Beyond Computer Science

James D. Herbsleb
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
+1 412 268 8933
jdh@cs.cmu.edu

ABSTRACT

Computer science is necessary but not sufficient to understand and overcome the problems we face in software engineering. We need to understand not only the properties of the software itself, but also the limitations and competences humans bring to the engineering task. Rather than rely on commonsense notions, we need a deep and nuanced view of human capabilities in order to determine how to enhance them. I discuss what I regard as promising examples of cognitive and organizational theories and propose research directions to develop new ways of representing run-time behavior and ways of thinking about project coordination. I conclude with observations on creating an interdisciplinary culture.

Categories and Subject Descriptors
D.2,0 [**General**]

General Terms
Measurement, Experimentation, Theory.

Keywords
Coordination, behavioral science, interdisciplinary, multidisciplinary.

1. INTRODUCTION

Nearly two decades ago, Fred Brooks, in his classic "no silver bullet" paper [5] warned us that certain of the difficulties of building software systems derive from the essential nature of software, hence are unlikely to be overcome by any single breakthrough. In fact, the more deeply we understand the nature of software development, the more we begin to grasp its true difficulties. Brooks makes this point with an analogy to medicine:

> The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions [6] p. 181.

I want to suggest in this talk that a deeper understanding of our own field leads us into the intersection of several scientific disciplines – an intellectual bramble that many of us would rather avoid than try to untangle.

The history of germ theory provides an informative historical analog. Between the time that John Snow published the first convincing epidemiological evidence for germ theory (the famous study showing the spread of cholera from the Broad Street Pump) and the general acceptance of those conclusions, was about 80-100 years [38]. The delay can be attributed to the entrenched, "common-sense" view prevalent at the time, which held that diseases were caused by "foul emanations from soil, water and air" [36] p. 377. This view often led to helpful actions, such as its emphasis on general cleanliness. But it also posed serious dangers, as when it was proposed that in order to promote hygiene, the streets of London should be meticulously cleaned up, including all the human waste, with the offending materials to be dumped into the Thames [38]. Unfortunately, the Thames provided the city's drinking water, so the proposal would have created an extremely effective channel for propagation of diseases like cholera and typhoid.

As Brooks points out, the difference between folk medicine and medicine as we currently know it is *a better understanding of the underlying mechanisms that produce the phenomena of interest*. In software engineering, we generally rely on computer science to provide explanations in terms of these underlying mechanisms, and that is often precisely the right place to turn. We need not, for example, rely on folk wisdom about how well various algorithms will scale with the number of inputs. We can analyze the algorithms, and understand precisely *why* they scale differently. Pointing out the utility of computer science for software engineering is to belabor the obvious.

The thesis of my talk, however, is that computer science is not enough. Neither is computer science plus common sense. We need not only to understand the properties and behavior of software, but also the behavior of software engineers, development teams, and organizations. The extraordinary complexity, conformance, changeability, and invisibility of software – the sources of its essential problems, according to Brooks [5] – severely challenge human capabilities. Understanding the underlying mechanisms in play as people tackle software engineering tasks – mechanisms rooted in human cognition, social practices, and culture – is critical to the progress of our field.

2. BRAINS AND SOFTWARE

The functional components of the human brain "were designed by natural selection to solve adaptive problems faced by our hunter-gatherer ancestors" [8]. The match of such components to the tasks of software development is not always what one would hope. The human cognitive landscape has many islands of specific, highly-specialized and very powerful subsystems such as natural language processing [32] and visual processing [39]; yet tasks requiring far simpler forms of cognition, e.g., simple logic problems, are often performed very slowly and with many errors [15] because of a poor match with cognitive competences.

