THE ENGINEERING OF SOFTWARE: VIEWS FROM TECHNOLOGY THEORY

Niels Jørgensen
Roskilde University
Roskilde, Denmark
nielsj@ruc.dk

Abstract

Parnas' information hiding approach to software modularization (Parnas 1972) is one of many indications that knowledge in software engineering cannot be adequately understood merely as "technical knowledge." Rather, insight into process issues such as coordination among module developers is also a constituent of such knowledge. With the aim of contributing to a broader interpretation of software engineering and its underlying epistemology, three views of technology are surveyed. The first is Shaw's theory of the immaturity of computer science as a supporting discipline of software practice, as sketched initially in "Prospects for an Engineering Discipline of Software" (Shaw 1990). The second is Vincenti's theory of vicarious models and technological evolution, based on his study of aeronautics in What Engineers Know and How They Know it *(Vincenti 1990). The third is Simon's interdisciplinary account of design in* The Sciences of the Artificial *(Simon 1996). The result of the survey is, on the one hand, that none of the three contributions account convincingly for a significant role in engineering for insights of a qualitative, process-oriented nature. On the other hand, the three studies of the relationship between scientific and engineering knowledge contain significant pointers for future research of the interplay between technical and process-oriented aspects of the construction of software artifacts. This includes the significance of codification of standard solutions to routine problems, of vicarious modeling to predict the behavior of proposed designs, and of satisficing (find a design that works) as opposed to optimizing (find the best design).*

Keywords: Software engineering, technology theory, module, technical knowledge, process

Introduction

Technical rationality is frequently viewed as an appropriate epistemology of computing's technical core, including the practical activity of programming and the theory of programming languages. The view is common both within technical and nontechnical communities. The 1989 ACM curriculum report (Denning et al. 1989) emphasized quantitative and formal knowledge, for example by defining computing as based on three paradigms: theory, rooted in mathematics; abstraction, rooted in (natural) science; and design, rooted in engineering. Also, proponents of soft methods for information systems development and their underlying epistemologies, radically different from technical rationality, tend to accept a *technical rationality for the technical core* premise. For example, Dahlbom and Mathiassen (1997) contrasted a mechanical and a romantic view, and argued that both were relevant and legitimate views, corresponding to engineering of artifacts versus facilitation of evolution and culture in organizations, respectively. Similarly, Avison et al. (1998), in arguing for the need to go beyond traditional, programming-based approaches to IS development, identified technical rationality as the epistemological basis of the original programming methods, such as object-oriented and structured programming.

However, in the history of software engineering and software architecture, there are indications of intimate interplay between technical and people issues. These include Parnas' (1972) definition of a software module as "a responsibility assignment rather than a subprogram" (p. 1054). With reference to Schön's (1983) postulate of a "rigor and relevance" dilemma, one may say that such considerations are not rigorous in the sense of quantifiable and formal, yet relevant in the technical construction of software artifacts. If this is true for a substantial part of the technical areas of computing, it seems reasonable to challenge the exclusive

rights of technical rationality to account for *technical knowledge* in computing. The present study attempts to shed light on the nature of technical knowledge in computing by drawing on three contributions from the emerging academic field of technology theory.

Mitcham, in his *Thinking Through Technology* (1994), distinguishes between two major approaches to the theory of technology: e*ngineering* philosophy of technology and *humanities* philosophy of technology. The former is an analytical approach originating within the technological and engineering communities, the latter an interpretative approach rooted in the social sciences and the humanities. A common denominator is rejection, as in Layton's "Technology as Knowledge" (1974), of reductionism in the sense of viewing technology as merely applied science; such a view may underestimate the importance of technological knowledge, distinct in form and origin from scientific knowledge, as well as social and other processes that shape nonlinear, nondeterministic paths of technological development. Aside from addressing such issues in research, there is promotion in educational institutions of the study of technology, as in Science, Technology and Society (STS) programs, and attempts to counter views of technology as academically subordinate to (natural) science. Indeed, it would be in line with these endeavors if the notion of software engineering could be freed of its mechanical, nonintellectual, and nerdy connotations.

The use of technology theory may supplement, and can indeed be seen as analogous to, the use of philosophy of science to shed light on information technology and information systems research. An example of the latter is Hirschheim and Klein's (1989) investigation of IS development paradigms, as defined in terms of concepts from the philosophy of science such as objectivism and subjectivism. A third type of investigation across different technological domains can be found in innovation theory, as in Rogers' (1995) stage model of the diffusion of innovations, and King's (1996) comparison of computerization and electrification in terms of productivity benefit, or lack thereof.

