Desert Island: Software Engineering—A Human Activity

Article *in* Automated Software Engineering · October 2002

DOI: 10.1023/A:1022972113929 · Source: DBLP

1 author:

Gerhard Fischer
University of Colorado Boulder
298 PUBLICATIONS **12,133** CITATIONS

SEE PROFILE

Desert Island Contribution — ASE Journal

Software Engineering — A Human Activity

Gerhard Fischer
University of Colorado, Center for LifeLong Learning and Design (L3D)
Department of Computer Science, Campus Box 430
Boulder, CO 80309-0430 - USA
gerhard@cs.colorado.edu

What are the fundamental problems of software engineering? Without a doubt, different people will have very different answers to this question; my answer is: *it should be (but too often is not) considered a human activity*. I firmly believe that system development is difficult not because of the complexity of technical problems, but because of the social interaction between users and system developers as they learn to create, develop, and express their ideas and visions. Software engineering (especially its upstream activities) is a human-oriented field, and as such will always have the openness of other design disciplines, such as architecture and graphic design, rather than the hard-edged formulaic certainty of downstream engineering.

My thoughts about software engineering as a human activity serve as selection criteria for the books I would take to a desert island. A basic belief of mine is that software engineers can learn a lot (1) by studying other design disciplines such as architectural design, graphic design, and organizational design, and (2) by acquiring a deep understanding of cognitive and social issues. For example, the limitations and failures of design approaches that rely on directionality, causality, and a strict separation between analysis and synthesis have been recognized in architecture for a long time. A careful analysis of these failures could have saved software engineering the effort expended in finding out that waterfall-type models can at best be impoverished and oversimplified models of real design activities. Assessing the successes and failures of other design disciplines does not mean that they have to be taken literally because software artifacts are different from other artifacts. Rather, they can be used as an initial framework for software design.

The books I have selected for my hypothetical desert island trip were chosen to help me to increase my understanding of these problems, which are fundamental to the future of software engineering. My selections all turned out not be considered "software engineering" books per se — they are books that software engineers should read, but in most cases do not.

The first book I would take would be **Herbert Simon's *The Sciences of the Artificial*** [Simon, 1996], which was originally written in 1969, with subsequent editions in 1981 and 1996. I have read this book several times already. It is a *fundamental book about design* that contains numerous important insights, observations, and challenging and inspiring themes that software engineers should know about and think about, and ask themselves what it means for their own work.

Presented here are a few of the arguments that I found inspiring: Simon provides a principled argument that complex systems will evolve much more rapidly from simple systems if there are stable intermediate forms than if there are not. This observation is nicely illustrated with the parable of two watchmakers. The first one assembles his watches by building stable subsystems, whereas the second one builds them from scratch, with the result that the first watchmaker turns out to have a very prosperous business and the second one ends up in bankruptcy. I consider this

story to be fundamental for software engineers to understand, and explore its impact and relationship for software reuse, component-based architectures, object-oriented design, and reliability of systems. Simon's thoughts on *nearly decomposable systems* provide further foundations for modularity of complex systems and the search for representation, which exploit the redundancy of a system to find more cognitively efficient representations of them.

Other sources for inspiration are Simon's observations about *domains*. Domains are not natural, God-given entities, but they are part of the "sciences of the artificial" — they are constructs that serve our needs. Domains have boundaries, but these boundaries are not absolute; they are structuring mechanisms that help human beings cope with a world in which there is too much to learn and too much to know. And as our needs change, so too will our domains. Domain models should be *designed* to fit what people want to do — first through participation with users and eventually by users themselves requiring support for design in use, end-user modifiability, and meta-design.

Simon argues that when a domain reaches a point at which the knowledge for skillful professional practice cannot be acquired in a decade, specialization increases, collaboration becomes a necessity, and practitioners make increasing use of reference aids. The notion of semantically rich domains can be used as a starting point to reflect about the relationship, importance, and complementarity of domain knowledge and computational knowledge, leading to such approaches as domain-oriented design environments that recognize the legitimacy of specialization to the domain by not serving all needs obscurely, but serving a few needs well.

Another concern spread throughout the book is Simon's attention to *evolution and evolutionary models* — required by the fact that design often has to proceed without final goals and therefore has to cope with fluctuating and conflicting requirements. Theoretical foundations of the ill-structured nature of design, as well as empirical evidence, have shown that it is impossible to have complete specifications because requirements fluctuate over time and conflict with each other.

The concern with evolution and considering systems as living entities provides the rationale for the second book that I would take along: **Richard Dawkins *The Blind Watchmaker*** [Dawkins, 1987]. This is an important book to read to gain a deeper understanding of *evolution*. Dawkins demonstrates that big-step reductionism cannot work as an explanation of mechanism, because we cannot explain a complex thing as originating in a single step — complex things *evolve*. I firmly believe that models from biology will be more relevant than models from mathematics to future software systems because we live in a world characterized by evolution — that is, by ongoing processes of development, formation, and growth in both natural and human-created systems. Biology tells us that complex, natural systems are not created all at once but must instead evolve over time. Evolutionary processes are ubiquitous and critical for social, educational, and technological innovations. For example, in our work we have developed the *seeding, evolutionary growth, reseeding process model* to cope with fluctuating and conflicting requirements. In this model, the goal of the seeding phase is to create an evolvable environment for a particular domain. Seeding entails embedding as much domain knowledge as possible into a system. But the design knowledge can never be considered as complete because each design project will address a problem that is in some respects unique, and will therefore generate new knowledge that can be added to the seed. The seed is therefore explicitly designed to evolve, which emphasizes evolution as the central design concept.