Understanding how to optimally engage this collection of competences in the highly complex tasks of software design and implementation is an exercise in mapping novel tasks onto cognitive machinery designed for very different purposes. It often results in what we might call "cognitive mismatch." For example, people tend to find the task of learning programming languages effortful, and most people never attempt it, or try then give up. On the other hand, nearly every intact human effortlessly performs the far more complex task of learning one or more natural languages, with no need for instruction or feedback, to an extremely high level of competence [32]. As a second example, it is well-known that small groups of people who spend time together naturally develop a "transactional memory" in which individuals specialize in storing and retrieving particular kinds of information so that the multi-individual system functions as a retrieval device far more effectively than the sum of non-interacting individuals [26]. This process is badly disrupted in geographically distributed teams where typical face-to-face interactions, which provide the information from which the specializations are generated, typically do not happen [22].

The overall point of this talk is that understanding how to do software engineering better requires a deepening of our understanding of 1) effective software engineering principles and practice, and 2) how these principles and practices line up against human cognitive, social, and cultural functioning. Current software engineering research, in my view, is making steady progress in the former, but constantly risks irrelevance as it neglects the realities of the latter. We tend to assume that humans can and will simply change in whatever ways are necessary. Human functioning is not nearly so malleable, however, and we ignore this fact to our great detriment.

If this major point is correct, there should be many critical areas of software engineering in which we can make better use of human capabilities. In order to do this, we need theories that shed light on the underlying mechanisms as software tasks are mapped onto specific human competences. I will focus on two examples: naïve psychology and coordination.

2.1 Naïve Psychology

One often hears descriptions of software that are strangely anthropomorphic, describing, for example, what a component "knows" or is "trying" to do. In fact, Edsgar Dijkstra was so offended by the frequency of such talk that he suggested instituting a system of fines to stamp it out [12]. No one advocates or teaches this style of description, so why do people use it instead of the more precise vocabulary of computer science?

Consider a team trying to understand the odd runtime behavior of a particular component (call it component *A*) in a very complex, distributed software system. Suddenly, someone says, "Ah, *A* thinks file *foo* is corrupted!" This insight explains the complex pattern of peculiar behavior, and everyone agrees quickly with the diagnosis. Consider, for a moment, the semantics of this statement. It carries very complex implications about the relation of the component and the file, alternative actions the component may engage in, ways that the component interprets state that relates to the file, and so on. Imagine trying to express the semantics fully in some other way, e.g., there is a Boolean field set to "false," various components call methods that read this field, etc. Translation of the full import of the semantics of "knows," without recourse to other anthropomorphic descriptions would be extremely long and difficult to understand, if it were possible in any practical sense at all. Talking about what a component "knows" is a very abstract description of system state that knits together, in a way instantly understood by others, many diverse system behaviors and internal states.

It is not surprising that humans favor this form of description. When closely examined, it reflects very sophisticated reasoning about what "people" (or things treated anthropomorphically) believe, are trying to do, and the ways in which these relate to behavior. It is now well-established that such reasoning is characteristic of a powerful cognitive subsystem that is highly specialized to navigate the complex human social world [28]. The course of development is increasingly understood [25], and the specific brain structures that are responsible for the capability have been identified [17, 34]. Poor functioning in these structures is linked with the syndrome of autism [1], in which people cannot interact socially, treating other people much as they do inanimate objects.

There is fairly strong evidence that professional software engineers make very heavy use of this cognitive system in collaborative software engineering activities [21], and it is not clear that there is any other choice for interactions among people in real time. Different cognitive systems operate at different speeds. Working alone, one is free to use any system that is appropriate, operating at any speed that allows the task to fit within scheduling constraints. Collaborative work, on the other hand, is socially constrained to operate at a rapid pace. The naïve psychology system is perhaps the only powerful, fast cognitive system that is well-adapted to modeling runtime behavior. There may simply be no other straightforward way for groups of people to think collaboratively about runtime behavior in various scenarios, or to debug difficult problems in distributed systems.

Although I have argued that the use of naïve psychology has advantages and may even be inevitable, I do not completely discount Dijkstra's concerns. Suppose, for example, that the runtime component states cannot be accurately described in the vocabulary of naïve psychology? This would likely make runtime behavior very hard to understand, but even harder for people to talk about if using naïve psychological concepts. Or it might well be the case that recording the conclusions reached in collaborative sessions is very error prone, since it requires a translation from naïve psychological talk to some more standard representation.

