Possible Core Theories for Software Engineering

Paul Ralph
Lancaster University
Lancaster, UK
paul@paulralph.name

I. INTRODUCTION

Following several calls for developing a general theory of software engineering (e.g., [1], [2]), Ralph et al. [3] suggested that a general theory may be developed by identifying and integrating a range of “core” theories. The purpose of this paper, then, is to review theories that are useful for analyzing software engineering (SE) behavior, but may be unfamiliar to many in the SE community. Five such theories are identified: the Theory of Cognitive Biases, Sensemaking-Coevolution-Implementation Theory, the Theory of Boundary Objects, Transactive Memory Theory and Complexity Theory. Rather than providing contradictory explanations, these theories apply at different units of analysis and may therefore be used simultaneously to understand the same software engineering phenomena.

II. FIVE POSSIBLE CORE THEORIES

A. Cognitive Biases

Beginning at the individual unit of analysis, a cognitive bias is a systematic deviation from optimal judgment [5]. Some existing research has examined the role of cognitive biases in software engineering e.g. [6], [7]. Many cognitive biases are extraordinarily robust – they apply to diverse individuals in heterogenous situations and persist despite explicit attempts to mitigate them [8]. Biases are caused by various cognitive phenomena including heuristics (mental shortcuts) and illusions (systematic misperceptions of reality) [9]. Some biases interact and reinforce each other, forming complexes of biases, or biasplexes [10]. Biasplexes, in turn, lead to behavioral antipatterns (i.e., systematic, repeated errors) [10], which may deleteriously affect software engineering projects [6]. Debiasing involves inhibiting biases or mitigating their effects [8], sometimes through sociotechnical intervention in the person-task system (Figure 1).

Cognitive biases may affect software project participants in many ways. For example, suppose a team is asked to produce a video game for the iPhone. Due to miserly information processing (the tendency to avoid thinking too much) [11] the team simply accepts that the client requires the iPhone platform, rather than Android, Windows or a game console. In the first design meeting, the lead developer suggests making an adventure game. Due to the bandwagon effect (the tendency for team members irrationally prefer the status quo). Miserly information processing, the bandwagon effect, confirmation bias and status quo bias thus combine to create the inertia biasplex, which systematically reduces solution space exploration [10].

Practically speaking, then, the Theory of Cognitive Biases may be used to identify the causal mechanisms underlying problematic behavior in SE teams and identify simple, inexpensive techniques to debias the team and reduce bias-related anti-patterns. Prominent examples include requesting several alternative solution concepts to improve solution space exploration and using planning poker [12] to inhibit insufficient adjustment bias during effort estimation. A less known
technique involves assigning a Devil’s Advocate [13] to increase critical reflection in design or retrospective meetings.

B. Sensemaking-Coevolution-Implementation Theory

Moving from the individual to the process unit of analysis, Sensemaking-Coevolution-Implementation Theory (SCI) posits that software development comprises three primary activities (Figure 2): forming and organizing beliefs about the project context (sensemaking); the iterative, mutual refinement of one’s understandings of the context and the software artifact (coevolution); and the realization of one’s beliefs in the form of a software artifact (implementation) [14]. SCI assumes a single design agent, which may be an individual or a team, but not multiple, conflicting teams. The design agent may engage in the three major activities in any sequence. This leads to distinguishing two iterative loops: evolutionary prototyping, i.e., iteratively improving a software artifact over weeks or months; and coevolution, i.e., “developing and refining together both the formulation of a problem and ideas for a solution, with constant iteration of analysis, synthesis and evaluation processes between the ... problem space and solution space” [Dorst:2001tq, p. 434], over minutes or hours.

SCI is consistent with the assumption that system requirements are often an illusion [15] and the empirical model of design [16]. A survey of over 1300 developers supported SCI’s core propositions [17].

Fig. 2. SCI Overview (adapted from [14])

SCI has many important implications for SE research and practice. For example, the realization that problem framing and problem solving are interconnected undermines the logic of outsourcing development through tendering and fixed-price/-schedule contracts. Similarly, it suggests that dividing team members into “analysts”, “designers”, “programmers” and “ testers” may engender a misleading sense of identity that hinders effective working as these role-names reflect a poor classification of activities. Furthermore, as SE projects depend on the effectiveness of coevolutionary processes, SCI motivates research on tools, techniques and methods for enhancing coevolution. In addition, SCI suggests that SE education programs should cover sensemaking, coevolution and implementation as core topics; however, an analysis of SE2004 (the ACM SE undergraduate model curriculum) revealed that it includes neither coevolution nor any other method of generating design candidates [18].

C. Boundary Objects

SCI does not explicitly model artifacts produced during SE other than the software itself. Boundary objects are a class of artifacts, which are simultaneously plastic enough to be used for different purposes in different domains but solid enough to retain their identities [19]. For example, a UML class diagram may be used to guide programming and to elicit feedback from a client, but obviously remains a class diagram in both cases. The term “boundary object” comes from the use of such diagrams to facilitate information sharing across the multifarious boundary between an SE team and a client organization. Although the concept is broad, not all objects are boundary objects. For example, source code is not usually plastic enough to be considered a boundary object, while the employee-management power dynamic is too effervescent.

