What Application Designers Know: Some Thoughts on Innovation in Interactive Systems

William Newman

Rank Xerox Research Centre, Cambridge UK

Abstract

This paper takes as a starting point Walter Vincenti's seminal book, "What Engineers Know and How They Know It," and explores the implications for interactive system design of some of his observations on engineering knowledge. In particular, it discusses the obstacles designers face in accessing relevant knowledge, and their consequent difficulties in engaging in design. The paper suggests that a crucial step in building knowledge about interactive system design knowledge is the establishment of *critical parameters* concerning the human activity that the application supports. It concludes with a discussion of the implications for innovative system design.

1 Introduction

Those who design and build software systems for direct use by people and organisations play an increasingly important role in society. These are, almost universally, *interactive systems* that are operated directly by their users via the system's user interface. Their designers must try to respond to users' demands for improved services and productivity by developing new applications. They must be prepared to follow up, when their designs are successful, with more powerful and reliable systems that keep pace with users' expanding needs. There is constant pressure on the designers of interactive systems to do more and to do it better.

In this paper I am concerned, not so much with whether software engineers are doing a good enough job of designing interactive systems, but how in the first place they can acquire the knowledge they need to do their job. In effect, I take as my starting point a paraphrase of a remark of Walter Vincenti's [27]: *what software engineers do depends on what they know.* My interest is to understand what software designers need to know, where this knowledge is to be found, whether there are significant barriers to the growth of knowledge about interactive systems and, if so, what effects these barriers might have.

To address this topic, I have chosen to try to apply general theories about engineering design knowledge to the special case of software design. There are dangers in adopting this approach, for it is not unanimously accepted that software design can be treated as an engineering exercise [29]. However, I try in the course of this paper to provide evidence that my approach is a sound one, and that it is valid to talk about interactive system design as "software engineering."

I therefore begin by outlining a basic taxonomy of engineering design knowledge, and then provide some evidence that it applies to software design. When applied to the design of interactive systems it yields interesting results, especially when contrasted with the design of *embedded software*, i.e., the non-interactive systems and components that support interactive software. I conclude that it is particularly hard for designers of interactive systems to acquire the knowledge they need. I have some final comments on how this affects innovation in interactive applications.

2 A taxonomy of engineering design knowledge

As I have pointed out, the question of how growth in engineering knowledge occurs is in effect two questions: what kinds of knowledge do engineers depend upon, and what causes this knowledge to accumulate? My first task is therefore to answer the first sub-question, by enumerating some major categories of engineering design knowledge, and elaborating on the categories in discussing how they contribute to design. I then compare the taxonomy with the findings from studies of software designers' knowledge.

2.1 A taxonomy

As a first cut at the categorization of engineering knowledge, I offer the following five categories, drawn from a number of previous analyses of engineering design [24, 23, 27]:

1 Domain knowledge relating primarily to designed artefacts and their constituent technologies
2 Domain knowledge relating primarily to environments in which artefacts are used
3 Knowledge of representations, particularly those applied to describing aspects of the domain
4 Known techniques for analysing the design or for simulating the proposed artefact's behaviour
5 Knowledge of the critical parameters against which the artefact's performance is measured.

The list is not exhaustive. It deliberately omits two categories of knowledge that play roles in software design, namely knowledge of procedures and processes, and general knowhow and rules of thumb. I do not regard this omission as crucial to the arguments I will lay out here.

2.2 Fitting the taxonomy to descriptions of engineering design

Accounts of engineering design refer frequently to the five categories of knowledge listed above. Henry Petroski [21], for example, describes structural engineering design thus (my own category numbers in brackets): "As each hypothetical arrangement of parts [1] is sketched either literally or figuratively on the calculation pad or computer screen [3], the candidate structure must be checked by analysis [4]. The analysis consists of a series of questions about the behavior of the parts under the imagined conditions of use after construction [2]." In a similar vein, Rogers identifies the engineer's need "to determine the size and shape of a piece of equipment [1] to perform a specified duty [5], or to predict the performance of the design [4] when it is called upon to function under other operating conditions [2]" [23].

