The sensemaking-coevolution-implementation theory of software design

Paul Ralph

PII: S0167-6423(14)00539-5
DOI: 10.1016/j.scico.2014.11.007
Reference: SCICO 1845

To appear in: *Science of Computer Programming*

Received date: 13 October 2013
Revised date: 18 February 2014
Accepted date: 25 June 2014

The Sensemaking-Coevolution-Implementation Theory of Software Design

Paul Ralph
University of Auckland
Auckland, New Zealand
paul@paulralph.name

Abstract

Following calls for greater theory development in software engineering, this paper formulates a process theory of software development practice. Sensemaking-Coevolution-Implementation Theory explains how complex software systems are created by cohesive software development teams in organizations. It posits that an independent agent (the development team) creates a software system by alternating between three categories of activities: making sense of an ambiguous context, mutually refining schemas of the context and design space, and manifesting their understanding of the design space in a technological artifact. This theory development paper defines, illustrates and conceptually evaluates Sensemaking-Coevolution-Implementation Theory. It grounds the theory's concepts and relationships in existing software engineering, information systems development and interdisciplinary design literature.

Keywords: Design, General Theory, Process Theory, Theory Development, Coevolution

1 INTRODUCTION

Cross explained that "we seem to have a fairly rich picture of design thinking, but we lack a successful, simplifying paradigm of design thinking … The lack of an adequate, simplifying paradigm is perhaps something which inhibits the transfer of knowledge from research into practice and education" [30]. To address this gap, this paper proposes a software design process theory, which attempts to incorporate many key findings from empirical research on design, especially software and information systems design. The proposed theory may contribute to a more general theory of software engineering.

Many academic disciplines emphasize general or unifying theories [80]. While Software Engineering (SE) has not historically done so, calls to develop a General Theory of Software Engineering (GTSE) are increasingly common (e.g. [72, 73, 122, 134]). A GTSE is needed to address the research-practice gap, to inoculate against fads and pseudoscience and to develop a cumulative body of knowledge [72, 73, 122]. However, developing a GTSE is extremely challenging due to the complexity and diversity of software development projects (e.g. real-time critical systems vs. multiplayer online games). Consequently, one way to proceed is to develop *core theories*, i.e., theories pertaining to discrete aspects of SE, and combine them to create a GTSE [117]. The software design process theory developed in this paper is intended as one such core theory. A theory of the design process is particularly needed because the design process has been historically conceptualized using methods, which oversimplify and over-rationalize reality [86, 115, 119, 148, 162].

Theories come in many different forms. Variance theories, which focus on the *causes* of a phenomenon [149], may posit one cause, many causes, a causal chain or a web of interconnected causes. Process theories, in contrast, focuses on *how* an entity changes and develops [149, 150]. Different theories have different purposes, including to explain, to predict, to analyze and to describe [63]. Theories also differ in their approach to causality [74], i.e., x causes y may mean 1) y is more likely given x (probabilistic); 2) y always follows x (regularity); 3) y would not occur without x (counterfactual); 4) y was instigated by an agent, x, with free will (teleological). Although process theories focus on how a phenomenon occurs, they often abstractly address *why* a phenomena occurs by adopting one of these four approaches to causation. For example, the Theory of Natural Selection explains why populations of organisms evolve using a probabilistic approach to causation; i.e., fitter animals have a high probability of reproducing.

Holistically understanding complex phenomena including software engineering may require diverse theoretical approaches, including both process and variance theories with different purposes and approaches to causality [122, 149]. This paper develops a teleological process theory to explain how teams design complex software systems. More specifically, the purpose of this paper is as follows.

> ***Purpose:*** *to formulate a process theory of software design practice, which explains how complex software systems are created by cohesive software development teams in organizations.*

Here, a software design process theory is an explanation of how and why a software system changes and develops. The term *practice* is used to denote concern with how software is developed in reality, including both effective and ineffective practices, rather than positing an idealized process or prescribing good ways of designing. Meanwhile, *design* refers to an agent specifying or building an object, which is intended to accomplish goals in a particular environment using a set of primitive components and subject to constraints [121]; "here, design encompasses all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems—not just the activity following requirements specification and before programming, as it might be translated from a stylized software engineering process" [51]. *Design* is used instead of *development* because thinking of design as a phase of rather than a synonym for development is one of the misconceptions the proposed theory is intended to address. Finally, the level of analysis varies from individual to team.

This paper proceeds as follows. The review of related work is divided into research needed to understand the proposed theory (§2) and situating it with respect to existing research (§5). The proposed theory is described in Section 3 and evaluated in Section 4. Section Six summarizes the paper's contributions and implications. Due to the novelty and complexity of the proposed theory, this paper focuses on providing a comprehensive presentation of the theory while empirical evaluation is discussed elsewhere.

2 THEORETICAL PRELIMINARIES

2.1 Review Structure and Sampling

As the design literature is spread across many fields and media, a search-based review, rather than a comprehensive analysis of a particular set of publications, was used. The review focused on three overlapping fields – information systems (e.g. MIS Quarterly), software engineering (e.g. the International Conference on Software Engineering) and the interdisciplinary design literature (e.g. Design Studies). As the review spread through bidirectional citation search, examining both what a work references and what references it, some particularly influential work from architecture, engineering and sociology was drawn in. To get to the proposed theory more quickly, this section focuses on the three frameworks needed to understand its concepts and contribution. The remainder of the literature review is presented in Section 5.

This review focuses on process theories. A *process theory* is an explanation of how and why an entity changes and develops [149]. A *design process theory* is an an explanation of how and why a *design artifact* changes and develops. A description qualifies as a design process theory if it meets the following three criteria.

1. It posits a formative relationship between a higher level phenomenon (design) and several lower-level phenomena (e.g. coding, modeling).
2. It includes a causal motor (dialectic, evolutionary, lifecycle, teleological) (Table 1).
3. It includes a claim to universality within a domain.

Table 1. Ideal Types of Process Theories (adapted from [113, 150])

Ideal Type	Proponents	Brief Description	Event Progression	Modern Example
Dialectic	Plato, Hegel, [150]	Changes result from shifts in power among conflicting entities	Recurrent, discontinuous sequence of conflict and resolution	Behavioral Negotiation Theory [97]
Evolutionary	Darwin, [150]	A population of entities changes as less fit entities expire and remaining entities change and recombine	Recurrent, cumulative and probabilistic sequence of variation, selection and retention	Change in populations of organizations [24]
Lifecycle	[92, 150]	An entity progresses through a series of stages in a predefined sequence	Linear & irreversible sequence of prescribed stages	The Organizational Lifecycle [75]
Teleological	[26, 138, 150]	An agent purposefully selects and takes actions to achieve a goal	Recurrent, agent-determined sequence of goal setting and action taking	Organizational decision making [91]

This paper discusses research contributions that fit the above criteria regardless of whether they are explicitly called process theories by proponents. This section discusses the Problem-Design Exploration Model (evolutionary), the Selfconscious Process (teleological), the Waterfall Model (because it is treated as a lifecycle theory even though it is a method) and the need for new theory. Section Five discusses the Basic Design Cycle (lifecycle), the Function-Behavior-Structure Framework (teleological) and Boomerang (teleological).

Process theories fundamentally differ from methods/methodologies. A *method* is a collection of prescriptions for developing systems or managing development projects. Types of prescriptions include practices, techniques, tools and phase models. Information systems and software engineering literature is predominately concerned with methods for developing systems and managing the development process [148] and methods literature is predominately normative or prescriptive [161]. Methods in general are inappropriate foundations for design theories [153] as a method purports to *prescribe an effective way* of creating software while a software design process theory purports to *explain* software creation, *both effective and ineffective* (cf. [63]). As this paper fundamentally concerns process theory, methods are not reviewed. One exception is the Waterfall Model as it is often treated as a process theory.

Process theories also fundamentally differ from process models. “A process model is an abstract description of an actual or proposed process that represents selected process elements that are considered important to the purpose of the model” [35]. A process model describes while a process theory explains. Moreover, process models (like methods) seek to describe one or a few sequences while process theories seek to encompass all the ways an entity changes. Process models are therefore not reviewed.

2.2 The Waterfall Model

The Waterfall Model (Waterfall) refers to a collection of methods that divide systems development into a series of phases including *analysis*, *design*, *coding* and *testing* or synonyms thereof. Different versions prescribe different numbers of phases and different degrees of interaction between them. Figure 1 shows three versions: 1) solid arrows show the forward-only version; 2) solid and dashed lines show a version allowing backtracking; 3) solid, dashed and dotted lines indicate a *clique* version, where transition between any two activities is allowed. The original, forward-only version from which the method gets its name was proposed by Royce [127] as a straw man, which he dismissed as too risky, in favor of a more iterative method.

Although, the forward-only version is now considered idealistic, many academics and practitioners continue to present it as fact. e.g. “in conventional software development, the development lifecycle in its most generic form comprises four broad phases: planning, analysis, design, and implementation” [47]; “regardless of the particular process model an organization may use … every software project will feature: (1) the requirements-definition and functional-specification phase; (2) the design phase; … (3) the implementation or the coding and testing phase; … and (4) the installation, operation, and maintenance phase” [45]. A popular introductory textbook states that “systems development … consist[s] of systems analysis, systems design, programming, testing, conversion and production and maintenance … which usually take place in sequential order” [82]. These phases are also explicitly adopted in the official IEEE Guide to the Software Engineering Body of Knowledge [19]. Finally, as of 5 July 2014, the Systems Development Lifecycle Wikipedia article states that it “adheres to important phases that are *essential* for developers, such as planning, analysis, design, and implementation” (italics added). Waterfall is a *method*; however, when its phases are presented as essential or generic, Waterfall is being *treated* as a theory.

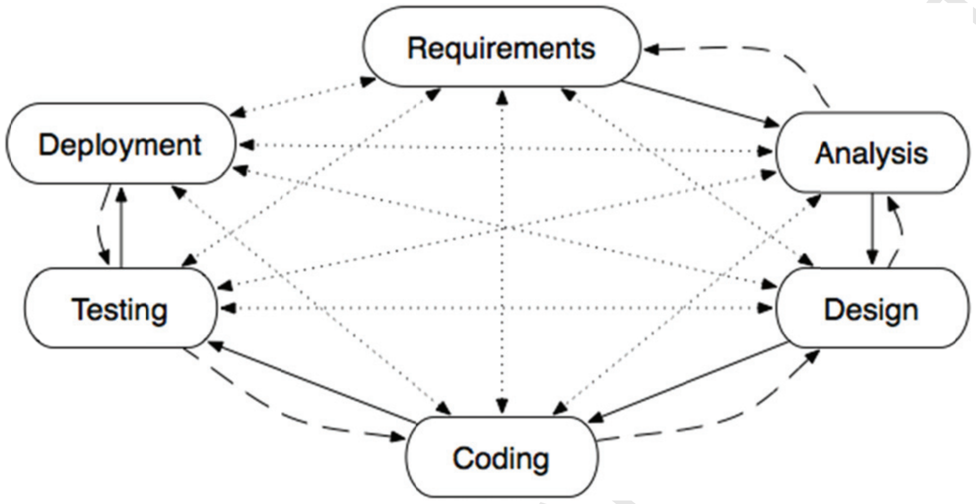

Figure 1. Generalized Waterfall Model (adapted from [127])

Notes: Solid arrows show Royce’s forward-only model; solid and dashed arrows show Royce’s bidirectional model; solid, dashed and dotted arrows show generalized model allowing any transition.

Waterfall, especially the forward-only version, is attractive in several ways. It is easy to understand and teach. The structure is consistent with basic methods of problem solving [109] – understand the problem, devise a plan to solve it, then execute the plan. It facilitates fixed-price/-schedule contracts [20]. It provides clear milestones and sense of progression. It allows managers to drive development through schedule and budget, and to divide work between specialists (e.g. analysts, designers, testers) as seen in the Unified Process [71].

However, Waterfall has been criticized in two specific ways – one widely known and one less known. First, Waterfall *oversimplifies* development by positing that all projects share the same (idealistically linear) sequence of phases [20, 61, 94]. Second, Waterfall *over-rationalizes* development by positing that analysis, design, coding and testing are cohesive, mutually exclusive activities. For example, testing is not cohesive because it includes heterogenous activities (e.g. debugging and acceptance testing). Meanwhile, some forms of testing (e.g. unit testing) violate mutual exclusivity by involving some analysis, design or coding (cf. [16]).

In summary, *Waterfall* refers to a collection of similar methods, but is often treated as a lifecycle process theory. It remains influential, partially due to its simplicity and compatibility with management methods. However, it over-rationalizes the design process through false dichotomies between analysis, design, coding and testing.

2.3 The Problem-Design Exploration Model

The Problem-Design Exploration Model [90] embodies a substantially different standpoint, modeling design as two interacting evolutionary systems – problem space *P* and solution space *S*. Distinguishing between problem and solution spaces is evident in analytical (e.g. [41]), empirical (e.g. [59]) and prescriptive (e.g. [25]) design research. Purao et al. described "the problem space … as the metaphoric space that contains mental representations of the developer's interpretation of the user requirements" and "the design space" as " the metaphoric space that contains mental representations of the developer's specific solutions" [111]. From this distinction, Maher et al. [90] suggest a formal model of exploration intended to describe how an artifact (not necessarily software) may be designed by genetic algorithms (Figure 2).

