Design Artifacts are Central: Foundations for a Theory of Software Engineering

Technical Report MSU-20150420

April 2015

Edward B. Allen*
Mississippi State University
edward.allen@computer.org

Abstract

Software engineering is widely acknowledged to lack a foundational theory similar to other fields of science and engineering. Software engineering does have microtheories that address a wide variety of issues, such as the IEEE Software Engineering Body of Knowledge (SWEBOK). More recently, Jacobson *et al.* have developed the Software Engineering Methods and Theory (SEMAT) Essence Kernel as a step toward a general theory. This paper argues that the core of software engineering is people making design artifacts under uncertainty that enable the production and delivery of executable software. Design artifacts are representations of the product that express design decisions. We argue that representations of the product are constructed from requirements definition through coding. The contribution of this paper is an explanatory theory for the core of software engineering in the form of an ontology of definitions and propositions. The structure of the ontology is formal, but the underlying semantics remain intuitive. We adopt the SEMAT vocabulary where approporiate and we adopt a cognitive-science definition of "design" from Visser. The ontology gives principled insight into software engineering practice. This is illustrated by a discussion of selected so-called "laws" of common wisdom, collected by Endres and Rombach. Several management issues are also discussed. The theory has implications for understanding software-engineering practice and for guiding software-development management.

Keywords: design, ontology, software engineering, theory

*Readers may contact the author through Edward B. Allen, Department of Computer Science and Engineering, Box 9637, Mississippi State University, Mississippi State, Mississippi 39762, (662)325-7449, edward.allen@computer.org.

1. Introduction

A general theory of software engineering is widely acknowledged to be needed [32], even though the field has a variety of microtheories that address specialized issues. The IEEE Software Engineering Body of Knowledge (SWEBOK) [18] is a prime example of a compendium of such microtheories. Johnson, Ekstedt, and Jacobson [22] argue that a unified theory of software engineering is needed. They counter arguments that software engineering doesn't need theory, that software engineering already has its theory, and that software engineering can't have a theory.

Theoretical physics, where mathematics seems to explain everything, is the prime exemplar of theory in science. In computer science, discrete mathematics is the foundation for the theory of computation [5], including topics such as computability, Turing machines, automata, and analysis of algorithms. Mathematics is very appealing as a foundation, because given trusted axioms, mathematically derived results are trusted as "truth," and predictions can be validated by quantitative empirical measurements.

In contrast, Simon [37] argues that "sciences of the artificial" are quite different from "sciences of the natural." Sciences of the natural have unchanging laws of nature. Sciences of the artificial are based on changeable human behavior. This implies that stable general theories for sciences of the artificial are much more difficult to formulate. Software engineering is certainly a science of the artificial, because software is an artificial construct and changeable human behavior is the salient characteristic of software-engineering processes.

Our goal is to fit together many familiar elements of software engineering into an interrelated whole that provides explanation and understanding when applied to real-world development projects. We propose that design artifacts are at the core of a general theory of software engineering. A mature predictive theory is well beyond the scope of this paper, but we offer a foundational explanatory theory.

Our theory of software engineering is structured as an ontology of definitions and propositions, rather than equations and theorems. An ontology in computer science is a model of real-world concepts, consisting of classes, attributes, relationships, and instances. Classes represent abstract concepts and instances of classes represent corresponding real-world items. The structure of the ontology is formal, but the semantics of the primitives remain intuitive. Our definitions specify classes and provide a vocabulary for the propositions which specify relationships. This paper does not explicitly itemize attributes, nor apply the ontology to case study instances.

We present an explanatory theory of software-engineering practice, not a normative prescription. Developing predictive theory elements is future work. Our ontology describes patterns of human activity, rather than laws of nature. Therefore, validating a proposition means gathering a preponderance of evidence that supports the proposition, rather than mathematical proofs. In this paper, we apply our theory to some so-called "laws" of common wisdom in software engineering from a variety of software-engineering contexts [6]. Empirical evidence for our theory can be acquired in the future through human-subject experiments, studies of artifacts, and observation of real-world software-development projects. In this paper, we assume that the context for software engineering is a project within an organization. Moreover, this paper is limited to software development; other types of engineering are not considered here.

Section 2 discusses related work that we have built upon. Section 3 presents our ontology of definitions and propositions. Section 4 discusses applications of the ontology to software-engineering practice and management. Finally, Section 5 summarizes our conclusions.

2. Related work

2.1 Theory in software engineering

Gregor [9] presents a taxonomy of theory in information-systems research consisting of the following types.

- I. Analysis — Stating what it is
- II. Explanation — Stating what it is, and how, why, when, and where
- III. Prediction — Stating what is and what will be
- IV. Explanation and Prediction — Stating what is, how, why, when, where, and what will be
- V. Design and Action —- Stating how to do something

Perry [28] points out that the above is a taxonomy of uses of theory. “Explanation” is a mapping of real world facts to abstract theory. “Prediction” by a theory is inference of something from real world facts that is not observable at that time. The theory provides the inference rules. Our theory falls largely in Category II. Explanation.

Johnson and Ekstedt [21] argue, “A unified theory of software engineering should thus be able to express, explain and predict those things that we usually call software engineering, including issues regarding programming languages, software design, software process management, requirements engineering, object orientation, code obfuscation, formal methods, contract programming, extreme programming, and more.” They discuss the benefits of a unified theory and characteristics of a mature theory with illustrations from three well known partial theories. Our work is motivated by their call for a general theory of software engineering.

Hannay, Sjøberg, and T. Dybå [13] found that theories explaining cause-effect relationships are not used much in papers reporting software-engineering experiments. About a quarter of the articles in their corpus (1993–2002) involved theory. Referenced theories were from publications in computer science, software engineering, information systems, social/behavioral sciences, cognitive psychology, management science, and economics. Hannay *et al.* call for increased use of theory in empirical software engineering where applicable. The limited use of theory of any kind and the wide variety of theories cited indicate that there is a need for a generally applicable theory of software engineering.

Johnson and Ekstedt [21] point out that software engineering does have an abundance of microtheories, even though there is no unified theory [40]. The Software Engineering Body of Knowledge (SWEBOK) [18] and textbooks are examples of collections of detailed software-engineering theory regarding how software engineering should be done. Similarly, Endres and Rombach [6] provide a compendium of "laws" from the software-engineering literature which are largely based on empirical evidence and common wisdom. Because the "laws" are from many authors for many contexts, this compendium is not a coherent theory, but it may provide hypotheses for a theory to address.

Several authors have proposed approaches to theory building [1] or proposed theories that might form a foundation for a general theory of software engineering [7, 8, 23, 27, 28, 31, 38]. Points of contact with our theory are noted in Sections 3 and 4 below where relevant.

Software Engineering Methods and Theory (SEMAT) [19,20] is a framework for software-engineering theory. In November 2014, the SEMAT Kernel transitioned to the Object Management Group (OMG) Essence standard, "Kernel and Language of Software Engineering Methods" (`http://www.omg.org/spec/Essence/1.0/`). The Essence Kernel defines vocabulary for a software project, and identifies possible states of first order elements ("alphas"). Fig. 1 depicts SEMAT relationships [19] and concepts plus a bit of context. In this paper, we adopt the SEMAT vocabulary [19] where applicable.