A particularly thorough treatment of design can be found in Herbert Simon's book *The Sciences of the Artificial*, in which he identifies roles for all of the taxonomy's components [24]. He places particular emphasis on the distinction between the "inner" environment of design—the substance and organization of the artifact itself [1]—and the "outer" environment in which the artefact operates [2]. He goes on to discuss designers' frequent difficulty in predicting how their designs will behave [4], the influence of choice of representation on design problem solving [3] and so-called "figures of merit" that permit comparison between designs [5].

2.3 How the taxonomy fits software design

Do these accounts of engineering design apply to the design of software? Or is software design fundamentally different in its knowledge requirements? Answers to these questions can be found in the studies of programmers and software designers by Adelson and Soloway [1], Visser [28], Guindon [12] and others. To a large extent the studies confirm that software designers do rely on the same five categories of knowledge and, in this respect, go about their work like other engineering designers.

For example, reliance on knowledge of existing designs can be seen in virtually every study of software design. Designers tend to start with a schema or model of the solution, which they construct from past experience or from a supplied solution. If the problem is familiar to them, the schema is likely to be well-formed and easily specified [1]. If the problem is unfamiliar, software designers are still likely to reuse an existing tested design, even though its adaptation to the new problem may be timeconsuming [12].

Software designers also rely on their knowledge of the environment of use. Like the engineers described by Petroski and Rogers, they apply their knowledge to constructing scenarios of use, which are then used in simulating the performance of the design. Guindon, for example, describes how designers of a lift control system created hypothetical configurations of the lifts for the purpose of testing designs [12]. Scenario-based simulation can be observed helping the designer to understand the behaviour of the system as a whole. Not surprisingly, designers who lack familiarity with the environment of use have difficulties constructing scenarios and are less able to simulate their designs' performance [1].

Knowledge of notations and other representations can also be seen contributing to software design. Experienced designers will often search among the notations familiar to them in order to choose one suitable for the problem [12]. If they are less experienced, they may go to the extreme of adopting a familiar notation to prepare the design, subsequently translating the finished design to the required notation [28].

2.4 Where the taxonomy falls short

There is plenty of evidence, therefore, of software designers' reliance on domain knowledge and on familiarity with representations; but what about techniques for analysis and simulation, and what about critical parameters? These kinds of knowledge are also in evidence, but not in great quantities.

The designers studied by Adelson and others showed no inclination to apply analytical techniques to the evaluation of their designs. Their simulation methods can only be regarded as primitive. They relied on simple inspections and scenario walkthroughs, whose purpose was just to check the functioning of the designs, and not to predict their performance. The almost total lack of attention to performance analysis is surprising, because programmers are known to take pride in squeezing speed improvements out of programs.

Software designers also showed little awareness of critical performance parameters. No "figures of merit" are mentioned in the studies; instead designers appear to adopt their own preferred evaluation criteria, such as reliability or simplicity. They use these to guide the choice of solution, but never apply any overall performance targets [12].

Again, the reason may lie in the designers' lack of experience with similar problems. Although presented with a precise problem definition, they mostly tended to continue elaborating on the stated requirements and constraints during design, illustrating software designers' well-known proclivity for "requirements drift." By continually adjusting the functional requirements for the system, they lost track of critical performance parameters as potential design targets.

3 Software designers' acquisition of knowledge

On the basis that it is valid to apply the knowledge taxonomy to software design, I will now look at the means available to software designers to acquire the knowledge they need. This is an exercise that has been carried out before, e.g., by Denning [7] and Curtis *et al.* [6]. However, these studies have not distinguished between the design of embedded software and interactive systems. Do designers in these two areas face the same problems in accessing information?