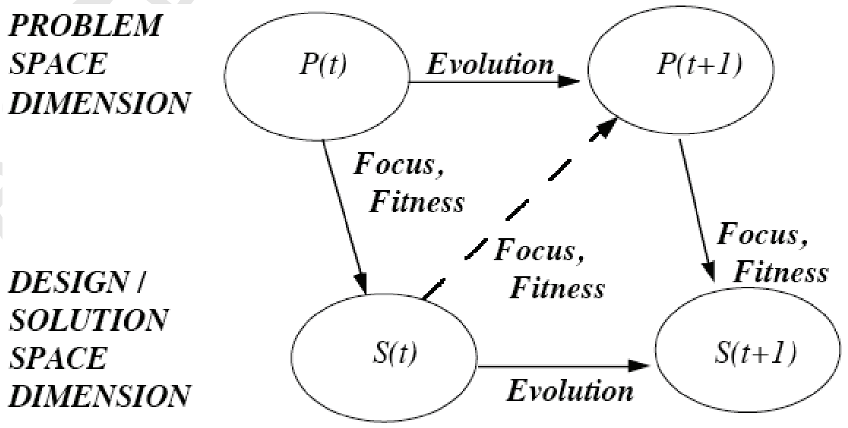

Figure 2. Problem-Design Exploration Model[1] (adapted from [90])

[1]*P(t) = problem at time t; S(t) = situation at a time t; dashed line indicates situation refocusing problem; diagonal downward movement indicates a search process.*

The Problem-Design Exploration Model is clearly a process theory – it relates design to lower-level activities (evolution, focus, fitness); it has an evolutionary causal motor; Maher et al. position it as a universal explanation within the design-by-genetic-algorithms domain. Moreover, it has several advantages including its overall simplicity and clear elucidation of the Coevolution concept. While problem-solution Coevolution has been observed in field studies of designers (cf. [30, 132]), the Problem-Design Exploration Model remains primarily a theory of applying genetic algorithms to design problems, rather than a theory of human behavior. Specifically, for genetic algorithms, the problem and solution spaces must be represented as a finite number of known dimensions, where for human designers, problem and solution spaces are merely metaphors and often lack such precise definition.

2.4 Alexander's Design Process Models

Differentiating *form* (the design artifact) from *context* (its environment), Alexander [2] proposed three "possible kinds of design process" (Figure 3). In the "Unselfconscious Process", the designer directly manipulates the design object to eliminate misfits between form and context; e.g., an igloo dweller creates vents when the temperature rises and eliminating them when the temperature falls. In the "Selfconscious Process", the designer works by iterating between "the conceptual picture of the context … and ideas and diagrams and drawings which stand for forms" (p. 75); e.g., a fashion designer sketches handbags from different angles, changing features based on ideas about aesthetics. In the third (unnamed) process, the designer creates a formal model of the mental pictures using set theory and solves problems using a divide and conquer strategy.

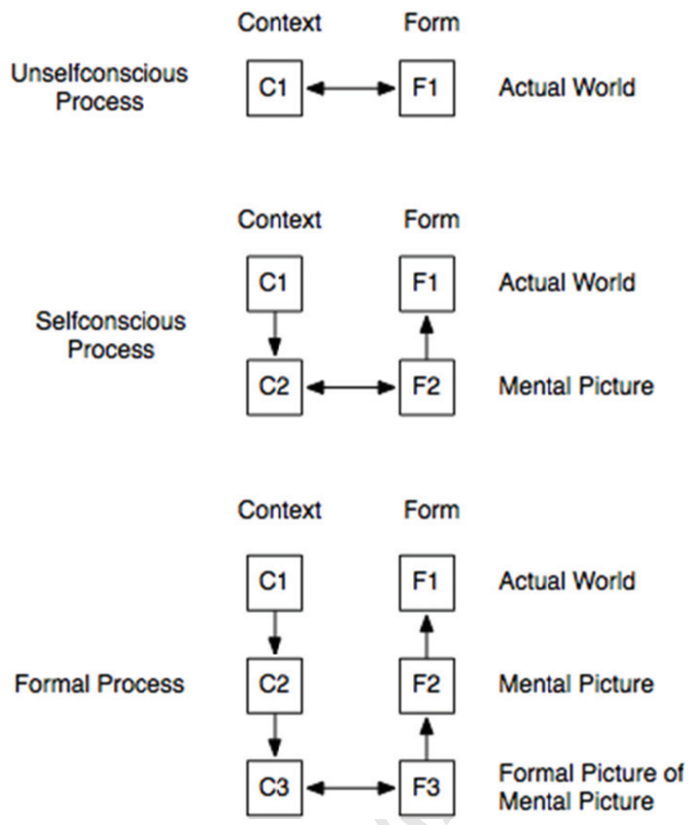

Figure 3. The Selfconscious Design Process[1] (adapted from [2])

[1]*arrows indicate interactions, not sequence; Alexander did not explicitly define these interactions*

These three models are not all process theories. The first model has only one interaction and therefore does not explain design in terms of lower level processes. As Alexander explicitly *prescribed* the third model to overcome the limitations of the Selfconscious Process, it is a method.

However, the Selfconscious Process does explain design in terms of three lower-level interactions, it has a (teleological) causal motor and Alexander presented it as a model of the design process generally used by architects and other designers. Moreover, it appears consistent with software design – from left to right, the first interaction is analogous to problem analysis, the second to Coevolution (as in the Problem-Design Exploration Model), the third to construction or implementation.

Benefits of the Selfconscious Process include its parsimony and inclusion of problem framing. However, its concepts and relationships need further clarification – Alexander admitted that the relationship between the mental pictures is not well understood. Furthermore, it was intended to explain architectural design rather than software design.

2.5 Summary and the Need for New Theory

None of the three frameworks discussed in this section is sufficient as a design process theory: Waterfall over-rationalizes the design process; the Problem-Design Exploration Model was never intended to explain software engineering in general; the Selfconscious Process lacks clear definitions of concepts and relationships. However, the Problem-Design Exploration Model informs design process theory by highlighting the centrality of Coevolution while the Selfconscious Process provides a basic three-activity structure.

3 SENSEMAKING-COEVOLUTION-IMPLEMENTATION THEORY

3.1 A Motivating Illustration

Suppose James is a software engineering researcher. He poses the research question, "what is the ratio of effort expended on analysis, design, coding and testing in co-located web development teams?" James begins observing developers at web development agency, Marshmallow, and analyzing their activities to determine the effort ratio.

James quickly encounters several difficulties. First, during planning meetings, the team members appear to rapidly oscillate between ideas about the problematic context and ideas about the system they are developing. New realizations about the context trigger reconceptualizing the system, but then the reconceptualized system triggers reframing of the context in an iterative fashion. This single activity appears to be both analysis and design.

Second, the programmers appear to make numerous design decisions independently and seamlessly while coding. The do not appear to iterate between designing and coding; rather, many design decisions seem to be part of the coding activity. The graphic designer, meanwhile seems to design, implement and test all at once - making small changes and immediately checking their effects. For her, designing and testing are simultaneous.

Third, Marshmallow extensively uses unit testing, which involves not only analyzing the system and context but also designing and coding the tests. James is not sure how to classify these activities - is coding the test suite ‘testing’ or ‘coding’? Furthermore, Marshmallow’s product owner uses acceptance testing (getting feedback from clients on prototypes) as an analytical tool.

James cannot assess the relative effort involved in analysis, design, coding and testing because his observations do not fit into these categories. He worries that these terms may be causing systematic misperceptions of developers’ actual activities. James needs different categories – a different vocabulary – to answer his research question.

3.2 SCI Overview

Sensemaking-Coevolution-Implementation Theory (SCI) was developed following the realization that *analysis*, *design*, *coding* and *testing*, do not describe highly cohesive, loosely-coupled activities (cf. [143]). Its purpose is to provide a better set of concepts to explain software development. SCI (Figure 4) makes two core claims: 1) a complex software system is developed by an independent, goal-oriented agent; 2) this agent performs the three high-level activities for which SCI is named. The design agent may engage in any of the three activities at any time, for any period, serially or in parallel. The concepts and relationships used in SCI are all rooted in existing literature (Table 2) – its contribution lies in their novel organization.

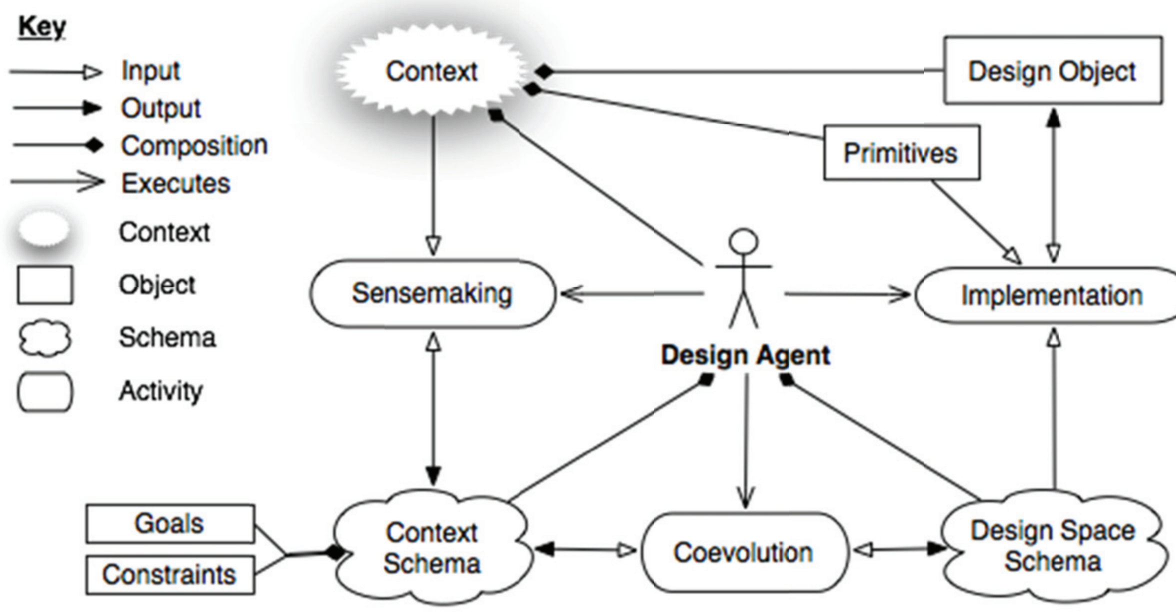

Figure 4. Sensemaking-Coevolution-Implementation Theory

Note: Arrows indicate relationships as shown, not a sequence of activities

3.3 The Design Agent

The Design Agent is the individual or group that specifies and creates the system. The Design Agent is centrally placed to emphasize SCI's *teleological* approach to causality [63, 74]. (In the remainder of this paper, "designer" and "design agent" are used interchangeably.) A teleological process theory is an explanation of how and why an entity changes where change is manifested by a goal-seeking agent engaging in activities in a self-determined sequence [150]. Here, the changing entity is the design object (software system). 'Why' it changes is explained by adopting the teleological approach to causality [74] – it changes because a person with free will chooses to change it. This is consistent with characterizations of design by Churchman – "design belongs to the category of behavior called teleological, i.e., 'goal seeking' behavior" [26] and Löwgren et al. – "Our basic assumption about the design process is that its form, structure, and qualities are not given or ruled by laws of nature ... [but] by designers' own thoughts, considerations, and actions" [87]. 'How' it changes is explained by the three SCI processes. Here, 'explain' is used in the reductionist sense – the higher-level activity (design) is explained by reducing it to a set of interconnected lower-level activities (Sensemaking, Coevolution, Implementation).

Table 2. Concepts and Relationships of SCI, Defined

Concept	Meaning	Key References
Coevolution	the process where the design agent simultaneously refines its design space and context schemas	[2, 30, 41, 90, 132]
Constraints	the designers beliefs about the restrictions on the design object's properties	[18, 19, 88, 121, 137, 140]
Context	the totality of the surroundings of the design object and agent, including the object's intended domain of deployment	[2, 25, 101, 103, 121, 132]
Context Schema	the collection of all of the design agent's beliefs about the context; the design agent's mental picture of the context.	[2, 90, 111]
Design Agent	an entity or group of entities capable of forming intentions and goals and taking actions to achieve those goals and that specifies the structural properties of the design object	[2, 30, 42, 121]
Design Object	the thing being designed; also any partial or complete manifestation thereof	[2, 42, 121]
Design Space Schema	the collection of all of the design agent's beliefs about the diversity of possible design artifacts, including design decisions and candidate solutions; the design agent's mental picture of the design object and its alternatives	[2, 90, 111]
Goals	optative statements about the effects the design object should have on its environment	[26, 37, 121]
Implementation	the process where the design agent generates or updates the design object based on its design space schema	[2, 18, 19, 127]
Primitives	the set of entities from which the design object may be composed	[95, 106, 121, 125]
Sensemaking	the process where the design agent organizes and assigns meaning to its perception of the context, creating and refining context schema	[40, 52, 128, 132, 158, 159]

SCI's agency assumption has several implications. First, some systems (e.g. Linux) are obviously developed by multiple agents, who may be uncoordinated, have different goals or even sabotage each other. SCI does not posit that all software is developed by a single agent; rather, it only applies to single-agent development. Second, while an individual can always be modeled as a single agent, groups are more complicated. A group that has high team cohesion [13], consistent goals, shared mental models and good coordination [43] may be considered a single agent. Groups with low cohesion, poor coordination, divergent mental models and conflicting goals should not be considered a single agent. In practice, some teams may fall between these extremes. Third, numerous theoretical perspectives may be applied to the design agent. For example, the design agent may be modeled as a transactive memory system [85, 157], rational agent (as in game theory) or political actor [83].