This line of thinking suggests several fruitful interdisciplinary research directions. One would be to work out the semantics of naïve psychology. This might make it possible, for example, to design components that can more accurately be described at runtime by statements about belief and desire, or even to assure that statements like "Component *A* believes proposition *p*" is atomically true or false for the component, i.e., it cannot fall into some inconsistent, hence confusing, state. Other work could seek to understand the contours of naïve psychology as applied to runtime components. One could, for example, design a series of experiments to determine when and to what extent the use of naïve psychological explanations can accurately describe runtime behavior, and the particular kinds of errors it generates. The long-term results could lead, for example, to new kinds of standard abstract representations that are very powerful yet very simple to understand. It might also lead to practices or tools designed to recognize or prevent errors induced by the irresistible tendency to think anthropomorphically about programs. The conception of "naïve psychology" and its cognitive and neural basis forms a theoretical thread to tie this research program together.

Naïve psychology, of course, is just one example, which I used to illustrate the point that human cognition exhibits unexpected properties, with very sharp gradients between complex tasks that can be performed effortlessly and relatively simple tasks that are effortful and error-prone. Visual processing, natural language processing, and many other types of cognition exhibit similar properties. In the next section, the discussion turns to an organizational level of analysis.

2.2 Coordination

One of the fundamental problems of software engineering is that design decisions constrain other design decisions in ways that are often hard to describe and difficult to understand completely. Despite the advances in software architecture, the use of information hiding in object-oriented design, and advances in programming languages, software is still full of interdependencies [11]. The problem is made much worse by the fact that design decisions are distributed across time and over people. Managing these interdependencies is the problem of coordination [29].

Coordination problems are pervasive in large projects (see, e.g., [10]. I would argue that many of what we consider the key advances in our field, such as modular design, are important precisely because they address the coordination issue. Modular software is better than monolithic software because individual teams can work on modules without being overwhelmed with the need to communicate about design decisions outside the team [7]. In Parnas's classic paper that began the movement toward information hiding and modularity, he was quite explicit that by "module" he meant "a responsibility assignment rather than a subprogram" [31] p. 1054.

Despite these insights of 30 years ago, we have not yet fully come to grips with the certainty that product architecture and organizational structure are intimately related. In fact, architectural innovation has led to the failure of product firms because they were unable to adjust organizationally [20]. An ethnographic study of software architects in a large firm found that they spent a large proportion of their time and energy engaged in "social engineering" in order to design an architecture that "fits" the organization [18]. If we assume that we can design architectures purely on technical grounds, we place our organizations and our customers at risk, but as yet we understand relatively little about how to think about this problem, beyond the speculation that there is a homomorphic relation between units of the product and units of the organization [7]. We need interdisciplinary research to understand the constraints that architectures impose on organizations, and that organizations impose on architectures, and how technical and organizational structures can co-evolve.

Extreme cases are often particularly interesting, and in open source software development we have and extreme case of geographically-distributed development. There has been much research interest in open source software, but the vast majority of work has been done by economists and management scientists, who are typically interested in questions of developer motivation and allocation of resources. Open source also presents questions of great interest to software engineers. For example, how can these widely-distributed projects succeed at all, give that they generally have no hint of defined process, relatively unsophisticated if serviceable tools [19], very sparse collaboration technology, virtually no management in the traditional sense [16], minimal if any plans, little if any requirements gathering and analysis beyond change requests and e-mail lists, [33] generally no system testing before release [30], and a host of other egregious violations of customary practice. And all these motley collections of volunteer techies have managed to do is create and maintain much of the software that runs the internet, a couple of highly-competitive operating systems, a web server that dominates the market, and much more.

Research to uncover the complex relationships between the code structure and organizational structure in open source is still in its infancy (see, e.g., [27]). Open source may have much to teach us about how to loosen the constraints between organization and architecture. Yet we really do not yet know much about the true capabilities of open source practices. For example, open source projects generally begin with a working system, built in the usual ways by an individual or co-located team. Open source practices are used for evolution and maintenance. Is it necessary to have a proto-system to create a common vision of the product? In what other ways can this coordinating function be performed? Can open source practices and tools be effective in earlier stages of development, such as high-level design? Are there particular architectural characteristics that support such a distributed development style? Can open source practices succeed in an industry environment? We have much to learn from open source.

