

# 35 years on: to what extent has software engineering design achieved its goals?

C.L. Simons, I.C. Parmee and P.D. Coward

**Abstract:** The term ‘software engineering’ was coined in 1968 to introduce the disciplines of established branches of engineering design to software manufacture. Some 35 years on, this paper attempts to gauge the success of software engineering against its original goals, with particular respect to the adoption of an industrial design process. The design issues raised in the 1968 NATO conference are examined and then modern examples of engineering design and software engineering are compared. While many aspects of design are found to be similar between the two, significant dissimilarities are also evident. Knowledge of such similarities and dissimilarities may offer opportunities for software engineering to learn lessons from engineering design, for example in the generation and evaluation of solution variants. Field studies are reviewed for empirical evidence of the success or failure of software engineering; results suggest a mixed picture over a diverse range of application domains. It is found that the issues surrounding software production identified 35 years ago remain unresolved today. Although considerable benefit was gained from adopting fundamental design practices from engineering design, the demands on software engineering continue to increase beyond the capabilities of current software engineering theory and practice.

---

## 1 Introduction

In 1968, The NATO Science Committee sponsored a working conference on software engineering to discuss all aspects of software, including the relation of software to hardware, the design of software, the production (or implementation) of software, and the distribution and service of software. At the conference, Bauer coined the term ‘software engineering’ to address the perceived crisis in the production of software at that time. As the report of the conference puts it [1], ‘the phrase ‘software engineering’ was deliberately chosen as being provocative, in 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.’

In 1968, established branches of engineering had drawn extensively from systems theory, a theory where systems are characterised by the presence of a boundary which cuts across its links with its environment. Such links reveal the external behaviour of the system such that it is possible to define a function expressing the relationship of inputs and outputs, and thus changes to the values of system variables. For complex systems, the system may be decomposed into subsystems for which it is also possible to define functions that relate inputs to outputs. Since technical engineering artefacts could be represented as complex systems, systems theory enabled the notion of solving complex technical

problems by applying systems theory in steps. In outline, these steps involved problem analysis, solution synthesis and solution evaluation. Established engineers drew on these steps as the basis for systematic and disciplined design methods for numerous branches of engineering, e.g. civil engineering, mechanical engineering, electronics engineering, chemical engineering, etc. Engineering design methods also differentiated between development and industrial production (manufacture/assembly) of the technical product, and generally referred to entire development (i.e. analysis, synthesis and evaluation) of the product as ‘design’.

Discussions at the NATO conference were organised under three main headings: Design of software, Production of software and Service of software. However, the report states that

‘the difficulties associated with the distinction between design and production in software engineering was brought out many times during the conference. Indeed, what is meant here by production in software engineering is not just the making of more copies of the same software package (replication), but the initial production of coded and checked programs.’

Therefore this paper adopts the view that for software engineering there is no essential difference between design and production, and so will henceforth designate the terms design and production to be included by the term software engineering design.

At the conference [1], Bauer suggested that software engineering should embody ‘... the establishment and use of sound engineering principles in order to obtain economically viable software that is reliable and works efficiently on real machines’. Naur endorsed this view, stating that ‘... software designers are in a similar position to architects

---

© IEE, 2003

*IEE Proceedings* online no. 20031198

doi: 10.1049/ip-sen:20031198

Paper first received 13th June and in revised form 5th November 2003

The authors are with the Faculty of Computing, Engineering and Mathematical Sciences, University of the West of England, Frenchay, Bristol BS16 1QY, UK

and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these ideas about how to attack the design problem.' In conference discussions there was general agreement that software engineering was in a very rudimentary stage of development as compared with the established branches of engineering, with Naur quoting in particular the construction architect Alexander [2] as a potential source of ideas for software engineering.

However, as software engineering drew on other disciplines (e.g. civil engineering, systems theory) at its birth, it emerged that some aspects of software engineering (e.g. the design life-cycle) were broadly similar to the other disciplines, whereas other aspects (e.g. problem analysis) were dissimilar due to the abstract, almost intangible nature of software. 35 years on, these similarities and dissimilarities persist. Knowledge of these similarities and dissimilarities may prove fruitful for software engineering research and practice by offering potential opportunities to learn lessons from other disciplines.

## 2 Generic design process

The activities of human design have proved elusive to define comprehensively. While there is agreement that a generic design process exists – 'the process of inventing physical things which display new physical order, organisation, form, in response to function' [2] – agreed definitions of how design occurs across a range of fields are less readily available. At the root of this lack of definition is the seemingly unbounded complexity of design problems. Modern design problems are increasingly complex to the point of perplexing human comprehension, being typically ill-structured and constrained by multiple conflicting requirements. Such design problem complexity contributes significantly to the poor understanding of the design process.

However, fundamental insight into the generic nature of the design process is offered by Alexander [2], who suggests that 'the ultimate object of design is form'. Alexander notes that

'...every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a property of this ensemble which relates to some particular division of the ensemble into form and context.'

Suh [3] relates the context of design to requirements, offering a view of the design process complementary to Alexander as

'determining the design's objectives in terms of specific requirements, which will be called functional requirements. Then, to satisfy these functional requirements, a physical embodiment characterised in terms of design parameters must be created. Design is defined as the mapping process from the functional space to the physical solution space.'

To illustrate the generic nature of the design process, this paper now examines the fields of building construction and engineering design as vehicles for generic design, rooted as they are in thousands of years' practice. It is notable, however, that despite this lengthy duration of time the exact

process of design remains largely undefined. Lawson [4] attempts to distil the experience of many years' building construction theory and practice into design process characteristics. He notes that building design problems are manifested across multi-dimensional, conflicting requirements, often involve a high degree of interaction among the various stakeholders of the design problem, and are difficult to quantify. Additionally, Lawson notes that the process of design involves the generation of alternative candidate design solutions, which are evaluated by quantitative and qualitative value trade-offs within the bounds of often conflicting constraints. Alexander [2] suggests that the evaluation of candidate design solutions is largely driven by the concept of misfit between form and context, a misfit being easier to detect than fit. Indeed, the absence of misfit is often taken as the presence of fit.

The design context may be viewed as a series of design constraints, whose nature may vary also. Alexander [2] notes that the design process is made difficult because we are trying to make a diagram of the forces (constraints) whose field (context) we do not yet understand. Simon [5] examines the nature of representing problem and solution spaces in the design process. He suggests that every problem solving effort begins with the creation of a representation of the problem to define the space in which the solution can take place. Simon advocates the use of abstraction to obtain a successful problem representation: '...focus of attention is the key to success – focusing on particular features of a situation that are relevant to the problem, then building a problem space containing these features, but omitting the irrelevant ones.'

Lawson [4] suggests that constraints that shape the problem space may be characterised as:

- radical or fundamental to the problem, where the problem/solution divide is directly addressed;
- practical, where the physical materials or technologies of the design product impinge;
- formal, where the organisation of solution component assemblies are addressed with respect to decomposition, proportion, geometries etc.; and
- symbolic, where the aesthetics of emergent solution properties dominate.

Lawson summarises the solution generators and constraints in a model of design problems in Fig. 1:

Lawson [4] goes on to suggest that design problems cannot be completely and comprehensively stated – rather it is the nature of the design process to employ subjective interpretations and to tend to organise (or decompose) design problems hierarchically. This decomposition of

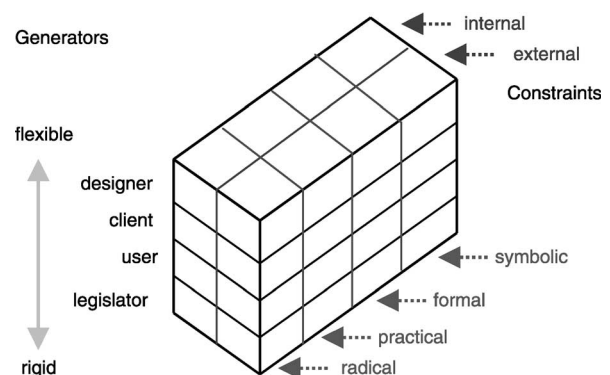


Fig. 1 Model of design problems, reproduced with permission from [4]

design problems is crucial to the design process, and is noted by other authors. Alexander [2] observes that the fitness of a form within its context can be decomposed into a number of fitness 'variables' that relate discrete, physical parts of the solution form to coherent, identifiable parts of the problem context. He further observes that the part of the solution and problem to which a fitness variable relates should ideally be independent of all other fitness variables within the problem context. This facilitates the independent variation of solution/problem parts with little or no impact on other solution/problem parts, resulting in a design that is robust yet flexible to change. While independent variation is a highly desirable attribute of all designs, Alexander notes that in designs of high complexity typical of the twentieth century, the opposite is frequently achieved. Such designs exhibit coupling between fitness variables such that changing a part of the solution to optimise a specific part of design form has an impact on other fitness variables. Such impact may be positive (i.e. increasing the overall fitness of the design) or negative (i.e. reducing the overall fitness of the design). Suh [3] offers a design axiom to help avoid design coupling which states, 'maintain the independence of functional requirements', and promotes the notion of clear traceability from independent functional requirements to independent design parameters. However, producing designs with decoupled fitness variables has proved difficult to achieve within multi-variate, heavily constrained design spaces.

