This sheds new light on the requirements phase, which is always acknowledged to be crucial and difficult. The point is that while this acknowledgement remains bound up in the conduit metaphor, the solutions on offer will fail to resolve the complexities. *IEEE Software* has focused on this topic a number of times in recent years, and in 1998 one issue focused on *requirements engineering*. The guest editors offered an introduction illustrating the conduit metaphor. [2] The general editor, on the other hand, offered a contrasting view that stressed that 'the field of requirements (and it no longer matters to me what word you like to append to *requirements* to make it sound more esoteric) has to do with understanding, not documentation' [8]. This realization had only come to him over the 20 years since had been confronted with a document entitled 'Software Requirements Specification'. This title of three nouns took him some time to decode, and resulted in him viewing requirements as being about documenting the external behaviour of a system; something which he now saw as partial and misleading.

In general the other contributors seem more in tune with Berry & Lawrence, than with Davis. Even those who mention scenarios and dialogue fail to follow this line of reasoning to its *toolmakers* conclusion; and it is understandable that they seek recourse to traditional software engineering recommendations such as better tools, formal descriptions, management and traceability. These are important, but ignore aspects for which Davis has prepared the ground when he stresses that anyone 'involved in requirements needs human skills, communication skills, understanding skills, feeling skills, listening skills' (paraphrasing DeMarco).

How much more effective might the requirements process become if viewed in terms of a dialogue between distinct parties, each with their own assumptions and cognitive processes; where achievement of mutual understanding is a prime objective, but one which will not be reached without communicative effort from all participants. Brooks tells us that changes in metaphor can have real effects on systems; 'teams can *grow* much more complex entities in four months than they can *build*'. So perhaps a change in the metaphor about communication during the requirements process can have a similar impact.

It is perhaps difficult for developers to appreciate the full impact of these aspects of software development because they are cognitively tuned to the engineering metaphor, and so unreceptive to these ways of understanding the issues. What we need is a collective jolt similar to the one that Brooks experienced when confronted by the construction metaphor. But the shift away from the engineering metaphor is rendered difficult because of the pervasive assumptions regarding the nature of engineering and the role of the engineer.

8 MOVING BEYOND *ENGINEERING* ?

The idea of *software engineering* has been central to software development since 1968, although Coplien [7] wonders if Peter Naur meant it as a joke! Whatever the motivation might have been, the term has stuck: But why? Mahoney [15] stresses the importance of the ways in which the term is used to anchor the field in a specific context;

‘every definition of software engineering presupposes some historical model’. He quotes the definition from the NATO conference implying ‘the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering’; and points out that the crucial terms are undefined. Mahoney argues that the ideas around the emergence of software engineering develop from multiple ‘mythic histories’. He identifies three specific views of engineering itself – applied science, mechanical engineering, and industrial engineering. This echoes Goldberg’s argument that there are three foci of software engineering – reliability, management and productivity [11].

The historian’s perspective offered by Mahoney, gives support to those who wish to jettison the engineering epithet; but there are those such as Parnas who remain convinced of its value and meaning. He argues that software engineering be treated as ‘an element of the set {Civil Engineering, Chemical Engineering, Electrical Engineering, ...}’. This taxonomy derives from his view that students in an ‘engineering’ discipline receive and require a different style of education from those in science-based ones. ‘Engineers are professionals whose education prepares them to use mathematics, science, and the technology of the day to build products that are important to the safety and well-being of the public.’ This accords with Hoare’s ideas from the 1970s, where he took engineers to represent the highest ideals of professionalism [13]. In this he was reiterating early work on professionals in general which saw them as imbued with a wide vision, a doctrine of service to the community, and a large degree of autonomy in their activities. Professionals had some form of certification from their own professional body, and this gave them independence from commercial and political pressures. Parnas develops the epistemological aspect of this by arguing that just as electrical engineering has its scientific basis in physics, so software engineering has its basis in computer science. But this is simply a 1999 version of a mythic history that originated in 1968. There is no need to repeat the arguments that stress the distinctions between engineering of *products* and software development; but Parnas has used an incorrect analogy. Some aspects of software engineering may well be based on computer science, but the impetus behind the development of software is fundamentally powered by factors derived from the widespread use and commercial aspects of computers. This is a key distinction, and can be illustrated using Parnas' own form of argument.