Dawkins's book lends support to the claim that software design needs to be understood as an evolutionary process in which system requirements and functionality are determined through an iterative process of collaboration among multiple stakeholders. The fact that requirements cannot be completely specified before system development occurs has led us to postulate the following claims serving as high-level guidelines for our research:

1. *Software systems must evolve; they cannot be completely designed prior to use*. Design is a process that intertwines problem solving and problem framing. Software users and designers will not be able to fully determine a system's desired functionality until that system is put to use.
2. *Software systems must evolve at the hands of the users*. Users (not developers) experience a system's deficiencies; therefore, they have to play an important role in driving its evolution. Software systems need to contain mechanisms that allow users to modify their functionality and content.
3. *Software systems must be designed for evolution*. Even recognizing that evolution is no panacea and creates its own problems, there are strong reasons to increase the efforts and the costs to include mechanisms for evolution (such as end-user modifiability, tailorability, adaptability, design rationale, and making software "soft") in the original design of complex systems. Experience has shown that the costs saved in the initial development of a system by ignoring evolution will be spent several times over during the use of a system.

Design for evolution provides foundations for recent developments in software engineering such as open source developments and meta-design extending the domain modeling approach to a *collaborative domain construction approach*. Design for evolution requires *"underdesign for emergent behavior":* it focuses not on creating final solutions, but on creating spaces in which users as developers and designers can create their own solutions to fit their needs.

The rationale for my third book is grounded in the observation that the individual human mind is limited. Complex design problems require more knowledge than any single person possesses because the knowledge relevant to a problem is usually distributed among many stakeholders. Creating a shared understanding among stakeholders requires bringing different and often controversial points of view together and can lead to new insights, new ideas, and new artifacts. Designers need to rely on the knowledge of other people and on external information. Relevant knowledge for complex design activities is distributed among multiple human beings and among artifacts, bringing together different sources of knowledge, none of which as the final authority. By exploiting the "symmetry of ignorance" and mutual competency, stakeholders can learn from each other. My choice for a book serving as a good inspiration to reflect about these problems is the volume ***Design at Work: Cooperative Design of Computer Systems*** **edited by Joan Greenbaum and Morton Kyng** [Greenbaum & Kyng, 1991]. This book emphasizes the prominent role of communication and mutual learning between domain practitioners and system developers, both in constructing an initial model of the domain rooted in domain practice and in evolving this model over time to suit the changing needs of practitioners. Mutual learning is required because developers need to learn about the current domain, and practitioners need to learn how current practices might be transcended with new technologies. Also needed are computational tools and new models of software development that promote communication and mutual learning by all stakeholders throughout the design. Specification errors often occur when designers do not have sufficient application domain knowledge to interpret the customer's intentions from the requirement statements — a communication breakdown based on a lack of shared understanding. We need to develop new kinds of languages, including usage scenarios, mock-ups, and simulations, that can serve as *boundary objects between different communities of practice*. Such boundary objects are fundamentally different from formal specifications whose strengths are that they can be manipulated by mathematics and logic and interpreted by computers. As such, these representations are often couched in the language of the computational system. However, such representations are typically foreign and unintelligible to users and get in the way of trying to create a shared understanding among stakeholders.

Communication and collaboration are critical because users know their requirements only vaguely at best, and system development is an ill-defined design task, in which the problem cannot be understood before attempts to solve it are made. New requirements emerge during development because they cannot be identified until portions of the system have been designed

or implemented. Because much of the user's knowledge is tacit, only a part of design knowledge can be expressed in verbal descriptions. Specification and implementation have to *co-evolve*, which requires the owners of the problems to have some control over the development. Systems must undergo sustained development, requiring extensible systems and social structures that include users who are able to change systems. These requirements are both social and technical. Promising technical approaches to enable continual evolution of systems include end-user modification and end-user programming.

Communication and coordination breakdowns were less of a problem in the early days of software engineering when systems were built for computer specialists. But as computers pervade more and more domains, software engineers no longer have the understanding of application domains that is required to precisely specify systems in advance of implementation. Software design and development is a cooperative design task between software developers and users. The basis of this cooperation is a spirit of mutual learning. To design useful and usable systems, software developers must understand the users' practice, and users must understand available technical possibilities.

I would take these three books, not because they provide specific operational answers to my theme of "*software engineering as a human activity,*" but because they provide interesting ideas, problems, and challenges to gain a deeper understanding for fundamental problems of software engineering and they can serve as an inspiration for future research. As I said before: the three books are not software engineering books per se, and this implies that we cannot blindly follow their lessons. Evolution in biology, for example, is different from evolution in the human-made world of complex software systems, because vast differences exist between the world of the born and the world of the made: one is the outcome of a random natural process and the other is the result of purposeful human activity. I would look forward to having the time on the desert island *to reflect on these issues in depth by using the inspiration provided by these three books.*

References

Dawkins, R. (1987) *The Blind Watchmaker,* W.W. Norton and Company, New York - London.

Greenbaum, J. & Kyng, M. (Eds.) (1991) *Design at Work: Cooperative Design of Computer Systems,* Lawrence Erlbaum Associates, Inc., Hillsdale, NJ.

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