The three studies of science and engineering surveyed here fall within Mitcham's category of *engineering* philosophy of technology. The studies focus on analytical topics such as the role of knowledge in engineering; they ask epistemological questions such as what sort of knowledge is relevant to engineering and where does it come from? The studies tend to focus on design in a constructive, engineering sense, as opposed to design in the sense of uncovering user needs. Specifically, Shaw's (1990) and Vincenti's (1990) studies rely on deep insights into the specific engineering fields of software and aircraft, respectively. Simon's (1996) account of design is interdisciplinary but focuses, as do the two others, on the inner structures of artificial systems and their construction. Surveying a selection of contributions sharing the same (analytical) approach facilitates reasoning across the contributions; it does not reflect a rejection of other (interpretative) approaches such as social constructivism, for example Pinch and Bijker's (1987) analysis of the early design history of the bicycle.

The result of the survey is, on the one hand, that none of the three contributions account convincingly for a significant role of insights of a qualitative, process-oriented nature in software engineering and architecture. On the other hand, the three studies contain significant pointers for future research of the interplay between technical and process-oriented aspects of the construction of software artifacts. This includes the significance of codification of standard solutions to routine problems (Shaw 1990), of cost-saving, vicarious modeling to predict the behavior of proposed designs (Vincenti 1990), and of satisficing (find a design that works) as opposed to optimizing (find the best design) (Simon 1996).

The present study employs software modularization, and more generally software architecture, as an example technical (sub)discipline. The nature (technical and other) of the field of software architecture may in itself be of significance to the Information Systems discipline, because the understanding of the nature of the field may be important for how it is taught at universities and business schools. The overall architecture of a system in a sense links the acquiring and the developing party, that is, the two target professional roles sketched in the IS curriculum: "acquisition, deployment, and management of it resources and services" versus "system development, system operation, and system maintenance" (Gorgone et al. 2002, p. 11). Both roles or sides of the negotiation table should understand software architecture and be able to communicate about it.

The paper is organized as follows: The next section recapitulates Parnas' concept of information hiding. The subsequent sections survey Shaw's, Vincenti's, and Simon's theories of science and engineering. In the final section, the conclusions are presented.

Parnas' Concept of Information Hiding

David Parnas' concept of information hiding addresses the software architectural problem of how to decompose a software system. Parnas' work is also interesting because it is early, widely acknowledged, and motivates information hiding by a rationale which is essentially process-oriented rather than technical. The term *process* is used here in the sense of activities and mechanisms for

coordination in an organization conducting software development; the notion of insight and research results of a *qualitative* nature is discussed later.

Software architecture can be defined as "the principled study of the overall structure of software systems, especially the relations among subsystems and components" (Shaw 2001, p. 657). Examples of software structures include client-server structures, generic class structures captured by design patterns, and the organization of networks into stacks of layered protocols. Software architecture also comprises the methods of communication and invocation among subsystems, such as message passing and procedure calls. Thus software architecture comprises the overall technical design of a software system.

Parnas (1972) suggested information hiding as the key decomposition criterion. The setting was a hypothetical team of software developers given the task of creating a program to meet certain requirements, starting out by defining a modular structure.

Parnas defined information hiding as the hiding of design decisions, for instance the details of a data structure and the procedures to access it. Recently, Parnas (1998) provided an up-to-date example: he claimed that there would have been no year 2000 problem if his idea had been widely accepted in practice; that is, if a program's data structure for representing a date is hidden within a single module, then checking the program for a Y2K error can be done by working with essentially just that module, and would not require inspection of the entire program for locally defined data structures for dates.

Parnas (1972) mentioned flexibility, comprehensibility, and shortening of development time as the three underlying goals. For example, development time could be reduced because modules could be worked on independently. All three goals are process or people oriented, in the sense of convenience and understandability from the point of view of the programmers, rather than technical in the sense of being oriented toward program execution time, for example.

Parnas contrasted information hiding with flowcharts (i.e., with defining a separate module for each major step in a flowchart). Parnas recognized that a flowchart-based module structure would often be more efficient, because program execution would jump less frequently between modules (i.e., execution would remain inside the first module, then inside the second, etc.). Information hiding-based decomposition may be less efficient because control would be transferred frequently from module to module, and this would entail a higher number of procedure calls.