3.3 The Context

The context of a software engineering project may become prodigiously complicated. It includes the design agent’s environment, i.e., not only the work environment but also the designer’s technological, social, economic and political environment. It also includes the development environment, i.e., the physical or virtual space in which prototypes and non-deployed software artifacts reside. Furthermore, it includes the intended deployment environment – again, not only technologically but also socially, politically and economically. The context may be analyzed using numerous theoretical approaches including Structuration Theory [60], Actor Network Theory [83] and Work Systems Theory [3].

SCI posits that the context is a complex system, in the technical sense of complexity science, where, a system is complex to the extent that it exhibits unpredictable (emergent) behavior [54, 70]. Here, a complex context is expected to be unstable, i.e., change both during and in response to development. Complexity means that the effects of development are not accurately predictable in advance. The context is, moreover, expected to contain stakeholders who conflict and disagree, and goals and constraints that are unknown, ambiguous, conflicting or unstable.

3.4 The Design Object

The design object is the system being developed, including partial representations thereof (e.g. prototypes). SCI posits that, in software engineering, the primary iterative artifact [17] is the software system itself. This separates software engineering from mechanical engineering, architecture, product design and industrial design where the primary iterative artifact is a design model (e.g. blueprint, sketch, technical drawing) [104].

SCI also posits that the design object is *complex*, again in the technical sense of complexity science., i.e., the design object will exhibit performance and effects not evident from its constituent parts (emergent behavior). Complex systems are not necessarily large or composed of many components and complexity is a spectrum rather than a binary property of a system. Like the agency assumption, SCI does not posit that **all** software systems are complex; rather, it only applies to systems that are.

Furthermore, rather than being created from nothing, SCI posits that the design object is constructed from a set of *primitives* [121]. Primitives may include not only existing components (as in component-based development [64]) but also programming languages, libraries, design patterns, data structures and services (as in Service Oriented Architecture [105]).

3.5 The Schemas

A *schema* is "a plan, diagram, or outline, especially a mental representation of some aspect of experience, based on prior experience and memory, structured in such a way as to facilitate (and sometimes to distort) perception, cognition, the drawing of inferences, or the interpretation of new information in terms of existing knowledge" [28]. SCI posits that the design agent forms two relevant schemas – the context schema includes all the designer's beliefs about the context while the design space schema includes all the designer's beliefs about the design object and its potential alternatives.

Both schemas may contain numerous and diverse constructs. SCI posits two universal components of the context schema. First, as teleological process theories posit a goal-oriented agent [150], SCI explicitly shows *goals* as part of the designer's context schema. Goals are also a central topic in requirements engineering research [37, 151]. Second, as the concept of *constraints* (restrictions on the design object) is central in both prescriptive and theoretical literature on software and engineering design [18, 19, 88, 121, 137, 140], the context schema explicitly includes constraints. The concepts of *Requirements* is intentionally omitted as many projects appear to lack meaningful requirements [118].

Schema may refer to a framework held wholly in the mind, wholly on paper (or another medium) or a combination thereof. In accordance with the cognitive science theory of the extended mind [27], the design agent may externalize all or part of a mental framework into a physical or virtual medium. The resulting conceptual models (context schema) or design models (design space schema) may aid memory, cognitive processing and communication.

In practice, some models (e.g. UML use case narratives) may combine elements of the two schemas. More fundamentally, the design agent itself may confuse the two schemas, possibly by incorrectly categorizing design decisions as part of the problem. Furthermore, where the design agent is a team, team members' schemas may differ. Large intrateam schema differences may violate SCI's single-agent assumption.

3.6 Sensemaking

"To convert a problematic situation to a problem, a practitioner must … make sense of an uncertain situation that initially makes no sense" [132]. This quotation suggests the construct *Sensemaking,* which is an important research topic in management [158, 159], human-computer interaction [52, 128] and knowledge management [40]. Sensemaking refers to "what people do to make sense of the information in their world" [52] and the way in which a person comes to understand, or give meaning to, a new or confusing phenomenon. While no formal definition of Sensemaking is widely accepted, it involves perceiving and encoding information to facilitate action [128] and noticing, bracketing, labeling, organizing and communicating ideas [159]. For the purposes of SCI, Sensemaking generates, organizes and refines the context schema.

Sensemaking is broader than requirements elicitation as software projects are affected by their entire social context [101]. Moreover, when a design agent engages in Sensemaking it may consider myriad aspects of the project context, including aspects of the design agent's work environment and the design object's intended deployment domain. This reflects the reality that internal factors including project budgets, interpersonal relationships and managerial pressure may affect design decisions.

Sensemaking includes the kinds of analysis that are focused on the context (e.g. interviewing stakeholders), but not the kinds of analysis that are focused on the design object (e.g. analyzing alternative design concepts, analyzing code using a debugger). Sensemaking also includes the forms of testing that focus on understanding the relationship between the design object and the context (e.g. acceptance testing), but not the forms of testing that focus on code and performance (e.g. debugging).

3.7 Coevolution

The relationship between the two schemas is labeled *Coevolution* (following [90]). Coevolution refers to the mutual, iterative refinement of the context and design space schemas. “In designing, ‘the solution’ does not arise directly from ‘the problem’; the designer's attention oscillates, or commutes, between the two, and an understanding of both gradually develops” [30]. During an observational study, Dorst and Cross witnessed a good example of coevolutionary behavior:

> *A seed of coherent information was formed in the assignment information, and helped to crystallize a core solution idea. This core solution idea changed the designer’s view of the problem. We then observed designers redefining the problem, and checking whether this fits in with earlier solution-ideas. Then they modified the fledgling-solution.* [41]

Many consider the idea of a designer mutually and iteratively refining schemas of the context(/problem/environment/problem space) and the design space(/design object/artifact/solution space/form) central to design [2, 17, 41, 111, 132, 152]. Important aspects of Coevolution include: 1) Schön’s interconnected trio of problem framing, tentative adjustments to a design concept, and conceptually evaluating consequences [132]; 2) organizing the system into components or subsystems [2, 107] and 3) creativity [41, 156].

In practice, many objects may coevolve (e.g. symbiotic organisms, hardware and software, source code and test suite). Here, *Coevolution* specifically refers to the designer’s rapid oscillating between the context and design space schemas. The defining feature of Coevolution is that new ideas about the context trigger reconceptualization of the design space *and vice versa.* Coevolution is the process by which designers formulate design concepts and make many design decisions.

3.8 Implementation

Implementation refers to creating and deploying the software system. Implementation involves not only programming but also all the activities tightly coupled with programming. These may include creating ancillary artifacts (e.g. installers, documentation) and most white-box testing (e.g. debugging, unit testing). Low-level design decisions made routinely during coding are part of Implementation.

3.9 Two Iterative Loops

While it is widely accepted that software development is often iterative [81], SCI distinguishes between two concentric iterative loops. The inner, Coevolutionary loop involves oscillation between ideas – the design agent iteratively refines its two schemas usually over minutes or hours. The outer, prototyping loop involves making sense of the context, Coevolution and creating or modifying software artifacts. The new(ly modified) artifacts alter the context [101], triggering additional Sensemaking, which alters the context schema, triggering further Coevolution, and so on. Distinguishing these two types of iteration – Coevolution and prototyping – is a core contribution of SCI.

3.10 Development of SCI

SCI was created by elaborating Alexander's Selfconscious Process [2] with concepts from software engineering (e.g. constraints), design (e.g. Coevolution), management (e.g. Sensemaking) and psychology (e.g. schemas) – see Section 4.3. Brief descriptions of SCI have appeared in several papers [112, 116, 117] and a limited empirical trial appeared in a conference paper [112]. A more comprehensive empirical trial has been conducted but not published [117]. Previous papers have not explained SCI's relationship to existing literature, or elaborated on its interpretation and implications. This paper presents SCI in much greater detail than previous work.

An earlier version of this paper comprised part of the author's dissertation [113]. The theory has since been improved in four main ways. First, the original theory distinguished between the *design agent's environment* and the *design object's environment* (Figure 5). Recognizing that these environments may overlap (e.g. if the system under construction includes hardware, the hardware may reside in the same room as the developer), the two environments were combined into a single *context*. Second, the original theory used the term *Mental Picture* following Alexander [2]. This has been replaced by the more precise term, schema (§3.5). Third, the *requirements* concept was removed from the context schema, following the realization that some projects have no meaningful requirements [118]. Fourth, the symbols have been modified to increase similarity to the Unified Modeling Language.

3.11 SCI as a General Theory of Software Engineering

In its current form, SCI is not a GTSE. First, SCI explains how an individual or cohesive group develops software. Non-cohesive groups (e.g. linux developers) exhibit inter-agent dynamics (e.g. goal conflict), which SCI does not attempt to model. Second, SCI explains development under complexity. When the context is complex, goals are ambiguous, vary by stakeholder or are simply unknown [25]. The designer therefore explores the context by designing [29], necessitating the inner iterative loop (Coevolution). When the design object is complex, the only way to determine its behavior is to build a prototype, necessitating the outer iterative loop. When both the context and design object are simple, no iteration is necessary and SCI does not apply. For example, the traveling salesman problem is a known, unambiguous, agreed problem. It does not evolve when we design algorithms to solve it. Moreover, we can logically establish whether our algorithm has lower time complexity than existing approaches without implementing it. SCI therefore does not explain this type of predictable, well-defined development.

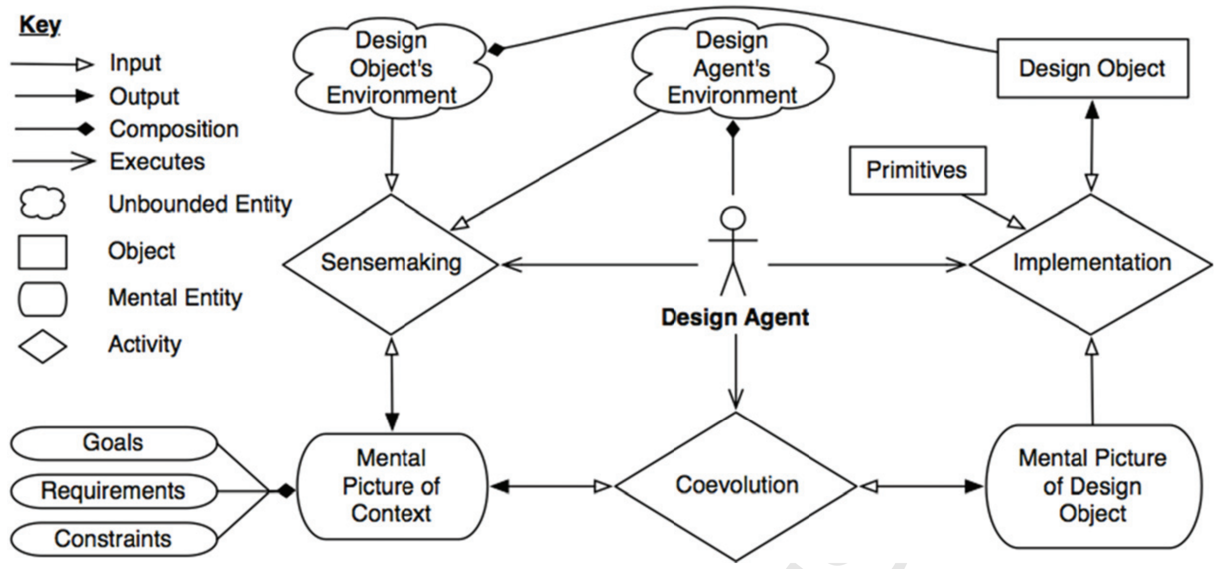

Figure 5. Earlier version of Sensemaking-Coevolution-Implementation Theory (from [113])

However, one route to developing a GTSE is to extend SCI to multi-agent development and simple systems. Briefly, SCI could be combined with a variant of FBS to model simple systems. Then, both could be combined with a quasi-independent theory of the design agent.