Beginning in 2012, workshops entitled "SEMAT Workshop on a General Theory of Software Engineering" (GTSE) have been held to support development and testing of "core, central, and general theories in the software engineering domain" (`http://semat.org/?page_id=1364`). Papers at these workshops [32] propose theories, discuss the nature of applicable theories, and present aspects of the SEMAT Essence Kernel standard. The 2014 workshop [32] concluded, "Although a widely accepted general theory of software engineering may still be some years off, objectives, positions and discussions are converging. Participants are no longer debating the need for theories or the kind of theories needed; rather, discussions center on how to develop and build support for a GTSE."

2.2 Software-engineering views of design

Design is often considered a mysterious art, perhaps because it is difficult to observe. Much of the literature in the field of software design presents either exemplar designs

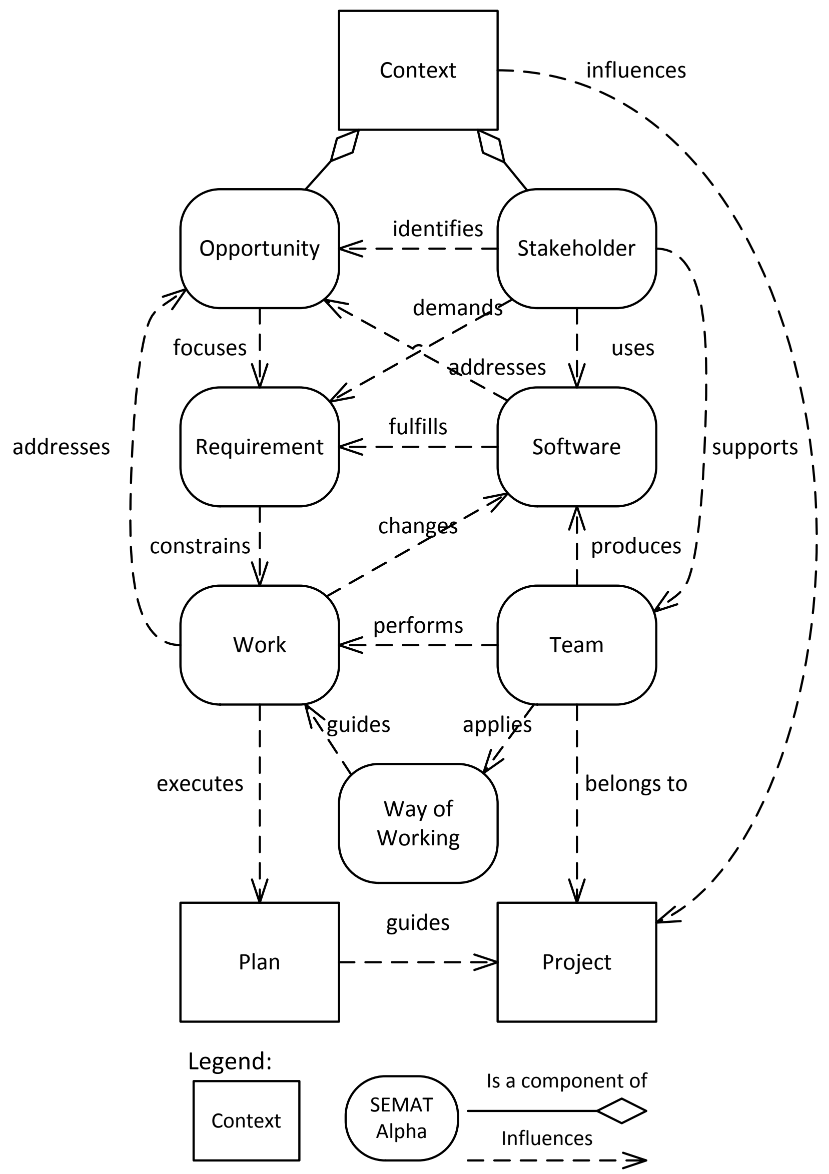

Fig. 1

SEMAT and context relationships

[10, 11, 15] or automation of supporting analyses [12, 29]. Brooks [3] emphasizes that exemplar designs are widely used to teach software design. Exemplar designs are helpful, because similar design problems recur in project after project. A successful design from the past can be adopted with some confidence if it fits the current problem. Exemplar designs fall in Gregor's Category I. Analysis. This type of theory is limited, but can form an empirical basis for more powerful theories.

Analysis of designs, often using tools, gives insight into strengths, weaknesses, trade-offs, and applicability. The goal of some tools is to identify certain design flaws or to reveal how the software would behave (e.g., simulations). Even though analysis tools are not software-engineering theory *per se*, they may have microtheories embedded in them [2].

2.3 Cognitive-science views of design

The role of cognitive skills should be an important element of a theory of software engineering. For example, one of the important cognitive skills employed during design is comprehension of previously generated artifacts. Beginning in 1986, a series of workshops [39] focused on discovering what individual programmers actually do, such as code comprehension. Various aspects of code comprehension have been studied by a number of researchers [42]. Comprehension of software-engineering diagrams is beginning to be studied as well. Mangano *et al.* [25] analyzed the role of sketches when pairs of software designers are working on design problems. Our prior research has addressed comprehension of Unified Modeling Language (UML) class diagrams [24, 43]. Beginning with a workshop in 1992 [30], the International Conference on Program Comprehension has welcomed research with cognitive-science components and approaches.

Visser [41] argues from a cognitive-science perspective that artifacts are central to understanding design processes in general. She presents several theories of design from the literature and then presents her synthesis. She says, "My own view on design . . . is to consider that design is most appropriately characterized as a construction of representations" [41, p. xviii]. However, design is not merely an intellectual activity; she points out that "Design results in artifacts" [41, p. xvii].

Visser's theory of design is based on empirical studies of industrial design of physical things. She says, "Designing consists in specifying an artifact, for example a machine tool — not in its implementation, its fabrication in the workshop. The result of design is a representation, the specifications of the machine tool. These representations are also artifacts, that is, entities created by people" [41, p. xvii]. Due to her context of industrial design, the term "requirement" is similar to a SEMAT "opportunity." She states, "We have defined design as an activity that consists in specifying an artifact product given requirements on that artifact" [41, p. 223].

To Visser, the product specification is the end result of design. She says, "Design consists in specifying an artifact (the artifact product) given requirements . . . At a cognitive level, this specification activity consists of constructing (generating, transforming, and evaluating) representations of the artifact until they are so precise, concrete, and detailed that the resulting representations — "the specifications" — specify explicitly and completely the implementation of the artifact product. . . . The difference between the final and the intermediate artifacts (representations) is a question of degree of specification, completeness, and abstraction (concretization and precision)" [41, p. 116]. In summary, Visser defines the process of design as the construction of cognitive artifacts that represent a product.

Visser's theory is attractive here, because artifacts are everywhere during software development, and a wide variety of artifacts are commonly used [25]. Construction of artifacts that represent the product is how software-design decisions get made. Artifacts that represent the product are the embodiment of design decisions.

