

Human-Centric Software Engineering

Gail C. Murphy
Department of Computer Science
University of British Columbia
murphy@cs.ubc.ca

ABSTRACT

Research into how humans interact with computers has a long and rich history. Only a small fraction of this research has considered how humans interact with computers when engineering software. A similarly small amount of research has considered how humans interact with humans when engineering software. For the last forty years, we have largely taken an artifact-centric approach to software engineering research. To meet the challenges of building future software systems, I argue that we need to balance the artifact-centric approach with a human-centric approach, in which the focus is on amplifying the human intelligence required to build great software systems. A human-centric approach involves performing empirical studies to understand how software engineers work with software and with each other, developing new methods for both decomposing and composing models of software to ease the cognitive load placed on engineers and on creating computationally intelligent tools aimed at focusing the humans on the tasks only the humans can solve.

Categories and Subject Descriptors

D.2.0 [Software Engineering]: General—*research direction, interdisciplinary*

General Terms

Experimentation, Human Factors

Keywords

human-computer interaction, methods, processes, tools, artificial intelligence, machine learning

1. INTRODUCTION

Science fiction writers often speculate about situations in which software is intelligent, sufficiently so to perhaps even program itself. Perhaps luckily, we have not yet entered into

situations where software can determine its own actions or evolve to meet new needs. Rather, software engineering is, and should remain, a human-intensive activity. Despite the central role of humans using the tools, methods and processes that support software engineering, the focus of much software engineering research is on improving artifacts that support, or are the end goal, of the engineering rather than on ensuring the abilities of the humans involved in the activities of engineering the software are amplified to the greatest degree possible. As one example, a substantial amount of research considers how to express the abstractions describing software in models rather than in source code. However, little attention has been paid to how the software engineers using the models reason about the eventual system through the models.

I argue that a human-centric approach to software engineering can help accelerate our ability to build complex software systems with desired qualities. A human-centric approach would involve research focused on how humans work with computational structures and with each other. A human-centric approach would also consider extensions to existing research to consider how humans can work with artifact-centric research results. Finally, such an approach would involve the development of limited intelligence models and tools to allow software engineers to focus those aspects of a development project requiring human creativity and judgement.

To give a sense of human-centered versus artifact-centered results, I first outline the differences between the two approaches in terms of vignettes in three areas of software engineering research results: tools, methods and processes. I then sketch how research agendas might change to accommodate human-centered software engineering.

2. HUMAN VERSUS ARTIFACT-CENTRIC

To motivate the need for human-centric software engineering and to clarify its differences from an artifact-centric approach, I present three vignettes. One vignette is provided for each of three major areas of software engineering research: tools, methods and processes. To help make the differences in approaches stand out when describing these vignettes, I use Alex as the name of a software engineer using artifact-centric tools, methods and processes and Henry as the name of a software engineer using human-centric tools, methods and processes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...\$10.00.

2.1 Tools

Software engineers use many tools every day to help build the artifacts that are part of the system to be deployed.

One of Alex's responsibilities is writing code in Java that will become part of the deployed product. When Alex needs to trace down a problem in the execution of the deployed product, he uses a debugger. The debugger requires Alex to take questions he has about execution and translate them into detailed artifact-centric actions, such as watching particular variables or breaking execution at particular points in methods.

One of Henry's responsibilities is the same as Alex, writing code in Java that will become part of the deployed product. When Henry needs to trace down a problem in the execution of the deployed product, he uses the Whyline tool [7]. Whyline allows Henry to answer many of his questions directly by asking why an object has a value that it does or why a piece of code was not executed. In answer to these kinds of questions, the Whyline tool provides an explanation that focuses on information the user is likely to find familiar or relevant. I claim that the design of the Whyline tool is human-centric; it answers a human's questions rather than forcing the human to think and act in terms of the artifacts comprising the system.

2.2 Methods

Software engineers also use methods to help design and build the artifacts that will comprise the system.

Before Alex codes, he uses UML models to express a design of the intended code. When expressing the design using UML, Alex breaks different aspects of the design into different models. For example, he uses a UML class diagram to capture intended Java code structure and he uses a UML sequence diagram to express how the code will behave to perform desired functionality. This separation of concerns between different models helps Alex focus on different aspects of the design individually. However, Alex's goal is to produce a design that creates a system with a set of desired behaviours. Reasoning about the behaviours of the system across these different models requires Alex to think really hard: he must compose the separated models cognitively to consider where the separate models compose to provide the desired system behaviours.

In contrast, Henry is designing a system using Statecharts with Statemate [4]. The method and supporting tools allow Henry to execute the design, which enables Henry to determine if the stated design provides the desired system behaviours. The fact that the method allows Henry to execute the design relieves Henry of the cognitive effort of composing different aspects of the Statecharts model together. I claim that the executable model approach is more human-centric; it allows Henry to expend more of his cognitive effort on the task of producing a desired design than Alex who wastes cognitive effort composing models mentally.

