The Work of Software Development as an Assemblage of Computational Practice

Susan Elliott Sim
Dept. of Informatics
University of California, Irvine
sesim@uci.edu

Marisa Leavitt Cohn
Dept. of Informatics
University of California, Irvine
mcohn@uci.edu

Kavita Philip
Dept. of Women's Studies
University of California, Irvine
kphilip@uci.edu

Abstract

Science and technology studies (STS) is a discipline concerned with examining how social and technological worlds shape each other. In this paper, we argue that STS can be used to study the work of software development as a complex, interacting system of people, organizations, culture, practices, and technology, or in STS terms, an assemblage. We illustrate the application of these ideas to the work of software development, where STS theory directs us towards examining at human-human relations, human-machine relations, and machine-machine relations. We conclude by discussing some of the challenges of applying STS in empirical software engineering.

1 Introduction

Software development is a complex activity with involving software, hardware, individuals, groups, and organizations. Many different disciplines have been applied to provide insight into some aspect of the activity. But each discipline comes with a set of intellectual commitments that partitions the problem in a particular way.

From a computer science perspective, source code is given primacy in the work of software development —programming, debugging, testing, and so on. Computer scientists tend to see source code as the foundation that makes everything else work, because it is the source of the program. In this view, it is the source, or origin, of a practice, that defines its essence. But this is only a partial view. From a software engineering perspective, design is given primacy in the work of software development—specification, implementation, and so on. In this view, it is the design activity that defines the essence of the technological practice. From a sociology of work perspective, people participating in social structures and contingencies are given primacy. In this view, it is the people who really decide the nature of technology. From a political science perspective, power is given primacy—the need to exert and accumulate influence and control. In this view, it is the ideological factors external to the technology that really shape it. While these, and other, perspectives are valid, they foreground one aspect of the activity at the expense of another.

It would be more accurate to say that the work of creating computer software is all of these things together. But it cannot be defined by simply defining the parts separately, and adding them up. The whole is, literally, greater than the sum of its parts. We need a theory that describes the interactive working of multiple elements together. This activity involves a large collection of people, factors, contexts, networks, machines, and artifacts that work together.

In this paper, we argue that Science and Technology Studies (STS) is an approach that will help us examine the work of software development as a complex whole. The field of STS examines the co-constitution of society and technology. Contemporary research in STS claims that science and technology both shape and are shaped by culture, society, politics and individuals. From this theory, we introduce the idea of the work of software development as an assemblage, that is, a complex of system of interacting parts shaped by internal and external influences. The particular form and structure of the assemblage constrains some activities, and energizes others. The capabilities and limitations of each element have an effect on the others. By adding this analytical lens to study the work of making computer programs, we are better-equipped handle the integrated complexity of this human endeavor, that is, the complex interacting whole and the constituent parts.

Applying STS to software development has a number of benefits, such as the availability of novel methods and analytical lenses. But there are drawbacks as well; the form and content of the products STS analyses are foreign to computer scientists and software engineers. The outputs are typically long essays that critique existing configurations and

challenge assumptions, which require reviewers to step outside of their comfort zones.

2 Science and Technology Studies

In this section, we give a brief introduction to STS. Well-known works in the field include Kuhn's *The Structure of Scientific Revolutions* [4], Latour's *Science in Action* [5], Knorr-Cetina's *Epistemic Cultures* [3]. The principle that society and technology are co-constituted is well established. Societal beliefs, social infrastructures, and political regimes, all shape technology. The relationships among humans, ranging from the economics of funding to political climate at the macro level, to individual designer's worldviews, shape technological design choices [6].

Scholars in STS have long argued that popular accounts of scientific knowledge production, as rigorous and methodical, are inadequate to the complexity of processes of scientific practice, as opportunistic and context-dependent. There are clear parallels between knowledge construction in the sciences, and system construction in software development. For instance, new understandings of the world external to the scientist or software developer are necessary to accomplish their respective tasks. STS has deployed several successful techniques to investigate the sociology of scientific and technological knowledge, including history (the analysis of scientific case studies from prior periods), ethnomethodology (observing practicing scientists), textual analysis (studying scientific explanations in documentation), and institutional studies (studying the working of scientific organizations) [1, 6, 8].

Methods in STS include close reading of texts and artifacts, observation and interruption of activity, interviewing and interrogating participants, and paying attention to history and social practices. Some of these methods are familiar to empirical software engineering, while others are not. One important distinction is social science methods attempt to collect data more "objectively," without perturbing the subject under study, whereas STS is less reluctant to engage and challenge individuals and processes to gather data. As well, the primary products of STS are critiques published as monographs, a practice more common to the humanities than the sciences. The only significant critique that we have in software development is Brooks' essay, *No Silver Bullet* [2].