Software engineers distinguish "requirements engineering," "software design," and "implementation" from each other. Requirements engineering includes analysis and definition of requirements. Software design includes architecture and detailed design. In contrast to Visser's definition, implementation typically includes algorithm design, coding, and unit testing. Thus, there is a mismatch between Visser's and the software-engineering community's definitions of "requirements," "design," and "implementation." In this paper, we note whose definitions are intended during the discussion.

Human cognitive abilities have definite limits in several dimensions. For example, Miller [26] found that a person's short term memory has limited capacity to remember chunks of information [6, p. 226]. Modern psychology has even more sophisticated models of how memory works and its limitations. Simon [36] argues that "bounded rationality" is an important aspect of human problem solving and design activities, in particular.

Hastie and Dawes [14, p. 249] point out that subjective utility theory, first explicated by von Neumann and Morgenstern, is the most popular normative definition of rational choice in the behavioral sciences. It is a purely mathematical theory based on logic and probability theory. Accordingly, the technical literature assumes that software engineers use highly rational thought processes and mathematical sophistication. However, Hastie and Dawes [14] present a substantial body of research showing that human behavior in making decisions does not consistently follow basic principles of rationality that conform to correct logic and probability theory, even when people have extensive training. This is due to a wide variety of cognitive phenomena, such as the following. Recall of past experience is often biased. Regression toward the mean is often not recognized. Irrelevant contexts often influence decisions. Correlation and causality are often confused. Subjective probability estimates are often subadditive. Probability theory is often not properly applied when dealing with inverse probability, conjunction and disjunction, and Bayes Theorem. These are summed up in the phrase “bounded rationality” [36].

Reason [33] proposes a taxonomy of human errors that helps explain why they happen and suggests ways to recover or prevent errors. A slip occurs when an action does not follow the person’s plan of execution, for example, a “slip of the tongue.” A lapse occurs when there is a memory failure which the person promptly notices. Skill-based errors, such as slips and lapses, occur when a person is performing routine actions in familiar situations. Rule-based mistakes occur when expertise is misapplied to the problem at hand. The rule worked in prior situations, but the wrong rule was applied in this case. Rule-based errors may be due to working memory problems or poor procedures. Knowledge-based mistakes occur when the person’s knowledge is inadequate to solve the problem at hand. The problem or environment is unfamiliar or expertise is lacking. These categories, namely, slips, lapses, rule-based mistakes, and knowledge-based mistakes, can be helpful when analyzing the root causes of bugs and the context of their genesis.

3. Foundations for a theory of software engineering

A unified predictive theory of software engineering [21] (Gregor's Category IV theory [9]) is well beyond the scope of this paper. This section presents some foundational explanatory ideas in the form of an ontology of definitions and propositions (Gregor's Category II theory [9]). Even though many of the details may be familiar in isolation, the overall structure is instructive.

Let us begin by asking, "What is a general definition of software engineering?" Shaw [34, 35] defines "engineering," in general, as the following.

> Creating cost-effective solutions to practical problems by applying scientific knowledge to building things in the service of mankind.

The things of interest in software engineering are software systems.

Definition 1 (Software)
A software system is a set of executable instructions and data for computers.

The execution of the software provides value [20, p. 15], namely, "solutions to practical problems" and "service of mankind." The set of instructions and data may be very large and distributed, and the computer system may consist of many widely distributed components, such as processing, data storage, and communication capabilities. Multiple software products may run concurrently on particular computers, may interact, and may form larger systems. Applying Shaw's definition [34] to software we arrive at a broad definition of software engineering.

Definition 2 (Software engineering)
Software engineering is creating cost-effective solutions to practical problems by applying scientific knowledge to building software in the service of mankind.

Producing software is a necessary part of an activity called “software engineering.”

3.1 Design artifacts are central

Shaw’s definition of “engineering” [34] above emphasizes that building things is the way that engineering serves society. Design is a critical prerequisite to implementing things, because design defines the thing that solves a practical problem. This section addresses the question, “What is software design?”

Definition 3 (Design (verb) [41, p. 116])
To design is to construct artifacts that represent a product.

Definition 4 (Design artifact)
A design artifact is a representation of a product.

The above two definitions state the same concept for “design,” the former as a verb and the latter as a noun. We adopt Visser’s definition of the verb “design” [41, p. 116]. If a design remains only a mental idea, it cannot be reliably communicated to others, such as to those who will make the product. In our context, the product of interest is a software system. Throughout this paper we use the word “design” where the product is software.

An example of a design artifact is a Software Design Description document which is defined by IEEE Standards as “a representation of a software system created to facilitate analysis, planning, implementation, and decision making” [16]. Artifacts range from documentation to wikis to source code files, all of which present views of the desired product.

We agree with Exman [8] who posits that static code is not software. We consider static code to be a representation of the executable software.

How are design artifacts created? What are elemental processes that result in such artifacts? We suggest that design decisions are key to answering these questions.

Definition 5 (Design decision)
A design decision creates design elements, defines attributes, or determines their configuration (relationships).

ISO Standard 24765 [17] defines "software design" in terms of "architecture, components, interfaces, and other characteristics." Definition 5 corresponds closely to the standard's definition. "Components" is included by "elements." "Other characteristics" is a synonym for "attributes." "Architecture" and "interfaces" correspond to "configuration." Lastly, the standard's phrase "conceiving, inventing, or contriving," corresponds to the word "creates." Creation of a design means the designer decides that the element, attribute, or relationship should be included in the product. "Creating" is essentially a sequence of decisions to include or not, and to draw relationships, or not.

Proposition 1 (Design artifacts express design decisions)
Software-design artifacts are expressions of sets of design decisions.

Rationale: As a representation of a product (Definition 4), a design artifact expresses characteristics of the product. Design decisions determine what characteristics are included in the product (Definition 5). Because the number of elements, attributes, and relationships of real-world software systems is huge, no one artifact is an adequate comprehensible representation. Thus, designers use abstraction to express subsets of the totality of design decisions, including some decisions and omitting others. ■

In practice, software engineers produce many design artifacts, representations of the system, because no one artifact can effectively communicate the huge number of design decisions that go into a real-world software system.

Proposition 2 (Information)
Design decisions are composed of information.

Rationale: A design decision (Definition 5) is, in essence, a choice among alternatives. One may choose among candidate elements, attributes, or relationships. One may choose that a design element is included in the design, or not. One may choose that a relationship of a particular type is included, or not. One may choose to connect this element to that, or not. Choice is fundamental to the concept of information. Software-design decisions expressed in artifacts (Proposition 1) are messages to various stakeholders of the software-development enterprise, some content is expected, some content is surprising, and some in between. If one expects the content of a message, then its arrival is not very informative, but if a message's content is a surprise, then its information content is high. ■

Information theory is a branch of mathematics that provides a way to measure the amount of information in messages in a probabilistic context [4]. Even though human discourse is not easily modeled by probability distributions, the ordinary concept of information is defined by choices in the context of expectations, similar to the information theoretic definition.

Engineering is performed by people, and software engineering, in particular, is a quintessentially human activity that deals with abstract concepts. Let us consider certain human characteristics that are especially relevant here.

Proposition 3 (Bounded rationality [36])
People have bounded rationality.