In answering these questions I have had to switch to a more subjective approach, because there is so little published information on how software designers acquire knowledge. I hope nevertheless to convince readers that the two problems of information access, concerning embedded software and interactive systems respectively, are significantly different problems, because of fundamental differences in the nature of the information.

3.1 Knowledge in support of embedded software design

What do the designers of embedded software need to know? And how can they come about this knowledge? To answer these questions I will look in turn at the taxonomy's five main categories of knowledge.

Domain knowledge about embedded software artefacts can be acquired from published accounts. Textbooks lay out a range of solutions, traditionally as coded algorithms but increasingly as object descriptions [9]. There are journals and conferences devoted to specific types of software systems, such as operating systems, distributed systems, image processing, computer graphics, databases, and so forth. Articles and papers describe new algorithms and compare them with past solutions. By reading this literature the software designer can gain familiarity with a range of designs, from which he or she can develop a schema for a new design problem. The fact that designs are presented as algorithms or patterns assists this process.

The same literature provides examples of the environments in which embedded software artefacts are used, because these environments are software systems too. They may not always be described to the same level of detail as are components, but they support the generation of scenarios for use in testing the design of the embedded component. A superficial knowledge of a CAD system will, for example, enable the designer of a graphics package to simulate the latter's performance in generating a display.

The representations needed in embedded software design are essentially those for the specification of software. They include programming languages, data representations, state-transition notations and the various other graphical notations offered by the software engineering methodologies [14]. Software designers get to know these representations early in their careers.

Techniques for the analysis of embedded software designs form a less explicit part of the designer's knowledge. There are, as I pointed out earlier, techniques for inspecting code and conducting walk-through analyses [2]. More rigorous methods of algorithmic analysis are also available, although empirical evidence suggests that these are used comparatively rarely. From these analyses the designer can gain a rough idea of the software system's performance. However, an alternative to analysis is always available, in the form of prototyping and testing. Through a combination of these, the designer is generally able to evaluate the design to an adequate level.

Finally, the critical parameters against which embedded software is tested appear to be widely understood. A number of universal parameters are applied to most software designs: speed of execution, reliability, use of resources. Speed, often the primary consideration, may be measured in terms of performance of standard benchmarks, or in terms of dimensions of the problem domain; thus speed may be proportional to n^2 or $\log(n)$. The existence of critical parameters that are tacitly agreed among designers, and localised to the embedded component in question, enables designers to set and achieve specific performance targets.

3.2 Knowledge in support of interactive system design

For the designer of an interactive system, acquisition of relevant knowledge is a very different matter. Again, the differences can be appreciated by looking at each type of knowledge in turn.

Like embedded systems, interactive systems are described in the literature. Furthermore, and unlike most embedded systems, they can sometimes be experienced first-hand, for example in the form of cash machines and on-line library catalogues; and some of them can even be purchased in computer stores for personal use. In a sense, therefore, knowledge about interactive systems is particularly accessible to designers. However, the availability of this knowledge is distinctly uneven: if the application cannot be purchased or accessed publicly, information about it is likely to be very hard to obtain. The "organisational" systems used by banks, police services, hospitals, government offices and military personnel are kept under wraps for the most part. Very few descriptions can be found in the literature, and then they are usually very superficial. Even the most celebrated organisational applications, such as the SAGE air defence system and the Sabre airline reservation system, have been documented to only a most perfunctory degree [8, 5]. As a result, designers' knowledge is steeply skewed towards those designs that are readily accessible.

The environments in which interactive systems are used are environments of human activity, altogether different from the software-systems environments in which embedded components are used. Learning about these environments is, again, relatively easy when access to the environments is itself easy. However, the difficulty here for the designer is to know what are suitable scenarios against which to test the design. It is tempting for the designer to say, "This is how *I* would use the system," but this carries a risk of applying a scenario that is weighted in the system's favour. Discovering patterns of behaviour in application environments is a specialised task, and few software designers have the skill or the time to carry it out. As a result there is a constant danger that interactive software will be designed on a basis of relatively superficial knowledge of the environment of use.