When candidate design solutions (or subsolutions) are generated, Lawson [4] suggests that there are potentially an inexhaustible number of designs possible for every design problem. He also suggests that typically no single, optimal design solution exists – a view endorsed by Simon [5]. Simon proposes the concept of 'satisficing', a term applying to a candidate design solution that is just good enough, i.e. the fitness of a candidate design cannot be demonstrated to be inferior to any other candidate while satisfying constraints to some degree. Lawson [4] also notes that design solutions may be holistic responses to problems, i.e. solutions 'appear' in response to components of the problem, which may then be assembled to address the problem as a whole.

Simon [5] also addresses the representation of design space, and notes the importance of making the problem space and solution space representations amenable to the process of design. He quotes Amarel [6], who goes so far as to suggest that 'solving the problem simply means representing it as to make the solution transparent'. While this is not always possible in all design situations due to the complexity and diversity of design contexts, it does elude to the significance of compatible problem/solution representations. The importance of problem/solution representation is recently explored by Do [7], who investigates the ways in which architects use sketches and informal diagramming of their designs to perform functional reasoning, structure mapping and knowledge acquisition. Do describes how designers' sketches become a reasoning process framework for functional reasoning about the design problem, in which the dimensional marks and calculations ascribed to graphical symbols validate functional reasoning. Sketches also afford the designer a means of visualising geometric shapes and the spatial relationships among them, supporting the exploration of the design space.

Also inherent in the design process is the discovery of further problem requirements as the problem becomes better understood as partial design solutions emerge, as is the inevitability of subjective value judgements concerning which constraints dominate. Lawson [4] suggests that the

strategies used by designers to arrive at optimal solutions include:

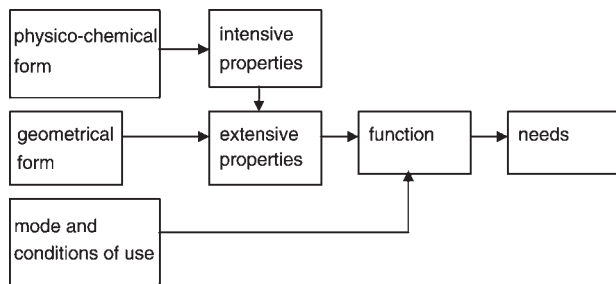
- scanning the problem context as it appears to explore and negotiate potential changes in boundary;
- using heuristics and design schemas based on previous experiential design knowledge;
- generating many alternative solutions;
- clustering together subelements of the overall problem and elevating this clustering to generate a form or structural architecture for the product; and
- keeping parallel lines of thought open to constantly evaluate alternatives within the bounds of the constraint which currently has the focal role (radical, practical, formal or symbolic).

The generation of alternative (or variant) solutions is seen by many engineering design authors as crucial to the success of the generic design process. A recent proposal by Liu *et al.* [8] addresses this facet by suggesting an 'ideal' approach for generating promising design alternatives. Liu *et al.* examine the balance to be struck between generating a wide range of conceptual design alternatives to prevent overlooking valuable variants, and evaluating/selecting promising variants soon enough to restrict their number from getting too large to allow meaningful consideration. The 'ideal' approach suggested involves three levels of solution divergence and convergence to achieve variant generation and selection. The first level involves generating topographical solutions based on various intrinsic properties of the physical/chemical form of solution variants, and screening with heuristics. The second level addresses promising spatial configuration possibilities. The third and final level addresses the generation and selection of physical embodiments of the solution variants. Liu *et al.*'s pursuit of an 'ideal' approach to solution variant generation is typical of the emphasis of placed on this aspect by generic engineering design.

However, possibly because of the large number of solution alternatives generated, design is, according to Lawson [4], potentially endless with no infallibly correct process. This aspect is also examined by Norman [9], who suggests that it is also inherent in the generic design process that, over time, design processes may evolve. Echoing Alexander's notion of design fitness, it seems likely that if problem contexts evolve over time, so should the design solutions. Should a design solution be robust yet flexible to change (as achieved by decoupled fitness variables), individual fitness variables may "evolve" independently of the overall design solution. Indeed, Norman [9] observes that "over time, this process results in functional, aesthetically pleasing objects". However, such evolutionary designs are only achieved when fitness variables of a design solution are sufficiently decoupled.

One possible explanation for the potentially endless nature of the generic design process is offered by Roozenburg and Eekels [10]. Roozenburg and Eekels differentiate between the form (or properties), the context (or mode of usage) and the function (or behaviour) of the design, and suggest that, of the many properties that a design may possess, only a few become evident under the specific conditions of a mode of usage. Roozenburg and Eekels suggest that

'a product with the required properties therefore functions in the intended manner only if it used in an environment and in a way that the designer has thought up and prescribed. The instructions for use are not given facts for the designer, but are thought up – together with the form of the product – and thus form an essential part of the design.'



**Fig. 2** Generic thought process of a designer, reproduced with permission from [10]

Roozenburg and Eekels suggest that a generic designer's thought process might be as in Fig. 2. (We interpret what Roozenburg and Eekels refer to as 'intensive properties' to be the internal properties of the design, whereas the 'extensive properties' are the external properties of the design.)

From the above Figure, Roozenburg and Eekels suggest that the functioning of a designed product depends on both its form and its use, as the arrows in the Figure denote causal relations. They propose that

'the designer, however, should reason against the direction of the arrows. Given a desired function, he should think up the form and its use, and this in such a way that if the user acts in accordance with usage instructions, the intended function is realised. This is the kernel of the design problem'.

If Roozenburg and Eekels are correct in identifying the kernel of the design problem, then a number of consequences follow. Firstly, it seems likely that considerable functional reasoning is required to solve a design problem. It also seems possible that the more alternatives are generated, the more the likelihood of design success. In addition, while much scientific knowledge is available for the transition from form to function, less knowledge is on hand for the transition from function to form, making this transition (the design process) largely dependent on the insight, creativity and experiential knowledge of the individual designer. Such consequences could conceivably give rise to the potentially endless nature of the design process, and are consistent with Alexander [2], Lawson [4] and Simon [5].

Attempts have been made to categorise designs as routine, innovative or creative. Parmee [11] suggests that routine designs are those where the problem context is largely well understood, resulting in designs that are evaluated from a well defined set of alternatives. Typically, fear of design failure is a prominent design driver, and so such designs are generally derived from similar, established problem contexts, and are characterised by precise, crisp, systematic and mathematical solutions. On the other hand, Navinchandra [12] suggests that '...innovative designs can be obtained by exploring a wide variety of alternatives. Innovative designs are not obtained through some deliberate attempt at producing them, but by generating lots of design alternatives and throwing away the bad ones.' He goes on: '...one source of innovation, in a given design domain, comes from the ability to use knowledge drawn from sources inside or outside the problem domain.' This view is consistent with Goel [13], who summarises several investigators' views of design as a problem space abstractly defined by reasoning goals and domain knowledge in the form of operators that enable problem space search. Reasoning goals are the design requirements and specific ranges of values they can take; the operators are specific to

the particular design domain. Goel's classification of creative design is as follows:

- 'If the design variables and the ranges of values they can take remain fixed during design processing, the design is routine;
- If the design variables remain fixed but the ranges of values change, the design is innovative; and
- If the design variables and the range of values both change, the design is creative.'

Goel [13] suggests analogy transfer as a useful technique to promote the creativity of design, in which analogical design involves 'reminding and transfer of knowledge about one design situation to another, where the transfer can occur in the services of any design task in the new situation'. Thus, according to Goel, analogical transfer requires the use of abstractions, where the abstractions typically express the structure of relationships between generic types of objects and processes.

Numerous authors [10, 14–17] have proposed design methods to promote a systematic and disciplined approach to design, resulting in a number of design steps. Pahl and Beitz [16] suggest that the activities of designers can be roughly classified as:

- conceptualising, i.e. searching for solution principles
- embodying, i.e. engineering a solution principle by determining the arrangement and preliminary shapes and materials of all components
- detailing, i.e. finalising production and operating details.

These design methods suggest that design activities are structured in a purposeful way so that the flow of work can be planned and controlled. However, these design methods also recognise the importance of avoiding excessive rigidity in design, which could stifle creativity. Pahl and Beitz [16] point out:

'Special emphasis is on the iterative nature of the approach and the sequence of steps must not be considered rigid. Some steps might be omitted, and others repeated frequently. Such flexibility is in accordance with practical design experience and is very important for the application of all design methods.'

The desire to promote creativity in design has more recently led to research into the potential of artificial intelligence in engineering design. Reffat and Gero [18] and Do [7] point out that computer aided design (CAD) tools have traditionally been confined to concrete representations and blueprints of products made at the latter stages of the design process after most of the major design decisions have been made. They suggest that bringing CAD systems to the early stages of design is potentially useful in providing designers with fruitful and applicable design knowledge during the generation of design concepts. However, the authors have adopted different approaches.

Reffat and Gero [18] have developed a system capable of discerning design knowledge in relation to its situation, which then modifies its behaviour as its situation from the design environment changes. The system structures its encountered situations and classifies them into situational categories in a hierarchical manner, enabling the design to potentially induce creativity by analogical transfer from one conceptual design situation to another. Do's research [7], on the other hand, has led to the development of design support tools that use freehand designer sketches to produce geometric visualisations using Virtual Reality Modelling Language (VRML). Such virtual reality representations of

early designs afford the designer a rapid prototyping tool to explore visualisations of early design concepts. In such ways, engineering design is seeking avenues to explore the softer conceptual aspects of early design, not dissimilar to that found in software design prototyping. Further research by Parmee and co-workers [19–21] also attempts to enhance design innovation and creativity, by utilising interactive evolutionary computation in support of the early, conceptual stages of design. Parmee emphasises designer interaction with the evolutionary computing design process by enabling human changes to design objectives, constraints and variable ranges in response to information generated from an evolutionary search process. An iterative user/machine based design environment is thus established where problem definition improves as relevant information is identified and utilised in successive design space modifications.