Rationale: We adopt Simon's concept of "bounded rationality" [36]. "Applying scientific knowledge" (Definition 2) means engineers must have intellectual reasoning skills, that is, rationality. Cognitive psychology research has shown that people have limited rational abilities [36], due to limited short-term memory, limited mental computational capacity, error-prone reasoning skills [14], and other limitations that result in slips, lapses, rule-based mistakes, and knowledge-based mistakes [33]. The term "bounded rationality" summarizes such human limitations [36]. ■

Even though ideal design decisions could be made using valid logic and probability theory, we do not assume that software engineers always achieve perfection. Design decisions may be cognitively difficult [14, p. 45] due to large numbers of alternatives, the degree of uncertainty, the number or difficulty of tradeoffs, the severity of loss if a choice is wrong, the cost of searching for alternatives, and the cost of deciding. Design decisions may also be influenced by non-rational factors [14, p. 45], such as values that may be supported or threatened, the intensity of emotions evoked, and time pressure. Facilitating comprehension in the face of bounded rationality is the reason that software engineers build such a wide variety of design artifacts.

Proposition 4 (Decisions under uncertainty)
At the time a design decision is made, one cannot be certain that it will be a wise decision, or that it will have good consequences.

Rationale: A complete design is the accumulation of many interdependent design decisions over time (Definition 4). Because of bounded rationality (Proposition 3), an individual person cannot comprehend all of the interrelated decisions of a real-world scale software system. Consequently, at the time of a particular decision, there are many things about the product that are as yet undetermined and unknown to the designer. Due to interdependence, all the consequences of a particular design decision in terms of software behavior cannot be anticipated until the complete product is tested and used. Therefore, one cannot be certain that the consequences will be good, or that the decision will be wise. ■

Judgment and decision making are routine activities in software development. Poor judgment and decisions often result in bugs. Software-design decisions are almost always made under uncertainty. Missing information, limited mental computational capacity, uncertain and non-stationary decision environments, and multifactor tradeoffs imply that probabilistic reasoning about risk should be used routinely in design decision making. Hastie and Dawes [14] point out that people do not consistently perform probabilistic reasoning correctly, even with substantial training.

Ralph [31] reviews five theories from the social sciences that can help explain human behavior at various levels of aggregation from individuals to projects. He suggests they could be incorporated into a general theory of software engineering. Future work will further apply more specific behavioral theories at various levels of aggregation.

In summary, design artifacts are central to the act of software design, and decisions are the elements from which artifacts are constructed. Because decisions are made by people with bounded rationality and uncertain information, there is always the risk of unwise decisions and unforeseen consequences.

3.2 Projects develop software

In this paper, we focus on a software-development project, namely, an aggregation of work in the context of an organization. Similarly, an evolution consists of a series of projects, namely, each delivery of a new version of the software. The organization might have a corporate identity or be just a collaboration, such as open-source projects. Let us examine some characteristics of projects.

Definition 6 (Product specification [41, p. 116])
A product specification is a final design artifact of the product with such precise detail that it can be directly implemented.

We adopt this definition by Visser [41, p. 116]. When producing a physical product, the product specification conveys enough detail to manufacture the product. Similarly, when producing an intellectual product, namely software, the product specification must convey all the detail necessary to manufacture the software's executable instructions, namely, source code, scripts, data, etc.

Proposition 5 (Goal)
Producing a final software-design artifact is a primary goal of software developers.

Rationale: In software engineering, the collection of source code, scripts, etc., corresponds to Visser's "final design artifact," or "product specification" [41, p. 116]. The mechanical process of compilation and installation are analogous to Visser's "implementation" [41, p. 116]. Similarly, manual installation procedures are included in "implementation" as well. For most software projects, source code and similar scripts are the final representation of the product created by the developers. Consequently, producing such artifacts is the goal of software developers, so the delivery tasks can proceed. Delivery tasks, such as packaging, distribution, and installation are usually a minor part of the effort. Moreover, packaging, distribution, and installation are often performed by people other than software developers. ■

Various software-development life-cycle models order tasks in many different ways. The SEMAT Way of Working alpha [20] is useful for specifying a particular life-cycle model and its methods, because it defines abstract tasks. A project plan can apply a particular life-cycle model to a specific project, because it defines concrete tasks, often based on selected Ways of Working.

A plan is not a design artifact, because it is not a representation of the project's product. However, a plan is necessary for efficient team performance. Planning includes choosing ways of working (e.g., methodologies), deciding what tasks are needed, assigning people to perform tasks, and determining when tasks should be performed. Because many tasks consist of building design artifacts, project plans take into account earlier design decisions that defined the existence of design elements represented by planned artifacts. Proper planning is necessary to accomplish the parallel design decisions that are mandated by "solutions to practical problems" (Definition 2). Planning is a traditional project-management function, and is a major factor in a project's success, or not.

Proposition 6 (Constraints)
Early design decisions constrain later design decisions.

Rationale: Given that a project's plan is guided by a chosen life-cycle model, some design artifacts must be produced before others. Later decisions must be consistent with and compatible with earlier decisions, and therefore, the later decisions are constrained by the earlier relevant decisions. ■

The complex dependencies among design decisions is one reason that project planning is difficult. Models, such as the waterfall life-cycle model, the spiral life-cycle model, incremental development, the Unified process, Scrum, and others serve to guide the order of tasks and events during development. Techniques, such as PERT and Critical Path Method, can analyze the dependencies and their impact on a project's schedule.

Definition 7 (Requirement)
A software requirement is a design decision regarding functional or holistic attributes of the software product at a level of abstraction meaningful to relevant stakeholders.

In the terminology of software engineers, requirements are distinct from design, because the decisions are often made by people who are not software engineers. A "practical problem in the service of mankind" [34] (a SEMAT [19] "opportunity") corresponds to Visser's requirements [41, p. 116], which is not stated as a representation of a product. In contrast, a set of requirements, as commonly defined by software engineers, portrays the desired product in stakeholder terms.

Proposition 7 (Design during the life cycle)
Design occurs during the software-development life cycle from requirements definition through coding.

Rationale: Because we adopt Visser's definition of design (Definition 3), rather than the more narrow software-engineering definition, requirements definition is included among design activities. A set of requirements is a representation of a system (Definition 4). We also include writing code as a design activity, because code is also a representation of the executable system (Definition 4), and when finished, it fits Visser's definition of a product specification (Definition 6). ■

On the way to a final design artifact, designers produce many intermediate artifacts that also represent the product. Visser's definition of design [41, p. 116] is broad enough to span the software-development life cycle, because it encompasses a wide variety of artifacts at various levels of abstraction and precision.

In Visser's terminology (Definition 3), software requirements form another artifact that represents the product, and thus, to define requirements is to design. Software architectures depict relationships among large software subsystems and modules. Detailed designs specify algorithms, data structures, and interactions of small software elements, such as classes and methods. Coding translates algorithms and data structures into a form that is both human readable and machine readable, ready for compilation. "Implementation" in Visser's terminology, is "compilation and delivery" in software-engineering terminology. Delivery consists of final compilation, packaging, distribution, and installation. Requirements through code are representations of the final executable software, the product, at various levels of abstraction, and thus, are design artifacts. The medium of a representation is immaterial; paper documents, wikis, Web pages, and source code files can all be design artifacts.