It seems permissible to make Parnas' analysis more explicit by asserting that the major scarce resource is developer time, not computing power. At an underlying level, one may assert that it is difficult and time consuming for a developer to build knowledge of the inner workings of a module, so in a sense limited developer *knowledge* is a constraint underlying the constraint on developer *time*.

In retrospect, Parnas said that his early work on information hiding "was not appreciated by those whose background was mathematics or science. They were looking for work that established fundamental limitations on computers; I was trying to overcome people's limitations" (1999, p. 48). Table 1 provides a summary of the information hiding concept.

Parnas' concept of information hiding as interpreted in Table 1 has a methodological as well as an epistemological aspect. It is a methodological concept in the sense of being oriented toward guidelines for programming (i.e., modularization), and providing a basis for the development of specific methods (e.g., structured programming). It is an epistemological position in the sense of asserting that there is a crucial nontechnical component of software architectural knowledge, for example of the knowledge embodied in a judgment of whether a certain design is appropriate for a given problem. The methodological (or software engineering) aspect appears to be the starting point, entailing the epistemological (or software architectural) aspect.

Table 1. Parnas' Concept of Information Hiding

Goals	Three process-oriented goals: flexibility, comprehensibility, shortening of development time
Relevant developer competencies (inferred by this author, not Parnas)	Understand process and how to balance process goals against performance

Technology View #1: Shaw's Paradigm for the Supporting Sciences of Engineering

The first view is from within computing: Mary Shaw's paradigm for software engineering. This view focuses on the supporting sciences, rather than engineering itself, and has software architecture research as the main example of a supporting field (Shaw 1990, 2001, 2002). A major aim of Shaw's work is to overcome a situation where "software engineering does not yet have a widely-recognized and widely-appreciated set of paradigms in the way that other parts of computer science do.…Poor external understanding leads to lack of appreciation and respect" (Shaw 2001, p. 662). Shaw presents a nuanced approach, recognizing the value of a variety of different paradigms, yet strongly favors among these a formal, quantitative approach.

Shaw defines the notion of an engineering discipline as the final state out of three in the historical evolution of the practical exploitation of a technology. The states are characterized by the way knowledge relevant for practice is generated and disseminated. The engineering state follows craftsmanship and commercial production, and is characterized by support from science. Shaw claimed that computer science is immature and largely unable to support software practice, and that consequently software practice has not evolved into a proper engineering discipline. Rather, it is seen as comparable to large-scale construction of buildings, infrastructure, etc., before it evolved into the field of civil engineering approximately 150 to 250 years ago, benefitting from support by the physical theories of statics and strength of materials. Exceptions to this immaturity include developments within the field of programming languages, such as high-level programming language constructs to support abstraction and modularization (Shaw 1990).

In Shaw's notion of engineering, the supporting science should research problems of relevance to the practitioner, and codify results in an accessible and useful form: "Engineering relies on codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice" (Shaw 1990, p. 16). There is an underlying assumption of some degree of pattern or regularity in the problems that confront practitioners in a given field. Shaw distinguishes between routine and innovative design: "Routine design involves solving familiar problems, reusing large portions of prior solutions. Innovative design, on the other hand, involves finding novel solutions to unfamiliar problems. Original designs are much more rarely needed than routine designs, so the latter is the bread and butter of engineering" (1990, p. 16). It is in this respect that computer science provides insufficient support for practitioners. As a result, "Software in most application domains is treated more often as original than routine" and so is perceived to require "virtuoso design" more often than actually needed (Shaw 1990, p. 16).

Shaw's proposal for evolving software practice into a proper engineering discipline prioritizes the codification of existing knowledge in the form of handbooks such as *Perry's Chemical Engineers' Handbook* (Perry and Green 1997). Such reference material is needed to aide the reuse of existing software (libraries, components, etc.), abstract entities (oversights over algorithms, class patterns, architectures, etc.), and analysis techniques. Educational activities should comprise the building of realistic systems aided by such material, rather than focus on fresh creation of small programs. These suggestions concern the codification and dissemination of existing knowledge, rather than point to entirely new areas of research.

Shaw indicates that the scientific results that are of the highest value are of a technical nature, in the form of general findings or truths and preferable in analytical or formal form. Knowledge about process issues, including life-cycle models, cost-estimation, and quality assurance, is written off as *software management* rather than proper software engineering (Shaw 1990). Shaw proposes a paradigm of software engineering research with a hierarchy of three classes of research results: findings, observations, and rules of thumb, based on work by Brooks (1988). Table 2 summarizes Shaw's hierarchy of research results.