There are theoretical views of coordination that help to explain the mechanisms underlying coordination, and for the most part they are theories from outside software engineering. One view, originating with Tom Malone [29] takes a radically interdisciplinary approach, noting that coordination problems and solutions have a similar character and structure in many fields. For example, the problems of humans competing for floor space and programs competing for memory have similar characteristics, since both are instances of a resource conflict. Independent of discipline, one could theoretically catalog all types of dependency patterns, and identify mechanisms (e.g., scheduling) that can resolve each type of conflict [9, 11]. Another view, with origins in sociology, views coordinated activity as, in effect, a distributed cognitive system that includes people, artifacts, and practices [24]. Careful ethnographic studies have revealed in great detail how many such systems work, including several radically different kinds of manual navigation, and the activity in the cockpit of an airplane. Finally, distributed AI deals with coordination among agents, and much of their machinery for doing so can be seen as a theory of coordination. For example, Durfee and colleagues [13, 14] have created hierarchical multi-dimensional spaces in which agents exchange information about their own activities in adaptive ways, attempting to minimize communication overhead while facilitating interaction.

Based on these theoretical considerations, colleagues and I have begun to think of coordination in software engineering as occurring along multiple dimensions [23]. For example, in a project where multiple teams or even multiple organizations are involved, one can impose different levels of uniformity. One can, for example, have more process or less process, i.e., processes defined at a very detailed level, or at a very high level, or anywhere in between. One can share design and implementation knowledge at a variety of levels, from just interfaces to the entire code base. Detailed project plans can be shared, or groups can plan independently, sharing only certain milestones. These and other "dimensions of coordination" certainly have complex interrelationships and enable a variety of tradeoffs. For example, can we reduce the detail in our process if we do more work to carefully define our technical interfaces? Understanding how to pick good points in this multi-dimensional space, and how to recognize early signs of breakdown and how to respond to them is our ongoing research program.

3. Barriers to Interdisciplinary Research

In hopes that I have convinced you there is good reason to think that interdisciplinary research is important to the future of our discipline, I want to conclude with a brief discussion of how to facilitate it. Some of the most important steps we can take are to create a culture that nurtures it. Based on my experience in institutions that have successfully fostered such research, I have several observations about the barriers that need to be eliminated in order to create such a culture.

The Universal Principle of Interdisciplinary Contempt. It is extremely difficult not to be dubious about the way someone in another discipline talks about a problem that is of interest to you. After all, we spend many years, from graduate school on, learning how to savage shoddy research. And people in other disciplines often talk in odd ways – they miss the point, they have no sense of rigor, they are unfamiliar with basic concepts and tools. Of course, these people are also sizing you up in similar fashion. Interdisciplinary collaboration requires a temporary "suspension of contempt" if I may call it that, until one understands something of the nature of the problems, approaches, and style of thinking typical of the alien discipline.

The Universal Management Principle: Everything I don't understand is simple. I can't find the reference, but if I recall correctly, this principle has its origin in Dilbert. It was explained as the principle that makes management possible. Just as what is near seems full of detail and what is distant seems just blobs lacking nuance, my discipline is incredibly challenging and important, and your discipline (of which I achieved sufficient understanding by reading an article in *USA Today*) is largely full of people who couldn't make it in my discipline. This very helpfully reinforces the contempt principle.

Administrivia. Roughly speaking, the administrivia burden increases approximately as the cube of the number of administrative units involved. If you collaborate outside the department on a grant or to teach a course, there will be contention over the revenue implications. Our students can't get full credit for taking their courses without forms, justifications, appeals to committees. And then of course, there are the really serious administrative issues. Publications in other disciplines won't count toward tenure or promotion. In fact, there may not even be an obvious place to publish the work.