One of the barriers to disseminating domain information about interactive systems is the relative lack of suitable representations. The designs of interactive systems are difficult to describe because they involve many linked software components and have complex, dynamic user interfaces. The notations for describing such systems are less well developed, in comparison with the notations for algorithms and software objects. So even when details of interactive systems are published, weaknesses in the descriptive representations may prevent designers from finding out what they need to know. For example, when details of the Xerox

Star system were made public in a number of articles [13, 25, 26], the descriptions appear to have been inadequate for those who tried to replicate what Xerox had built [30].

Designers of interactive systems are poorly equipped with analytical tools. The techniques available to the embedded software designer do not apply here, because the problem is to analyse and predict the behaviour of an interactive system in the hand of its users. This is a complex domain of analysis, and there is a tendency for analysis to be timeconsuming and difficult to learn. Some simple methods have been developed, including Keystroke-Level Analysis and Cognitive Walkthroughs [3, 22, 17]. However, they do not appear to have achieved widespread use.

Perhaps the most pervasive problem for the designers of interactive systems is the lack of known critical parameters, analogous to the execution-speed parameters that govern the design of most embedded software components. In the case of specialised applications, such as cash machines, critical parameters do tend to be known within the organisations that develop them, but are unavailable to other designers. A well-documented example of this is the study, known as "Project Ernestine," of workstations for toll-and assistance operators, or TAOs [11]. In this application the time taken by the TAO to complete each call is a critical parameter in the workstation's design; however, the times are different for each type of call, and information about times and call-types is considered confidential by telephone companies [10]. Each application has its own critical parameters; in many cases, the research has not yet been done to establish what they are.

In summary, therefore, the design of interactive systems is significantly less well provided with essential information, and levels of knowledge in designers are bound to be lower as a result. Particular problems are patchiness in knowledge of existing designs, superficial knowledge about application environments, and an almost complete dearth of knowledge about the critical parameters that apply to individual applications.

3.3 Is this engineering?

With so many barriers in the way of acquiring knowledge, we might find it hard to view interactive system design as an engineering activity. After all, if engineering design is a knowledge-dependent activity, and the knowledge is unavailable, can engineering design be said to take place?

The answer lies, I believe, in the evidence that application designers try, in almost every case, to access the information they need to design and build systems in an organised way; and this organised use of available knowledge is the ultimate stamp of the engineer [23]. Even if they abandon the search for information in some areas, e.g., for information regarding existing solutions to the problem at hand, they will still attempt to search methodically for it in others. They will not start from scratch unless there is no alternative. I would claim, therefore, that we are discussing a design practice that exhibits the main features of an engineering activity. As such, we can compare it with other engineering practices and use them as a basis for understanding how interactive system design might develop into a stronger discipline.

3.4 The search for critical parameters

I have mentioned the problem of identifying critical parameters for interactive systems, and I will wrap up this section by expanding on this problem and suggesting how it might be attacked.

The essence of a critical parameter is that it provides a basis for quantifying a requirement. Vincenti quotes an example in his study of the development of requirements for aircraft flying qualities [27]. By means of extensive testing, engineers were able to isolate the parameter *stick force per* g *acceleration* as a basic determinant of maneuverability. Once this parameter had been identified, and accepted by the aircraft industry, it became a standard means of specifying flying qualities. Vincenti quotes figures of six pounds per *g* on fighter-type aircraft and up to 50 pounds per *g* on bombers and transports. He adds, "Aeronautical engineers today express amazement that any meneuverability criterion besides stick force per *g* ever existed." One of the hallmarks of engineering is, I claim, this tendency for critical parameters to become tacitly and universally accepted [18].