### 3 Comparison of engineering design and software engineering

#### 3.1 Methodological approaches

Since 1968, the desire to apply the disciplined, systematic approach of industrial engineering design to software has led to the emergence of numerous diverse software engineering methodologies [22–29] and much-cited texts [30, 31]. Each methodology offers a design approach prescribed by phases, procedures, rules, techniques, documentation, tools, model notations and practices to develop a software system. Furthermore, each methodology is underpinned by a belief system or philosophy about software design. For example, data-flow based methodologies elevate the importance of representing the flow of data in design whereas object-oriented methodologies believe that representing the system as objects is necessary for successful software development. It is notable that, across the range of methodologies, only two aspects are common to all – structural modularity (expressed in various forms) and the use of abstraction. However, despite this commonality, the philosophy of any one methodology, e.g. Yourdon [29] may be at total variance with the philosophy of others e.g. Jacobson *et al.* [28]. Given the original aims of the software engineering field, it is useful to examine an example of the systematic disciplined engineering design processes that software engineering strives to emulate, and compare them with an example of a mainstream software engineering methodology in order to evaluate the extent to which current software engineering design has achieved its original aims – the establishment and use of sound engineering principles.

Such a systematic approach to engineering design has been proposed by numerous authors [10, 14–17]. Pahl and Beitz [16] are typical of the systematic approach and suggest that ‘the main task of engineers is to apply their scientific and engineering knowledge to the solution of technical problems, and then to optimise those solutions within the requirements and constraints set by material, technological, economic, legal, environmental, and human-related considerations.’ Pahl and Beitz break the engineering design process down into four main phases: product planning and clarifying the task, conceptual design, embodiment design and detail design. The terms ‘conceptual design’ and ‘embodiment design’ are consistent with original proposals by French (later editions of this work may be found in French [15]).

According to Pahl and Beitz [16], product planning and clarifying the task involves eliciting customer requirements while conceptual design involves abstracting the essential

problems and function structures which drive the generation and selection of a principle solution (concept). Embodiment design takes the design concept and embodies it to produce a definitive layout of the proposed technical systems and its component decomposition in accordance with constraints. Detail design then realises the components of the design.

Pahl and Beitz [16, p. 4] propose that the engineering design process is generic to a number of branches including ‘mechanical engineering, electro-mechanical engineering, chemical and process engineering, transport engineering, precision engineering and software.’ It is interesting that Pahl and Beitz refer to ‘software’ as a branch of engineering design rather than ‘software engineering’.

The Unified Software Development Process (USDP) [28] is an example of a modern mainstream software methodology, and represents a significant achievement in unifying previously disparate unifying object-oriented notations, semantics and development processes. The Unified Software Development Process (USDP) [28] also breaks the engineering design process down into four main phases: inception, elaboration, construction and transition. However, unlike engineering design, the USDP phases are really chronological periods in which activities (named ‘workflows’) occur. The workflows are requirements, analysis, design, implementation and test. At a high level, a comparison of the purpose of the engineering design phases and USDP workflows yields an approximate correspondence between the phases/workflows shown in Table 1.

Immediately noticeable is a difference in the terminology of ‘design’ across the two. In engineering design, the term design refers to all activities involved in the design and production of the product once the task is planned and clarified. In USDP, the term design more narrowly refers to the physical, tangible design of conceptual analysis model prior to implementation and test. Also noticeable is that engineering design has no separate phase for testing; testing is deemed to be an inherent component of each phase.

Also comparing at a high level, engineering design offers numerous general methods and techniques for finding and evaluating solutions whereas USDP provides little in this area. Engineering design suggests three categories of solution finding methods [16, pp. 71–98] including conventional methods (e.g. literature search, analysis of natural and existing systems), intuitive methods (brainstorming, method 635, synectics) and discursive methods (e.g. systematic search of physical processes, classification schemes, design catalogues and methods for combining solutions). Engineering design also offers methods for solution selection and evaluation [16, pp. 99–106]. Such methods involve the identification and quantification of evaluation criteria followed by the weighting of evaluation criteria. Objective trees may then be derived which subsequently enable the use of matrices of solution variants against weighted criteria. Each solution variant may be

**Table 1: Correspondence between engineering design phases and USDP workflows**

Engineering design	USDP
Product planning and clarifying the task	Requirements
Conceptual design	Analysis
Embodiment design	Design
Detail design	Implementation Test

awarded a score enabling a ranking of solution variants from optimal to least optimal.

### 3.2 First phase of design - product planning

At the first phase, engineering design suggests product planning and clarifying the task. The emphasis here is on product planning rather than project-based planning, and the prioritisation and quantification of requirements where possible. In the comparative phase of inception, USDP [28] proposes expanding the system vision and setting evaluation criteria. Both engineering design and USDP propose the collection of required product features into a requirements or features list. However, terminology over categories of requirements differs as shown in Table 2.

Engineering design is rigorous in its capture of both design objectives and property requirements; it advocates that design objectives directly address the problem, and that properties are expressed qualitatively and quantitatively. Engineering design reflects the maturity of its field by providing a checklist of 17 headings for quantitative properties, each heading having numerous examples [16, pp. 130–137]. For example, under the heading of geometry, example properties of size, height, breadth, length, diameter, space requirements, arrangement, connection, extension, etc. are given. The properties checklist extends to hundreds of example properties for consideration. Engineering design also emphasises the quantification of requirements where possible so as to enable objective evaluation. USDP [28], on the other hand, emphasises functional requirements over nonfunctional requirements, reflecting the less concrete nature of the software product. To counter the inherently abstract nature of software, functional requirements are simulated with interaction scenarios known as ‘use cases’. However, USDP advises that nonfunctional requirements that are general (in the sense of not being linkable to a single functional requirement) are captured as ‘supplementary’ requirements. USDP offers no nonfunctional requirement checklists but does emphasise the quantification of nonfunctional requirements where possible.

### 3.3 Second phase – conceptual design

In the second phase, engineering design advocates activities termed as ‘conceptual design’ whereas USDP proposes activities termed as ‘analysis’. In engineering design, conceptual design begins with abstracting to identify the essential problems. Pahl and Beitz [16, p. 141] offer question checklists to guide the designer, for example: is the crux of the problem to improve technical function, or to reduce weight, space, costs, delivery time, etc.? Next follows problem formulation in solution neutral terms, emphasising quantitative aspects over qualitative. When formulating the problem, Pahl and Beitz [16, p. 149] advocate establishing a function structure, i.e. an overall function and its subfunctions representing a decomposition of the problem space. Each function is shown with inputs and outputs based on the flow of energy, materials and

signals. Next Pahl and Beitz suggest searching for a working principle to ‘concretise’ the function, offering many rich catalogues and ‘classification schemes’ to the designer. The working principles are then combined to synthesise candidate solution variants. A key issue at this point is the generation of as many candidate solutions as possible. A potentially factorial number of different variant combinations provide an effective search of the problem space. Obvious misfits are rejected by the designer based on experiential knowledge but principle solution variants are firmed up and evaluated against technical and economic criteria. Again, Pahl and Beitz [16, p. 179] provide evaluation checklists to guide the designer. Each solution variant is placed in a matrix against weighted criteria and evaluated using either a numerical measurement or an ordinal value (e.g. high, medium, low) to give an overall value for the solution variant. This enables systematic evaluation of solution variants to establish which variants are optimal when compared to others. Overall, conceptual engineering design reflects the maturity of the field in offering numerous checklists, mechanisms for generating numerous solution variants and their subsequent evaluation against often conflicting criteria. The conceptual design phase results in a principle solution concept.

In the second phase of USDP [28] (elaboration), analysis begins with the elaboration of the functional requirements represented as use cases (use cases provide a simulation of interactions within the problem space). From the use cases, an initial structure of the problem space is then derived. However, unlike engineering design (and early software methodologies) where a function-structure is established, USDP advocates a structural view expressed by logical (or conceptual) objects and components. During analysis, USDP dwells at some length on the concept of ‘software architecture’, explaining it as a set of significant decisions by which the structural decomposition of the problem is initially represented while simultaneously addressing required software functionality. As software systems become larger, USDP highlights that software architecture becomes more crucial to understanding the representation of the internal structure of the problem space. However, because USDP analysis is confined to exploration of the problem space rather than the solution space, the notion of an analysis ‘software architecture’ lacks dimensionality and is at variance with engineering design, whose conceptual design is more quantitative. Thus the USDP treatment of quality requirements (known as ‘nonfunctional requirements’ in USDP) during analysis is at best intuitive and less systematic than engineering design. While the drivers of software architecture are discussed in USDP, checklists and catalogues of exemplars of best practice – a reflection of design maturity – are absent from USDP. (Such checklists and catalogues are also largely absent from software methodologies in general.)

The analysis phase in USDP is, however, very necessary for the synthesis of a candidate outline software solution structure, and as such represents the software problem space at a level of precision sufficiently detailed to enable the USDP design to commence. Indeed, the analysis model of the software-to-be is viewed by USDP (and nearly all software methodologies) as a ‘conceptual model’ to the extent that it is an abstraction and it avoids implementation (solution) issues. Thus a key difference with conceptual engineering design is that USDP analysis does not address the solution space, and in particular the generation and evaluation of solution variants. USDP suggests the solution space is addressed by a separate workflow termed design.