Proposition 8 (Intellectual effort)
Making decisions costs intellectual effort.

Rationale: The act of making a design decision (Definition 5) is surrounded by a wide variety of cognitive activities such as the following [41]: generating ideas, transforming ideas, acquiring information (Proposition 2), and analyzing and evaluating consequences (Proposition 4). Each of these requires intellectual effort to accomplish, often by many people. ■

This proposition implies that making design decisions is not cost free and it does not happen instantly.

Proposition 9 (Result of work)
Creation or modification of artifacts is the result of software-engineering work.

Rationale: Design decisions require effort (Proposition 8), and creating or modifying representations of those design decisions require additional effort. The phrase "software-engineering work" summarizes this cumulative effort. Similarly, the SEMAT Kernel [19] states, "Work updates and changes a software system" where "software system" includes all the design artifacts. ■

The consequences of this proposition is that real-world software development costs the work of a team over a considerable period of time. Because software developers are typically paid salaries for their work, a project usually has a financial cost. Because the process takes time, the project plan has a schedule.

3.3 Supporting activities are necessary

A design decision is not just a moment in time when a choice is made. It typically takes considerable effort to marshal necessary inputs to the final moment of choosing. Consequently, a wide variety of software-engineering activities support the moment of choosing.

Definition 8 (Tentative or definite decisions)
Design decisions may be tentative or definite.

If a design decision is easily subject to change, we call it "tentative." If a design decision is intended for inclusion in the product specification (Definition 6), we call it "definite."

Proposition 10 (Analysis and evaluation)
Tentative decisions are analyzed and evaluated, often using tools.

Rationale: Because the consequences of tentative design decisions are uncertain (Proposition 4), further information is needed to determine if they are wise. Analysis and evaluation provide that necessary information. Due to the huge scope of relevant information for real-world software systems, automated tools are often necessary to produce the desired information. ■

Analysis and evaluation of tentative designs often consume a major part of a project's effort, for example, testing and quality assurance. The results of analysis and evaluation activities are important contributors to finally arriving at a product specification of definite decisions.

Tools are often necessary to acquire the results of analysis and evaluation. There is an extensive software-engineering literature and commercial marketplace that presents algorithms and sophisticated tools for producing valuable inputs to analysis and evaluation processes.

Prototyping is an example of analysis. A prototype implements some, but not all, aspects of the product, so that an evaluation of the prototype can be input to subsequent design decisions. For example, prototype user interfaces are commonly used to elicit opinions regarding tentative requirements.

Rework becomes necessary when a tentative design decision is found to have undesirable consequences, such as a bug. Rework replaces the initial decision with a better tentative decision, which in turn, is then analyzed and evaluated.

Proposition 11 (Transition from tentative to definite)
To become incorporated into the final product specification, a tentative design decision must transition to a definite design decision. This transition is itself a design decision as well.

Rationale: The product specification (Definition 6) is by Definition 8 composed of definite design decisions. Due to associated uncertainty (Proposition 4), most design decisions are initially tentative, until sufficient analysis and evaluation (Proposition 10) has been done. The results of analysis and evaluation are inputs to the decision to transition the tentative decision to a definite decision. ■

For example, software testing is one of the most common forms of analysis. A tentative design artifact (source code) is compiled and tested. The test results are input to decisions on whether tentative design decisions embodied by the source code should become definite, answering the question, "Is the source code satisfactory?"

3.4 The core of software engineering is design

We consider the core activities of software engineering to occur during requirements definition through coding (Proposition 7). The following proposition summarizes the key point of our theory.

Proposition 12 (The core of software engineering is design)
The core of software engineering is people making design artifacts under uncertainty, based on scientific knowledge, that enables production of executable software that solves practical problems in a cost-effective way in the service of mankind.

Rationale: Executable software (Definition 1) is the end result of software engineering. A software product is built by expressing in artifacts the many decisions that determine the executable instructions and data in the product (Proposition 5). Such design decisions are necessary for software to be created. Thus, the core of software engineering is making design decisions expressed in artifacts. At the time such design decisions are made, it is often uncertain (Proposition 4) whether the resulting software will solve the practical problem at hand in a cost-effective manner (Definition 2). ■

Perry [27, 28] presents a formal calculus for manipulating theories, models, and evaluations. When applied to a software product, theories corresponds to problems (SEMAT opportunities); models correspond to software design decisions; and evaluations correspond to analysis and evaluation of design decisions.

Adopting Visser's broad definition [41, p. 116] of the verb "to design" (Definition 3), we consider the core of software engineering to be decisions which are expressed in artifacts that represent the executable software product. Supporting artifacts, such as test cases and project plans, are not considered "design artifacts," but they are necessary for cost-effective development.

3.5 Ontology structure

Fig. 2 shows the structure of our definitions and propositions, and for convenient reference, Table 1 lists our definitions and propositions presented above.

We adopt Visser's definition of design [41, p. 116] (Definition 3), and we adopt Simon's concept of bounded rationality [36] (Proposition 3), both of which are based on cognitive science. We argue that design artifacts are constructed throughout the development life cycle, from requirements definition through coding (Definition 6, Definition 7, Proposition 7). Design artifacts are composed of design decisions made under uncertainty (Definition 5, Proposition 1, Proposition 4). Design decisions typically are initially tentative and become definite after analysis and evaluation (Definition 8, Proposition 10, Proposition 11). We conclude that the core of software engineering is people making design artifacts (Proposition 12).

4. Discussion

An explanatory theory is validated by marshaling evidence that the theory gives useful insights into real world phenomena. In this section, we consider how the proposed ontology gives insight into some issues in software-engineering practice and management.

Table 2 lists some common wisdom selected from those collected by Endres and Rombach [6]. They limited their selection of so-called "laws" from the literature to those that are "supported by direct experimentation, documented case studies, or by the collective