Table 2. Shaw's Hierarchy of Research Results (Shaw 2002)

Type of Research Results	Characteristics
Findings	Well-established scientific truths, judged by truthfulness and rigor
Observations	Reports on actual phenomena
Rules of thump	Generalizations, signed by their author but perhaps incompletely supported by data

Findings, the highest form in Shaw's hierarchy of research results, is also characterized by precision, formality, and statistical validation. In the absence of findings, rules of thumbs and observation may provide some guidance to practitioners; they are also "the groundwork for the research that will, in time, yield findings" (Shaw 2002, p. 4). In other words, there is an assumption that research can eventually evolve into a quantitative form.

An initial observation is that in Shaw's hierarchy of types of research results (see Table 2), Parnas' initial formulations of the concept of information hiding would fit into the lowest category.

Second, in emphasizing rigor and statistical validation (for the highest category), Shaw's research hierarchy is well in-line with a paradigm of technical rationality. However, it is tempting to note that the rhetoric of scientific truthfulness seems beyond what is justifiable on the basis of critical rationalism in the sense of Popper's (1972) work, that is, the discussion of the asymmetry between verification and falsification, and the assertion of the impossibility of establishing in an absolute sense the truth of a general hypothesis by means of empirical testing.

Finally, it is interesting to note the persistence of qualitative results in one of the fields considered by Shaw to be mature: the field of programming languages. The field comprises notions such as data abstraction, and insights into how programming languages support it, that can be seen as the contemporary approach to what Parnas initiated with the information hiding concept. The most extensive realization of data abstraction is in object-oriented programming languages; Java, for example, allows for declaring data as *private*, thus effectively hiding it. The field of programming languages has produced a range of textbooks that resemble the handbook-type reference publications that Shaw suggests are crucial for supporting engineering. However, programming language textbooks such as *Concepts of Programming Languages* (Sebesta 1993) and *Comparative Programming Languages* (Wilson and Clark 1993) are conceptual and qualitative, rather than formal and quantitative, in their presentation of programming language constructs (statements, classes, etc.), concepts (modules, abstraction), and specific languages. There are also formal theories in the field of programming languages, such as denotational semantics, that can be used to give a formal definition of the meaning (in the sense of execution effect) of a statement in a programming language, for example. Yet, available research-based publications that attempt to give an overview over the usefulness of existing languages and the principal aspects of their components are conceptual and qualitative. For instance, it is significant that the choice of programming language for a given problem, such as the choice between Java and C, which is more machine-oriented and provides less support for information hiding, remains a problem for which there is no formal, quantitative method available. In such design choices one is left with what is essentially a qualitative judgment of the relative importance of factors such as time performance requirements, reliability requirements, and the developer team's knowledge. Sebesta (1993) mentions four criteria for evaluating a programming language: source code writability (one subcriterion of which is support for abstraction), readability (with subcriteria including simplicity), reliability (e.g., type checking), and cost (e.g., compilation and execution time, as well as programmer training). Sebesta notes that while most computer scientists would agree on the importance of his criteria, "it may be impossible to get even two computer scientists to agree on the value of a given language characteristic relative to others" (p. 7). One may note that it is precisely this sort of dependence on personal judgment that one seeks to eliminate by formalization and quantification. Table 3 below attempts to summarize the implications of Shaw's view.

Table 3. Summary of the Implications of View # 1: Shaw's Paradigm for the Supporting Sciences of Engineering

View of technology	• Engineering is an advanced state of the practical exploitation of technology which is characterized by support from a mature science. • Supporting sciences aid practitioners in applying standard solutions to routine problems. • Supporting sciences should codify relevant and reliable knowledge in handbooks
Implications	• The most reliable and useful results are based on quantitative or formal models. • Other result types are of some use to practitioners, and a basis for further research. • Process considerations or 'software management' is not a part of software engineering.

Technology View #2: Vincenti's Theory of Technological Evolution and Vicarious Models

The second view, Vincenti's *What Engineers Know and How They Know it* (1990), is an account of aeronautics. Vincenti's book contains five case studies of the evolution of flight technology in the first half of the 20th century, as well as a proposal for a generic epistemology of engineering that focuses on the evolution of engineering knowledge.