Two related concerns are the diversity within software engineering and the possibility that SCI applies to other fields. First, many software engineering subfields have concepts peculiar to their circumstances; e.g., play-testing in digital game development [141]. SCI does not attempt to model such subfield-specific concepts. Moreover, it seems obvious that attempting to capture all subfield-specific concepts would create an overly large, unwieldy theory. Second, while SCI was created to understand development of software and information systems, it could apply to other domains. However, this is not unusual. For example, the Theory of Natural Selection happens to explain numerous economic phenomena, but this neither makes it an economic theory nor undermines its biological merits. Similarly, if SCI happens to explain some industrial design phenomena, for example, it does not cease being a theory of software engineering. Moreover, SCI is not a general theory of design because, for many physical products, the outer iterative loop is too slow to be meaningful to the design agent.

3.12 Summary

In summary, SCI explains software development in terms of a design agent who engages in three lower level activities – Sensemaking, Coevolution and Implementation – in a self-directed order. SCI only applies to design of complex systems, in complex contexts, by a single agent (an individual or a cohesive team). It is not a GTSE; however, it may provide a basis for developing one. In the interests of parsimony, SCI focuses on the concepts believed to be instrumental to software development, omitting a plethora of activities, concepts and artifacts that are either optional, or peculiar to specific subfields. SCI differs from existing theories in its teleological approach to causality, dual iterative loops and emphasis on complexity.

SCI is intended to replace lifecycle depictions of the development process. These depictions have two problems – 1) design is depicted as a linear sequence of phases; 2) the terms *analysis*, *design*, *coding and testing* confuse academics and practitioners because they do not describe cohesive, mutually exclusive categories of activities. While Agile methods have illuminated the linear-sequence problem [15], SCI is novel in illuminating the bad categories problem.

4 CONCEPTUAL EVALUATION

SCI was motivated by Cross's call for a simplifying paradigm for design and Brooks' call for a communicable model of design (Section 1); therefore, SCI should be evaluated for simplicity and communicability. Furthermore, SCI can be evaluated using general criteria for theories including novelty, usefulness and testability. This section evaluates SCI's on these five criteria.

4.1 Simplicity

Cross elucidated the need for a "simplifying paradigm" for understanding and communicating design [30]. Sections 2 and 5 review several design process theories, including the Selfconscious Process and the Situated FBS Framework. The Selfconscious Process appears perhaps *too* simple, i.e., so simple that it is difficult to draw useful propositions from it. In contrast, Situated FBS with its three worlds, twelve artifacts and twenty processes is perhaps so complex that it is difficult to understand and apply. SCI is somewhere between. Favoring simplicity over specificity in SCI's development was intentional as successful theories are often extended over time.

4.2 Communicability

Education drives the need for a communicable model of software design [20]. SCI's communicability is enhanced by clear definitions of each concept and relationship. The discussion of SCI's core concepts in Section 3 further enhances its communicability.

However, some students and academics have mistaken SCI for a method, as evidenced by questions including 'why would developers choose SCI over Waterfall?'. Of course, developers do not choose SCI when programming anymore than surgeons choose germ theory when washing. SCI is a theory, not a method. Empirical studies investigating the understandability of SCI to different groups (e.g. students, programmers, managers) are a potentially fruitful area of future work.

4.3 Novelty

SCI's novelty lies in its synthesis of existing literature into an organized process theory. The Selfconscious Process, on which SCI's structure is based, exhibits one problematic limitation – its concepts and relationships are not clearly defined. For example, Alexander explained that the interaction between the designer's conceptualizations of form and context "contains both the probing in which the designer searches the problem for its major 'issues', and the development of forms which satisfy them; but its exact nature is unclear" [2]. To clarify and elaborate the Selfconscious Process, SCI:

- gives the three relationships names that link them to existing research
- adds goals, constraints and primitives
- explicitly shows the design agent
- renames "mental pictures" to "schemas" to increase precision

Moreover, this paper explains the relationships both within SCI's concepts and between SCI and existing research to a greater degree than previous work.

4.4 Usefulness

SCI is a theory intended for use by researchers and educators, not a method intended for use by practitioners. SCI is useful for research and education in several ways.

For researchers, SCI is useful for organizing, categorizing and analyzing empirical data from field studies (e.g. ethnography, action research, case study) of software development teams and projects. Researchers can use SCI to conceptualize the diverse technologies and practices they encounter. For example, conceptual modeling languages are predominately Sensemaking tools, while integrated development environments are predominately Implementation tools. Facilitating empirical software engineering research is the primary intended use of SCI.

The other theories discussed in this paper do not support qualitative analysis as well as SCI does. Waterfall (§2.2), the Basic Design Cycle (§5.1) and FBS (§5.3) over-rationalize development, leading to the problems described in Section 3.1. The Problem Design Exploration Model (§2.3) and Boomerang (§5.2) model unique aspects of particular subfields and therefore cannot be effectively applied to software development generally. Finally, SCI improves on Alexander's process models (§2.4), which are too general and ill-defined to apply directly (§4.3).

SCI is also useful for generating and refining research questions. For example, a researcher beginning with a solid understanding of SE methods would struggle to formulate a question like "How do conflicting goals affect Coevolution in mobile application development?" because methods do not emphasize goal conflict and Coevolution.

For educators, SCI provides an alternative to Waterfall as a way to introduce software development fundamentals. Educators can use SCI to conceptually organize software engineering tools and practices. For example, interviewing stakeholders, conceptual modeling and planning poker are all Sensemaking practices, while test-driven development, refactoring and unit testing are all

Implementation practices. Similarly, SCI usefully illuminates the dearth of tools and practices for facilitating Coevolution. SCI can also be used to evaluate courses and curricula. For instance, using SCI to analyze the ACM/IEEE model curriculum for SE undergraduate programs revealed that it does not cover how to generate design concepts [114]. Furthermore, SCI is useful for explaining why fixed-price/-schedule contracts increase overall project risk – since the context schema is expected to change significantly during the project, estimates based on the original context schema are unlikely to be accurate.

4.5 Testability

SCI should produce testable propositions. However, process theory propositions may be less clear than variance theory propositions as different types of process theories make different types of truth claims.

For example, in Lifecycle theories (e.g. Waterfall if it were a theory), "the trajectory to the final end state is prefigured and ... [each stage] must occur in a prescribed order, because each piece sets the stage for the next. Each stage of development is seen as a necessary precursor of succeeding stages" [150]. A lifecycle theory of process P therefore claims: 1) that P is divisible into a specific set of phases; 2) that later phases are impossible without earlier phases. "Unlike life-cycle theory, teleology does not prescribe a necessary sequence of events or specify which trajectory development ... will follow" [150]. Teleological theories including SCI, posit that independent agents choose their own activity sequence. Therefore, a teleological theory of process P claims 1) that P is divisible into a specific set of activity categories; 2) that an independent agent chooses transitions between these activities at will to achieve goals.

As lifecycle and teleological theories both describe activity categories or phases, both may be incorrect in four ways (analogous to the four types of modeling grammar deficiencies [155]) – phases or categories may be overloaded, redundant, extraneous or missing. In addition, a teleological theory may be deficient if it posits a non-existent agent and a lifecycle theory may be deficient if it posits an incorrect sequence (Table 3).

Table 3. Possible Deficiencies in Teleological and Lifecycle Process Theories[1]

Deficiency	Definition	Hypothetical Waterfall Deficiency
Overload	Dissimilar activities map into one process or phase	Debugging and user observation both map into testing despite involving different people, places, times, goals and actions
Redundancy	One activity maps into several processes or phases	Acceptance testing maps into both analysis and testing
Excess	No activities map into a process or phase	No requirements gathering or analysis is observed
Deficit	An activity does not map to any process or phase	The team undertakes domain specific training
No Agent[2]	No actors map into the hypothesized independent, goal-seeking agent	n/a
Incorrect Sequence[3]	The activity sequence does not conform to the hypothesized sequence of phases	The project starts with coding

Notes: [1] Teleological theories have processes while Lifecycle theories have phases; [2] No agent applies only to teleological theories; [3] Incorrect sequence applies only to lifecycle theories

Furthermore, in modern epistemology, testability is treated as a mutual property between two theories. “That some propositions are testable, while others are not, was a fundamental idea in the philosophical program known as logical empiricism. That program is now widely thought to be defunct”; rather, “testing is an inherently contrastive activity – testing a hypothesis means testing it against some set of alternatives” [139]. Consequently, the testable propositions derivable from SCI depend on the process theory against which it is tested.

For example, treating Waterfall as a process theory, SCI is testable against it on the first four dimensions of Table 3. Suppose we observe a testing specialist applying several kinds of interconnected testing techniques in a manner loosely coupled with analysis, design or programming activities. This would favor Waterfall’s view of testing and indicate activity overload or deficit in SCI. In contrast, suppose we observe two distinct kinds of testing – acceptance testing, where an analyst reviews a prototype with a client, and unit testing, where developers build test suites while building the software. Further suppose acceptance testing appears tightly coupled with other activities aimed at understanding the client’s views and desiderata (Sensemaking), while unit testing appears tightly coupled with programming (Implementation). This would favor SCI's view of testing and suggest that Waterfall's "testing" phase is overloaded.

Although methodological advice for evaluating process theories is less abundant than for variance theories, some authors recommend questionnaires and field studies [110, 160]. For questionnaires, items may be generated for each contrasting prediction on bipolar scales such that, for example, a 'strong disagree' would indicate beliefs consistent with SCI where 'strong agree' would indicate beliefs consistent with Waterfall. Meanwhile, collecting in-depth, longitudinal data from a small number of software development teams may provide a rich evidentiary base for each theory. This evidence may be compared using a coding scheme based on the two theories to determine which is more veracious. Furthermore, combining the two approaches enables multi-method triangulation – the survey allows for random sampling and reliability while the field study facilitates gathering deep insights into developer behaviors and cognitive processes. While this paper primarily employs conceptual evaluation, due to the complexity of theoretically justifying SCI, this discussion demonstrates that SCI is testable in principle.

4.6 Limitations of and Future Additions to SCI

The presentation of SCI given in this paper is limited in several important ways. First, the purpose of this paper is to provide a comprehensive account of SCI – empirical evaluation is handled elsewhere [112, 117]. Such an account is necessary as SCI integrates diverse research from numerous disciplines, which may be unfamiliar to many software engineering researchers.

Second, SCI focuses on activities strictly necessary for software design to occur. It does not comprehensively enumerate all design-related activities. For example, many developers draw diagrams; however, as diagrams are not strictly necessary to construct software, SCI does not include a 'diagramming' activity. This is consistent with existing design process theories – none of the process theories reviewed in this paper attempt to capture all possible optional design activities. Sim and Duffy proposed an "ontology of generic engineering design activities" including "abstracting", "decision making" and "searching" [136]. Mapping Sim and Duffy's ontology onto SCI may be a fruitful area for future research.

Similarly, accounting for each aspect of design in one process theory would significantly increase its complexity. Like the other process theories reviewed, SCI omits important concepts including architecture, environmental factors, individual differences, intra-agent dynamics, negotiation, politics, power, quality, success and time. Attempting to account for all these dimensions would hamper the theory's communicability and usefulness as a simplifying paradigm for understanding design, undermining the motivation given in the introduction. Therefore, exploring these and other dimensions of design is left to future work while the current theory focuses on explaining design behavior as simply and understandably as possible.

5 RELATED WORK

This section reviews the three process theories omitted from Section 2 and situates SCI in relation to three major themes in the software engineering and information systems development literature. These themes were chosen simply because they are useful in this context and do not represent a comprehensive review.

5.1 The Basic Design Cycle

The Basic Design Cycle models design as a primarily linear sequence of problem analysis, solution synthesis and simulative testing, culminating in a decision to proceed with the proposed solution "or try again and generate a better design proposal" [126] (Figure 6). It meets all three process theory criteria: 1) it relates design to lower-level activities (synthesis, simulation, evaluation); 2) it has a lifecycle causal motor; 3) it includes a claim to universality – "someone who claims to have solved a design problem has gone through this cycle at least once" [126]. Its structure is not unlike Waterfall (above).

The Basic Design Cycle mostly shares Waterfall's benefits and limitations – it is easy to understand and communicate and is intuitively appealing. However, it appears inconsistent with empirical observations of expert designers [30, 132] and over-rationalizes complex development activity by implying that analysis and design are mutually exclusive, problems are given and the artifact's properties are predictable *a priori* (cf. [20, 148, 162]). To be clear, however, as it was devised to explain *engineering* design it can hardly be faulted for not accurately explaining *software* design. The Basic Design Cycle differs from SCI in several ways – 1) it prescribes a sequences of phases; 2) it proposes activity categories that are neither cohesive nor mutually exclusive; 3) it assumes problems are given and static.