Border Defense. Software engineering is usually housed in computer science departments or schools, and we are often regarded as a sort of fringe discipline by the more purist among our computer science colleagues. Interdisciplinary research may seem completely beyond the pale. In some environments, I have heard people asking the question, of one research program or other, "But is that really computer science?" Many people spend significant time and energy worrying about this question, apparently. The best answer I've heard was given by my colleague Randy Pausch, co-director of the Entertainment Technology Center in the School of Computer Science at CMU. When a student asked whether a project he, the student, was considering was "really" computer science, Randy replied, with a hint of impatience, "Do something great; we'll decide what to call it later!"

Practical application on a per-paper basis. In software engineering, we have a strong if unfortunate tendency to think that every paper should show a practical result that is immediately useful. We don't always achieve that, by any means, but it is rarely questioned as a goal. I would like to question it. To return to the medical analogy I borrowed from Brooks at the beginning of this paper, medicine builds on fundamental research into how biological systems work. We are well aware that it may take many years for a discovery to lead to some useful result. Applying to medicine the rather impatient standard we apply to ourselves, all energy would be diverted to clinical trials of someone's latest brainstorm, rather than the steady formulation and test of theories to identify fundamental disease processes, principles of pharmacology, and so on. I think it is quite reasonable to expect *programs* of research to lead to practical results, but we need to spend some time and energy understanding how things work.

4. Conclusion

As a field we have benefited enormously from our borrowings from behavioral science. A notable example is adapting and applying empirical methods to experimentally validate and accumulate results (e.g., [3, 37]). We have borrowed theories, especially from economics, and turned them to good advantage as tools to use in technical and project decision-making (e.g., [4, 35]). We need to continue in this strong interdisciplinary path, and as others have also argued [2], nurture our own theoretical tradition. We will need to draw broadly on many disciplines in order to succeed.

5. References

[1] Baron-Cohen, S., et al., Another advanced test of theory of mind: evidence from very high functioning adults with autism or asperger syndrome. Journal of Child Psychology and Psychiatry, 1997. 38: p. 813-822.

[2] Basili, V.R. The Role of Experimentation in Software Engineering: Past, Current, and Future. in 18th International Conference on Software Engineering (ICSE 18). 1996. Berlin, Germany: IEEE Computer Society Press.

[3] Basili, V.R., et al. The Software Engineering Laboratory--An Operational Software Experience Factory. in International Conference on Software Engineering. 1992. Melbourne, Australia.

[4] Boehm, B., Software Engineering Economics. 1981, Upper Saddle River, New Jersey: Prentice Hall.

[5] Brooks, F.P., No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, 1987. 20(4): p. 10-19.

[6] Brooks Jr., F.P., The Mythical Man-Month: Essays on Software Engineering. 20th Anniversary ed. 1995, Boston, MA.: Addison-Wesley Publishing Company Inc.

[7] Conway, M.E., How Do Committees Invent? Datamation, 1968. 14(4): p. 28-31.

[8] Cosmides, L. and J. Tooby, Evolutionary Psychology and the Emotions, in Handbook of Emotions, M. Lewis and J.M. Haviland-Jones, Editors. 2000, Guilford: NY, NY.

[9] Crowston, K., A taxonomy of organizational dependencies and coordination mechanisms, in Tools for Organizing Business Knowledge: The MIT Process Handbook, T.W. Malone, K. Crowston, and G. Herman, Editors. 2003, MIT Press: Cambridge, MA.

[10] Curtis, B., H. Krasner, and N. Iscoe, A field study of the software design process for large systems. Communications of the ACM., 1988. 31(11): p. 1268-1287.

[11] Dellarocas, C., A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components, in Center for Coordination Science. 1996, Massachusetts Institute of Technology: Cambridge, MA.

[12] Dijkstra, E., On the cruelty of really teaching computer science. Communications of the ACM, 1989. 32(December): p. 1398-1404.

[13] Durfee, E.H., Organisations, Plans, and Schedules: An Interdisciplinary Perspective on Coordinating AI Systems. Journal of Intelligent Systems, 1993. 3(2-4): p. 157-187.