A major aim of Vincenti's work is to demonstrate the relative independence of technology vis-à-vis science, as opposed to a view of technology as applied science: "Technology, though it may *apply* science, is not the same as or entirely *applied* science" (Vincenti 1990, p. 4, emphasis in original). By science, Vincenti means the natural sciences, in particular physics; by technology, he understands mainly the knowledge used in the creative engineering of technological artifacts (e.g., aircraft). He adopts a conventional distinction between scientific and technological knowledge, that is, between (knowledge for) explanation of natural phenomena versus practical utility. Vincenti's concept of technological knowledge includes explicit and formal knowledge such as measurements from wind tunnel tests of wing profiles, catalogs over wing profiles with known good performance, and theories of airflow on wing surfaces. Technological knowledge is seen as rooted in its own communities and organizations, including research institutions and commercial aircraft companies. Vincenti argues that technological knowledge is independent from scientific knowledge in a double sense: it is not only a different body of knowledge, but it is also *generated* mainly within the field of technology itself, rather than by transfer from science. Vincenti's analysis is also of interest because from a historical perspective, computing may resemble aeronautics more than civil engineering, for example, since aeronautics is a young field and was backed early on by organized research and commercialization. Finally, Vincenti's theory of the evolution of technology is interesting because it focuses on design: "Decreasing uncertainty in the growth of knowledge in a technology comes, I suggest, mainly from the increase in scope and precision…in the vicarious means of selection" (Vincenti 1990, p. 250). By vicarious selection, Vincenti refers to selection among proposed designs by means of theoretical or experimental modeling, as a cost and time saving alternative to building full prototypes.

Aircraft design is subject to obvious quantitative requirements pertaining to speed and load, and derived requirements pertaining to the propulsion generated by propellers, for instance. There are also requirements of a less easily quantifiable nature, including stability and maneuverability. Vincenti's work is widely recognized, and is a sophisticated analysis in favor of a quantitative paradigm for engineering progress.

The so-called Davis wing, the subject of one of Vincenti's case studies, illustrates how accumulation of technological knowledge enhances the predictive power of modeling. A certain wing profile (developed by a D.A. Davis) was chosen by the designers of the American second world war bomber plane B-24, of which more than 19,000 were eventually produced, the highest number for any bomber plane in history. A wing's profile (airfoil section) is its contour when viewed from the side. The Davis wing had a novel profile, one aspect of which was that its point of maximum thickness was further aft (to the rear) compared to other profiles.

The wing profile for the B-24 was chosen mainly because it performed well in wind tunnel tests; indeed, the test results were so favorable that they were not trusted until repeated wind tunnel tests had confirmed them. In addition, a prototype had been built and successfully flown, but the wing profile decision had to be made before significant test results had been gathered about the prototype. Thus the design decision was based on a vicarious model, in this case an experimental rather than a theoretical one. Of course, the advantage of using a vicarious model is the elimination of the time and financial costs associated with building and testing a full prototype; the disadvantage is the uncertainty associated with the method, that is, the validity of the model testing results. Vincenti terms this *technological uncertainty*, and views the accumulation of knowledge to reduce the uncertainty of modeling as a key aspect of technological development.

The uncertainties associated with wind tunnel tests of wing profiles in the 1930s included uncertainty related to the scaling up from the size of the model wing to the actual wing size, and scaling up of wind speed to actual flight speed. There was no theoretical explanation of the favorable test results of the model, or theoretical justification that it would perform similarly in practice. In retrospect, the successful wind tunnel test of the Davis profile can be partly explained on the basis of the distinction between laminar and turbulent air-flow in the so-called boundary area (the section of air immediately surrounding the wing). Moving the maximum thickness point rearward may extend the section of the wing over which the air-flow is laminar, and so reduce overall drag. However, it is also known that the effect of reduced turbulent air-flow is more significant in the wind tunnel than on a full-scale plane, and that other contemporary profiles would in fact have performed similarly in practice, regardless of the fact that they were less impressive in the wind tunnel. Modern aeronautics use computer simulation models that utilize a range of theoretical and practical results in fluid dynamics. According to Vincenti (Chapter 2), it has been refined to a point where such vicarious models can reliably predict the performance of a wing profile design.

Table 4. Summary of the Implications of View #2: Vincenti's Theory of Technological Evolution and Vicarious Models	
View of technology	• Technology is a body of knowledge relatively independent from scientific knowledge. • Technological evolution provides designers with increasingly precise vicarious models. • Vicarious models let designers predict performance of proposed designs.
Implications	• Quantitative modeling paradigm of a more pure or radical nature than Shaw's. • Fluid dynamics is a major supporting science, which suggests that increased precision in modeling is due (at least partly) to applicability of natural, physical laws. • Computer science is an engineering discipline, not a science.