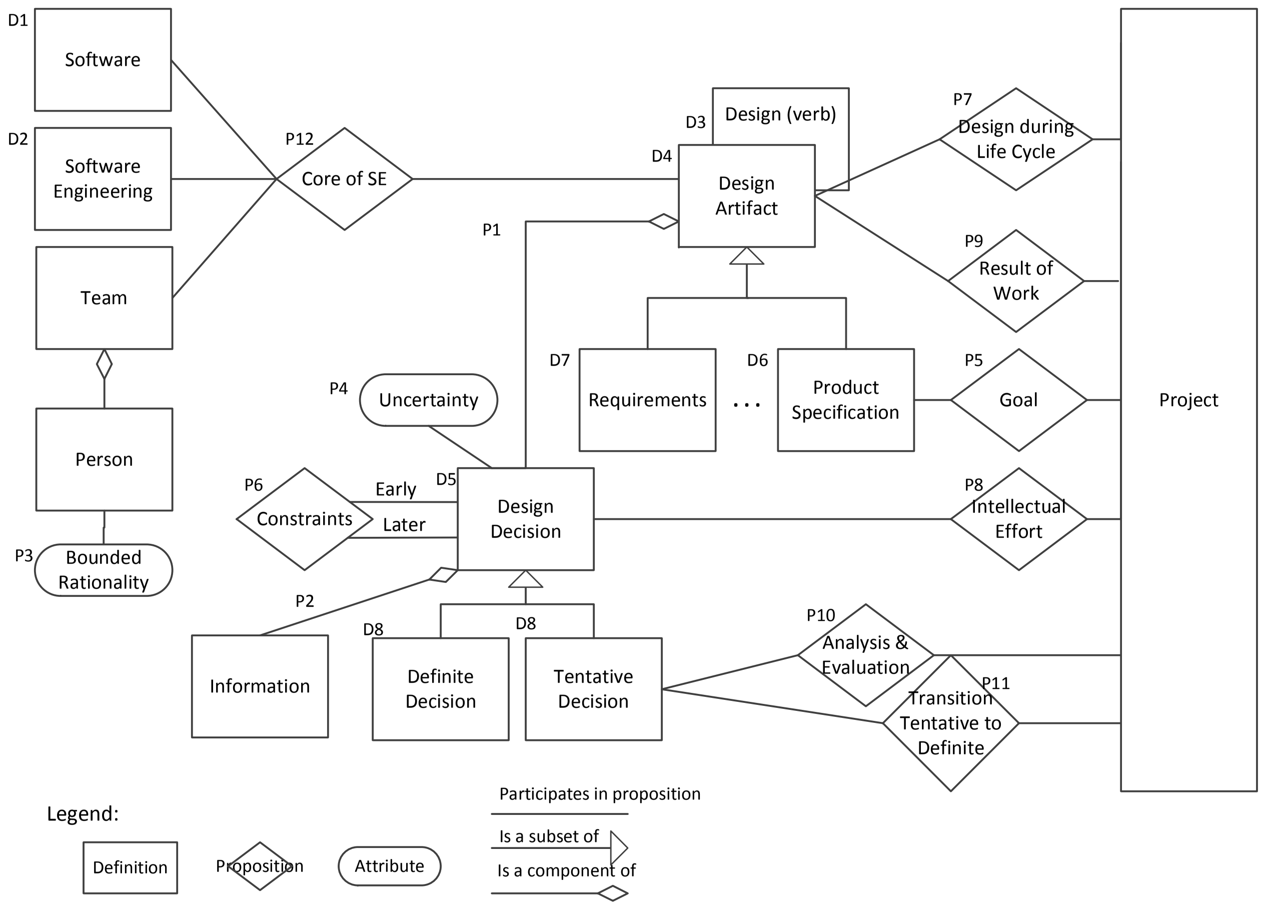

Fig. 2

Ontology for software engineering theory

Table 1

Definitions and Propositions

ID[1]	Title	Definition/Proposition
D1	Software	A software system is a set of executable instructions and data for computers.
D2	Software engineering	Software engineering is creating cost-effective solutions to practical problems by applying scientific knowledge to building software in the service of mankind.
D3	Design (verb) [41, p. 116]	To design is to construct artifacts that represent a product.
D4	Design artifact	A design artifact is a representation of a product.
D5	Design decision	A design decision creates design elements, defines attributes, or determines their configuration (relationships).
P1	Design artifacts express design decisions	Software-design artifacts are expressions of sets of design decisions.
P2	Information	Design decisions are composed of information.
P3	Bounded rationality	People have bounded rationality.
P4	Decisions under uncertainty	At the time a design decision is made, one cannot be certain that it will be a wise decision, or that it will have good consequences.
D6	Product specification [41, p. 116]	A product specification is a final design artifact of the product with such precise detail that it can be directly implemented.
P5	Goal	Producing a final software-design artifact is a primary goal of software developers.
P6	Constraints	Early design decisions constrain later design decisions.
D7	Requirement	A software requirement is a design decision regarding functional or holistic attributes of the software product, at a level of abstraction meaningful to relevant stakeholders.
P7	Design during the life cycle	Design occurs during the software-development life cycle from requirements definition through coding.

Table 1

(Continued.) Definitions and Propositions

ID[2]	Title	Proposition
P8	Intellectual effort	Making decisions costs intellectual effort.
P9	Result of work	Creation or modification of artifacts is the result of software-engineering work.
D8	Tentative or definite decisions	Design decisions may be tentative or definite.
P10	Analysis and evaluation	Tentative decisions are analyzed and evaluated, often using tools.
P11	Transition from tentative to definite	To become incorporated into the final product specification, a tentative design decision must transition to a definite design decision. This transition is itself a design decision as well.
P12	The core of software engineering	The core of software engineering is people making design artifacts under uncertainty, based on scientific knowledge, that enables production of executable software that solves practical problems in a cost-effective way in the service of mankind.

experience of practitioners" [6, p. xv]. Let us consider how this common wisdom is elucidated by the ontology and its emphasis on design decisions and bounded rationality.

4.1 Understanding software-engineering practice

Bounded rationality (Proposition 3) limits the amount of information (Proposition 2) and the degree of complexity that a software engineer can reliably comprehend and mentally manipulate. Accordingly, common wisdom asserts that large size and high complexity increase the likelihood of bugs and other defects. Large size results from the aggregation of many design decisions. Complexity is a result of the interactions of many design decisions. Several of the "laws" in Table 2 address problems that arise from large complex structures.

The Gestalt Laws from cognitive science [6, p. 225], as summarized by Endres and Rombach, state, "Humans tend to structure what they see to form cohesive patterns." The Gestalt Laws address a particular aspect of bounded rationality, namely, perception of patterns (Proposition 3). People seek to form patterns that fit within their capability for comprehension. Cohesive patterns, in particular, are easier to comprehend, because regular structures are not as complex as random patterns.

Simon's Law [6, p. 40] illustrates the importance of structuring information, stating, "Hierarchical structures reduce complexity." Arbitrary patterns of connections require more information to describe than regular patterns. Consequently, regular structures, such as hierarchy, are less complex than arbitrary structures of similar size.

Table 2

Common wisdom regarding design [6]

Title	Quote	Page in [6]
Gestalt Laws	Humans tend to structure what they see to form cohesive patterns.	224
Simon's Law	Hierarchical structures reduce complexity.	40
Parnas' Law	Only what is hidden can be changed without risk.	45
Lehman's Second Law	An evolving system increases its complexity unless work is done to reduce it.	165
McIlroy's Law	Software reuse reduces cycle time and increases productivity and quality.	77
Basili-Boehm COTS Hypothesis	COTS-based software does not eliminate the key development risks.	85
Davis' Law	The value of a model depends on the view taken, but none is best for all purposes.	22
Corbató's Law	Productivity and reliability depend on the length of a program's text, independent of language level used.	72
Glass' Law	Requirement deficiencies are the prime source of project failures.	16
Boehm's First Law	Errors are most frequent during the requirements and design activities and are the more expensive the later they are removed.	17
Boehm's Second Law	Prototyping (significantly) reduces requirement and design errors, especially for user interfaces.	19
Boehm's Hypothesis	Project risks can be resolved or mitigated by addressing them early.	201
DeMarco-Glass Law	Most cost estimates tend to be too low.	194

Bounded rationality (Proposition 3) is also a motivation for the principle of information hiding. Parnas' Law [6, p. 45] states, "Only what is hidden can be changed without risk." When design decisions in one module are hidden from other modules, there are no interactions for the programmer to understand. This simplifies the job of making changes to the hidden design decisions, and thus, reduces the risk of mistakes.