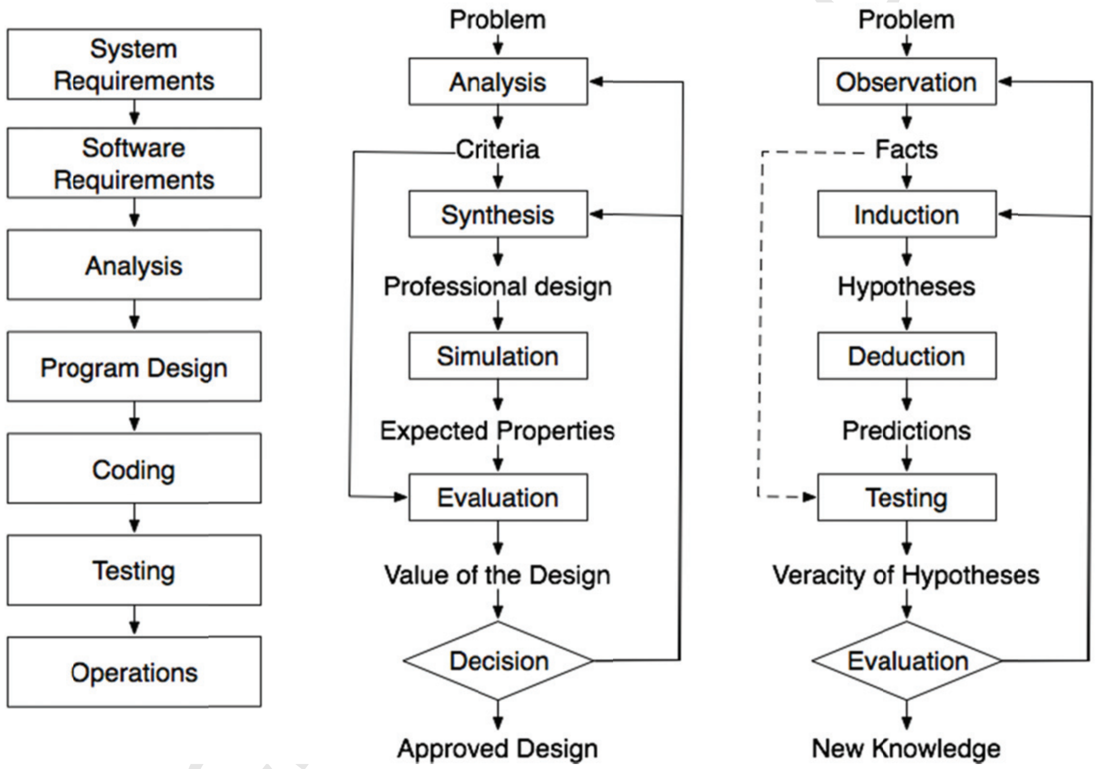

Figure 6. The Basic Design Cycle (centre) with Waterfall (left) and the Basic Cycle of Scientific Inquiry (right), (adapted from [126])

5.2 Boomerang

The Play-Test Boomerang (Figure 7) is a model of video game development centered on *play-testing* [141]. Video games differ from many software artifacts in that their primary quality dimension – *is it fun?* – is experiential an unpredictable. As such, game developers play-test prototypes, i.e., they play parts of the game to evaluate its aesthetic experience. Boomerang posits that once an initial prototype is available, play-testing (rather than coding or planning) drives the development process. Specifically, play-testing informs reconceptualization, redesign and coding.

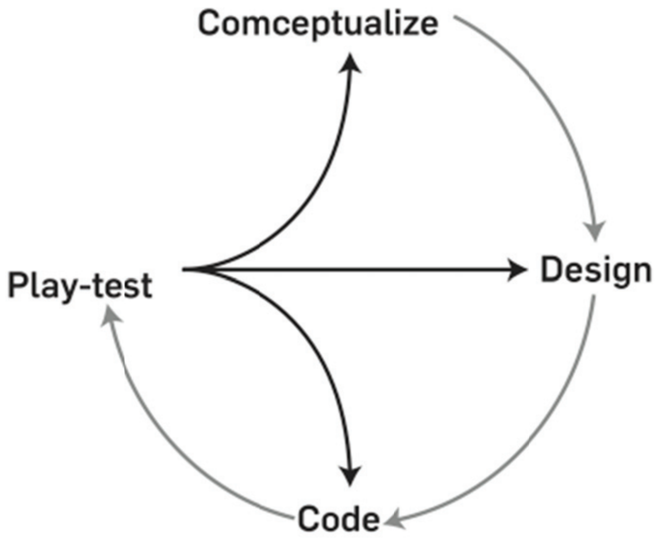

Figure 7. Boomerang (from [141])

While its proponents do not call it a theory, Boomerang meets two of the three process theory criteria: 1) it relates game development to lower-level activities (play-test, conceptualize, etc.); 2) it has a teleological causal motor. However, its proponents do not explicitly claim universality or discuss generalizability.

In any case, Boomerang is easy to understand and communicate, and is grounded in empirical observation. However, as it specifically highlights the unique aspects of game development, it is unlikely to generalize to software development more broadly. It is similar to SCI in its focus on improvisation and iteration. However, it differs from SCI in the centrality of play-testing rather than Coevolution. The relationship between play-testing and Coevolution is a potentially productive area for future research.

5.3 The Function-Behavior-Structure Framework

The Function-Behavior-Structure Framework (FBS) claims that "the purpose of designing is to transform *function*, F (where F is a set), into a *design description*, D, in such a way that the artifact being described is capable of producing those functions" [55] (original italics). The original FBS (Figure 8) had five artifacts (Table 4) and eight processes (Table 5).

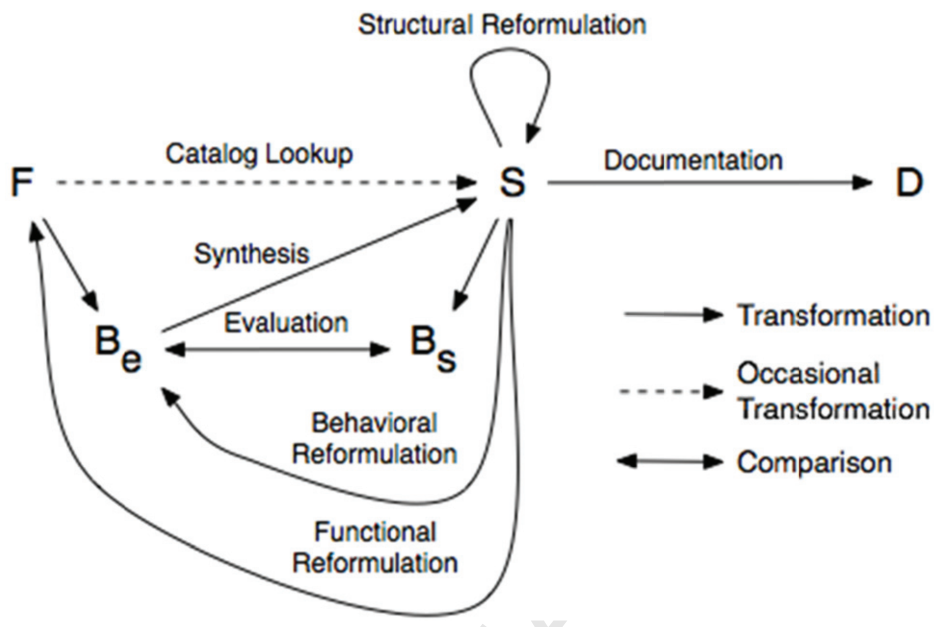

Figure 8. The Function-Behavior-Structure Framework (adapted from [78])

Table 4. FBS Artifacts (adapted from [55])

Symbol	Meaning
Be	expected (desired) behavior of the structure
Bs	"the predicted behavior of the structure" (p. 3)
D	a graphically, numerically and/or textually represented model that transfers "sufficient information about the designed artifact so that it can be manufactured, fabricated or constructed" (p. 2)
F	"the expectations of the purposes of the resulting artifact" (p. 2)
S	"the artifact's elements and their relationships" (p. 2)

Gero and Kannengiesser updated FBS to include the idea of *situatedness*, meaning that "the agent's view of a world changes depending on what the agent does" [58]. They therefore distinguish between three "worlds".

The external world is the world that is composed of representations outside the designer or design agent. The interpreted world is the world that is built up inside the designer *or design agent in terms of sensory experiences, percepts and concepts… The expected world is the world imagined actions will produce.* [58]

Table 5. (Situated) FBS Processes (after [56])

Process	**Meaning**	**Situated Activities**
Formulation	deriving expected (desired) behaviors from the set of functions	1 through 10
Synthesis	"expected behavior is used in the selection and combination of structure based on a knowledge of the behaviors produced by that structure" (p. 3)	11, 12
Analysis	the process of deriving the behavior of a structure	13, 14
Evaluation	comparing predicted behavior to expected behavior and determining whether the structure is capable of producing the functions	15
Documentation	transforming the structure into a design description that is suitable for manufacturing	12, 17, 18
Structural Reformulation	modifying the structure based on the structure and its predicted behaviors	9 (possibly driven by 3, 6 or 13)
Behavioral Reformulation	modifying the expected behaviors based on the structure and its predicted behaviors	8 (possibly driven by 2, 5, 14 or 19)
Functional Reformulation	modifying the set of functions based on the structure and its predicted behaviors	7 (possibly driven by 1, 4, 16 or 20)

They argued that function, behavior and structure have representations in each world and mapped the original eight FBS processes onto 20 activities across the three worlds. The three-world metaphor, twelve representations, eight processes and 20 activities comprise *Situated FBS* (Figure 9, Table 5).

As Situated FBS can be somewhat overwhelming, an informal exposition may help. FR^e, BR^e and SR^e are the functional, behavioral and structural requirements provided to the designer. Based on these requirements, the designer produces a structure, S^i. Be^i and Fe^i are the behaviors and functions that the designer desires S^i to produce, while B^i and F^i are the behaviors and functions the designer predicts S^i will actually produce. S^i is evaluated by comparing the desires (Be^i and Fe^i) with the predictions (B^i and F^i). It is not clear how S^i and Se^i differ. Once the designer is satisfied, she externalizes the functions, behaviors and structure (*F^i*, *B^i* and *S^i*), creating the design specification (*F^e*, *B^e*, *S^e*), which enables manufacture of the design object.

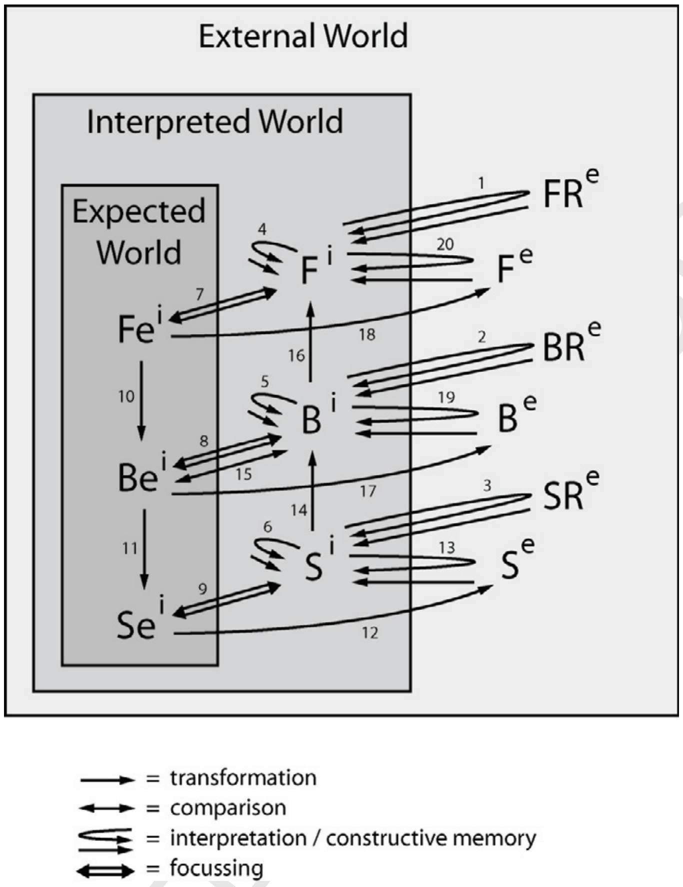

Figure 9. The Situated FBS Framework (after [56])

Note: numbers are labels and do not indicate sequence.

FBS and Situated FBS are clearly process theories – they relate design to the lower-level activities shown in Table 5, have a (teleological) causal motor and their proponents claimed that "the eight processes depicted in the FBS Framework are … fundamental for all designing" [58]. FBS was originally applied to engineering design but later generalized to software development [57, 78].

The benefits of FBS include its meticulousness and the intuitive appeal of its main constructs – developers speak in terms of a software system's requirements (functions), behavior (features) and architecture (structure). However, it is very complicated, difficult to explain to non-experts and difficult to apply in practice. More fundamentally, FBS oversimplifies and over-rationalizes design work (and differs from SCI) by assuming that requirements are known and static, and that designing is effectively a process of symbol manipulation [53].

5.4 Interpersonal Interactions in Development

Sambamurthy and Kirsch [129] identified seven core development concepts – *tasks* (of the development team including planning, analysis, design), *stakeholders*, *agenda* (stakeholders' goals), *transactions* (stakeholder actions), *context* (outside of the stakeholders and team), *structure* (e.g. methods, tools, policies) and *outcomes*. Similarly, Curtis [36] proposed a "layered development model", which gives the possible units of analysis for studying development, including "individual", "team", "project", "company" and "business milieu". Meanwhile, Bansler [8] organized development research into three traditions: the *systems theoretical*, which seeks to increase efficiency through technology; the *socio-technical*, which seeks to improve productivity and wellbeing through greater consideration of human factors; and the *critical*, which seeks to improve the employee-employer power balance. From these perspectives, SCI explains how individuals and teams behave at the task level, while abstracting the remaining concepts and layers into the *context*, from a viewpoint similar to Bansler's socio-technical perspective.