Vincenti's view of the increased modeling precision attained as a product of technological development bears resemblance to Shaw's hierarchical classification of software engineering results, and the expectation that research will eventually produce results of the highest form in the hierarchy. In software design, a vicarious model (in Vincenti's sense) would be a software architecture design that is proposed and not (yet) implemented.

A significant difference between aeronautics and software architecture is, of course, the physical nature of air-flow phenomena and wings. The fact that these natural and artificial phenomena obey natural laws of physics, such as those of gravity and fluid dynamics, is a key source of the increased precision and validity of modeling that has been attained in aeronautics.

If one adopts Vincenti's view of technology as centered on knowledge for design and rooted in its own communities and organizations, it is tempting to suggest that we label as *technology* or *engineering* a large portion of computing disciplines, including portions of information systems and software engineering. This would be justified on the basis that these fields, or portions of them, generate knowledge mainly for the purpose of practical utility. Shaw characterizes the fields of programming languages and software architecture as scientific fields within computer science, but it is unsatisfactory to view these fields as *explaining* the behavior of the phenomena they study; rather they create artifacts (i.e., languages and architectures) that may be useful for certain applications, and prove or argue otherwise that they have certain useful properties. While such a view may be controversial to some, it is consistent with Brooks' later view of computer science: "I submit that by any reasonable criterion the discipline we call 'computer science' is in fact not a science, but a synthetic, an engineering, discipline. We are concerned with building things" (1996, p. 6). Brooks asserted that as a consequence, the field should be more concerned with users and their needs, rather than "climbing into our ivory towers [and writing] to each other in ever more esoteric vocabularies" (1996, p. 6).

Technology View #3: Simon's Theory of Satisficing and the Sciences of the Artificial

The third view, Herbert Simon's *The Sciences of the Artificial* (1996), is the broadest of the three. Simon's unified approach to design encompassed classical engineering fields as well as medicine and architecture, and even nontechnical fields such as business (e.g., business strategy design), administration (e.g., organization design), and public policy (e.g., design of public infrastructure). Simon provided reflections on what can be known about these fields of artifacts, as well as many reflections on and characterizations of design processes.

Simon adhered to complexity as a major characteristic of artificial systems. He expressed this informally as "the whole is more than the parts" (p. 183), and "it is typical of many kinds of design problems that the inner system consists of components whose fundamental laws of behavior—mechanical, electrical, or chemical—are well known. The difficulty of the design problem often resides in predicting how an assemblage of such components will behave" (p. 15).

Specifically referring to computers, Simon strongly emphasized that the hardware is relatively unimportant in determining the properties of a computer system, and that instead what is important are the properties of the computing system as a whole. A theory of an individual component (such as a computer's hardware) "may indeed by simply irrelevant" (p. 19). If a reliable theory of the full system is not available, developers must take an empirical approach. Referring to the development of the first time-sharing operating systems (such as the OS/360 project that Brooks headed for a period), Simon states that since there was no theory available to predict how a proposed design would behave, development proceeded essentially by building a system and seeing how it worked.

Two of Simon's characterizations of the design process may be of particular interest: First, *satisficing* is a term introduced by Simon to capture the situation when a designer asks "does this alternative satisfy all the design requirements?" rather than "Of all possible worlds…which is the best" (p. 121). The concept is rooted in Simon's work in artificial intelligence, including classical optimization problems such as the traveling salesman problem that has no known efficient solutions. Satisficing was also seen as relevant when there is limited knowledge of design alternatives, or limited ability to predict their behavior; or if global optimization is outright impossible because there is no common utility function that captures the individual preferences of stakeholders, even if each of these are known.

Second, Simon emphasized that design should be conscious about resource allocation to the design process itself. "There are two ways in which design processes are concerned with the allocation of resources. First, conservation of scarce resources may be one of the criteria for a satisfactory design. Second, the design process itself involves management of the resources of the designer" (Simon 1996, pp. 124-125). This applies to a computer program requiring time for its search for a solution to an instance of the traveling salesman problem (i.e., a route design), as well as to human designers constrained by project deadlines.

Simon's assertion of the irrelevance of hardware properties to the understanding of the entirety of a complex computer system underpins the point made earlier (in the discussion of view #2) of the difference between physical systems (such as aircraft) and nonphysical software systems. A similar point has been made by the complexity theorist Hartmanis (1995) in his Turing award lecture. Hartmanis described the lack in computing of hard constraints, impossible to violate or circumvent as the laws of physics: "Computer science deals with information, its creation and processing, and with the systems that perform it, much of which is not directly restrained and governed by physical laws" (p. 19). Hartmanis described the pleasure he felt when eventually finding resemblance of hard constraints in complexity theory: "I loved physics for its beautifully precise laws that govern and explain the behavior of the physical world. In Shannon's work, for the first time, I saw precise quantitative laws that governed the behavior of…information. [This was] surprising and immensely fascinating" (p. 8).