Complexity growth is a natural result of ongoing system evolution from one release to the next, as stated by Lehman's Second Law [6, p. 165], "An evolving system increases its complexity unless work is done to reduce it." This is based on the observation that evolution of systems tends to keep adding design decisions (information and interactions) unless there is a concerted effort to refactor. Thus, the scope of what a designer must comprehend keeps growing until it is beyond the bounds of rationality, precipitating a crisis.

The cost of making decisions (Proposition 8) and the uncertainty inherent in design decisions (Proposition 4) motivate software reuse. McIlroy's Law [6, p. 77] recommends reusing software. Choosing an off-the-shelf component (COTS) is essentially the adoption of all the constituent decisions that went into the design of the component without itemizing what they were. This implies that all the internal structures of the reused component are accepted as definite decisions in one decision (Proposition 11), dramatically reducing the cost of making decisions. Reuse also simplifies the representation of software architectures, because a COTS component's internals can be ignored in subsequent views of the remaining design decisions, alleviating the constraints of bounded rationality (Proposition 3).

In contrast, the Basili-Boehm Hypothesis [6, p.85] asserts that adoption of COTS components is not a panacea, even though reuse might help reduce complexity. All of the COTS component's internal design decisions remain a risk, because they are only superficially analyzed and evaluated (Proposition 10). The wisdom of the decision to adopt a COTS component is uncertain until the component is tested and used (Proposition 4).

Software engineers create many kinds of artifacts. IEEE Standards identify a host of software-engineering documents that a project may create. Some of these, such as a Software Requirements Specification, a Software Architecture Description, and a Software Design Description, include models of the product software, and thus, we consider them design artifacts (Definition 4).

Davis' Law [6, p. 22] states, "The value of a model depends on the view taken, but none is best for all purposes." Artifacts embody various views of a model. Each view is a subset of the design decisions that go into the product (Proposition 1). Views of a model purposely leave out information to keep the task of comprehension within human limitations (Proposition 3). Accordingly, a common strategy in software engineering is to limit the number of features in a view of a model, such as a diagram, that need to be comprehended simultaneously to achieve a particular purpose.

Corbató's Law [6, p. 72] addresses the tasks of creating, comprehending, and analyzing source code. The longer the source code text is, the more information it contains (Proposition 2) which directly impacts a person's ability to perform the tasks reliably, due to bounded rationality (Proposition 3). Corbató's point is that the choice of programming-language level is a minor contributor to the complexity of the tasks compared to the length of the text.

According to our broad definition of design (Definition 3), software requirements are essentially design decisions (Definition 7). Software requirements are important, because "early design decisions constrain later design decisions" (Proposition 6). Glass' Law [6, p. 16] draws attention to the importance of getting requirements right, saying "Requirement deficiencies are the prime source of project failures." Similarly, Boehm's First Law [6, p. 17] focuses on how errors in requirements are costly to fix. This is because later design decisions that depend on earlier erroneous requirements are subject to rework (Proposition 6). Boehm's Second Law [6, p. 19] offers an approach for early analysis of requirements through prototyping (Proposition 10). Boehm's Hypothesis [6, p. 201] sums up the implications of Proposition 6 saying, "Project risks can be resolved or mitigated by addressing them early."

4.2 Guiding management of software development

Our ontology emphasizes that software engineering is a people-intensive discipline, and that the characteristics of people drive many phenomena of software development.

Why are multi-person projects necessary? "Cost-effective solutions to practical problems" (Definition 2) almost always entail too many design decisions and require too many diverse skills for one person. The obvious solution is for management to form teams [20, p. 15] to work on design decisions in parallel, so that the executable product has a timely delivery. Even though the extent of the design may be large due to replication, management can strive to keep the number of interdependent design decisions that any one person is expected to make small (Proposition 3). Thus, methods for building large complex systems often utilize abstraction of design, regularity of design structures, and partitioning of design decisions for multiple designers.

Making design decisions is fundamental to progress on a project (Proposition 12). If the decision-making process of an organization is dysfunctional, then progress to the product will be difficult or even impossible. For example, a system architect might insist on making all final design decisions, creating a bottleneck, or perhaps, company politics plays out in committee meetings that are supposed to make design decisions, frustrating real progress. Decision-making processes deserve careful management attention so that wise timely decisions are normal practice (Proposition 8).

Why do we have cost overruns and late projects? The DeMarco-Glass Law [6, p. 194] states, "Most cost estimates tend to be too low." Planning generates expected costs and due dates according to dependencies among design issues (Proposition 6). Uncertainty over the amount of work (Proposition 4) entails uncertain costs, and the effort needed to arrive at wise design decisions is uncertain (Proposition 8). Thus, when planning assumptions turn out to not come true, the impact is almost always more work, not less, and more cost and more time than was expected. Reducing uncertainty as early as possible will help make planning assumptions more realistic (Boehm's Hypothesis [6, p. 201]).

4.3 Contexts

A theory of software engineering must also consider the multiple contexts of a project. These include the technology context, the organizational context, and the business context. Events in a project's context can significantly affect the work. Smolander and Päivärinta [38] emphasize the importance of a project's organizational context. Organizations are adaptive, learning entities where human behavior is not strictly governed by technical rationality. Erbas and Erbas [7] show the applicability of microeconomic theory to the business context of software development. Future work could further incorporate such contextual concepts into a general theory of software engineering.

The above discussion illustrates how the proposed ontology gives principled insight into disparate "laws" from a variety of software-engineering contexts. The proposed ontology emphasizes the importance of design artifacts and the limitations of human rationality. We claim that design occurs from requirements definition through coding, and that supporting activities are necessary. The above also illustrates how our theory provides insight for managers of software development. Bounded rationality (Proposition 3), decision-making (Proposition 12), uncertainty (Proposition 4), and design dependencies (Proposition 6) drive many management plans and actions.

5. Conclusions

A general theory of software engineering is needed to provide a foundation for the field. Although software engineering has a variety of microtheories, a unifying theory of software engineering would give coherence to the patterns of human activity that we call "software engineering" [21].

The contribution of this paper is a theory consisting of an ontology of definitions and propositions to help explain software-engineering phenomena. Our theory falls in Gregor's Category II. Explanation [9]. We have adopted Visser's definition [41, p. 116] of design (Definition 3) and Simon's concept of human bounded rationality [36] (Proposition 3), both of which have roots in cognitive science. We conclude that the core of software engineering is people making design artifacts (Proposition 12). However, many supporting activities are also necessary for cost-effective development of software.

We apply the ontology to various "laws" collected by Endres and Rombach [6] to illustrate its relationship to common wisdom in the field. The ontology also provides insights for project managers in certain areas.

A variety of research questions remain open. For example, is this theory scalable down to the individual and up to large projects? Is it scalable even to communities of actors not confined by projects? What kinds of analysis and evaluation are necessary and cost-effective? What is a complementary theory for management of software projects? Lagerström and Ekstedt [23] summarize a theory of software implemented as a model. The theory is especially applicable to business enterprise information-technology contexts. Future work on our proposed theory might model projects using the Unified Modeling Language (UML), the Object Constraint Language (OCL), and the Predictive, Probabilistic Architecture Modeling Framework (P2AMF) as they did. Future work also includes empirically validating the propositions presented here. Proposing additional ontology elements to describe other areas of software engineering more precisely, and proposing predictive propositions is also future work.