Moreover, developing large systems involves communication, learning and negotiation processes [36]. Furthermore, as the number of stakeholders and participants increases, coordination becomes increasingly challenging. This raises questions concerning what we know about communication, learning, negotiation and communication and how they relate to SCI.

While knowledge is often defined as *true, justified belief*, Orlikowski found that "knowing is not a static embedded capability or stable disposition of actors, but rather an ongoing social accomplishment, constituted and reconstituted as actors engage the world in practice" [102]. Furthermore, "organizations fail to learn from their experience in systems development because of limits of organizational intelligence, disincentives for learning, organizational designs and educational barriers" [89] and, more specifically, "when team members are immersed in a design activity, they are often unable (or unwilling) to acquire knowledge that cannot be immediately put to use" [154]. In summary, we know that acquiring knowledge in design projects is often difficult.

In software projects, knowledge is often exchanged in a dialectic process [36, 154], through discussions where criticism of old ideas generates new ideas and discussants revise beliefs. Many of these knowledge-producing interactions involve current or intended users of the design object. A climate that encourages user participation does not guarantee success but appears to help [22]. More specifically, user participation leads to critical encounters [98, 124] and conflict/resolution cycles [98, 123, 124] that may improve designs and change project trajectories. However, while some longterm trends (e.g. the Agile movement) encourage increased user participation, others undermine it. For example, the trend toward off the shelf software (instead of custom development) transfers power and participation away from users [10]. Moreover, users may be expected to take responsibility for projects even when their involvement is shallow [14].

Coordination, communication and control are tightly interconnected concerns within software projects. Adding human resources to a software project may not increase its velocity partly due to increasing coordination overhead [21]. Furthermore, as team size increases, informal, ad hoc coordination mechanisms become insufficient [96]. This insight is at the heart of document-based development methods, which seek to protect the organization from knowledge loss and miscommunication by recording crucial details in requirements specifications, software documentation and other documents [62, 71, 79]. Indeed, team coordination is improved by shared mental models [43]. However, documentation does not reduce the amount of communication required; rather, "Artificial (often political) barriers to communication among project teams [create] a need for individuals to span team boundaries and to create informal communication networks" [35]. This partly explains designers' need for both vertical coordination (coordination between developers and users, through authorized entities including project managers) and horizontal coordination (between developers and users directly) [99]. Geographic distance negatively affects coordination; however, this relationship is mediated by shared knowledge [44]. Finally, coordination issues are complicated by users, clients and managers who use formal and informal modes of control to influence development [76, 77, 100]. Formal controls include defining appropriate business processes (behavioral control) and performance targets (outcome control). Informal controls include individuals setting their own targets (self-control) and groups that share values, approaches and goals (clan control).

SCI focuses on design *activities* from the perspective of the design agent. The term 'knowledge' was not used in SCI due to difficulty defining it. Rather, SCI posits that designers generate, organize and revise beliefs and ideas about the project's context and design object through Sensemaking and Coevolution. Of course, whether the design agent is an individual or a team, one would expect complex intra-agent knowledge generation and communication processes. However, SCI sits at an intermediate level of abstraction between the individual and the organization. As such, intra-agent phenomena including the cognitive process of individual designers and coordination within design teams are not represented. Drilling down to a lower level of abstraction is left to future research.

Similarly, design projects may involve management processes (including coordination) parallel to the design process. Management activities are omitted from SCI not only to control scope but also as they are not inherent to design in the same way as SCI's activities. For example, consider a "guerrilla project" where well-meaning employees develop an application "under the radar" of management, relying on clan control (cf. [77]). Rather than a relationship between design agent and context, which could be shown in SCI, coordination in this case exists primarily within the design agent, i.e., at a lower level of abstraction [36] than SCI. This caveat acknowledged, interconnections between management process and core design activities represent another potentially beneficial area for future research.

Considering user participation raises the related consideration of when a user becomes part of the design agent. Beck [15] recommends having a user onsite at all times to answer questions from the design team. Such a user could conceivably become intermingled with the design team, sharing much of the team's context schema and contributing to design decisions. At this point, the user is part of the design agent. This is related to the research stream on user-led systems design [50, 84]. The reverse question, *when does the design team split into multiple agents*, is discussed next.

5.5 The Scope of the Design Agent

Software development is increasingly distributed and global [67]. Distributed work takes more time than co-located work [68] as it requires more people [66] and coordination [20]. Consequently, how to mitigate the productivity impact of distributed work has become a key concern in the methods literature (cf. [145]). One form of distributed development that has received particular attention is Free/Libra Open Source Software Development (FLOSSD).

FLOSSD differs from traditional software engineering in its transparency [131], lack of formal methods, budgets, schedules or rule structures [130, 131], developer self-assignment of tasks [34] and success measures [32, 33]. For example, the DeLone and McLean model of information systems success [38, 39, 108] is based on a development and deployment process very different from that observed in FLOSSD; the latter requires different success indicators including number of developers, level of development activity, individual reputation and movement from alpha to stable release [33].

FLOSSD projects are organized into layers – a small group of core developers, a larger group of ad hoc developers who make minor changes and bug fixes and an even larger group who report problems [96]. Core developers create extensive but incomplete shared mental models [133]. Their adherence to the FLOSSD ideology enhances communication and trust, amplifying team effectiveness [144]. Team effectiveness is dominated by team composition and organizational context rather than tools or other technical factors [31]. Rather than traditional forms of vertical and horizontal coordination (§5.4; [99]), FLOSSD projects are coordinated using "informalisms", e.g., discussion forum threads, project wikis and to-do lists that describe or prescribe aspects of the project, which replace formalisms including requirements specifications and system documentation [130].

Furthermore, FLOSSD may be evolving from FLOSSD 1.0, where a core person or group of developers haphazardly developed a product with erratic support under GPL or a similar license, to FLOSSD 2.0 where major players are choosing open-source strategies and providing better support under a variety of licensing schemes [49]. This shift may lead to substantive changes in or bifurcation of FLOSSD norms and patterns.

FLOSSD and distributed development more generally raise questions about SCI's structure and agent concept. Curtis et al. proposed a "layered development model" that identifies the multiple units of analysis (individual, team, project, company, business milieu) involved in development research [36]. SCI uses the design agent concept to cross the individual and team levels. However, some FLOSSD projects may involve simultaneous development by two or more separate teams exhibiting competition, rivalry, contradictory goals and weak coordination ties. Moreover, while a FLOSSD project's core developer group may act as a single agent, it would be incredulous to extend the design agent concept to the outer layers. Therefore, SCI does not currently generalize to FLOSSD or distributed development generally. Exploring the implications of multi-agent design is a potentially fruitful avenue for future research.

5.6 Methods and their Limitations

Avison and Fitzgerald [6, 7] divided the history of SDMs into four eras. In the "pre-methodology era", developers trained in programming but not the "contexts of use" built software from a superficial understanding of users' needs and goals, leading to many failures. In the "early methodology era", development was divided into phases inspired by Waterfall. In the "methodology era" proper, practitioners began using the term "methodology" and many approaches emerged including structured programming, prototyping, object-oriented development and participative development. In the (current) "post-methodology era", many practitioners have rejected methodologies in general. For example, developers may omit elements of methods not out of ignorance but because they perceive them as irrelevant in their context [48]. Possible explanations for methodology rejection include perceived ineffectiveness or poor usability of specific methods and perceived ineffectiveness of greater methodicalness *in principle*. These explanations have starkly different implications – the first implies a need for better methods; the second implies a need for something other than methods.

Many studies continue to defend methods and methodicalness in principle. For example, higher process maturity (i.e., greater methodicalness) is associated with higher product quality, customer satisfaction, productivity and morale [65], which justifies Capability Maturity Model Integration (CMMI) and software process improvement [1] more generally. Many organizations do not adopt CMMI despite believing in its benefits as they think they are too small, CMMI is too costly or they lack time [142]. More recently, the Agile project management framework Scrum has been linked to higher productivity [23]. Several surveys report that projects using Agile approaches have higher success rates than projects using linear or ad hoc approaches [4, 5, 146].

However, many studies also argue that academic literature is biased toward formal methodologies [46] and methodicalness while ignoring "the possibility that amethodical development might be the normal way in which the building of [software] systems actually occurs in reality" [148]. *Amethodical* systems development refers to "management and orchestration of systems development without a predefined sequence, control, rationality, or claims to universality" [148]. Although the case for amethodical development is complex, at least three dimensions are evident:

1. Many practitioners do not use methods as prescribed [9, 93, 107].
2. Several field studies found that practitioners acted in a fundamentally amethodical manner characterized by improvisation, opportunism and either absent or impotent formal controls [11, 12, 35, 154, 162].
3. In many situations, the desiderata – goals, requirements, requests, wants, etc. – are unknown, ambiguous, disputed, inarticulable, changing, misunderstood or conflicting [20, 30, 36, 61, 69, 127, 132, 135, 147, 154, 162].

This debate raises a host of questions. Do more methodical development efforts outperform less methodical development efforts? If so, which methods are best? If not, what can increase the effectiveness of amethodical development efforts? Additionally, to what extent are these relationships context-dependent and does system complexity moderate the relationship between methodicalness and performance?

SCI is a process theory, not a method; therefore, it does not make prescriptions. However, SCI may be used to analyze prescriptions. For example, SCI posits that envisioning the design object changes the designer's understanding of the context. If correct, this implies that "freezing requirements" prior to design concept generation is irrational. Regarding the methodical/amethodical debate, SCI's assumptions are broadly more consistent with the worldview of amethodical development than with formal methods (especially plan-driven methods). For example, the former involves a reactive, improvising agent (like SCI) while the latter often assume known goals or imply that analysis, design and programming are temporally and physically separable (unlike SCI).

6 CONCLUSION

The purpose of this paper was to formulate a process theory of software design practice, which explains how complex software systems are created by cohesive software development teams in organizations. The resulting theory, SCI, posits three primary design activities – Sensemaking, Coevolution and Implementation – enacted by an independent design agent in a self-determined sequence. Each of SCI's concepts and relationships are theoretically justified by reference to existing research. Therefore, rather than proposing new constructs, the paper's contribution lies its innovative synthesis and clarification of existing constructs.

Software design process theories such as SCI are needed for two basic reasons. First, without a clear, explanatory process theory of software development, methods (especially Waterfall) are used instead, leading to oversimplified, over-rationalized views of reality. Second, lack of a clear theory of software development permits a kind of pseudo-scientific cult of method, where both well-meaning experts and avaricious con artists evangelize systems of prescriptions without sufficient empirical testing to separate the penicillin from the snake oil.

Moreover, new theory is needed as existing theories either oversimplify and over-rationalize development (Waterfall, FBS, Basic Design Cycle), do not apply to software in general (Problem-Design Exploration Model, Boomerang), or are simply ambiguous (Selfconscious Process). This is not to say that FBS, etc. are bad theories. A better interpretation would be that FBS and the Basic Design Cycle are theories about designing predictable artifacts to solve well-defined problems while SCI and Boomerang are theories about designing complex artifacts to address ambiguous situations. Some web developers, for example, have been observed employing a linear, Waterfall-like process to create very simple, static-content pages, i.e., the kind of project that one or two people can complete in a day [120].

SCI suggests numerous practical implications for software engineering practice. For example, dividing a team into business analysts, system architects, programmers and quality assurance specialists is likely to undermine coordination and cause conflict. As these roles are based on Waterfall categories, which are neither cohesive nor mutually exclusive, tasks and responsibilities cannot be divided cleanly between them. Similarly, planing a development project as a series of phases dedicated to analysis, design, etc. is likely counterproductive as the actual tasks performed during development do not divide along these lines. Additionally, SCI explains why tendering (where firms bid for a outsourcing contract based on a specification) so often produces poor, late and over-budget software. Because Coevolution changes the context schema and building the system changes the context itself, a software development projects actively obsolesces its own contract. This creates conflict between the client (whose needs change), the developers (whose understanding evolves) and the lawyers and accountants (who often fixate on the obsolete contract).

Due to the complexity of expressing and theoretically justifying the new theory, only conceptual evaluation of SCI is attempted. Conceptual evaluation suggests that SCI is reasonably simple and communicable, highly novel, potentially useful and definitely testable. Initial empirical evaluation suggests that SCI is more consistent with empirical observations of designers than FBS or Waterfall phases [112, 117].

In conclusion, this paper presents and conceptually evaluates a novel software design process theory, theoretically justifies its elements and discusses its relationship to existing process theories and themes in the development literature. SCI is a theory, not a method; it aims to explain development phenomena, not to prescribe a specific approach. SCI is not a GTSE as it focuses on complex systems and cohesive teams. However, SCI may provide one of the core theories from which a general theory could be constructed.