Indeed, within the computing disciplines, complexity theory may be the field where laws in the strict sense of being impossible to violate play the greatest role, aside from disciplines of computer hardware technology. Complexity analysis may be defined as the study of resources required during computation to solve a problem, in particular time (cpu) and space (memory) resources. For decades, algorithmic textbooks have provided reference-type information of relevance for practitioners, such as the average or worst-case time consumption of alternative sorting and searching algorithms. However, complexity theory is a principled study, producing results such as classifications of computation problems relative to one another, including the equivalence (in a certain sense) of all so-called NP-complete problems (Garey and Johnson 1979). It is not commonly expected of complexity theory to eventually become useful for vicarious modeling in the sense of Vincenti (i.e., predictive modeling of cost and benefits of software architectural designs).

From the point of view of computing, Simon's notions of the design process as a resource consumer may be of interest in connection with the increased emphasis in the computing industry on shortening the development time. This is likely to make designers less interested in finding the software architecture that is technically optimal, and focus merely on finding an architecture that simply works. Time as a constraint of increased importance was emphasized by Brooks (1995) when he reviewed his theses; he noted that the emergence of the shrink-wrapped industry had created a setting for software development where time constraints were becoming more important, whereas in the classical software industry, schedule was more easily negotiable. These events in the software industry are also reflected in the interest in flexible software development methods, such as methods involving frequent releases, including those practiced by Microsoft and Netscape and recorded by Iansiti and MacCormack (1997).

Table 5. Summary of the Implications of View #3: Simon's Theory of Satisficing and the Sciences of the Artificial

View of technology	• Complexity as a common denominator of artificial systems • Complex systems can be perceived as aggregates of subsystems. • Yet a system's behavior cannot be inferred from that of its subsystems.
Implications	• Computer software systems cannot be understood on the basis of physical laws. • Satisficing (a design that works) as an alternative to optimizing (the best design). • The time spent on the design process itself as a crucial constrained resource.

Conclusion

The goal of this investigation was to examine concepts and other elements within the technology theories of Shaw, Vincenti, and Simon that would account for the role of process-oriented and qualitative insights in software technology, such as Parnas' concept of information hiding.

On the one hand, in a basic sense this has been unsuccessful. Shaw and Vincenti propose a view of technological knowledge as evolving to a state of precise results of quantitative and/or formal nature, and they see qualitative insights as intermediate and less valuable. Simon's account of knowledge for design is more pragmatic and does not grade different kinds of insights, but neither does it account positively for the relevance of qualitative forms of insight. It may follow from Vincenti's and Simon's analyses that there is significant room for nonquantifiable insights into software systems because they do not obey physical laws, but this remains unexplored.

On the other hand, if one accepts the premise that qualitative and process-oriented insights will remain of relevance to a technical field such as software architecture, even as knowledge is accumulated, then several key concepts of the technology theories surveyed may provide some useful input: Shaw's distinction between routine and innovative problems does not presuppose Shaw's quantitative paradigm, and may be useful also in a conceptual context where the identification of a routine problem is the qualitative judgment of the practitioner. Vincenti's concept of a vicarious model (i.e., a theoretical or practical model built to save the cost of full scale prototype implementations) is closely linked to an ideal of finding an optimal design; however, the notion can be reinterpreted with support from Simon's concept of satisficing so as to stress the importance of finding, at an early stage, a software design that is estimated to work, whether optimal or not. Simon's focus on the resource consumption of the design process is in line with Parnas' concern for development time, and by implication, the limited time for developers to build knowledge of a problem at hand. Attaining a vicarious model embodying a satisficing design can then be seen as a crucial step in the knowledge-building process in a development project. Perhaps the integration of both analytical and interpretative approaches, rather than merely analytical ones as in the this paper, is a road to capturing the composite nature of such knowledge.

References

Avison, D. E, Wood-Harper, A. T., Vidgen, R. T., and Wood, J. R. G. "A Further Exploration into Information Systems Development: The Evolution of Multiview 2," *Information Technology and People* (11:2), 1998, pp. 124-139.