Automotive engineering would be one member of his engineering set, and clearly has a basis in science - physics, mechanics etc. But automotive engineering has to involve more than use of ‘mathematics, science, and the technology of the day to build products that are important to the safety and well-being of the public’. Parnas, like Hoare, takes it for granted that the engineer’s combination of technical expertise and consummate professionalism are necessary and sufficient for discipline and practice. Compare this, however, with Merton who argued that far from being natural decision-makers, engineers were in fact imbued with a ‘trained incapacity for thinking about and dealing with human affairs’ (quoted in [14]).

Parnas’ terms originate in what Boguslaw, in 1965, called the new utopianism of computer system developers who ‘retain their aloofness from human and social problems presented by the fact or threat of machined systems and automation. … People problems are left to the after-the-fact efforts of social scientists’ [quoted in 12].

Merton and Boguslaw were both social scientists, but this critique of an inherently narrow rationalism has always been present in software development (widely understood) through the writings of people such as Ackoff, Ehn, and Winograd & Flores [1, 9, 24]. Moreover an implicit critique of the engineering metaphor was present within the software development community in the 1960s. Mahoney alludes to an intervention by Sharp at the 1969 NATO conference, where he argued that ‘one ought to think rather in terms of *software architecture* (= design)’; and that architects are trained differently from engineers. Sharp even went as far as arguing that ‘I don't believe for instance that the majority of what [Edsger] Dijkstra does is theory -- I believe that in time we will probably refer to the 'Dijkstra School of Architecture''.

The architectural metaphor was eclipsed by the engineering one; but it has its own history and has become more influential in the past decade. Coplien, writing as guest editor in the issue of *IEEE Software* immediately preceding the one devoted to software engineering, in which Parnas’ article is published, notes that in 1961 or 1962 Brooks raised the idea of an architectural metaphor. So there is some basis to argue that it predates the engineering one.

Although Coplien locates software architecture as a chapter in software engineering, he also recognizes the profound challenge that it poses to current orthodoxy. Thus the emergence of patterns involves recognition that ‘architecture, like any system discipline, is about relationships between system parts and … between people’. This means questioning the strategy of grand designs and modular construction, because the detail will only emerge as the project develops. Coplien quotes Gabriel who sees software development as proceeding by ‘piecemeal growth and rarely through thorough design. … planned development alienates those developers who are not also the planners’.

Schön would describe the architecture or engineering distinction as one of *frame conflict*. The solution to this he terms ‘frame restructuring ...constructing a new problem-setting story, one in which we attempt to integrate

conflicting frames by including features and relations drawn from earlier stories, yet without sacrificing internal coherence or the degree of simplicity required for action'. The architectural metaphor, seen through the writings of those influenced by Alexander perhaps moves some way towards this. Coplien seems to be laying the groundwork for this in his argument that the true basis of software architecture was lost as software engineering gave in to 'formal envy'; although this is now being rectified as developers 'embrace and capitalize on uncertainty' and 'honour the human constraints'.

Any moves towards a resolution, however, are in danger of being deflected by what McConnell has termed the 'Gold Rush', with the 'present day equivalent of the tin pan and the wooden sluice being a desktop computer, fast Net connection and software compiler' [quoted in 6]. There is no chance of enforcing any principles of software development in a context where code-and-fix is the order of the day; with developers aiming to meet excess demand. Again this is not a new problem, although its scale is probably larger than ever. Mahoney notes that the issue of 'industrial strength software in a competitive market' was a challenge from the start; and he quotes Bauer, in 1971, to the effect that these issues are 'too difficult for the computer scientist'.