ACKNOWLEDGEMENTS

Thanks are due to Jeffrey Parsons, Olga Volkoff and the participants of the First SEMAT Workshop on General Theory of Software Engineering for their constructive comments on SCI, and to the National Sciences and Engineering Research Council of Canada for financial support.

BIBLIOGRAPHY

[1] Aaen, I., Arent, J., Mathiassen, L. and Ngwenyama, O. 2001. A conceptual map of software process improvement. *Scandinavian Journal of Information Systems*. 13, 123–146.
[2] Alexander, C.W. 1964. *Notes on the synthesis of form*. Harvard University Press.
[3] Alter, S. 2013. Work system theory: overview of core concepts, extensions, and challenges for the future. *Journal of the Association for Information Systems*. 14, 2, 72–121.
[4] Ambler, S. 2010. 2010 IT project success rates. *Dr. Dobb's Journal*. http://www.drdobbs.com/architecture-and-design/226500046.
[5] Ambler, S. 2008. Agile adoption rate survey results. *Ambysoft*. http://www.ambysoft.com/surveys/agileFebruary2008.html.
[6] Avison, D. and Fitzgerald, G. 2003. *Information Systems Development: Methodologies, Techniques and Tools*. McGraw-Hill Education.
[7] Avison, D. and Fitzgerald, G. 1999. Information Systems Development. *Rethinking Management Information Systems: An Interdisciplinary Perspective*. W. Currie and R. Galliers, eds. Oxford University Press. 250–250.
[8] Bansler, J. 1989. Systems development research in Scandinavia: Three theoretical schools. *Scandinavian Journal of Information Systems*. 1, 1.
[9] Bansler, J. and Bødker, K. 1993. A Reappraisal of structured analysis: Design in an organizational context. *ACM Transactions on Information Systems*. 11, 2, 165–193.
[10] Bansler, J. and Havn, E. 1994. Information systems development with generic systems. *Proceedings of the Second European Conference on Information Systems*. W. Baets, ed. Nijenrode University Press. 707–715.
[11] Baskerville, R. and Pries-Heje, J. 2004. Short cycle time systems development. *Information Systems Journal*. 14, 3, 237–264.
[12] Baskerville, R., Travis, J. and Truex, D.P. 1992. Systems without method: The impact of new technologies on information systems development projects. *Proceedings of the IFIP WG8.2 Working Conference on The Impact of Computer Supported Technologies in Information Systems Development*. North-Holland Publishing Co. 241–269.

[13] Beal, D.J., Cohen, R.R., Burke, M.J. and McLendon, C.L. 2003. Cohesion and Performance in Groups: A Meta-Analytic Clarification of Construct Relations. *Journal of Applied Psychology*. 88, 6, 989–1004.

[14] Beath, C.M. and Orlikowski, W.J. 1994. The Contradictory Structure of Systems Development Methodologies: Deconstructing the IS-User Relationship in Information Engineering. *Information Systems Research*. 5, 4, 350–377.

[15] Beck, K. 2005. *Extreme programming eXplained: Embrace change*. Addison Wesley.

[16] Beck, K. 2002. *Test driven development: By example*. Addison-Wesley Professional.

[17] Berente, N. and Lyytinen, K. 2006. The Iterating Artifact as a Fundamental Construct for Information System Design. *1st International Conference on Design Science in Information Systems and Technology*.

[18] Boehm, B. 1988. A spiral model of software development and enhancement. *IEEE Computer*. 21, 5, 61–72.

[19] Bourque, P. and Dupuis, R. eds. 2004. *Guide to the software engineering body of knowledge (SWEBOK)*. IEEE Computer Society Press.

[20] Brooks, F.P. 2010. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional.

[21] Brooks, F.P. 1995. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc.

[22] Butler, T. and Fitzgerald, B. 1997. A case study of user participation in the information systems development process. *Proceedings of the 18th International Conference on Information Systems*. AIS. 411–426.

[23] Cardozo, E., Neto, J., Barza, A., França, A. and da Silva, F. 2010. SCRUM and Productivity in Software Projects: A Systematic Literature Review. *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*.

[24] Carroll, G. and Hannan, M.T. 1989. Density Delay in the Evolution of Organizational Populations: A Model and Five Empirical Tests. *Administrative Science Quarterly*. 34, 411–430.

[25] Checkland, P. 1999. *Systems Thinking, Systems Practice*. Wiley.

[26] Churchman, C.W. 1971. *The design of inquiring systems: Basic concepts of systems and organization*. Basic Books.

[27] Clark, A. and Chalmers, D. 1998. The Extended Mind. *Analysis*. 58, 1, 7–19.

[28] Colman, A. 2008. *A Dictionary of Psychology*. Oxford University Press.

[29] Cross, N. 1992. Research in Design Thinking. *Research in design thinking*. N. Cross, K. Dorst, and N. Roozenburg, eds. Delft University Press.

[30] Cross, N., Dorst, K. and Roozenburg, N. 1992. *Research in design thinking*. Delft University Press.

[31] Crowston, K., Annabi, H., Howison, J. and Masango, C. 2005. Effective Work Practices for FLOSS Development: A Model and Propositions. *Proceedings of the 42nd Hawaii International Conference on System Sciences*. IEEE. 197a.

[32] Crowston, K., Annabi, H., Howison, J. and Masango, C. 2004. Towards a portfolio of FLOSS project success measures. *Proceedings of the 4th Workshop on Open Source Software Engineering, 26th International Conference on Software Engineering*. IET. 29–33.

[33] Crowston, K., Howison, J. and Annabi, H. 2006. Information systems success in free and open source software development: Theory and measures. *Software Process: Improvement and Practice (Special Issue on Free/Open Source Software Processes)*. 11, 123-148.

[34] Crowston, K., Li, Q., Wei, K., Eseryel, U.Y. and Howison, J. 2007. Self-organization of teams for free/libre open source software development. *Information and Software Technology*. 49, 6, 564–575.

[35] Curtis, B., Kellner, M.I. and Over, J. 1992. Process Modeling. *Communications of the ACM*. 35, 9, 75–90.

[36] Curtis, B., Krasner, H. and Iscoe, N. 1988. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*. 31, 11, 1268–1287.

[37] Dardenne, A. and Lamsweerde, A. 2010. Goal-directed Requirements Acquisition. *Science of Computer Programming*. 20, (Aug.), 3–50.

[38] DeLone, W.H. and McLean, E.R. 1992. Information systems success: The quest for the dependent variable. *Information management*. 3, 60–95.

[39] DeLone, W.H. and McLean, E.R. 2003. The DeLone and McLean Model of Information Systems Success: A Ten-Year Update. *Journal of Management Information Systems*. 19, 4, 9–30.

[40] Dervin, B. 1998. Sense-making theory and practice: an overview of user interests in knowledge seeking and use. *Journal of Knowledge Management*. 2, 2, 36–46.

[41] Dorst, K. and Cross, N. 2001. Creativity in the design process: Co-evolution of problem-solution. *Design Studies*. 22, (Sep.), 425–437.

[42] Eekels, J. 2000. On the Fundamentals of Engineering Design Science: The Geography of Engineering Design Science. Part 1. *Journal of Engineering Design*. 11, (Dec.), 377–397.

[43] Espinosa, A., Kraut, R., Lerch, J., Slaughter, S., Herbsleb, J. and Mockus, A. 2001. Shared mental models and coordination in large-scale, distributed software development. *Proceedings of the 22nd International Conference on Information Systems*. AIS. 513–518.

[44] Espinosa, J.A., Sandra, A.S., Robert, E.K. and James, D.H. 2007. Familiarity, Complexity and Team Performance in Geographically Distributed Software Development. *Organization Science*. 18, 4, 613–630.

[45] Ewusi-Mensah, K. 2003. *Software Development Failures*. MIT Press.

[46] Fitzgerald, B. 1996. Formalized systems development methodologies: a critical perspective. *Information Systems Journal*. 6, 1, 3–23.

[47] Fitzgerald, B. 2006. The Transformation of Open Source Software. *MIS Quarterly*. 30, 3, 587–598.

[48] Fitzgerald, B. 1997. The use of systems development methodologies in practice: a field study. *Information Systems Journal*. 7, 3, 201–212.

[49] Fitzgerald, B., Hartnett, G. and Conboy, K. 2006. Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*. 15, 2, 200–213.

[50] Franz, C.R. and Robey, D. 1984. An investigation of user-led system design: rational and political perspectives. *Communications of the ACM*. 27, 12, 1202–1209.

[51] Freeman, P. and Hart, D. 2004. A Science of design for software-intensive systems. *Communications of the ACM*. 47, 8, 19–21.

[52] Furnas, G.W. and Russell, D.M. 2005. Making sense of sensemaking. *CHI '05 extended abstracts on human factors in computing systems*. ACM. 2115–2116.

[53] Galle, P. 2009. The Ontology of Gero's FBS Model of Designing. *Design Studies*. 30, 4, 321–339.

[54] Gell-Mann, M. 1999. Complex adaptive systems. *Complexity: Metaphors, models and reality*. Westview Press. 17–45.

[55] Gero, J.S. 1990. Design prototypes: A knowledge representation schema for design. *AI Magazine*. 11, 4, 26–36.

[56] Gero, J.S. and Kannengiesser, U. 2007. A Function-Behavior-Structure Ontology of Processes. *Artificial Intelligence for Engineering Design Analysis and Manufacturing*. 21, 4, 379–391.

[57] Gero, J.S. and Kannengiesser, U. 2007. An ontological model of emergent design in software engineering. *16th International Conference on Engineering Design*.

[58] Gero, J.S. and Kannengiesser, U. 2004. The Situated Function-Behaviour-Structure Framework. *Design Studies*. 25, 4, 373–391.

[59] Gero, J.S. and McNeill, T. 1998. An Approach to the Analysis of Design Protocols. *Design Studies*. 19, 1, 21–61.

[60] Giddens, A. 1984. *The constitution of society: Outline of the theory of structuration*. John Wiley & Sons.

[61] Gladden, G.R. 1982. Stop the Life-Cycle, I Want to Get Off. *SIGSOFT Software Engineering Notes*. 7, 2, 35–39.

[62] Great Britain Office of Government Commerce 2009. *Managing successful projects with PRINCE2*. Stationery Office Books.

[63] Gregor, S. 2006. The Nature of Theory in Information Systems. *MIS Quarterly*. 30, 3, 611–642.

[64] Heineman, G.T. and council, W.T. 2001. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman.

[65] Herbsleb, J., Zubrow, D., Goldenson, D., Hayes, W. and Paulk, M. 1997. Software quality and the Capability Maturity Model. *Communications of the ACM*. 40, 6, 30–40.

[66] Herbsleb, J.D. and Mockus, A. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Transaction on Software Engineering*. 29, 6, 481–494.

[67] Herbsleb, J.D. and Moitra, D. 2001. Global software development. *IEEE Software*. 18, 2, 16–20.

[68] Herbsleb, J.D., Mockus, A., Finholt, T.A. and Grinter, R.E. 2001. An empirical study of global software development: distance and speed. *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society. 81–90.

[69] Herbsleb, J.D., Paulish, D.J. and Bass, M. 2005. Global software development at Siemens: experience from nine projects. *Proceedings of the 27th International Conference on Software Engineering*. ACM. 524–533.

[70] Holland, J.H. 1992. Complex Adaptive Systems. *Daedalus*. 121, 1 (Jan.), 17–30.

[71] Jacobson, I., Booch, G. and Rumbaugh, J. 1999. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc.

[72] Johnson, P., Ekstedt, M. and Jacobson, I. 2012. Where's the Theory for Software Engineering? *IEEE Software*. 29, 5, 94–96.

[73] Johnson, P., Ralph, P., Goedicke, M., Ng, P.-W., Stol, K.-J., Smolander, K., Exman, I. and Perry, D.E. 2013. Report on the Second SEMAT Workshop on General Theory of Software Engineering (GTSE 2013). *SIGSOFT Software Engineering Notes*. 38, 5, 47-50.

[74] Kim, J. 1999. Causation. *The Cambridge Dictionary of Philosophy*. R. Audi, ed. Cambridge University Press. 125–127.

[75] Kimberly, J. and Miles, R. 1980. *The Organizational Life Cycle*. Jossey-Bass.

[76] Kirsch, L.J. 1997. Portfolios of Control Modes and IS Project Management. *Information Systems Research*. 8, 3, 215-239.

[77] Kirsch, L.J., Sambamurthy, V., Ko, D.-G. and Purvis, R.L. 2002. Controlling Information Systems Development Projects: The View from the Client. *Management Science*. 48, 4, 484–498.

[78] Kruchten, P. 2005. Casting software design in the Function-Behavior-Structure Framework. *IEEE Software*. 22, 2, 52–58.

[79] Kruchten, P. 2003. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional.

[80] Kuhn, T.S. 1996. *The Structure of Scientific Revolutions*. University of Chicago Press.

[81] Larman, C. and Basili, V.R. 2003. Iterative and Incremental Development: A Brief History. *IEEE Computer*. 36, 6, 47–56.

[82] Laudon, K., Laudon, J. and Brabston, M. 2009. *Management Information Systems: Managing the Digital Firm*. Pearson, Prentice Hall.