**Table 2: Requirements terminology in engineering design and USDP**

Engineering design term	USDP term
Design objectives (that the intended solution is to satisfy)	Functional requirements
Properties (of the solution)	Nonfunctional requirements

In addition, the USDP techniques for the identification of candidate analysis objects and components and their subsequent composition into a coherent analysis hierarchy appear to be based on fragments of case studies for the purpose of exemplifying analysis activities and artefacts, rather than guidance for the software engineer on object-oriented analysis. Furthermore, the resolution of often conflicting functional and quality requirements during USDP analysis is deferred until design, when realisation issues are addressed. Undoubtedly USDP offers a comprehensive approach to software engineering analysis. However, when compared against conceptual engineering design that has mitigated failure by continual refinement of many decades of practice, USDP analysis appears to offer little guidance in the application of systematic, disciplined engineering principles aspired to at the birth of software engineering in 1968.

### 3.4 Third phase – embodiment design

At the third phase, engineering design advocates activities termed as ‘embodiment design’ whereas USDP proposes activities termed as ‘design’. Embodiment design starts with the principle solution concept and realises the design to the point where subsequent component detail design can lead directly to production. Pahl and Beitz [16, p. 199] suggest that, during the embodiment design phase, designers

‘must determine the overall layout design (general arrangement and spatial compatibility), the preliminary form designs (component shapes and materials) and the production processes, and provide solutions for any auxiliary functions. In all this, technological and economic considerations are of paramount importance.’

The nature of engineering embodiment design is alluded to by Pahl and Beitz, who suggest that embodiment design

‘involves a large number of corrective steps in which analysis and synthesis constantly alternate and complement each other. This explains why the familiar methods underlying the search for solutions and evaluation must be complemented by facilitating the identification of errors (design faults) and optimisation.’

In other words, embodiment design is the design phase where mismatches between the problem space and the solution space are identified and rectified. Importantly, the selection of solution evaluation criteria focuses the designer’s attention on design objectives and – crucially – desired properties of the solution. Indeed, design requirements that were ill-understood previously may become illuminated in the light of many solution variants and so better understood by the designer. The designer may revisit requirements to re-prioritise design objectives and perhaps tighten or even relax constraints. Equally important is the notion of identification of errors in embodiment design, where errors may be the failure to fulfil technical objectives within design constraints. This is consistent with the suggestion of Alexander [2] that misfit between form and context is easier to detect than fit. Design optimisation relies on the generation of numerous and novel solution variants and the rejection of suboptimal variant embodiments against constraints before proceeding to detail design.

Pahl and Beitz [16, p. 206] provide numerous principles, guidelines and checklists for the designer conducting embodiment design. Principles include force transmission, division of tasks, self-help and stability and bi-stability. Guidelines include areas such as design to allow expansion, requirement creep and relaxation, design against corrosion

damage, design for ergonomics, design for aesthetics, design for quality, design for production, design for ease of assembly, design to standards, design for ease of maintenance, design for recycling, and evaluating embodiment designs. The authors complement the embodiment design guidelines with numerous checklists.

The USDP [28] equivalent of engineering embodiment design is termed ‘design’. Whereas USDP analysis is concerned with the problem space, USDP design is concerned with the solution space. USDP suggests that the purpose of design in software engineering is to acquire an in-depth understanding of issues regarding nonfunctional requirements and constraints for the first time since requirements were elicited. Based on that understanding, then to decompose the software solution into manageable pieces and capture the interfaces between those pieces, and thus create a seamless abstraction of the software solution’s implementation. In other words, USDP design is concerned with defining and decomposing the solution space in terms of physical software objects and components derived from analysis abstractions. Solution decomposition in USDP design also represents the ‘software architecture’ of the solution, which addresses nonfunctional requirements and constraints.

The transformation of problem space analysis abstractions represented as objects to solution space design objects directly traceable to software program code is also addressed by USDP. The transformation involves two aspects: derivation of design objects from abstract objects in the problem space, and other transformations necessary to comply with the concrete physicality of implementation constraints on which the software solution is to be deployed.

USDP [28] portrays the derivation of design solution objects from analysis abstractions as an essentially uncomplicated step. For example, if an analysis object such as ‘Customer’ exists, then the design object ‘Customer’ must also exist. In one sense this has merit as the process is straightforward and promotes traceability between the problem and the solution. However, the demerit of this approach is that no attempt is made to either generate candidate solution variants, or evaluate solution variants against often conflicting constraints and technical and economic criteria. The ‘goodness’ or ‘fitness’ of the resulting design object is thus just as likely to be determined by simple intuition and guesswork and traceability to the corresponding analysis abstraction as by any systematic, disciplined approach. Design is less straightforward but typically intuitive when the second aspect of the transformation is taken into account. Nonfunctional requirements, which played little or no part in the identification of analysis abstractions, now play a significant role in the derivation of the solution objects, especially at a holistic level when designing the software architecture of the solution. The nonfunctional requirements specify the quality attributes of the solution and typically include scale or size of the solution in terms of numbers of users and/or numbers of data to be processed, among others. These numbers determine the designer’s choice of implementation and production environments, and any available generic solution objects and components that typically ship with the environments. However, nonfunctional design constraints may impact on each other and conflict, e.g. maximum performance and minimum cost. Intuition and experiential knowledge is typically used by the designer to resolve such conflicts. USDP offers few principles, and little or no guidance or checklists to aid the designer in this facet of design. Thus USDP software design is a combination of two transformations of analysis abstractions: the one-to-one,

almost traceability-driven derivation of physical solution objects from problem space abstractions with little or no generation and evaluation of solution variants, combined with the informal and often intuitive application of an implementation and production environment generic objects and components. In these respects, USDP design reveals a stark contrast to engineering embodiment design. USDP affords no mechanisms for generating and evaluating solution variants. While engineering design promotes the generation of promising design alternatives followed by their evaluation, USDP does not address this facet. USDP offers no guidelines or checklists for the transformation of analysis abstractions into solution objects analogous to the guidelines and checklists provided by engineering design in abundance.

### 3.5 Software architecture in conceptual and embodiment design

In recent years, however, other authors have noted the apparent weakness of mainstream software engineering methodologies to address the influence of nonfunctional requirements on design, and proposed alternative approaches. For example, Shaw and Garlan [32] were among the first to systematically address the importance of solution-generic objects and components. Shaw and Garlan noted various styles of organisation and decomposition of solution-generic objects and components, together with trade-off mechanisms for selection among design alternatives. Shaw and Garlan deemed such styles of organisation and decomposition ‘software architectures’, and offered some guidance for ‘architectural design’ within a design space. Bosch [33] harks back to the systematic disciplined approaches of building architects and civil engineers in advocating the production line metaphor for software engineering design. Bosch places equal emphasis on functional requirements and quality attributes (nonfunctional requirements). Bosch then suggests ways in which functional requirements may be used to structure the problem space into structural component architectures, which are subsequently assessed against quality attribute profiles driven from quality requirements by methods such as scenarios, performance simulations and experiential knowledge. Bosch also offers guidance as to the transformation of the software architecture by the application of architectural styles and practices according to the quality attribute profiles. Clements *et al.* [34] propose an approach for evaluating software architectures termed ‘Architecture Analysis Tradeoff Method’ (ATAM). ATAM also places emphasis on quality attributes (nonfunctional requirements) even at the expense of functional requirements and so is well suited to the evaluation of solution-generic object and component architectures. Clements *et al.* pay particular attention to the quantification of quality attributes and suggest the weighting attributes within a ‘quality attribute utility tree’ that enables evaluation of architectural variants. To provide a coarse degree of solution architecture variant ‘fitness’, Clements *et al.* suggest the use of scenario-based simulations to estimate the final performance/utility of the architecture variant. The ‘quality attribute utility tree’ approach appears not dissimilar to the solution evaluation matrices suggested in engineering conceptual design by Pahl and Beitz [16]. Lastly, Apperley *et al.* [22] advocate the specific design activity of ‘technical architecture’ to design or assemble the solution-generic objects and components aimed at the production platform for the solution, based on nonfunctional requirements and constraints. These ‘technical’ solution-generic objects and

components are then integrated with design solution objects to produce the software solution.

Although Alexander [2] originally noted potential conflicts between desirable solution qualities, Sigman and Liu [35] more recently proposed an argumentation methodology for capturing and analysing solution design rationale specifically to trace the impact of nonfunctional requirements on design. The argumentation methodology represents weighted nonfunctional requirements within an objectives tree, but identifies the impact between requirements as either positive or negative. The objectives trees again appear not dissimilar to solution evaluation matrices of Pahl and Beitz [16]. Designers use experiential knowledge to differentiate between solution variants. However, while Sigman and Liu offer techniques to capture and model quality attributes of the problem space, they have little to suggest on how these might translate into the solution space.