As I have said, critical parameters are different for each application. The parameter governing the design of TAO workstations—call-completion time—cannot be assumed to apply to calls to directory-assistance or ambulance-service operators [15]. The reason for this variation lies in the dependence of application-design parameters on the *supported activity*. If the activity changes, critical parameters may no longer apply.

In comparison with other fields of engineering, very little research has been done into establishing critical parameters for software systems, of any category. As regards embedded systems, the tendency has been to assume that the standard parameters—speed, reliability, use of resources, etc.—apply in all cases. In the design of interactive systems, the main concern has been to provide adequate functionality (where "adequate" can mean "more than the competitor's"). A secondary concern has been usability; but the establishment of an interactive system's usability often includes deciding what to measure, and different parameters—not necessarily critical to the application—may be selected on an arbitrary basis for each usability evaluation.

The difficulty of determining critical parameters for interactive systems should not be underestimated. The concept that human activity is sufficiently repeatable to allow measurement of recurring parameters may seem controversial; some might argue that such parameters cannot be found, or cannot be used as a basis for design. Certainly my own experience has been that deliberate attempts to find them often fail, and that the parameters sometimes emerge in the course of looking for something else [19]. The strongest arguments for their existence are that they have been detected in some applications, e.g., in TAO activities [11], and that people's ability to plan their activities suggests an ability to make their own estimates of how long the activities will take [20]. If we can tap into this tacit knowledge, we may be able to discover the critical parameters we need to know.

Difficult though critical parameters are to determine, there is a strong argument for trying to do so, because critical parameters lie at the root of the acquisition of software engineering design knowledge. They provide a basis for enhancing designs, for without them the decision as to whether enhancement has been achieved becomes arbitrary. They also provide a means of selecting scenarios for testing designs: if the scenario is couched in terms of critical parameters, e.g., it describes a task whose performance-time is critical, then it will provide a valid basis for testing. Critical parameters also help us to understand what kinds of analyses we need to carry out, and therefore what tools need to be developed. I would suggest, therefore, that if more attention is paid to the identification of these parameters, faster progress may be achieved towards providing interactive system designers with the knowledge they need to do their job. I will conclude with some remarks on this point.

4 Thoughts on innovation in interactive systems

To work in the computer field is to experience unceasing innovation on a massive scale. This is true whether one works in hardware, embedded software, interactive systems or some other related area. In a sense, innovations must proceed in tandem in all areas because advances in one area depend on advances in another.

What does it mean, then, if one area—interactive systems—experiences particular difficulties in acquiring essential knowledge? How can this area keep pace with the rest? I have worked in the interactive systems area for some years, and have meanwhile kept an eye on developments in other areas, and my sense is that innovations in interactive systems do *not* keep pace with the rest, in the true sense. Although novel ideas for interactive systems are generated at the same rate as in other areas, or even faster [16], the *innovation* process by which these ideas are brought into general use often falters.

Innovation is indeed a process, not an instantaneous "eureka"-type event. Ideas for new designs do not work perfectly first time, unless they are very minor enhancements to existing designs. The ideas for interactive systems that flood the computer business tend to be relatively radical. They offer totally new ways to do things, e.g., to access library books, to hold meetings, to carry out surgical operations. Ideas of this kind tend to bring with them a host of side-effects and performance problems, which need to be worked out over time. The goal of this innovative process is, as Constant has put it, to reduce the original "radical technology" to a generally accepted and established "normal technology" [4]. During the innovative process, designers and users alike are motivated by the prospect of gaining an advantage over the previous technology. Thus reading books on-line should be advantageous in comparison with reading them on paper, holding a videoconference should offer advantages over holding it face-to-face, remote surgery should be preferable, at least some of the time, to traditional surgery.

As I have pointed out, knowing whether we have improved on an existing design involves knowing the critical parameters for that design problem; and in the design of interactive applications the critical parameters are generally unknown. There is also a need to know about existing designs, and about techniques for analysing and predicting their performance. The lack of these forms of knowledge has two undesirable effects.