Analyzing SE artifacts and processes using boundary objects reveals many interesting patterns. For example, designers use boundary objects to promote shared representation, mobilize for action and transform and legitimize design knowledge [19]. In a recent (unpublished) study we found that SE project participants externalize their cognition into boundary objects to remember (e.g., to-do list), communicate (e.g., informal diagram on whiteboard during design meeting) and manage complexity (e.g., entity-relationship model showing tables but not attributes). This suggests a fourth core activity in SCI where design agents externalize their cognition about the project context into conceptual models and their cognition about the design space into design models (Figure 3).

While the Theory of Boundary Objects does not imply specific guidelines or practices for SE, it is useful for both researchers and practitioners as an analytical lens. For example, we found that some project participants feel pressured to externalize their understanding of the context or design space so that, when they are away, their teammates could still access their knowledge. However, their understanding is often so complex that any external representation detailed enough to use effectively is too voluminous to use efficiently. This boundary objects paradox is an example of a behavioral antipattern discussed above.

D. Transactive Memory Theory

SCI assumes that software is developed by a design agent, which may be a team if the team acts in coordinated fashion toward a common agenda. This raises questions regarding the
relationship between coordination, cognition and performance, which Transactive Memory Theory may address.

Shifting, then, to the group unit of analysis, Transactive Memory is a theory of group mind, which posits that individuals have both memory and meta-memory [20]. It further posits that that team members become specialists (Figure 4), maintaining detailed knowledge of their specialization (memory) and superficial links to the specialties of other team members (meta-memory). Team performance therefore depends on both task familiarity (memory) and team familiarity (meta-memory) [21], i.e., high performance necessitates that team members need to know not only how to do their part but also to whom to route tasks and requests outside of their specialty.

E. Complexity Theory

SCI seeks to explain how complex software systems are designed [14]. In Complexity Theory, “complexity” refers to the extent to which a system manifests behavioral properties not evident from its constituent parts [23], [24]. These “emergent” behaviors may arise from the (large) number of interconnections between components, from non-linear interactions between components or from poor understanding of the domain. Complex systems are also self-organizing, in that a high-level order spontaneously emerges from low-level interactions among initially unordered components. Complexity Theory has been used to explain phenomena as diverse as increasing returns (in economics) and how more complex organisms evolve from less complex ones [25]. Complexity Theory refers to the scientific study of complex systems. (Like game theory, it is a collection of concepts and an area of study rather than a set of hypotheses.)

Moving up to the project unit of analysis, SE projects and the organizations that execute them can be modeled as complex systems. More specifically, SE projects are complex adaptive systems [26], [27], i.e., complex systems that modify themselves based on feedback (Figure 5). Doing so entails several implications for SE practice including a controversial outlook on the relationship between project complexity and planning. Complexity theory posits that higher complexity implies lower predictability; therefore, planning is more difficult and less valuable in more complex projects. This further implies that more linear, plan-driven methods will be more effective for simple, routine SE projects while more iterative, reactive methods will be more effective for more complex, innovative SE projects. The idea that the utility of planning is inversely related to project complexity directly opposes much of the methods literature where proponents of both Agile and plan-driven methods agree that plan-driven methods are more appropriate for larger, more complex projects while Agile methods are more appropriate for smaller, less complex projects [12], [28]. In a recent (unpublished) study, we observed an SE team confronted with an initially simple project that appeared to gain complexity almost daily. Consistent with Complexity Theory, the team abandoned planning initiatives and adopted practices to become more responsive rather than proactive.

Fig. 4. Transactive Memory

Viewing a SE team as a transactive memory system implies that team performance fundamentally depends on individual skill and familiarity between team members. It suggests that interactions including peer programming, peer code reviews and daily stand-up meetings will have longer term positive performance impacts by increasing team familiarity [21], cf. [22]. Furthermore, in a recent (unpublished) study, we that transactive memory degradation helped explain the abandonment of an initiative to transition from an ad hoc methodology to Scrum. Practically speaking, Transactive Memory Theory implies that more direct attempts to share areas of expertise, rather than expertise itself, will increase team performance.

Fig. 5. Model of a Complex Adaptive System (from Wikimedia Commons)
depend on the complexity of the project. In addition, Transactive Memory Theory clarifies SCI’s design agent assumption; Complexity Theory clarifies SCI’s complex design object assumption; cognitive biases explain common problems encountered during coevolution (e.g., poor solution space exploration) and boundary objects suggest a fourth core SCI process (i.e., design agents externalize their cognition using boundary objects).

In conclusion, this paper describes five theories from SE and reference disciplines, which are useful for explaining and understanding SE behavior. These theories do not individually or jointly constitute a general theory of SE; rather, they are part of the theoretical foundation necessary to construct a more general theory.

Therefore, this paper’s key contribution is to advance the general theory of software engineering agenda by 1) reaffirming that SE theory is important; 2) acknowledging that a general theory may be developed by integrating existing relevant theories from SE and reference disciplines; and 3) discussing five relevant theories that may be new to many SE researchers. Future research is needed to identify more relevant theories and better integrate them to form a more general theory. This paper begins this process by noting that the reviewed theories are broadly consistent but apply at different units of analysis.

ACKNOWLEDGMENT

Thanks are due to Gerasimos Balis, Ammar Hamamra, Daniel Kershaw, Eduardo Narros, Petr Shportun and Ben Shreeve for their efforts on the three unpublished studies mentioned in this paper.

REFERENCES