### 3.6 Evolutionary software design

In recent years, attempts have been made to address the lack of evaluation of solution variants by a number of ‘evolutionary’ (or ‘agile’) software design approaches [36–39]. Such design approaches abandon the formality and ceremony of other software engineering methodologies, instead emphasising organisational aspects such as small teams of designers and developers working in close co-operation with the user community to deliver regular increments of software solutions, thus allowing the solution to evolve over time. This evolution is founded on the premise that full evaluation of an executable software solution variant by the user community is preferable to the evaluation of design models and documentation. Evolutionary design has also brought with it an emphasis on testing, as well as an acceptance of the inevitably high rate of change present in many design problem spaces. For example, Beck [36] advocates an elevation of the importance of testing as an affective technique to focus on functional and quality requirements as the criteria for solution evaluation. Cockburn [37] suggests that, as the size of design problem increases, knowledge, understanding and communication of the design problem between the designers is crucial. In fact, a characteristic of evolutionary software design processes is that the level of formality in design varies as the scale of the design problem. For small design problems, minimal formality and ceremony (in the sense of producing and quality reviewing layout diagrams of objects and components) is striven for while for larger, ill-structured design problems of increased complexity more layouts diagrams are necessary to communicate the knowledge of the design solution. Such variation of formality of design is reminiscent of Alexander [2], who draws the distinction between a ‘mental picture’ of the design space and a ‘formal picture of the mental picture’ within self-conscious design. It seems likely that when design problems are small in scale, designers can communicate design solutions verbally and possibly by informal and unstructured sketches. On the other hand, when design problems are large in scale, designers need object and component layout diagrams and blueprints to effectively communicate software design solutions.

Evolutionary design has also brought the design practice of ‘refactoring’ to the fore. Fowler [39] describes the process of re-factoring a solution design as ‘changing a software system in such a way that it does not alter the external behaviour of the (program) code yet improves the internal structure.’ In one sense, it appears somewhat odd to advocate changing the design after the software solution

product is finished, particularly as the nature of design is essentially sequential – conceptual, embodiment and detail. However, Fowler is bringing the iterative aspect of design to the fore, an aspect not lost by other authors [16, 26, 28, 36, 37]. In effect, refactoring is an attempt to achieve a quality attribute requirement for the design – that of maintainability. In a design where the problem space is ill-defined and complex, Fowler suggests that, as the chances of achieving an optimal design solution on the first iteration are slim, the maintainability quality attribute is best achieved after any design misfits become apparent, i.e. after the initial design is complete.

Some authors have attacked evolutionary approaches as unsystematic and undisciplined, citing lack of layout diagrams and software architecture as evidence of lack of software design. Citing empirical studies, Rosenberg and Stephens [40] cast doubt on the success of evolutionary design in the production of large scale software products, and also weaker design practices such as ‘oral documentation’ in eXtreme Programming. However, while posing the question ‘Is design dead?’, Fowler [41] suggests a distinction between planned design and evolutionary design. Fowler proposes that planned design represents the systematic, disciplined element of software design where solution decomposition and variant generation occurs. When the problem space is large, formally produced layout diagrams of the initial design solution are necessary to enable the organisation of development and production of the solution. However, when the problem space is complex, ill-structured and driven by requirements that may change and evolve in the light of a better understanding of the problem space as the solution emerges, the nature of software design becomes more evolutionary, i.e. focusing of incremental design with refactoring. Fowler concludes that software design is indeed alive in both planned and evolutionary incarnations, and suggests that evolutionary design, with its emphasis on solution evaluation, is a plausible strategy for software design.

One limitation of evolutionary design is pointed to be van Gorp and Bosch [42] who suggest the phenomenon of design erosion. Van Gorp and Bosch report that, no matter how thorough the initial design, software solution designs tend to erode over time with evolutionary design in the sense that refactoring design decisions accumulate to the point of becoming suboptimal and invalid against the full range of design criteria. When such a point is reached, van Gorp and Bosch suggest that the only viable solution is a step change, and the introduction of a completely fresh solution variant to replace the current solution design. Thus, according to van Gorp and Bosch, evolutionary design is limited to a period of viability dependent on rate of change of design constraints.

### 3.7 Similarities and dissimilarities

Overall, there appears to be considerable common ground between engineering design as typified by Pahl and Beitz [16] and software engineering design as typified by USDP [28]. This is not entirely unexpected when the original objectives of software engineering of 35 years ago are taken into account. Design activities of both fields may be approximately mapped: product planning to requirements, conceptual design to analysis, embodiment design to design, and detail design to implementation. Both engineering design and software engineering clearly differentiate between problem and solution spaces, and offer techniques and representations for exploring, bounding and structuring those spaces. Both fields rely on two categories of

requirements: behavioural objectives and quality constraints. Both also depend heavily on technological and economic constraints as success criteria. Furthermore, both fields are also characterised by a diverse range of branches or areas of application; in engineering design from mechanical and electronic to chemical and process engineering; in software engineering from real-time and safety-critical to business domains and games software. Because of the diverse range of application, both fields exhibit a degree of flexibility in design approach appropriate to the product being designed. These findings are consistent with Diaz-Herrera [43], who examines a number of definitions of engineering and shows their correspondence with software development in order to define the software engineering profession and its relation to computer science. Diaz-Herrera compares the Pahl and Beitz engineering design approach with a ‘waterfall’ software development life-cycle and finds much similarity. Diaz-Herrera suggests these similar features that cut across all engineering disciplines are found in software engineering as well. An additional aspect common to engineering design and software engineering is the application of an evolutionary design process. Changing the internal design of an engineering product after production is evident particularly for novel, creative products constructed from innovative materials where product failure is a distinct possibility. In such conditions engineering design becomes inherently evolutionary in a manner analogous of design re-factoring in software engineering.

However, there are some significant aspects of divergence between engineering design and software engineering. Firstly, in terms of design theory, engineering design views the whole process as ‘design’, whereas software engineering views the whole process as ‘development’ underpinned by distinct workflows such as requirements, analysis, design, implementation and test. It is interesting to speculate as to why software engineers chose to depart from engineering design in their terminology, as this might suggest a departure in underlying theory. It appears that the inventors of the original software engineering life cycle model where the terminology was introduced were influenced by systems theory (as indeed was engineering design). It is also possible that the inventors named the development process to reflect the more abstract nature of software. However, it is more likely that since 1968, both fields have independently evolved in parallel and arrived at their terminologies in the light of emerging theory and practice. In software engineering, the visualisation and representation of software designs has proved difficult theoretically and has led to varying terminology within varying methodologies.

Secondly, in terms of design practice, engineering design has much mature advice and guidance concerning solution variant generation and evaluation within the solution space. Despite some comparatively recent work on software architecture, software engineering remains weak in this area. The difference in guidance and advice provided by engineering design and software engineering is illustrated by the checklists available in both disciplines. Engineering design offers checklists throughout the design process across a variety of engineering products [44–46], whereas software engineering typically offers checklists for inspection and review [47–49]. Due to the inherent difficulties of representation and visualisation of the software product, mature advice and guidance in the form of proven techniques and checklists across the development life-cycle are less readily available in software engineering, reflecting the immaturity of the field. Building on its mature

guidance, engineering design allows the rejection of suboptimal solution variants at conceptual and embodiment design phases, before production. In software engineering design, rejection of suboptimal design variants is more problematic, as mature advice and guidance is less readily available.

Perhaps because of this weakness (the paucity of software design evaluation mechanisms), evolutionary design approaches have emerged in software engineering in the past five years. Within software engineering design, however, evolutionary approaches are paradoxical: do evolutionary approaches represent a systematic, disciplined engineering approach despite their apparently informal and idiosyncratic approach? The debate remains on-going: proponents [36–39] advance the view that solution evaluation with user community feedback is a more systematic way to economically achieve reliable software; opponents [40] argue that the typical lack of layout diagrams of objects and components and software architecture in evolutionary design is inherently a breach of the original aims of software engineering, i.e. it is informal and idiosyncratic.

Thirdly, it is interesting to draw this section to a close by reflecting on how far the dissimilarities between engineering design and software engineering are due to the nature of the product being engineered. Mechanical, chemical and electrical products, and buildings, for example, are technical artefacts that may be measured in dimensions suitable for representation as functions with input and output variables consistent with systems theory. Analysis, synthesis and evaluation of the product under design is enabled by the tangible, quantifiable nature of the product. Engineered products may be visualised and later manufactured as assemblies of technical components and subcomponents. Software, on the other hand, is intangible and abstract during analysis and synthesis, leading to difficulties in representation and visualisation (although significant progress has been made in this aspect by USDP), and especially in evaluation. Software analysis and design is also dimensionless (in the sense that it cannot be absolutely quantified), adding to difficulties in evaluation. Software products are also composed of assemblies of components and objects (software ‘building blocks’) but such hierarchies are often very complex in terms of multiple state changes affected within components, and articulations across component interfaces. Both engineering design and software engineering arose from systems theory and a desire to impose a systematic and disciplined approach to design. However, while the design process of both engineering design and software engineering shows fundamental similarities in terms of the design process (i.e. analysis, synthesis and evaluation), significant dissimilarities also exist due to differences in the nature of the entities being designed.

#### 4 Current software engineering practice

In addressing the extent to which the adoption of disciplined, systematic methods in software engineering has been a success, it is useful to examine the empirical evidence of software engineering design practice to evaluate the success or failure of software engineering projects, and validate the underlying engineering theories. Unfortunately, empirical evidence of current software engineering design practice is not widely available. Reasons for this may be many: software engineers may have insufficient time to record the success or otherwise of their project for posterity;

others may be understandably reticent to publish findings that place software production in an unflattering light.

Nevertheless, taking the available empirical evidence of software engineering design currently available in the literature, a number of themes emerge that illuminate the extent to which software engineering design may be considered successful. The first theme concerns the empirical evidence of the success or failure of software engineering projects. The second theme concerns studies revealing differences between software engineering theory and practice. Lastly, the third theme concerns the diversity of applications of software engineering.