2.3 Processes

Software engineers rely on processes to organize the activities they undertake when building a system.

Alex works in a group that uses a strict waterfall approach to software development. Alex participates in the design of the system as part of his team. When the design is considered complete, as the team is small, the team moves on to the implementation phase of the project. When problems asso-

ciated with incomplete design decisions arise during implementation, decisions are made primarily by individual team members rather than the team as a whole reconsidering the design. Centering the process around artifacts does not allow Alex and his team members to easily reconsider design decisions as a group, leading eventually to a drift between the design documents and the implementation.

In contrast, Henry works as part of group that uses an agile approach to software development. Henry and his team organize their work into sprints that represent deliverable features. The team meets each day to prioritize work and consider effects of an individual's work on the team. The team is able to deal with decisions that effect the design as a group when necessary. I claim that this agile process is more human-centric; it allows Henry and his team mates to spend their energy on the problems at hand and admits the need for iteration of humans to produce complex artifacts.

3. A WAY FORWARD

How do we make software engineering and software engineering research more human-centric?

One way is to have better descriptions and theories available of how software developers work. For example, how do software engineers think about designs expressed as separable models? How do software engineers manage inconsistencies that arise between artifacts, such as designs and implementations? What vocabulary do software engineers use to explain parts of software artifacts, such as code? What does this vocabulary tell us about how software engineers reason about software?

Gathering information to answer these kinds of questions will require empirical studies into how software engineers work individually and collaboratively to produce software systems. Some empirical studies conducted over the past forty years have started to gather this information. For example, Soloway and Ehrlich performed empirical studies to show that expert programmers use programming plans and rules of programming discourse that distinguish them from novice programmers [8]. More recently, Ko and colleagues analyzed the work of Microsoft software engineers to identify the information needs of software engineers at work [6]. Many more studies like these are needed to cover the wide range of activities of software engineers.

Information from such studies can trigger and support the creation of new tools, methods and processes. For example, the code bubbles tool for programming [2] addresses the need for programmers to form, manage and change working sets of code when making changes to a system, determined from a previous empirical study [5]. The information gathered can also be used to form theories about why certain tools, methods and processes may be more effective than others. These theories can be used to design approaches that will be more effective for humans. Storey discusses some ways in which empirical results and theories for program comprehension can be used [9]. More work in developing, evolving and using theories to spawn the next generation of results is needed.

Another way to move forward is to improve a software engineer's work by aiming to amplify human intelligence. For example, can we use results from machine learning and artificial intelligence to increase the information density presented to software engineers? Can results from human-computer interaction be used to provide collaborative in-

formation, such as team awareness results, in ways that do not take a software engineer away from cognitively intensive tasks, such as fixing bugs? Can we introduce agents into the tools software engineers use to help sift through, filter and direct the attention of software engineers to likely pertinent information?

Relatively few projects focus on amplifying human intelligence. As one example, Fritz and colleagues introduced a model that maps the knowledge of a software engineer in a code base based on how the engineer and their team authors and changes code [3]. Such a model can be used in several ways, including filtering bug notifications to focus on those related to code for which an engineer has a high degree-of-knowledge. As another example, Bessey and colleagues describe different trade-offs they have made in showing results produced by the Coverity static analysis tool to humans [1]. Many more possibilities exist to amplify a software engineer's intelligence and focus their effort on tasks that matter rather than tasks that can be performed by a machine.

4. SUMMARY

Software is created by humans, often for, humans. In this position paper, I have argued we need to take an increased human-centric approach to software engineering and software engineering research. Such an approach may involve increased studies to understand how developers work. Such an approach may involve the use of techniques from other disciplines to amplify human intelligence in the cognitively challenging process of creating large, complex, correct software systems.

5. ACKNOWLEDGMENTS

Thanks to Meghan Allen, Thomas Fritz, Emerson Murphy-Hill and David Notkin for helpful comments on an earlier version of this position paper.

6. REFERENCES

- [1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.
- [2] A. Bragdon, S. P. Reiss, R. C. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proc. of International Conference on Software Engineering*, pages 455–464, 2010.
- [3] T. Fritz, J. Ou, G. C. Murphy, and E. R. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proc. of International Conference on Software Engineering*, pages 385–394, 2010.
- [4] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Software Eng.*, 16(4):403–414, 1990.
- [5] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ideas: a detailed study of corrective and perfective maintenance tasks. In *Proc. of International Conference on Software Engineering*, pages 126–135, 2005.
- [6] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of International Conference on Software Engineering*, pages 344–353, 2007.
- [7] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proc. of International Conference on Software Engineering*, pages 301–310, 2008.
- [8] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10(5):595–609, 1984.
- [9] M.-A. D. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.