[83] Law, J. and Hassard, J. eds. 1999. *Actor network theory and after*. Blackwell Publishers.

[84] Lawrence, M. and Low, G. 1993. Exploring Individual User Satisfaction within User-Led Development. *MIS Quarterly*. 17, 2, 195–208.

[85] Lewis, K., Lange, D. and Gillis, L. 2005. Transactive Memory Systems, Learning, and Learning Transfer. *Organization Science*. 16, 6 (Nov.), 581–598.

[86] Lewis, M.W. 2000. Exploring Paradox: Toward a More Comprehensive Guide. *The Academy of Management Review*. 25, 4 (Jan.), 760–776.

[87] Löwgren, J. and Stolterman, E. 2004. *Thoughtful Interaction Design: A Design Perspective On Information Technology*. MIT Press.

[88] Lyytinen, K. 1987. A Taxonomic Perspective of Information Systems Development: Theoretical Constructs and Recommendations. *Critical Issues in Information Systems Research*. R. Boland and R. Hirschheim, eds. John Wiley & Sons, Inc. 3–41.

[89] Lyytinen, K. and Robey, D. 1999. Learning failure in information systems development. *Information Systems Journal*. 9, 2, 85–101.

[90] Maher, M., Poon, J. and Boulanger, S. 1995. Formalising design exploration as co-evolution: A combined gene approach. *Preprints of the Second IFIP WG5.2 Workshop on Advances in Formal Design Methods for CAD*. 1–28.

[91] March, J.G. and Simon, H.A. 1958. *Organizations*. Wiley.

[92] Markus, M.L. and Robey, D. 1988. Information technology and organizational change: causal structure in theory and research. *Management Science*. 34, 5, 583–599.

[93] Mathiassen, L. and Purao, S. 2002. Educating reflective systems developers. *Information Systems Journal*. 12, 2, 81–102.

[94] McCracken, D.D. and Jackson, M.A. 1982. Life cycle concept considered harmful. *SIGSOFT Software Engineering Notes*. 7, 2, 29–32.

[95] Meyer, B. 1988. Reusability: the case for object-oriented design. *Software reuse: emerging technology*. IEEE Computer Society Press. 201–215.

[96] Mockus, A., Fielding, R.T. and Herbsleb, J.D. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering Methodology*. 11, 3, 309–346.

[97] Neal, M.A. and Northcraft, G.B. 1991. Research in organizational behavior. *Research in Organizational Behavior*. L.L. Cummings and B.M. Staw, eds. JAI Press. 147–190.

[98] Newman, M. and Robey, D. 1992. A Social Process Model of User-Analyst Relationships. *MIS Quarterly*. 16, 2, 249–266.

[99] Nidumolu, S. 1995. The Effect of Coordination and Uncertainty on Software Project Performance: Residual Performance Risk as an Intervening Variable. *Information Systems Research*. 6, 3, 191–219.

[100] Nidumolu, S.R. and Subramani, M.R. 2003. The Matrix of Control: Combining Process and Structure Approaches to Managing Software Development. *Journal of Management Information Systems*. 20, 3, 159–196.

[101] Orlikowski, W.J. 1993. CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development. *MIS Quarterly*. 17, 3, 309–340.

[102] Orlikowski, W.J. 2002. Knowing in Practice: Enacting a Collective Capability in Distributed Organizing. *Organization Science*. 13, 3, 249–273.

[103] Orlikowski, W.J. 1993. Learning from notes: Organizational issues in groupware implementation. *The Information Society*. 9, 3, 237–250.

[104] Pahl, G., Beitz, W., Feldhusen, J. and Grote, K.-H. 2007. *Engineering Design: A Systematic Approach*. Springer-Verlag.

[105] Papazoglou, M.P. 2003. Service-oriented computing: Concepts, characteristics and directions. *Proceedings of the Fourth International Conference on Web Information Systems Engineering*. IEEE.

[106] Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 15, 12, 1053–1058.

[107] Parnas, D.L. and Clements, P.C. 1986. A rational design process: How and why to fake it. *IEEE Transaction on Software Engineering*. 12, 2, 251–257.

[108] Petter, S., DeLone, W. and McLean, E. 2008. Measuring information systems success: models, dimensions, measures, and interrelationships. *European Journal of Information Systems*. 17, 3, 236–263.

[109] Polya, G. 1957. *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press.

[110] Poole, M., Van de Ven, A.H., Dooley, K. and Holmes, M.E. 2000. *Organizational change and innovation processes theory and methods for research*. Oxford University Press.

[111] Purao, S., Rossi, M. and Bush, A. 2002. Towards an Understanding of Problem and Design Spaces during Object-oriented Systems Development. *Information and Organization*. 12, 4, 249–281.

[112] Ralph, P. 2010. Comparing Two Software Design Process Theories. *Global Perspectives on Design Science Research: Proceedings of the 5th International Conference, DESRIST 2010*. R. Winter, J.L. Zhao, and S. Aier, eds. Springer. 139–153.

[113] Ralph, P. 2010. *Fundamentals of Software Design Science*. University of British Columbia. PhD Dissertation.

[114] Ralph, P. 2012. Improving coverage of design in information systems education. *Proceedings of the 2012 International Conference on Information Systems*. AIS.

[115] Ralph, P. 2011. Introducing an Empirical Model of Design. *Proceedings of The 6th Mediterranean Conference on Information Systems*. AIS.

[116] Ralph, P. 2012. Sensemaking-Coevolution-Implementation Theory: a model of the software engineering process in practice. *Proceedings of The 1st Semat Workshop on a General Theory of Software Engineering*. SEMAT.org.

[117] Ralph, P. 2013. Software Engineering Process Theory: A Multi-Method Comparison of Sensemaking-Coevolution-Implementation Theory and Function-Behavior-Structure Theory. *(Working Paper)*. arXiv:1307.1019 [cs.SE].

[118] Ralph, P. 2013. The Illusion of Requirements in Software Development. *Requirements Engineering*. 18, 3, 293–296.

[119] Ralph, P. 2013. The Two Paradigms of Software Design. *(Working Paper)*. arXiv:1303.5938 [cs.SE].

[120] Ralph, P. and Narros, E. 2013. An Exploratory Case Study of the Complexity/Agility Relationship in Small Software Design Projects. *Proceedings of the Pacific Asia Conference on Information Systems*. AIS.

[121] Ralph, P. and Wand, Y. 2009. A Proposal for a Formal Definition of the Design Concept. *Design Requirements Engineering: A Ten-Year Perspective*. K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. Robinson, eds. Springer-Verlag. 103–136.

[122] Ralph, P., Johnson, P. and Jordan, H. 2013. Report on the First SEMAT Workshop on a General Theory of Software Engineering (GTSE 2012). *SIGSOFT Software Engineering Notes*. 38, 2, 26–28.

[123] Robey, D. and Farrow, D. 1982. User Involvement in Information System Development: A Conflict Model and Empirical Test. *Management Science*. 28, 1, 73–85.

[124] Robey, D. and Newman, M. 1996. Sequential patterns in information systems development: an application of a social process model. *ACM Trans. Inf. Syst.* 14, 1, 30–63.

[125] Robey, D., Welke, R. and Turk, D. 2001. Traditional, iterative, and component-based development: A social analysis of software development paradigms. *Information Technology and Management*. 2, 1, 53–70.

[126] Roozenburg, N. and Eekels, J. 1995. *Product Design: Fundamentals and Methods*. Wiley.

[127] Royce, W. 1970. Managing the development of large software systems. *Proceedings of WESCON*. IEEE. 1–9.

[128] Russell, D.M., Stefik, M.J., Pirolli, P. and Card, S.K. 1993. The cost structure of sensemaking. *Proceedings of the INTERACT '93 and CHI '93 conference on human factors in computing systems*. ACM. 269–276.

[129] Sambamurthy, V. and Kirsch, L.J. 2000. An Integrative Framework of the Information Systems Development Process. *Decision Sciences*. 31, 2, 391–411.

[130] Scacchi, W. 2007. Free/Open Source Software Development: Recent Research Results and Methods. *Advances in Computers*. V.Z. Marvin, ed. Elsevier. 243–295.

[131] Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S. and Lakhani, K. 2006. Understanding Free/Open Source Software Development Processes. *Software Process: Improvement and Practice*. 11, 2, 95–105.

[132] Schön, D.A. 1983. *The reflective practitioner: how professionals think in action*. Basic Books.

[133] Scozzi, B., Crowston, K., Yeliz Eseryel, U. and Qing, L. 2008. Shared Mental Models among Open Source Software Developers. *Proceedings of the 42nd Hawaii International Conference on System Sciences*. IEEE. 306–306.

[134] SEMAT Call for Action: *http://semat.org/?page_id=2*. Accessed: 2012-11-21.

[135] Shenhar, A.J., Dvir, D., Levy, O. and Maltz, A.C. 2001. Project Success: A Multidimensional Strategic Concept. *Long Range Planning*. 34, 6, 699–725.

[136] Sim, S.K. and Duffy, A.H.B. 2003. Towards an ontology of generic engineering design activities. *Research in Engineering Design*. 14, 4, 200–223.

[137] Simon, H.A. 1996. *The Sciences of the Artificial*. MIT Press.

[138] Singer, E.A. 1959. *Experience and Reflection*. University of Pennsylvania Press.

[139] Sober, E. 1999. Testability. *Proceedings and Addresses of the American Philosophical Association*. 73, 2, 47–76.

[140] Sommerville, I. 1996. *Software Engineering*. Addison Wesley.

[141] Stacey, P. and Nandhakumar, J. 2008. Opening up to agile games development. *Communications of the ACM*. 51, 12, 143–146.

[142] Staples, M., Niazi, M., Jeffery, R., Abrahams, A., Byatt, P. and Murphy, R. 2007. An exploratory study of why organizations do not adopt CMMI. *Journal of Systems and Software*. 80, 6, 883–895.

[143] Stevens, W.P., Myers, G.J. and Constantine, L.L. 1974. Structured design. *IBM Systems Journal*. 13, 2, 115–139.

[144] Stewart, K.J. and Gosain, S. 2006. The Impact of Ideology on Effectiveness in Open Source Software Development Teams. *MIS Quarterly*. 30, 2, 291–314.

[145] Sutherland, J., Schoonheim, G., Rustenburg, E. and Rijk, M. 2008. Fully Distributed Scrum: The Secret Sauce for Hyperproductive Offshored Development Teams. *Proceedings of Agile*. 339–344.

[146] The State of Agile Development: 2011. *http://www.versionone.com/state_of_agile_development_survey/11/*. Accessed: 2012-07-02.

[147] Truex, D.P., Baskerville, R. and Klein, H. 1999. Growing systems in emergent organizations. *Communications of the ACM*. 42, 8, 117–123.

[148] Truex, D.P., Baskerville, R. and Travis, J. 2000. Amethodical systems development: the deferred meaning of systems development methods. *Accounting, Management and Information Technologies*. 10, 1, 53–79.

[149] Van de Ven, A.H. 2007. *Engaged scholarship: a guide for organizational and social research*. Oxford University Press.

[150] Van de Ven, A.H. and Poole, M.S. 1995. Explaining development and change in organizations. *The Academy of Management Review*. 20, 3, 510–540.

[151] van Lamsweerde, A. 2004. Goal-oriented requirements engineering: a roundtrip from research to practice. *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*. IEEE. 4–7.

[152] Venable, J. 2006. A framework for Design Science research activities. *Information Resources Management Association International Conference*. (May), 184–187.

[153] Vermaas, P.E. and Dorst, K. 2007. On the Conceptual Framework of John Gero's FBS-Model and the Prescriptive Aims of Design Methodology. *Design Studies*. 28, 2, 133–157.

[154] Walz, D.B., Elam, J.J. and Curtis, B. 1993. Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration. *Communications of the ACM*. 36, 10, 63–77.

[155] Wand, Y. and Weber, R. 2002. Research Commentary: Information Systems and Conceptual Modeling--A Research Agenda. *Information Systems Research*. 13, 4, 363–376.

[156] Wang, D. and Ilhan, A.O. 2009. Holding Creativity Together: A Sociological Theory of the Design Professions. *Design Issues*. 25, 1, 5–21.

[157] Wegner, D.M. 1987. Transactive memory: A contemporary analysis of the group mind. *Theories of group behavior*. B. Mullen and G.R. Goethals, eds. Springer. 185–208.

[158] Weick, K. 1995. *Sensemaking in Organizations*. Sage.

[159] Weick, K.E., Sutcliffe, K.M. and Obstfeld, D. 2005. Organizing and the Process of Sensemaking. *Organization Science*. 16, 4, 409–421.

[160] Wolfe, R.A. 1994. Organizational innovation: review, critique and suggested research directions. *Journal of Management Studies*. 31, 3, 405–431.
[161] Wynekoop, J. and Russo, N. 1997. Studying system development methodologies: an examination of research methods. *Information Systems Journal*. 7, (Jan.), 47–65.
[162] Zheng, Y., Venters, W. and Cornford, T. 2011. Collective agility, paradox and organizational improvisation: the development of a particle physics grid. *Information Systems Journal*. 21, 4, 303–333.