##### 4.1 Success or failure?

Unfortunately, empirical evidence of the success or failure of software engineering in practice is not widely available in research literature. On the one hand, successful software designs are self-evident in the world around us: compared to software systems of 1968, modern software is several orders of magnitude more reliable and its functionality underpins much modern business and economic infrastructure. On the other hand, there are many anecdotal sources suggesting a mixed picture of success. Flowers [50] discusses a number of large scale software design projects in the UK public and private sectors that were abandoned before delivery or cancelled shortly after delivery. Yourdon offers survival guidance for impossible projects, entitled ‘Death march’ [51]. Glass [52] reveals numerous instances of commercial organisations in the USA attempting to follow a systematic, disciplined design process, but failing in the sense of commercial or product failure. It is difficult to draw conclusions on the success or failure of software engineering design, as software engineering projects may fail for many reasons, e.g. changing requirements, technical implementation issues, or organisational problems. Available evidence, at face value, suggests a mixed picture.

##### 4.2 Differences between software engineering theory and practice

As software engineering is a relatively new field, it is expected that new theories of software engineering will be continually emerging. However, there is evidence to suggest that evaluation of the emerging theories is less rigorous than might be expected. Zerkowitz and Wallace [53] examined 600 papers in the computer science community to determine how well the community was succeeding in validating its theories. They observed that up to 20% of papers in some journals had no validation component for the proposed theory, either experimental or theoretical.

In a different survey of the software engineering research literature, Glass *et al.* [54] examined 369 papers over 5 years in 6 leading research journals in the software engineering field. Although the survey is limited by the choice of journal and selection of papers, a number of interesting aspects emerge. In terms of the most frequent research topic occurring in the literature, design concepts (such as software representation and visualisation) hugely dominated over problem domain specific topics. Furthermore, in terms of research approaches, Glass *et al.* note that ‘the dominant categories had to do with conceptual analysis and concept implementation. Perhaps surprisingly, there were very few instances case/field studies, simulations, or mathematical proofs.’ It would appear that software engineering research literature is concentrating on conceptual formulations of software rather than evaluative studies of the proposed concepts. Given the heavy conceptual bias of software engineering research, Glass *et al.* conclude that

'there is a severe decoupling between research in the computing field and the state of practice in the field. This is particularly problematic in the software engineering field. Studies have shown that some technical advances are ignored almost entirely, and others take, on average, 15 years to move from initial discovery to general practical usage.'

Glass's finding concerning this time taken for technical advance to have an impact is generally true for all technical disciplines, however, and is not just confined to software engineering.

Evidence that the theory of software engineering is decoupled from its practice emerged as early as 1986, when Parnas and Clements [55] suggested that designers should 'fake' the theoretical, rationale design process. Parnas and Clements put forward the notion that a rational, systematic software design process will always be an idealisation. They offer several reasons to explain this, including the impossibility of complete and comprehension requirements elicitation, the inability of human beings to comprehend the complexity and plethora of design detail, and continual changes in design context. However, Parnas and Clements advocate attempting to follow a rational design process (suggesting an example). They then proceed to offer guidance as how to 'fake' the documentation as if the designers had followed the process. One possible limitation of the paper, however, is that although Parnas and Clements offer suggestions as to what deliverables of a 'fake' process might look like, they offer no description of what the 'fake' design process is.

Further evidence of the decoupling of software engineering theory and practice was presented by Fenton in 1993 [56]. Fenton analysed the benefits claimed by the proponents of a number of systematic, disciplined software methodologies available at that time, but found that the claims were rarely backed up by hard evidence. Fenton states that 'In some cases where empirical evidence does exist, the results are counter to the anecdotal views of the experts. Thus the software industry places trust in unproven, often revolutionary, methods.'

One area that might be expected to illuminate the extent of the theory/practice decoupling would be use of design guidance and checklists provided by methodologies. Unfortunately, no empirical evidence exists in the research literature to evaluate the usefulness or extent of use of checklists in either software engineering or engineering design.

Where empirical evidence does exist, it confirms the lack of evaluation of software engineering theory as espoused, for example, in structured analysis and design methodologies [24, 25, 29]. Structured analysis and design methodologies emphasise the concept of the flow of data through the software system, and lay down a sequence of activities and technique for the design of a software product. However, in a field study of its use in 1996, Wastell [57] reports a project that adopted SSADM as its design methodology 'to embody a rigorous engineering approach to system development.' As the software engineers being studied were encountering the methodology for the first time, their adherence to the methodology was strict. However, the study reveals that 'far from facilitating the development process, SSADM encouraged a rigid and mechanical approach in which the methodology was applied in a ritualistic way which inhibited creative thinking.' As the design process became more problematic, the software engineers adhered to the methodology theory more slavishly, only making the design process more

problematic. Wastell suggests that the software engineers being studied abdicated responsibility for design decision to the methodology in 'a fetish of technique', rather than solving the design problem to hand.

On the other hand, in a field study of the practice of five different software engineering methodologies, Fitzgerald [58] reports a 'wide difference between the formalised sequence of steps and stages prescribed by the methodology and the methodology in action uniquely enacted for each development project.' Fitzgerald goes on to suggest that 'developers omit certain aspects of methodologies not from a position of ignorance, but from the more pragmatic basis that certain elements are not relevant to the development environment they face.' Such findings suggest that the theory of the methodology was at best being practically adapted by the software designer, and at worst ignored. It is interesting to speculate on the reasons why one field study noted the slavish adherence to a methodology while another observed departure. It seems reasonable to suggest that design skill levels are a causative factor. Novice designers with shallow understanding of design might adhere to a methodology more closely than expert designers who employ a deeper understanding of design to interpret what they perceive the design process is trying to achieve.

Nandhakumar and Avison [59] report the findings of a field study of design in a large organisation where software engineering 'methodologies are treated primarily as a necessary fiction to present an image of control or to provide a symbolic status, and are too mechanistic to be of much use in the detailed, day-to-day, organisation of system developer's activities.'

In a further field study of 128 software developers in a large organisation using a software engineering methodology, Reimenschneider *et al.* [60] report an active resistance by software designers to the adoption of the methodology. Such was designer resistance to the design methodology that Reimenschneider *et al.* report that adoption was enforced by organisational mandate, but only where a degree of compatibility already existed between the methodology and how developers actually performed their work.

In a further recent field study, Barry and Lang [61] surveyed 100 organisations developing multi-media solutions, and compared the design methodology practised with a further 100 organisations developing 'traditional' solutions. Barry and Lang found that designers 'generally agree that systematic approaches are desirable in order to beneficially add structure to development processes, but are predominantly using their own in-house methods rather than those prescribed in the literature.' The authors found not only that no single method was used by software designers for either 'traditional' or multimedia systems development, but also that the software designers failed to use methods widely available in the literature (e.g. USDP [28]). Barry and Lang speculate that such widely available methodologies might be perceived by software designers to be neither sufficiently proven nor universally applicable.

Reasons for the decoupling of current software engineering theory and practice are not easily identified. One speculation is that modern software engineering methodology theory has concentrated on the conceptual representation and visualisation of software at the expense of recognising and exploiting the cognitive problem solving process evident in software designers. Empirical evidence to support this view of software design as cognitive problem solving is provided in a much cited paper of Guindon [62].

In a study of how six expert designers attacked a lift controller design problem, Guindon noted that the designers

found the problem specification incomplete and ambiguous, with no definite criteria to evaluate whether a solution is reached, and suggested the term 'no stopping rule'. The designers therefore set about elaborating the requirements, to decrease incompleteness and ambiguity. This included logically inferring new constraints on the solution which necessarily or possibly follow from the initial requirements, and the addition of new, desirable but optional, inferred requirements. The constraints served to decrease the range of possible design solutions by acting as simplifying assumptions, while inferred requirements contributed to problem structuring.

Based on her empirical study, Guindon notes that

'...ill-structured problems, such as design problems, are characterised by the lack of stopping rule and by an incomplete specification of the goals. The designers in this study circumvented this problem by adopting, very early in the session, a small set of personalised criteria that guided the search for a solution and the selection on a satisfactory solution. We call this small set of criteria their preferred evaluation criteria.'

Guindon explains that

'Preferred evaluation criteria, if wisely selected, can effectively reduce the daunting complexity of the design process, without imposing constraints that are too binding on the design solution. However, the need to reduce the complexity of a problem may also mean that designers commit themselves to wrong choices that are difficult to recover from.'

These findings are consistent with Glass [63], who suggests that incomplete requirements are 'missing requirements', which constitute a large number of logical errors in the software design when eventually produced.

Guindon [62] also suggests that the software designers in the lift controller problem relied heavily on simulations of their design solutions, but these were shallow in the sense of being limited to one level of abstraction, and generating few solution variants. Furthermore, use of the preferred evaluation criteria constrained the generation of solution variants such that as soon as one solution variant (from typically two or three candidates at most) appeared satisfactory (as opposed to optimal), the designers proceeded quickly to detail design. In one sense this finding suggests ineffective design since, as Glass [63] points out, 'there is seldom one best design solution to a software problem'. Glass reinforces the notion of how difficult it is for designers to arrive at an 'optimal' solution. Typically there are so few variants generated by designers, who have limited evaluative techniques at their disposal – usually only mental simulations of how they think the solution variant might perform. Glass suggests that the complexity of the solution process is at the heart of the matter: complexity driven by inferred requirements and scale of the problem space.

Such empirical evidence suggests that the sequence of cognitive problem-solving design activities may be at variance with the sequential and step-wise activities typically mandated by software engineering methodologies.