Schön demonstrates that frame restructuring and the making of generative metaphor are closely related. 'In both processes, participants bring to the situation different and conflicting ways of seeing ... there is an impetus to map the descriptions ... but [they] resist mapping ... the participants work at the restructuring of their initial descriptions - regrouping, reordering, and renaming elements and relations ....'.

Wittgenstein observed that metaphor is the way we make sense of things, but it is also the way we produce nonsense when there is a mismatch between the language that we use and our practices. We have to avoid nonsense. We have to start the process of frame restructuring, taking the strengths of the engineering, the architectural and other relevant metaphors; simultaneously coping with the commercial pressures of the 'code rush'.

Schön offers us a strategy and a conclusion - 'when we interpret our problem-setting stories so as to bring their generative metaphors to awareness and reflection, then our diagnoses and prescriptions cease to appear obvious and we find ourselves involved, instead in critical inquiry'. In the 1950s and 1960s the introduction of the engineering metaphor moved us forward in the critical activity of developing a discipline for software development; we now have to move forward to the next stage.

REFERENCES

1. Ackoff, R. *Redesigning the Future*, Wiley, 1974
2. Berry, D.M. & Lawrence, B. guest editors' introduction on Requirements Engineering, *IEEE Software*, March/April 1998
3. Berman, B. The Computer Metaphor: Bureaucratizing the Mind, *Science as Culture* 7, 1989
4. Black, M. More about Metaphor, in Ortony, (ed) 1993
5. Brooks, F.P. No Silver Bullet: Essences and Accidents of Software Engineering, *IEEE Computer*, April 1987, pp 10-19
6. Chapman,G. Gold Rush Mindset Undermining Programming Field, *Los Angeles Times*,
7. Coplien, J. Reevaluating the Architectural Metaphor: Toward Piecemeal Growth, *IEEE Software*, September/October, 1999, pp 40-44
8. Davis, A. The Harmony in Rechoirments, editorial in *IEEE Software*, March/April 1998
9. Ehn, P. *Work-oriented Design of Computer Artifacts*, Lawrence Erlbaum, 1989
10. Fowler, H.W. *A Dictionary of Modern English Usage*, 2nd edition, revised by E Gowers, Oxford, 1965
11. Goldberg, R. Software engineering: An emerging discipline, *IBM Systems Journal*, vol25, 3-4, 1986, pp 334-353
12. Greenbaum, J. & Kyng, M. *Design at Work*, Lawrence Erlbaum, 1991
13. Hoare, C.A.R. Software Engineering, *Computer Bulletin*, December 1975, pp6-7
14. Johnson, T. *Professions and Power*, Macmillan, 1972
15. Mahoney, M. Finding a History for Software Engineering, plenary delivered at *Foundations of Software Engineering Conference*, November 1998
16. McDermid, J. (ed) *Software Engineer's Reference Book*, Butterworth Heinemann, 1991
17. Ortony, A. (ed) *Metaphor and Thought*, 2nd edition, Cambridge, 1993
18. Parnas, D.L. Software Engineering Programs Are Not Computer Science Programs, *IEEE Software*, November/December 1999, pp19-30
19. Pressman, R. *Software Engineering*: *A Practitioner's Approach*, 3rd edition, McGraw Hill, 1992
20. Reddy, M. J. The conduit metaphor: A case of frame conflict in our language about language, in Ortony, 1993
21. Schön, D.A. Generative Metaphor: A perspective on problem setting in social policy, in Ortony, 1993
22. Sommerville, I. *Software Engineering*, 5th edition, Addison-Wesley, 1996
23. Wasserman, A. Toward a Discipline of Software Engineering, *IEEE Software*, November 1996, pp23-31
24. Winograd, T. & Flores, F. *Understanding Computers and cognition*, Addison Wesley, 1986