Abstract—While the search for a grand theory of software engineering is an honourable search like that of the Arthurian knights of old, a theory of software engineering must be useful to practitioners otherwise such theory is nothing more than a textbook curiosity. We argue that prior attempts to develop rational theories of software engineering have lead to the creation of methodologies that are not used. Further we argue this pursuit is not addressing the impediments practitioners are experiencing which are social in nature. Our research has demonstrated that we can generate useful software engineering theory by gathering field data and generating theory using methods such as Grounded Theory. Using such an approach we were able to generate the theory of Reconciling Perspectives, which explains how practitioners manage the process of software development.

Keywords—theory; software engineering; grounded theory; social processes

I. INTRODUCTION: WHY A THEORY OF SOFTWARE ENGINEERING

Why should we be interested in creating a theory of software engineering? Theory is the foundation of science and provides explanations of phenomena as well as testable predictions. Engineering emerged as a profession in response to the failure of constructed works such as collapsing bridges and exploding boilers [1]. Engineering began applying natural sciences theory to create theoretical models that explained and predicted the behaviour and performance of engineered systems before they were built. Our great grandparent engineers used such models to predict the load that a bridge could support before it was built, or to predict the pressure a boiler could withstand and the amount of steam it would generate.

Software engineering emerged forty years ago during a NATO scientific conference [2] in a response to the realization that large software systems were frequently delivered late – if delivered at all – were over budget with feature deficiencies, and had suspect quality. The hope is we could develop or discover a set of theories that will do for us what the application of theory has done for other more mature engineering disciplines. The “Holy Grail” of software engineering is generating theory that will enable us to codify knowledge [3] and predict the outcome of interventions on software engineering efforts.

However, in our search for the Holy Grail we must ask are we looking for the right one? Engineering is an applied science and this means finding or developing theories that are useful to practitioners. Are we searching for a theory that will help us resolve the problems that are impeding software engineering? In this short position paper, we argue a useful theory of software engineering will be a social theory and present examples of how we have generated a useful theory.

II. ARE SOFTWARE ENGINEERING THEORIES USEFUL?

Engineering is applied science and a theory of software engineering must be useful to practitioners. Practitioners will not use something if they do not perceive it as useful or does not provide a useful explanation of the phenomena they are experiencing. A case in point: our research was motivated by trying to understand why software engineers are reluctant to adopt software methodologies despite evidence software methodologies improve outcomes. Our conclusion was engineers do not find methodologies useful because the mechanistic reductionist models most software methodologies are based on are limiting, and even impeding to the way individuals actually create software [4].

Parnas and Clements [5] characterized the well-intentioned search for the theoretical Philosopher’s Stone of software engineering, a rational software process that would give us a good reason for doing what we’re doing. They expressed their ideal vision for a software methodology, writing: “ideally we would like to derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof”. Parnas and Clements also stated clearly that the Philosopher’s Stone did
not exist because the complexity of software makes an ideal rational process impossible. Despite this, they argued it is still beneficial to approximate or, as they put it, “fake it”.

Most software methodologies try to approximate or “fake” a rational process by following a reductionist framework describing the successive transformation of work products as they flow from one stage to the next [6]. The classic waterfall embodies this reductionist framework and models software development as a linear series of processes where the outputs of the prior process flow down as inputs to the next [7].

While modern software engineering methodologies have moved towards more iterative and incremental models of software development, such as the Unified Process (UP) [8] and its variants, they still model software development as a technical process of reductionist transformation. Each process stage can be described in terms of an Input-Process-Output model (IPO), where inputs arrive and are then processed by transforming them into outputs. This model enables methodology designers to specify the inputs to a process, the outputs, the quality of those outputs, the steps for transforming the inputs into outputs, and the individuals responsible for animating the process. The IPO model for software methodologies is standardized in the Software Process Engineering Meta-model (SPEM) [9]. Many of the modern UP software methodologies, including the Rational Unified Process (RUP) and OpenUP, are built using SPEM as a meta-model.

The problem with software methodologies and the efforts to standardize their definition is it seems no one actually follows them unless coerced to do so [10-13]. Time and time again, when we ask developers why they do not follow the corporate software development methodology, or adopt a “rational” methodology, the response we get is “…because if we ever did it like that we would never get it done”.

In contrast to Parnas’ idealized “faked” rational process, our observation of the methodology-in-use is that it is usually an ad hoc collection of practices agreed to on a situational basis by those immediately engaged in the software development process. The “faking” part is more often the arbitrary and coercive creation of work products (e.g. detailed documentation) that produces a veneer or fiction of a rational process that conceals the irrational process [12]. The strategy of “faking” a rational process is not serving development teams well in practice, and Nandhakumar and Avison concluded from their study that “formal methodologies are too mechanistic to be of much use in the detailed, day-to-day organization of developers activities”. Further, they asserted the risk of faking such a process is that “seeking to impose methodologies to improve the productivity of the developers’ task may be counter productive.”

The IPO model behind most software methodologies simply did not address the issues that are relevant to the practitioner or explain the phenomena they are experiencing. The effort to discover an rational software engineering theory will not address the issues that are impeding people from getting the job done. Despite their theoretical basis on input-process-output models, software methodologies are not used because they do not address the main concerns of practitioners. The lesson is if we do not address the main concerns of practitioners then any grand theory of software engineering will not be useful.