In a further paper, Guindon [64] described the nature of software design as essentially opportunistic, where software designers exploit opportunistic thoughts via simulations, rather than any systematic search of the solution space. Such findings are confirmed by Curtis [65], who suggested broadening the scope of software design research to how aspects of cognitive psychology may improve software design productivity and quality. Although a comprehensive

discussion of the cognitive psychology of software engineering design is beyond the scope of this paper, it is useful to briefly examine the mental thought processes of design to see if this sheds light on the decoupling of software engineering theory and practice.

For a specific, finite design problem, logical reasoning enables the identification of an evaluated design solution. The nature of such logical reasoning has recently been surveyed by Perkins [66]. Perkins begins by providing much empirical evidence to support the notion that, by and large, people can and do reason according to standard logic. Perkins goes on to examine the question of by what mechanisms standard logic is carried out. Two theories are advanced: rule theories and mental models.

Rule theories have been advocated by several investigators, notably Rips [67]. Rule theories suggest that logic might apply in the following (simplified) manner:

'If X, then Y. X, so Y'

The inference proves easy when the rule happens to exactly match its need. However, Johnson-Laird and Byrne [68] propose a different conception of logical reasoning via mental models in which reasoning might follow three stages:

- *Comprehension*: reasoners use their knowledge to understand the premises. They construct an internal model of the state of affairs that the premises describe, resulting in perception dependent models.
- *Description*: reasoners try to formulate a parsimonious description from the models. (Where no putative conclusion arises, nothing follows from the premises.)
- *Validation*: reasoners now try to search for alternatives of the premises in which the putative conclusion is false. If there is no such model, the conclusion is valid. However, if there is a model, reasoners return to the second stage of the process to search for further examples to validate.

As Johnson-Laird and Byrne point out, 'Because the number of mental models is finite[...], the search can in principle be exhaustive. If it is uncertain whether there is an alternative model to the premises, then the conclusion can be drawn in a tentative and probabilistic way.'

Perkins attempts to evaluate which mechanism better describes human reasoning, but, based on much empirical evidence, concludes that human beings use different mechanisms in different circumstances. Where relevant experiential knowledge is readily available in the form of a rule applicable to the problem context, it seems likely that application of the rule will be an effective reasoning mechanism. Jefferies *et al.* [69] provide evidence to support this view. Among many aspects of the design process, they investigated differences between the reasoning of experts and novices. They found that experts were able to use rules based on past experience to derive many candidate solutions, while novices, having little or no past experience, were hard pressed to reason even one solution to many problems. However, when little or no relevant experiential knowledge is available, or when the problem is novel, ill-defined, complex and constrained, it seems likely that mental models will provide the more effective reasoning mechanism, providing a mental framework for the generation and evaluation of design alternatives.

Based on this empirical evidence, it is not possible to conclusively identify the causes of the decoupling of software engineering theory and practice. However, it is interesting to speculate that the mismatches in activity sequencing of software engineering methodologies and

cognitive problem-solving activities may contribute, at least in part, to such decoupling as it exists.

### 4.3 Diversity of software engineering applications

In a recent study of practice of software engineering design, Avison and Fitzgerald [70] note a huge diversity in design approaches used by practitioners currently. Software engineering methodologies may be formal or evolutionary, sequential or incremental and iterative, or may be contingent on the criticality, available resource, designer skill levels and time to market of the product solution. As Avison and Fitzgerald found that in recent years, 'For some organisations, the problem is not the concept of a methodology but the inadequacy of current methodologies, prompting them to keep looking for different and better ones. Still others seek, not better methodologies, but alternatives to traditional in-house systems development'. Thus Avison and Fitzgerald further suggest that although software engineering methodologies claim to be appropriate for a wide variety of design domains, designer adaptation of the methodology in question is a significant factor. It seems possible that the inflexibility of methodologies to adapt to various domains may also contribute to the decoupling of design theory and practice.

## 5 Conclusions

Software engineering was born 35 years ago to establish the use of sound engineering principles to obtain economically viable software that is reliable and works efficiently on real machines. In comparison with established engineering design methodologies, e.g. Pahl and Beitz [16], modern mainstream software engineering design methodologies, e.g. Jacobson [28], have much in common. Both use the concepts of problem space and solution space, and rely on two categories of requirements: behavioural objectives and quality constraints. Engineering design and software engineering also have comparable 'phases,' i.e. product planning/requirements, conceptual design/analysis, embodiment design/design, detail design/implementation. Both depend on technical and economic constraints as success criteria and both are characterised by a flexibility that enables their application across a diverse range of problem domains. Empirical evidence also suggests that both have been applied to a diverse range of applications, resulting in diverse varieties of design processes, including evolutionary design.

However, engineering design and software engineering also differ in a number of ways. Firstly, terminology differs. Engineering design views the entire process as design, whereas software engineering views design as the phase in which the logical statement of the problem space is transformed into physical solution variants and their evaluation. In terms of practice, engineering design is more mature than software engineering design. This maturity is evidenced not only by the greater period of time that engineering design practice has been refined, but also from the widely available engineering design checklists and guidance, e.g. in Pahl and Beitz [16]. However, possibly the most significant divergence between the two lies in solution variant generation and evaluation. Engineering design offers many techniques for generating solution variants and evaluating those variants prior to production; software engineering is weaker in this aspect. Furthermore, in software engineering, support for solution variant generation is absent. While techniques exist in software engineering for evaluating development artefacts during

analysis and design, evaluation is most effective when solution performance is acceptance tested with the user community. Compared to engineering design, such evaluation across the development cycle is less mature. While some promising progress has been made recently in the evaluation of software architecture [34], research in this area is on-going.

Has the disciplined, systematic approach of engineering design been a success or failure in software engineering? Despite there being little definitive evidence in the research literature, available evidence suggests a mixed picture in software engineering projects. Both literature surveys and empirical evidence suggest a decoupling of software engineering theory and practice, the causes of which are difficult to identify. However, it is interesting to speculate that mismatches between activity sequencing of software engineering methodologies and cognitive problem-solving activities may contribute, at least in part, to such decoupling as it exists.

Difficult software design issues that were noted in 1968 remain. Some (e.g. incremental development, user involvement, writing tests before design, component assembly) have even been 'rediscovered' in recent years. Furthermore, despite significant progress in standardising design methodologies (e.g. Jacobson *et al.* [28]), software engineering has struggled to address the inherently abstract nature of software. Consequently, difficulties remain with scoping and structuring ill-structured problem spaces due to issues of problem representation and visualisation, with consequent communication difficulties. Software engineering has also failed to contain the inherent complexity arising from 'inferred' requirements (those that arise from initial solution variant generation) which has led to confusion as to how the software design process may effectively and efficiently iterate. Both the theory and practice of what constitutes a desirable software engineering design process remains poorly understood.

In a discussion of software engineering and society in 1968 [1], Kolence suggests that 'the basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time.' Empirical evidence of software engineering projects suggests this crucial issue remains valid 35 years after it was stated. It appears that as fast as software engineering makes progress, so the demands made on it continue to increase beyond its capabilities.

Given the mixed success of software engineering, was it valid for software designers to attempt to emulate their engineering design counterparts? Was it suitable to impose a systematic, disciplined approach on a poorly understood design process when the nature of the software product was abstract? In 1968, systems theory offered the basis of a systematic approach to software development while engineering design was put forward as a mature (if not shining) example of design. However, since the abstract nature of software mandated novel adaptations of systems theory, an immediate emulation of systematic, disciplined software engineering design could not have realistically been expected. On the other hand, the mature fundamentals of design, common to numerous branches of engineering, proved to be an effective starting point for a field in its infancy.

In the future, it appears likely that the success of software engineering will be enhanced by resolving the decoupling of software engineering theory and practice. Improved knowledge of software engineering design processes may contribute to this, as might an enhanced understanding of the impact of a diverse range of design applications. It is

suggested that software engineering could learn lessons from engineering design in the area of solution variant generation and evaluation, and in time develop the guidance and checklists analogous to those available to practitioners in engineering design.

This comparison of engineering design and software engineering has been carried out as part of on-going research into supporting and enhancing software engineering analysis and design by the use of interactive evolutionary computing approaches within complex and ill-structured software design domains. Future work will investigate the introduction of Parmee's interactive evolutionary design strategies [19–21] to software engineering.