First, *radical ideas may be pursued when existing solutions could easily be improved*. A new idea for an interactive system has an attraction all of its own; because it is new, it generates interest in its further development. But it may not offer any real advantages over existing systems. There may be simple ways of enhancing an existing design. We may not know enough about the existing design to tell. Unless we know the critical parameters against which to assess the two approaches, and are familiar with the other design, we cannot make an informed choice here.

Second, *the innovative process may degrade rather than improve performance*. Innovation involves a long series of design changes. Since the critical parameters are unknown, there is no way to tell whether the design is being changed for the better. Even if they are known, analytical techniques may be inadequate, as in the case of Project Ernestine [10]. What can designers do in these circumstances? One common recourse is to add features, because these are seen as "improvements" in the competitive sense. Interactive systems thus gradually become more feature-rich and resource-demanding; but do they offer improved support to their users?

There is a solution to these problems, in the form of research to build the kinds of knowledge that I have suggested is so hard for interactive system designers to acquire. Foremost among these is to establish the critical parameters of interactive applications. In tandem, work is needed to develop analytical models for similating and predicting the outcomes of design. Efforts need to be made to document existing applications. As I have said, these are hard areas of research; but they have the potential to offer considerable benefits. I believe they could enable the process of innovation in interactive systems to proceed a lot better than it currently does.

Acknowledgements

I am grateful to the many people with whom I have had helpful discussions and correspondence on the topics covered in this paper. They include Walter Vincenti, Wayne Gray, Bonnie John, Mary Shaw, Stuart Card, Richard Harper, Marge Eldridge, Bob Anderson, Hervé Gallaire, Thomas Green, Ruven Brooks, Patrick Waterson, Susan Gasson, Robert Rist, Willemien Visser and Terrie Shaft.

References

[1] B. Adelson and E. Soloway: The Role of Domain Experience in Software Design. In: IEEE Trans. on Software Engineering, Vol SE-11, no. 11 (November 1985), pp. 1351-1360.

[2] J. L. Bentley: Writing Efficient Programs . Englewood Cliffs NJ: Prentice-Hall, 1985.

[3] S. K. Card, T. P. Moran and A. Newell: The Psychology of Human Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.

[4] E. W. Constant II: The Origins of the Turbojet Revolution. Johns Hopkins Univ. Press, Baltimore, MD, 1980.

[5] D. G. Copeland, R. O. Mason and J. L. McKenney: Sabre: The Development of Information-Based Competence and Execution of Information-Based Competition. In: IEEE Annals of the History of Computing, Vol. 17 no 3 (1995), pp 30-57.

[6] B. Curtis, H. Krasner and N. Iscoe: A Field Study of the Software Design Process for Large Systems. In: CACM Vol 31, no. 11 (1988), pp. 1268-1287.

[7] P. J. Denning: Computing, Applications and Computational Science. In: CACM Vol. 34, no. 1 (October 1991) 129-131.

[8] R. R. Everett, C. A. Zraket and H. D. Bennington: SAGE: A Data Processing System for Air Defence. In: Proc. Eastern Computer Conf. (1957), pp. 144-157.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides: Design Patterns. Reading MA: Addison-Wesley, 1995.

[10] W. D. Gray, B. E. John and M. E. Atwood: The Précis of Project Ernestine or, An Overview of a Validation of GOMS. In: Proceedings of CHI '92 Human Factors in Computing Systems (May 3-7, 1992, Monterey, CA) ACM/SIGCHI, N.Y., pp. 307-312.

[11] W. D. Gray, B. E. John and M. E. Atwood: Project Ernestine: Validating a GOMS Analysis for Predicting and Explaining Real-World Task Performance. In: Human Computer Interaction Vol 8 (1993), pp 237-309.

[12] R. Guindon: Knowledge exploited by experts during software system design. In: Intnl. J. of Man-Machine Studies, Vol. 33, no. 3 (1990), pp. 279-304.