Brooks, F. P. "The Computer Scientist as Toolsmith II," *Communications of the ACM* (39:3), March 1996, pp. 61-68.

Brooks, F. P. "Grasping Reality Through Illusion: Interactive Graphics Serving Science," in *Proceedings of the 1988 ACM SIGCHI Human Factors in Computer Systems Conference*, E. Soloway, D. Frye, and S. B. Sheppard (Eds.), ACM Press, New York, 1988, pp. 1-11.

Brooks, F. P. *The Mythical Man-Month. Essays on Software Engineering: 20th Anniversary Edition*, Addison-Wesley, Reading, MA, 1995.

Dahlbom, B., and Mathiassen, L. "The Future of Our Profession," *Communications of the ACM* (40:6), June 1997, pp. 80-89.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. "Computing as a Discipline," *Communications of the ACM* (32:2), January 1989, pp. 9-23.

Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

Gorgone, J. T., Davis, G. B., Valacich, J. S., Topi, H., Feinstein, D. L., and Longenecker, Jr., H. E. *IS 2002: Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems*, ACM, AIS, and AITP (available online at **http://www.aisnet.org/Curriculum/IS2002-12-31.pdf**).

Hartmanis, J. Turing Award Lecture: On Computational Complexity and the Nature of Computer Science," *ACM Computing Surveys* (27:1), March 1995, pp. 7-16.

Hirschheim, R., and Klein, H. K. "Four Paradigms of Information Systems Development," *Communications of the ACM* (32:10), October 1989, pp. 1199-1216.

Iansiti, M., and MacCormack, A. "Developing Products on Internet Time," *Harvard Business Review*, September-October, 1997, pp. 108-117.

King, J. L. "Where Are the Payoffs from Computerization? Technology, Learning, and Organizational Change," *Computerization and Controversy: Value Conflicts and Social Choices* (2nd ed.), R. Kling (Ed.), Academic Press, Orlando, FL, 1996, pp. 239-260.

Layton, E. L. "Technology as Knowledge,"*Technology and Culture* (15:1), January 1974, pp. 31-41.

Mitcham, C. *Thinking Through Technology*, University of Chicago Press, Chicago, 1994.

Parnas, D. L. "On the Criteria To Be Used in Decomposing Systems Into Modules," *Communications of the ACM* (15:12), December 1972, pp. 1053-1058.

Parnas, D. L. "Parnas on Parnas: A Life of Indecision," *Software Engineering Notes* (24:4), July 1999, pp. 47-49.

Parnas, D. L. "Successful Software Engineering Result," *ACM SIGSOFT Software Engineering Notes* (23:3), May 1998, pp. 64-68.

Perry, R. H., and Green, D. W. (Eds.). *Perry's Chemical Engineers' Handbook* (7th ed.), McGraw-Hill, New York, 1997.

Pinch, T. J., and Bijker, W. E. "The Social Construction of Facts and Artifacts, or How the Sociology of Science and the Sociology of Technology Might Benefit Each Other," *The Social Construction of Technological Systems*, W. Bijker, T. P. Hughes, and T. J. Pinch (Eds.), MIT Press, Cambridge, MA, 1987, pp. 17-50.

Popper, K. R. *Objective Knowledge: An Evolutionary Approach*, Clarendon Press, Oxford, England, 1972.

Rogers, E. M. *Diffusion of Innovations* (4th ed.), Free Press, New York, 1995.

Schön, D. *The Reflective Practitioner*, Basic Books, New York, 1983.

Sebesta, R. C. *Concepts of Programming Languages* (2nd ed.), Benjamin/Cummings, Redwood City, CA, 1993.

Shaw, M. "The Coming-of-Age of Software Architecture Research," in *Proceedings of the International Conference on Software Engineering*, May 2001, pp. 657-664.

Shaw, M. "Prospects for an Engineering Discipline of Software," *IEEE Software* (7:6), November 1990, pp. 15-24.

Shaw, M. "What Makes Good Research in Software Engineering?," *International Journal on Software Tools for Technology Transfer* (4:1), 2002, pp. 1-7.

Simon, H. A. *The Sciences of the Artificial* (3rd ed.), MIT Press, Cambridge, MA, 1996.

Vincenti, W. G. *What Engineers Know and How They Know it: Analytical Studies from Aeronautical History*, Johns Hopkins University Presss, Baltimore, MD, 1990.

Wilson, L. B., and Clark, R. G. *Comparative Programming Languages* (2nd ed.), Addison-Wesley, Wokingham, England, 1993.