The methods of STS were created to study science and technology in general and we believe that they can easily be applied to computational systems and programming practices. In studying the work of software development, it is possible interactively observe software developers, and conduct close textual analyses of code and software documentation, and draw on institutional and historical studies of the rise of computational technology and its attendant social practices.

A key idea in STS is assemblage**,** which Murphy defines as "as an arrangement of discourses, objects, practices, and subject positions that work together within a particular discipline or knowledge tradition. It is not the list of elements that make an assemblage consequential, it is what they made possible by the ways they articulated each other" [7]. Applying this concept to the work of software development draws our attention to not just the actors and the technology, but also how they mutually define each other and the possibilities created and eliminated by particular arrangements. Consequently, we can symmetrically consider the contexts of machinic work (running, compiling) and human work (debugging, documenting, implementing).

3 The Sites of Code Work

In this section, we apply STS principles to the work of software development to illustrate the kinds of perspectives and inquiries that are possible. We apply the concept of assemblage to source code. This mode of analysis directs us to examine how artifacts and participants are mutually co-constructed. In other words, in what ways has source code come to be as a result of being created by people working in certain contexts. By the same token, how have people been shaped by the exigencies of source code.

For those who seek to understand the technical challenges of software development, we wish to demonstrate that source code is not the whole story. For those who believe that social worlds shape computational problems, we wish to demonstrate that social pressures are not the whole story, either. The rest of the story has three parts: human-machine relations, human-human relations, and machine-machine relations. Our approach to opening up the boxes of social and technical interaction is based on investigating these three areas, which together form an assemblage that affects the work of software.

3.1 Human-Human Relations

> *Pay attention to the various levels at which software is embedded within human-human relations*.

Our attention is drawn to the types of human-human interactions that are central to the software development process. For instance, what kinds of

human-human relations are prescribed by different kinds of software processes? There currently exist many different methodologies for software development, and each facilitates a different mode of operation and community of practice. Observing and interacting with developers who participate in Agile and open source leads us to ask questions about human-human relations in these processes, such as: What is the coder's role in the overall design process? What is considered good software work in this model? Does the developer have contact with the users of the software? Are they given room to interpret specifications or make design decisions? What are the values of the communities in which software developers work?

These contrasting processes also influence the software itself, how it is envisioned, and the worlds that the software constitutes. The way that a piece of software frames the technological work done by users differs greatly when that software is continually evolving with the users' work practice versus when it does not. However, when it comes to software, these social arrangements are often overlooked. They may impact the lives of the software engineer, but are not considered relevant to the outcomes of software. It seems unlikely that they have no impact at all. User participation in the design of software has not been extended to participation in coding practice. Because coding is deemed to occur only when a programmer sits in front of a keyboard, we have very few alternative notions of what participation in coding could look like. Should the act of coding not also include standing in front of a whiteboard, brainstorming about design options, and talking to users?

3.2 Machine-Human Relations

Interpret coding as a form of writing by which humans instruct computers, and investigate the broader social and technical construction of human-code interactions.

In some ways, the work of software development as machine-human relations is well understood. Entire areas of computer science are devoted to this topic, such as programming languages, compilers, program comprehension, software process, human-computer interaction, and computer-supported collaborative work. But the activity still warrants further scrutiny. Drawing on the lessons from STS, it is not sufficient to look at the apparent action. It is necessary to examine also the meanings of the activity, people's beliefs when they engage in the activity, and how people are affected when they engage with the technology. In other words, we must investigate human-machine relations not just in terms of direct instruction, but in terms of its broader social embeddedness.

To begin, what should properly be construed as computer programming? The first image that comes to mind is a specialist software developer sitting at a keyboard entering instructions in a programming language, such as Java or C. Clearly, this activity is very different from creating punch cards for an IBM 704 computer or connecting processor units in the ENIAC using wires. Yet those activities are also a form of programming. What of graphical user interface programming, where the application developer is engaged in prescribing windows, menus, and dialog boxes by clicking and dragging? What of end user programmers who are implementing business rules and scientific equations in a spreadsheet? Finally, what kind of activity are users engaged in when they have the know-how to press a key combination, such as control-alt-delete? The action of programming the computer is commonly seen as less sensitive than other macro-social factors, but with a more diverse conception of computer programming also comes a broadening of the category of who can be properly labeled a programmer.