[13] D. E. Lipkie, S. R. Evans, J. K. Newlin and R. L. Weissman: Star graphics: An object-oriented implementation. In: Computer Graphics (SIGGRAPH '82 Proceedings) Vol 16. no 3 (1982). pp. 115-124.

[14] J. Martin: Recommended diagramming standards for analysts and programmers. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[15] M. J. Muller, R. Carr, C. Ashworth, B. Diekmann, C. Wharton, C. Eickstaedt and J. Clonts: Telephone Operators as Knowledge Workers: Consultants Who Meet Customer Needs. In: Proceedings of CHI '96 Human Factors in Computing Systems (April 13-18, 1995, Vancouver BC) ACM/SIGCHI, N.Y., pp. 130-137.

Donald T. Campbell (1960)

↳ CamWorks User Interface

[16] W. M. Newman: A Preliminary Analysis of the Products of HCI Research, Based on Pro Forma Abstracts. In: Proceedings of CHI '94 Human Factors in Computing Systems (April 24-28, 1994, Boston, MA) ACM/SIGCHI, N.Y., pp. 278-284.

[17] W. M. Newman and Lamming M. G. (1995) Interactive System Design. Wokingham: Addison-Wesley.

[18] W. M. Newman: The Place of Interactive Computing in Tomorrow's Computer Science. In: Wand I. and Milner R., eds.: Computing Tomorrow. Cambridge: Cambridge Univ. Press, 1996.

[19] W. M. Newman: Models of Work Practice: Can they Support the Analysis of System Designs? Conference Companion, CHI '96 Human Factors in Computing Systems (April 13-18, 1996, Vancouver BC) ACM/SIGCHI, N.Y., pp. 216.

[20] W. M. Newman, M. A. Eldridge and R. H. R. Harper: Modelling Last-minute Authoring: Does Technology Add Value or Encourage Tinkering? In: Conference Companion, CHI '96 Human Factors in Computing Systems (April 13-18, 1996, Vancouver BC) ACM/SIGCHI, N.Y., pp. 221-222.

[21] H. Petroski: To Engineer is Human: The Role of Failure in Successful Design. New York: St Martin's Press, 1985.

[22] P. G. Polson and C. H. Lewis: Theory-based design for easily learned interfaces. In: Human Computer Interaction., Vol 5 (1990), pp 191-220.

[23] G. F. C. Rogers: The Nature of Engineering: a Philosophy of Technology. London: Macmillan, 1983.

[24] H. A. Simon: The Sciences of the Artificial, Second edition. Cambridge MA: MIT Press, 1981.

[25] D. C. Smith, C. Irby, R. M. Kimball, E. Harslem and H. L. Morgan: The Star User Interface: An Overview. In: Proc. AFIPS National Comp. Conf., Vol. 51 (June 1982), pp. 515-528.

[26] D. C. Smith, C. Irby, R. M. Kimball, W. Verplank and E. Harslem: Designing the Star User Interface. In: BYTE, vol. 7 no. 4 (April 1982).

[27] W. G. Vincenti: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. Baltimore: Johns Hopkins Univ. Press, 1991.

[28] W. Visser: More or less following a plan during design: opportunistic deviations in specification. In: Intnl. J. of Man-Machine Studies, Vol. 33, no. 3 (1990), pp. 247-304.

[29] T. Winograd, ed.: Bringing Design to Software. Reading MA: Addison-Wesley, 1996.

[30] G. H. Woodmansee: The Visi On Experience—From Concept to Marketplace. In: Human Computer Interaction—INTERACT '84 (Shackel B., ed.), pp. 871-875. Amsterdam: North Holland, 1984.

Address of author

Dr William Newman
Rank Xerox Research Centre
61 Regent Street, Cambridge UK CB2 1AB
Email: newman@cambridge.rxrc.xerox.com