III. SOFTWARE DEVELOPMENT AS A SOCIAL PROCESS

It is immediately clear to an observer that software development is a social process. Other researchers have made the same observation: “Software Engineering is primarily a social and creative process, where the creativity, skill, and cooperation of developers, users, and procurers determine the quality and effectiveness of the developed software” [14]. The social processes are not a side effect of the software development process; the social processes are the control mechanism – as was highlighted by Nandhakumar and Avison’s “The evidence suggests that social controls, such as norms promoting collaboration with colleagues, professional design practices and established routines appeared to be a more significant influence on developers’ work practice at LMC than the requirements of a methodology” [12]. Put in modern agile shorthand: “people trump process” [15].

Over thirty years ago Boehm [16] reported the influence of social factors on the outcome of software engineering efforts: “Personnel attributes and human relations activities provide by far the largest source of opportunity for improving software productivity”. If social factors are the biggest cost drivers and explain variances in the productivity of software development teams, then theories that help us identify and understand social processes in software engineering should be useful and yield significant benefit to the industry.

IV. GENERATING USEFUL SOFTWARE ENGINEERING THEORY

While a grand theory of software engineering may elude us because of the dominant influence of social processes on software engineering we can still generate useful theory that practitioners can use to understand and explain their situation. Grounded theory is one tool available to us to generate substantive theories that explain what is going on. A Grounded Theory is a set of integrated conceptual hypotheses systematically generated to produce a theory about a substantive area [17]. Glaser and Strauss [18] proposed the method as having application for both qualitative and quantitative data. When working with qualitative data, and when compared to other qualitative research methods such as narrative or ethnography, Grounded Theory generates a substantive theory that explains participants’ behavior as a set of integrated hypotheses.
Co-discoverers Barney Glaser and Anselm Strauss called the method “grounded” because a theory is systematically obtained from a broad array of data through a rigorous process of constant comparison—it is “grounded” in the data [18, 19]. Grounded Theory differs from logico-deductive methods of inquiry because, rather than developing a theory without relying on data and then systematically seeking out evidence to verify the theoretical constructs, researchers using Grounded Theory set out to gather data and then systematically develop a mid-level substantive theory derived directly from the data [18]. The goal of Grounded Theory is to generate concepts and categories that account for a pattern of behavior, which is relevant and problematic for those involved [20].

In our field study of practitioners, we used Grounded Theory to generate a substantive theory of how people manage the process of software development. We learned that the main concern of people involved in the process of software development is getting the job done and that different points of view and expectations create impediments—a perspective mismatch. People use a four stage process of Reconciling Perspectives [21] to remove these impediments (cf. fig. 1). When a perspective mismatch is discovered, people converge their mismatched perspectives by reaching out and negotiating a consensual perspective. Constructing the work products and evaluating them validates the consensual perspective. The process may yield accepted work products, providing objective evidence the perspective mismatch was reconciled, or may result in waste.

Reconciling Perspectives is a social process moderated by social dynamics, and creating accepted work products depends on individuals’ abilities to reach out and engage in negotiations while also sheltering themselves from distracting interruptions. This creates a tension in the process that must be managed because to discover and converge perspective mismatches, people must be open to interruptions while also remaining focused on getting the job done.

What Reconciling Perspectives highlights is the need for managed communications and the roles leaders (or “critical individuals”) must play to manage communications. When we explained the theory to practitioners a common comment was “yeah, that’s my life!” From Reconciling Perspectives we were able to explain the role of communications in software projects and that more is not necessarily better. We were able to generate a set of recommendations for practitioners to help them better understand their communication and appropriate interventions leaders could take to avoid the creation of waste. Furthermore, Reconciling Perspectives is generated from field data and therefore gains an element of “street credibility” with practitioners that a logico-deductive approach may not enjoy.

Other researchers have used grounded theory. Ferreira and colleagues [22] study of how two different communities of developers; user interaction designers and agile software developers integrate their work describes a process similar to Reconciling Perspectives. Whitworth’s and Biddle’s study of XP teams investigated the social processes that contribute to the team’s success [23]. Their findings suggest agile practices were seen by the study participants to increase the ability for team members to work together with others in an environment of constant feedback and progress. Hoda, Noble, and Marshall’s [24] investigation of self-organizing teams highlighted the influence of individuals in the roles they play facilitating the organizing of self-organizing teams.

This approach opens the opportunity to create a collection of mid-level theories that can explain the main concern of individuals and teams and how they are resolving their main concern. While a collection of mid-level theories that explains what is going on may not be as mentally satisfying as an elegant grand theory, such a collection will nevertheless is useful and perhaps better captures the messy nature of socio-technical systems. It captures what people are really doing rather than what experts believe they should be doing.

V. THE SEARCH FOR A USEFUL HOLY GRAIL
The effort to find a grand theory of software is much like the legendary search for the Holy Grail: a doomed and honourable test of courage. However, like the Arthurian knights of legend, the search for the Grail was a quest to develop and demonstrate moral leadership in a time when people needed a way to teach moral principles. We need to make decision and interventions on the basis of some useful theoretical model rather than “gut feelings” and hearsay.

A theory of software engineering must explain social processes or it will not be useful to practitioners. Furthermore, we certainly do not believe that attempting to incorporate social practices into a methodology following a formal metamodel such as SPEM will work, or even that it is desirable. Our failure with OpenUP [25] is a clear demonstration of this. Instead, we recommend employing an empirical approach, following the principles of validated learning, and what has widely become known as “Lean Start-up” [26], to design and
deploy a socio-technical methodology. A good start would be using Reconciling Perspectives to explain and highlight the social aspects of existing software methodologies such as Scrum. Feedback from practitioners can then be used to design socio-technical methodologies with greater scope.

REFERENCES