## 6 References

- 1 Naur, P., and Randell, B. (Eds.): 'Software engineering'. Proc. Conf. sponsored by NATO Science Committee, NATO, Garmisch, Germany, 1968
- 2 Alexander, C.: 'Notes on the synthesis of form' (Harvard University Press, 1964)
- 3 Suh, N.: 'The principles of design' (Oxford University Press, 1990)
- 4 Lawson, B.: 'How designers think – the design process demystified' (Oxford Architectural Press, 1997)
- 5 Simon, H.: 'The sciences of the artificial' (MIT Press, 1996)
- 6 Amarel, S.: 'On the mechanisation of creative design', *IEEE Spectr.*, 1996, (3), pp. 112–114
- 7 Do, E.: 'Drawing marks, acts and reacts: towards a computational sketching interface for architectural design', *Artif. Intell. Eng. Des. Anal. Manuf.*, 2002, **16**, pp. 149–171
- 8 Liu, Y.-C., Bligh, T., and Chakrabati, A.: 'Towards an "ideal" approach for concept generation', *Des. Stud.*, 2003, **24**, pp. 341–355
- 9 Norman, D.: 'The design of everyday things' (MIT Press, 1998)
- 10 Roozenburg, N., and Eekels, J.: 'Product design: fundamentals and methods' (John Wiley & Sons, 1995)
- 11 Parmee, I.: 'Evolutionary and adaptive computing in engineering design' (Springer-Verlag, 2001)
- 12 Navinchandra, D.: 'Exploration and innovation in design' (Springer-Verlag, 1991)
- 13 Goel, A.: 'Design, analogy, and creativity', *IEEE Expert Intell. Syst. Appl.*, 1997, **12**, pp. 26–70
- 14 Cross, N.: 'Engineering design methods' (John Wiley & Sons, 1989)
- 15 French, M.: 'Conceptual design for engineers' (Springer-Verlag, 1999, 3rd edn.)
- 16 Pahl, G., and Beitz, W.: 'Engineering design' (Springer-Verlag, 1996, 2nd edn.)
- 17 Pugh, S.: 'Total design: integrated methods for successful product engineering' (Addison-Wesley, 1991)
- 18 Reffat, R., and Gero, J.: 'Computational situated learning in design', 'Artificial intelligence in design '00' (Kluwer Academic Publishers, 2000), pp. 589–610
- 19 Parmee, I.C., Cvetkovic, D., Watson, A.H., and Bonham, C.R.: 'Multi-objective satisfaction within an interactive evolutionary design environment', *Evol. Comput.*, 2000, **8**, pp. 197–222
- 20 Parmee, I.C.: 'Improving problem definition through interactive evolutionary computing', *Artif. Intell. Eng. Des. Anal. Manuf.*, 2002, **16**, pp. 185–202
- 21 Parmee, I.C., Cvetkovic, D., Watson, A.H., Bonham, C.R., and Packham, I.: 'Introducing prototype interactive evolutionary systems for ill-defined design environments', *J. Adv. Eng. Softw.*, 2001, **32**, pp. 429–441
- 22 Apperley, H., and Simons, C., *et al.*: 'Service- and component-based development' (Addison-Wesley, 2003)
- 23 Booch, G.: 'Object-oriented analysis and design with applications' (Benjamin Cummings, 1994)
- 24 DeMarco, T.: 'Structured analysis and system specification' (Prentice-Hall, 1979)
- 25 Eva, M.: 'SSADM version 4: a user's guide' (McGraw-Hill Education, 1994)
- 26 Firesmith, D., and Henderson-Sellers, B.: 'The OPEN process framework: an introduction (OPEN series)' (Addison-Wesley, 2001)
- 27 Jackson, M.: 'Principles of program design' (Academic Press, 1975)
- 28 Jacobson, I., Booch, G., and Rumbaugh, J.: 'The unified software development process' (Addison-Wesley, 1999)
- 29 Yourdon, E.: 'Structured design: fundamentals of a discipline of computer program and system design' (Prentice-Hall, 1979)
- 30 Pressman, R.: 'Software engineering, a practitioner's approach - European adaptation' (McGraw-Hill, 2000, 5th edn.)
- 31 Sommerville, I.: 'Software engineering' (Addison-Wesley, 2001, 6th edn.)
- 32 Shaw, M., and Garlan, D.: 'Software architectures: perspectives on an emerging discipline' (Prentice Hall, 1996)
- 33 Bosch, J.: 'Design and use of software architectures: adopting and evolving a product-line approach' (Addison-Wesley, 2000)
- 34 Clements, P., Kazman, R., and Klein, M.: 'Evaluating software architectures: methods and case studies' (Addison-Wesley, 2002)
- 35 Sigman, S., and Liu, X.: 'A computational argumentation methodology for capturing and analysing design rationale arising from multiple perspectives', *Inf. Softw. Technol.*, 2003, **45**, pp. 113–122
- 36 Beck, K.: 'Extreme programming explained: embrace change' (Addison-Wesley, 2000)
- 37 Cockburn, A.: 'Agile software development' (Addison-Wesley, 2002)
- 38 Schwaber, K., and Beedle, M.: 'Agile software development with scrum' (Prentice Hall, 2002)
- 39 Fowler, M.: 'Refactoring: improving the design of existing code' (Addison-Wesley, 1999)
- 40 Rosenberg, D., and Stephens, M.: 'XP refactored' (Addison-Wesley, 2003), in press
- 41 Fowler, M.: 'Is design dead?' *Softw. Dev. Mag.*, 2001, **9**, (4), available at <http://www.sdmagazine.com>
- 42 Van Gorp, J., and Bosch, J.: 'Design erosion: problems and causes', *J. Syst. Softw.*, 2002, **61**, pp. 105–119
- 43 Diaz-Herrera, J.: 'Engineering design for software: on defining the software engineering profession', Proc. 31st IEEE Annual Frontiers in Education Conf. on Impact on Engineering Science and Education, vol. 1, Reno, NV, 10–13 October 2001, pp. 685–688
- 44 Wimmer, W.: 'The ECODESIGN checklist method: a redesign tool for environmental product improvements'. Presented at Int. Symp. on Environmentally Conscious Design (EcoDesign) and Inverse Manufacturing, Tokyo, Japan, 1–3 February 1999
- 45 Hendershot, D.: 'A checklist for inherently safer chemical reaction process design and operation'. Presented at 17th Annual Int. Conf. of American Institute of Chemical Engineers, Jacksonville, FL, October 2002
- 46 Brandstotter, M., Knoth, R., Kopacek, B., and Kapacek, P.: 'Checklist for optimised design for re-use of printed wire boards and components'. Presented at IEEE Int. Symp. on Electronics and the Environment, May 2003
- 47 Belli, F., and Crisan, R.: 'Towards automation of checklist-based code reviews'. Presented at 7th Int. Symp. on Software Reliability Engineering, White Plains, NY, 30 October–2 November 1996
- 48 Miller, J., Wood, M., and Roper, M.: 'Further experiences with scenarios and checklists', *Empir. Softw. Eng.*, 1998, **3**, pp. 37–64
- 49 de Almeida, J., Jr., Camargo, J., Jr., Basseto, B.A., and Paz, S.M.: 'Best practices in code inspection for safety-critical software', *IEEE Softw.*, 2003, **20**, pp. 56–63
- 50 Flowers, S.: 'Software failure, management failure' (Wiley, 1996)
- 51 Yourdon, E.: 'Death march' (Prentice-Hall, 1999)
- 52 Glass, R.: 'Computing calamities: lessons learned from products, projects and companies that failed' (Prentice-Hall, 1999)
- 53 Zalkowitz, M., and Wallace, D.: 'Experimental validation in software engineering', *Inf. Softw. Technol.*, 1997, **39**, pp. 735–743
- 54 Glass, R., Vessey, I., and Ramesh, V.: 'Research in software engineering: an analysis of the literature', *Inf. Softw. Technol.*, 2002, **44**, pp. 491–506
- 55 Parnas, D., and Clements, P.: 'A rational design process: how and why to fake it', *IEEE Trans. Softw. Eng.*, 1986, **12**, pp. 251–257
- 56 Fenton, N.: 'How effective are software engineering methods?'. Proc. 2nd Int. Conf. on Achieving Quality in Software (AQUIS), Venice, Italy, 1993, pp. 295–305
- 57 Wastell, D.: 'The fetish of technique: methodology as a social defence', *Inf. Syst. J.*, 1996, **6**, pp. 25–40
- 58 Fitzgerald, B.: 'The use of systems development methodologies in practice: a field study', *Inf. Syst. J.*, 1997, **7**, pp. 201–212
- 59 Nadhakumar, J., and Avison, D.: 'The fiction of methodological development: a field study of information systems development', *Inf. Technol. People*, 1999, **12**, pp. 176–191
- 60 Reimenschneider, C., Hardgrave, B., and Davies, F.: 'Explaining software developer acceptance of methodologies: a comparison of five theoretical models', *IEEE Trans. Softw. Eng.*, 2002, **28**, pp. 1135–1145
- 61 Barry, C., and Lang, M.: 'A comparison of 'traditional' and multimedia information systems development practices', *Inf. Softw. Technol.*, 2003, **45**, pp. 217–227
- 62 Guindon, R.: 'Knowledge exploited by experts during software system design', *Int. J. Man-Mach. Stud.*, 1990, **33**, pp. 279–304
- 63 Glass, R.: 'Facts and fallacies of software engineering' (Addison-Wesley, 2003)
- 64 Guindon, R.: 'Designing the design process: exploiting opportunistic thoughts', *Hum.-Comput. Interact.*, 1990, **5**, pp. 305–344
- 65 Curtis, B.: 'Insights from empirical studies of the software design process', *Future Gener. Comput. Syst.*, 1992, **7**, pp. 139–149
- 66 Perkins, D., *et al.*: 'Standard logic as a model of reasoning: the empirical critique', in Gabbay, D., (Ed.): 'Handbook of the logic of argument and inference: the turn towards to the practical' (Elsevier Science, 2001)
- 67 Rips, L.: 'The psychology of proof: deductive reasoning in human thinking' (MIT Press, 1994)
- 68 Johnson-Laird, P., Byrne, R.: 'Deduction' (Lawrence Erlbaum Associates Limited, 1991)
- 69 Jefferies, R., Turner, A., Polson, P., and Atwood, M.: 'The processes involved in designing software', in Anderson, J., (Ed.): 'Cognitive skills and their acquisition' (Hillsdale, 1981)
- 70 Avison, D., and Fitzgerald, G.: 'Where now for development methodologies?' *Commun. ACM*, 2003, **46**, pp. 79–82