Reducing the activity of programming to typing also reduces the programmer to a kind of intelligent input device. So the question arises, what precisely is being programmed? Perhaps it is the human operator who must deal with a myriad of interfaces and existing pieces of software in order to build new software. As well, a large body of "technical" knowledge is necessary to program effectively. Perhaps it is the human operators who will be using the software who must be trained to use an interface, prepare their data in a particular fashion, or change how they do business to fit within the parameters of the software.

3.3 Machine-Machine Relations

Pay attention to the ways that machine functionality is embedded in a technological stack.

Machine-machine relations are the ways in which computer programming is constrained by other technologies, both past and present. The most obvious example is the QWERTY keyboard, which is known to be less efficient than alternative designs, but whose persistent legacy constrains future keyboard design. Another example is the E-13B font, created in the 1960s to encode bank, branch, and account information on cheques, and is still with us today. Issues of interoperability and application program interfaces (APIs) are ongoing concerns machine-machine

relations, but are relatively superficial ones in the context the technological matrix supporting computer programming.

Programming languages and software engineering have been concerned with simplifying the work of software development by transferring more of the labor of creating and testing software applications to the computer. Early computer programs were written entirely in binary and machine instructions. Assembly language introduced human-readable codes and mnemonics, which could then be translated into machine instructions, at first by humans and later by a computer program. The first high-level programming language, FORTRAN, was created to further simplify programming and make the profession of programming accessible to more workers. This "automatic" programming was made possible by a compiler that translated statements into machine instructions and arranged them optimally in the computer's memory. Many other innovations have been introduced including time-sharing machines, video display terminals, editors, debuggers, integrated environments, version control systems, and recently workbenches for model-driven architecture. Coding practices such as testing and documentation have always been seen as peripheral and not-quite coding, but we take all these layers are relevant to the construction of software.

These technologies made programming more "abstract" and "easier" for people. The result is that software developers are progressively more distant from the controlling computer hardware. In reality, the software developer is no longer instructing the machine directly. Rather, the coder interacts with a stack of technology that has been created by people working within socio-technical systems at different points in time. But this layering of elements of the computer is due to the historical layering of design decisions that define contemporary programming practices, and not to an inherent computational need. Hardware design develops in ways that are constrained by and contingent on machine-machine interactions of various kinds. For example, the compiler is not a necessary or determined technical construct; it both shapes and is shaped by computer software. Our inherited models of this technological stack can limit our ability to conceive of what is "computable."

4 Discussion

The empirical and analytical methods of STS provide a different approach to examining the work of software development. Despite the richness of the research questions and complexity of the answers, applying STS to software development is challenging in a number of ways.

Scientific and technical approaches prefer to simplify problems, to distill them down to an equation or model. In contrast, STS seems to revel in the complexity of problems. So a description that appears to be marvelous to an STS scholar appears unfinished to an engineer.

Results in STS tend to be book length monographs of carefully constructed prose. Such texts are virtually impenetrable to someone schooled in computer science and trained to read technical papers with section headings. But the aspect of these essays that is most antithetical to software engineering is these critiques question existing configurations and challenge assumptions. These results require reviewers to step outside of their comfort zones within the dominant paradigm. Last, but not least, STS is very effective in raising questions, but less effective in answering them. The results do not provide ready solutions or prescriptions for the technology-builder to apply.

However, we believe that STS can be particularly valuable for asking blue-sky questions about what alternative technologies might look like. By identifying hidden assumptions and constructions, we can think outside the box and ask what-if questions about how computing might be different. For instance, what might software development look like if the first application domain were music, and not ballistics tables? Insights from STS can point the way to software technologies that were not previously considered.

5 References

[1] Wiebe E. Bjiker, Thomas Parke Hughes, and Trevor J. Pinch, *The Social Construction of Technological Systems*: The MIT Press, 1990.

[2] Frederick P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer,* vol. 20, pp. 10-19, 1987.

[3] Karin Knorr-Cetina, *Epistemic Cultures: How the Sciences Make Knowledge*: Harvard University Press, 1999.

[4] Thomas Kuhn, *The Structure of Scientific Revolutions.* Chicago: University of Chicago Press, 1962.

[5] Bruno. Latour, *Science in Action*. Cambridge: Harvard University Press, 1988.

[6] Donald A. MacKenzie and Judy Wajcman, *The Social Shaping of Technology, Second Edition*: Open University Press, 1999.

[7] Michelle Murphy, *Sick Building Syndron and the Problem of Uncertainty*: Duke University Press, 2006.

[8] Trevor J. Pinch and Ronald Kline, "Users as Agents of Technological Change: The Social Construction of the Automobile in the Rural United States," *Technology and Culture,* vol. 37, pp. 763-795, 1996.