[14] Durfee, E.H. and T.A. Montgomery, Coordination as Distributed Search in a Hierarchical Behavior Space. IEEE Transactions on Systems, Man and Cybernetics, 1991. 21(6): p. 1363-1378.

[15] Evans, J.S., Logic and human reasoning: an assessment of the deduction paradigm. Psychological Bulletin, 2002. 128(6): p. 978-996.

[16] Fielding, R.T., Shared Leadership in the Apache Project. Communications of the ACM, 1999. 42(4): p. 42-43.

[17] Frith, C.D. and U. Frith, Interacting Minds: A Biological Basis. Science, 1999. 286: p. 1692-1695.

[18] Grinter, R.E. Systems Architecture: Product Designing and Social Engineering. in International Joint Conference on Work Activities, Coordination, and Collaboration. 1999. San Francisco, CA.

[19] Halloran, T.J. and W.L. Scherlis. High Quality and Open Source Practices. in Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering. 2002. Orlando, FL.

[20] Henderson, R.M. and K.B. Clark, Architectural innovation: The reconfiguration of existing product technologies and hte failure of established firms. Administrative Science Quarterly, 1990. 35(1): p. 9-30.

[21] Herbsleb, J.D. Metaphorical Representation in Collaborative Software Engineering. in International Joint Conference on Work Activities, Coordination, and Collaboration. 1999. San Francisco, CA.

[22] Herbsleb, J.D. and A. Mockus, An Empirical Study of Speed and Communication in Globally-Distributed Software Development. IEEE Transactions on Software Engineering, 2003. 29(3): p. 1-14.

[23] Herbsleb, J.D., D.J. Paulish, and M. Bass. Global Software Development at Siemens: Experience from Nine Projects. in International Conference on Software Engineering. 2005. St. Louis, MO.

[24] Hutchins, E., Cognition in the Wild. 1995, Cambridge, MA: The MIT Press.

[25] Leslie, A.M., Pretense and representation: The origins of "theory of mind". Psychological Review, 1987. 94(4): p. 412-426.

[26] Liang, D., R. Moreland, and L. Argote., Group versus individual training and group performance: The mediating role of transactive memory. Personality and Social Psychology Bulletin, 1995. 21: p. 384-393.

[27] MacCormack, A., J. Rusnak, and C. Baldwin, Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code, in Harvard Business School Working Paper. 2004: Boston, MA 02163.

[28] Malle, B.F., Folk theory of mind: Conceptual foundations of human social cognition, in The New Unconscious, R. Hassin, J.S. Uleman, and J.A. Bargh, Editors. 2005, Oxford University Press: New York.

[29] Malone, T.W. and K. Crowston, The interdisciplinary theory of coordination. ACM Computing Surveys, 1994. 26(1): p. 87-119.

[30] Mockus, A., R. Fielding, and J.D. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology, 2002. 11(3): p. 309-346.

[31] Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 1972. 15(12): p. 1053-1058.

[32] Pinker, S., The Language Instinct: How the Mind Creates Language. 1995, New York: Harper Collins.

[33] Scacchi, W., Understanding the requirements for developing open source software systems. IEE Proceedings on Software, 2002. 149(1): p. 24-39.

[34] Shaw, P., et al., The impact of early and late damage to the human amygdala on `theory of mind' reasoning. Brain, 2004. 127: p. 1535-1548.

[35] Sullivan, K.J., et al., Software Design as an Investment Activity: A Real Options Perspective, in Real Options and Business Strategy: Applications to Decision Making, L. Trigeorgis, Editor. 1999, Risk Books: London. p. 215-262.

[36] Susser, M., Glossary: causality in public health science. J Epidemiol Community Health, 2001. 55: p. 376-378.

[37] Tichy, W.F., Should Computer Scientists Experiment More? IEEE Computer, 1998. 31(5): p. 32-40.

[38] Vandenbroucke, J.P., Changing images of John Snow in the history of epidemiology. Soz.- Präventivmed., 2001. 46: p. 288-293.

[39] Zeki, S., Vision of the Brain. 1993, Oxford: Blackwell Science.