Figure, Ground, and Media: Toward Human-Centered Software Engineering

Position Paper for the Software Engineering Methods and Theory Initiative

Larry L. Constantine, IDSA, ACM Fellow
Institute Fellow, Madeira Interactive Technologies Institute
Professor, University of Madeira, Funchal, Portugal

Author's Note. I have chosen to express my position without reference or reaction either to those position statements of other signatories already posted or to the subsequent exchange regarding those postings.

Introduction

I have been inspired by the literal metaphor of a *position* paper to locate my position on the future agenda and direction for the SEMAT initiative in two distinct dimensions. These dimensions might be labeled in many ways, but I will cast them as *figure* and *ground*.

By *figure*, I mean what is and should be the focus—the subject of attention and the scope of subject matter—in synthesizing a sound basis for software engineering going forward. What should we be talking about and from what perspective or point of view? By the second dimension in which my position may be located, I intend somewhat of a play on the word *ground*, which is not in this case mere background or the dimmed and out-of-focus context of the picture we are painting, but the literal and figurative ground upon which the subject stands, the grounding or foundation on which the scope of our attention rests. A less literary but not necessarily more informative or accurate reading might be attained by referring to these axes as *scope* and *theory*.

Figure – Scope

Why do we build software-based systems? We build them for use. Yet use, users, user interfaces, and usability have, historically, been either ignored in software engineering methods or, at best, treated as afterthoughts, to be grafted onto an existing scaffold or shoehorned into an ill-fitting framework. It was understandable, even if regrettable, that the earliest work in software engineering methods largely neglected the interfaces between the software and its human users. As the architect of structured design, arguably one of the first software engineering methods, I also stand accused and enter a plea of guilty. Although it was always my intention to incorporate what we then thought of as "input-output design" in the magnum opus that Ed Yourdon and I wrote, we stopped short of the finish line and went to press with the

keystone of our structure still missing. Our sins have been replicated by nearly every method and methodologist since. User experience design was patched onto the Unified Process and the so-called Unified Modeling Language rather late in the defining game, tossing user experience and interaction designers the scraps of models and notations devised for other purposes and ill-suited to the realities of interaction design.

Agile methods followed suit, and the agile community has since been playing an aggressive game of catch-up ball to incorporate usability and interaction design into their practices. Even in the most open forums, though, the dialogue is almost exclusively about how to adapt usability and interaction design practice to fit with agile approaches and rarely about altering agile methods and manifestos to better fit with the nature of user interface and interaction design.

And now we have more contemporary work on practices over processes which continues to write the same sad history. It is nothing short of appalling that, after something approaching a half century of programming and software engineering methodology and evolution of techniques and technology, the first purpose of software systems is still an afterthought, given at best a small placeholder often labeled “to be determined.”

We design, engineer, and build software systems to be used. For that reason, users, user interfaces, and user experience broadly conceived are not merely a peripheral part of what we do; they are the center, the kernel, the starting point rather than an afterthought. Any method and theory going forward must start from and build upon that premise. Practices need to revolve around and reflect the reality of user requirements. Use, users, user interfaces, and user experience are not merely pieces of the whole, to be slotted into a template where there happens to be space to squeeze them in; they need to be recognized as the figure of the gestalt.

Ground – Theory

Ivar Jacobson is fond of quoting Kurt Lewin, the founder of social psychology, that nothing is so practical as a good theory. However, good theory is far more than practical, because it also transcends and outlasts practice. Practices change, but sound theory abides. It outlasts techniques and the promoters of techniques, branded methods and the marketers of methods; it outlives fashion and style and popularity and success alike.

Good theory, genuine theory as distinguished from mere models (of which software engineering has no shortage), is not just descriptive; it offers explanation and prediction of results as well as the possibility of control over outcomes. Genuine theory is, of course, testable, particularly in Popper’s sense of falsifiability.

My position is that we should begin our work from a base built upon the best and most researched theories and clearest and consistent evidence and then work our way outward and upward. Eventually, we should be aiming to put software engineering on a foundation that is both evidence-based and theoretically well grounded. These are related but not equivalent objectives. Evidence, real data, is a required basis for all engineering practice, but evidence absent theory, whatever its heuristic utility, highlights crucial and potentially risky holes in our understanding. Good data, such as the excellent cumulative work of Capers Jones and his colleagues, is valuable and extremely important but is not enough. It only tells us what is and has been, not what it is about or where we might go.

Software engineering has little in the way of fundamental theory and even less in the way of well tested theory. Good theory does lie behind structured programming, although real-world research has been somewhat scatter-shot. Modern database design and normalization is fairly well-grounded in mathematical formulations, although, here, too, research trails adoption by too much. Program complexity models, such as the venerable cyclomatic metric and the often overlooked software science measures based in information theory, might also be candidates for inclusion in the theoretical groundwork. And, of course, I cannot omit the human-centered theory of software complexity embodied in the constructs of inter-component coupling and

intra-component cohesion, which have been the subject of hundreds of studies and have demonstrated both predictive ability and robustness in the face of fundamental changes of programming paradigm.

The final content of the theoretical grounding remains to be determined, and there may be many more candidates that warrant consideration or inclusion, but the main point is that testable and tested theory—not practices or techniques or concerns or muddled mixtures of these in the form of methods—should be the framework upon which software engineering is organized.

Not surprisingly, I am convinced that any theory of software engineering that does not take into account at its core the fact that software is specified by humans, to be understood by humans, to be modified and extended by humans, and ultimately to be used by humans is fatally flawed.

This is the kernel of software engineering theory and methods, and this is the position where I stand. Anything less than a synthesis that comprises a human-centered software engineering is, in my view, a failure, a failure with precedent in a long legacy of failure, but a failure nonetheless.

There are, of course, many more concerns within our scope and much more to the process ahead for SEMAT, but if we do not start from the right place we will almost certainly end up in the wrong place. Again.

Media - Models and Notation

I would have preferred to stay within the clean and comfortable landscape of the two-dimensional world of figure and ground on which I first plotted my position, but as I finished, I realized I had left out something crucial. Figure and ground must be expressed in some form, pictures are painted in some media. The main media of expression for software engineering are models and modeling languages.

Projecting outward into this third dimension, my position is that a genuine synthesis must begin with a review and, if not a complete reconstruction, at least a radical renovation of the models and modeling notations used as the expressive media of modern software engineering. From the foregoing delineation of my position, it should be quite predictable that the revision and redaction ahead needs to be human-centered. First, notations need to become more usable and user friendly. Arbitrary, unmemorable, and counterintuitive representations (which is most of those in common use today) need to be replaced with symbology that is simple and intuitable while remaining rich and expressive. This is a job for visual and interaction designers who are also software engineering methodologists. (There are two or three of us on the planet.)

Second, the core models of software engineering need to be constructed as an integral and seamless whole rather than a patchwork collection thrown into a bin with commonality and connections buried in their expression within some meta- or meta-meta-language. At the same time, their content, capability, and expressive style need to be suited to the human professionals that use them for different purposes and distinctive ends. The models and notations that facilitate designing software architecture are not necessarily those that aid the design of interaction architecture. (Designing user interfaces with screen stereotypes and sequence diagrams is like wearing shoes of the wrong shape and two-sizes too small.)

Any “unified” modeling language must include models especially tailored to designing all aspects of software systems, including most centrally the interfaces with users. And any truly usable “unified” modeling language must be a language that is itself a well-designed interface to the process of software engineering, one that is tailored to the abilities and limitations of its human users: the analysts, designers, engineers, and programmers who must generate and interpret and revise the models.