Acknowledgments

The author thanks G. Bradshaw and the Empirical Software Engineering Research Group at Mississippi State University for helpful discussions.

References

[1] S. Adolph and P. Kruchten, "Generating a Useful Theory of Software Engineering," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 47–50.

[2] D. Batory, "Why (Meta-)Theories of Automated Software Design Are Essential: A Personal Perspective," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 19–22.

[3] F. P. Brooks, Jr., *The Design of Design: Essays from a Computer Scientist*, Addison Wesley, Upper Saddle River, New Jersey, 2010.

[4] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, 1991.

[5] P. J. Denning, "Great Principles of Computing," *Communications of the ACM*, vol. 46, no. 11, Nov. 2003, pp. 15–20.

[6] A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*, Addison Wesley, Harlow, England, 2003.

[7] C. Erbas and B. C. Erbas, "On a Theory of Software Engineering: A Proposal Based on Transaction Cost Economics," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 15–18.

[8] I. Exman, "Speeding-Up Software Engineering's Escape from Its Pre-paradigmatic Stage," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 1–4.

[9] S. Gregor, "The Nature of Theory in Information Systems," *MIS Quarterly*, vol. 30, no. 3, Sept. 2006, pp. 611–642.

[10] S. Gregor and A. R. Hevner, "Positioning and Presenting Design Science Research for Maximum Impact," *MIS Quarterly*, vol. 37, no. 2, June 2013, pp. 337–355.

[11] S. Gregor and D. Jones, "The Anatomy of a Design Theory," *Journal of the Association for Information Systems*, vol. 8, no. 5, 2007, Article 2.

[12] W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software*, vol. 23, no. 1, Jan. 2006, pp. 51–60.

[13] J. E. Hannay, D. I. K. Sjøberg, and T. Dybå, "A Systematic Review of Theory Use in Software Engineering Experiments," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, Feb. 2007, pp. 87–107.

[14] R. Hastie and R. M. Dawes, *Rational Choice in an Uncertain World: The Psychology of Judgment and Decision Making*, Sage Publications, Thousand Oaks, California, 2001.

[15] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly*, vol. 28, no. 1, Mar. 2004, pp. 75–105.

[16] *IEEE Std.1061-1998, IEEE Recommended Practice for Software Design Descriptions*, IEEE, Piscataway, New Jersey, 1998.

[17] *ISO/IEC/IEEE Std.24765, International Standard: Systems and Software Engineering — Vocabulary*, International Organization for Standardization, Geneva, Switzerland, 2010.

[18] *ISO/IEC TR 19759-2005, Guide to Software Engineering Body of Knowledge (SWEBOK)*, Tech. Rep., International Organization for Standardization, Geneva, Switzerland, Feb. 2005.

[19] I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman, "The Essence of Software Engineering: The SEMAT Kernel," *Communications of the ACM*, vol. 55, no. 12, Dec. 2012, pp. 42–49.

[20] I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman, *The Essence of Software Engineering: Applying the SEMAT Kernel*, Addison Wesley, Upper Saddle River, New Jersey, 2013.

[21] P. Johnson and M. Ekstedt, "In Search of a Unified Theory of Software Engineering," *Proceedings: 2nd International Conference on Software Engineering Advances*, Cap Esterel, France, 2007, IEEE Computer Society.

[22] P. Johnson, M. Ekstedt, and I. Jacobson, "Where's the Theory for Software Engineering," *IEEE Software*, vol. 29, no. 5, Sept. 2012, pp. 94–96.

[23] R. Lagerström and M. Ekstedt, "Extending a General Theory of Software to Engineering," *Proceedings: 3rd SEMAT Workshop on a General Theory of Software Engineering*, Hyderabad, India, 2014, ACM, pp. 36–39, Position paper.

[24] K. Lemon, E. B. Allen, J. C. Carver, and G. L. Bradshaw, "An Empirical Study of the Effects of Gestalt Principles on Diagram Understandability," *Proceedings: The First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, Sept. 2007, IEEE Computer Society, pp. 156–165.

[25] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek, "How Software Designers Interact with Sketches at the Whiteboard," *IEEE Transactions on Software Engineering*, vol. 41, no. 2, Feb. 2015, pp. 135–156.

[26] G. A. Miller, "The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review*, vol. 63, no. 2, Mar. 1956, pp. 81–97.

[27] D. E. Perry, "A Unifying Theoretical Foundation for Software Engineering," *Proceedings: 20th International Conference on Software Engineering and Data Engineering*, Las Vegas, Nevada, June 2011, International Society for Computers and their Applications, pp. 1–6.

[28] D. E. Perry, "A Theoretical Foundation for Software Engineering: A Model Calculus," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 39–46.

[29] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420–432.

[30] V. Rajlich et al., eds., *Worshop Notes: Workshop on Program Comprehension*, Orlando, Florida, Nov. 1992. IEEE Computer Society, Part of the Conference on Software Maintenance 1992. Notes distributed to workshop attendees. Predecessor of the International Conference on Program Comprehension.

[31] P. Ralph, "Possible Core Theories for Software Engineering," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 35–38.

[32] P. Ralph, I. Exman, P.-W. Ng, P. Johnson, M. Goedicke, A. T. Kocataş, and K. L. Yan, "How to Develop a General Theory of Software Engineering: Report on the GTSE 2014 Workshop," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 6, Nov. 2014, pp. 23–25.

[33] J. Reason, *Human Error*, Cambridge University Press, Cambridge, 1990.

[34] M. Shaw, "Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 7, no. 6, Nov. 1990, pp. 15–24.

[35] M. Shaw, "Continuing Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 26, no. 6, Nov. 2009, pp. 64–67.

[36] H. A. Simon, *Models of Bounded Rationality*, MIT Press, Cambridge, Massachusetts, 1982.

[37] H. A. Simon, *The Sciences of the Artificial*, 3rd edition, MIT Press, Cambridge, Massachusetts, 1996, Originally published in 1969.

[38] K. Smolander and T. Päivärinta, "Forming Theories of Practices for Software Engineering," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 27–34.

[39] E. Soloway and S. Iyengar, eds., *Empirical Studies of Programmers: Papers Presented at the First Workshop on Empirical Studies of Programmers*, Washington, DC USA, June 1986. Ablex Publishing.

[40] K.-J. Stol and B. Fitzgerald, "Uncovering Theories in Software Engineering," *Proceedings: 2nd SEMAT Workshop on a General Theory of Software Engineering*, San Francisco, California, May 2013, SEMAT, pp. 5–14.

[41] W. Visser, *The Cognitive Artifacts of Designing*, Lawrence Erlbaum Associates, Mahwah, New Jersey, 2006.

[42] A. von Mayrhauser and A. M. Vans, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, June 1996, pp. 424–437.

[43] K. D. Wilson, *The Effects of Gestalt Principles on Diagram Comprehension: An Empirical Approach*, Ph.D. dissertation, Mississippi State University, 2012, Advised by Edward B. Allen.