

Title	Teaching theorizing in software engineering research
Authors	Stol, Klaas-Jan
Publication date	2024-12-25
Original Citation	Stol, K. J. (2024) 'Teaching theorizing in software engineering research', in Mendez, D., Avgeriou, P., Kalinowski, M., Ali, N.B. (eds) Handbook on Teaching Empirical Software Engineering. Springer, Cham, pp. 31-69. https://doi.org/10.1007/978-3-031-71769-7_3
Type of publication	Book chapter;book-chapter
Link to publisher's version	10.1007/978-3-031-71769-7_3
Rights	© 2024, the Author, under exclusive license to Springer Nature Switzerland AG. This version of the chapter has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-71769-7_3
Download date	2026-10-13 05:16:07
Item downloaded from	https://hdl.handle.net/10468/17344

Teaching Theorizing in Software Engineering Research

Klaas-Jan Stol

Abstract This chapter seeks to support software engineering (SE) researchers and educators in teaching the importance of theory as well as the theorizing process. Drawing on insights from other fields, the chapter presents 12 intermediate products of theorizing and what they mean in an SE context. These intermediate products serve different roles: some are theory products to frame research studies, some are theory generators, and others are components of a theory. Whereas the SE domain doesn't have many theories of its own, these intermediate products of theorizing can be found widely. The chapter aims to help readers to recognize these intermediate products, their role, and how they can help in the theorizing process within SE research. To illustrate their utility, the chapter then applies the set of intermediate theorizing products to the software architecture research field. The chapter ends with a suggested structure for a 12-week course on theorizing in SE which can be readily adapted by educators.

1 Introduction

The ancient Greeks are widely acknowledged to have made major advances in the sciences [75]. Why might this be? Nisbett describes [75, p. 4]:

The Greeks, far more than their contemporaries, speculated about the nature of the world they found themselves in and created models of it. They constructed these models by categorizing objects and events and generating rules about them that were sufficiently precise for systematic description and explanation. [...] Whereas many great contemporary civilizations, as well as the earlier Mesopotamian and Egyptian and the later Mayan civilizations, made systematic observations in all scientific domains, only the Greeks attempted to explain their observations in terms of underlying principles.

Klaas-Jan Stol
Lero and University College Cork, Western Road, Cork, Ireland e-mail: k.stol@ucc.ie

The success of the ancient Greeks thus lay in recognizing the value in creating models, categories, concepts, rules, and principles to establish an understanding of the world. How does research today fare in terms of these activities? More recently than ancient Greek civilization, Suddaby [114] lamented that a common problem with Grounded Theory is “a failure to ‘lift’ data to a conceptual level,” and indeed, this is also a shortcoming within SE research [111]. Robson [89] argued that “*without theory the research may be easier and quicker, but the outcome will often be of little value.*”

The last statement should hit a nerve among SE researchers. The SE community produces thousands of research papers each year, many of which have gone through a tedious review process and have found a willing audience among fellow researchers, and yet their value appears to be of little relevance or interest to SE professionals [11, 122]. At the same time, a lack of attention for theory and conceptualization has been observed within the SE community [57, 109]. I argue that these two observations, namely (1) that much of SE research is not valued by practitioners, and (2) SE research lacks attention for theory and conceptualization, are correlated. Perhaps, as a community, we should learn a lesson from the ancient Greeks, and focus on explaining our observations beyond the immediate settings we study, and on identifying underlying principles of our findings. This chapter seeks to draw attention to increase the reader’s sensitivity to not only the need to theorize, but also to suggest a light-weight approach to get started on this endeavor. Or, with apologies to Larry Wall:¹ “Theory development is hard, Let’s go theorizing...”

The SE community has made major strides in improving researchers’ level of expertise in using a wide range of research methods. However, conducting an excellent case study or a carefully designed experiment does not, on its own, add to the body of knowledge within SE. Whereas the remaining chapters in this volume focus on teaching specific research methods, we also need to consider the role of theory to advance science. Perhaps the single-most important aspect of any study is its motivation, its grounding in prior work, and how this feeds back to the literature. Theorizing can help to address this, because it encourages to build on or challenge prior work, and theorizing effectively can suggest promising research directions. Theory development is a fundamental research strategy [110], but one that has not received much explicit attention. SE researchers simply aren’t trained to theorize, and as a result, the publication norms within the SE community are such that nobody appears to miss it, either.

1.1 The Need for Theorizing

Why do we need theorizing? There are several reasons. The first reason to engage in theorizing is that this distinguishes science apart from consulting or journalism. Presenting some findings that remain completely disconnected to other findings

¹ Perl creator Larry Wall’s 11th State of the Onion speech was entitled: “Programming is Hard, Let’s go Scripting...” <https://www.perl.com/pub/2007/12/06/soto-11.html/>

without drawing any parallels does not help a scientific field forward. In other words, without theory, there is no cumulative tradition, no ‘glue’ that holds studies together, nor is there any hope for parsimony: tens or hundreds of studies might find similar findings, all slightly different, without any attempt to synthesize, generalize, and unify such findings into a coherent theory. Each study stands on its own, and while it may reference other studies on a particular topic, there is no attempt to provide any link at a conceptual level that helps us understand trends, causes, and effects. Dubin distinguished between “being a good reporter” and “being a good scientist”; the former can present new and useful information to the body of knowledge, but the latter is able to find relationships between ‘things’ [35]. This can readily be seen in many SE papers: ‘Related Work’ sections read as obligated shopping lists of references to please reviewers, rather than an attempt to synthesize the state of knowledge relevant to the study that is presented (“good reporting,” not “good science”). The common practice in SE papers of placing such a section *at the end* of a paper exacerbates this, and is therefore misguided.

Second, and closely related to the argument above, is that due to a lack of theory or conceptualization, we remain stuck at the level of the idiosyncratic, the specific, without seeing the ‘big picture.’ Theory helps us to focus on what is important, rather than what is incidental. Without theory we tend to focus on the ‘here and now,’ rather than on what we can learn to serve us in the future. That is, rather than focusing on the latest trends, which at the time of writing seems to be the use of Large Language Models (LLM) in Software Engineering (SE), and highlighting how useful specific features of a given version of ChatGPT might be, we ignore more general questions that consider fast-emerging technologies in general and how they can be leveraged in SE in practice. By the time a paper is published, the study may be outdated as the next version of the LLM is released! What can we learn from quickly emerging AI technology *in general*? Or more generally stated, what can be a *lasting contribution* to the field, rather than one that is outdated by the time the paper is presented at a conference?

Third, theorizing can help to explain seemingly ‘obvious’ findings. Many papers receive review comments stating that “*the findings are obvious.*” The first line of defense against this is to state that, without an empirical study, we cannot know whether something is true; perhaps our assumptions are not true at all! Theorizing plays a key role here because it can help to unpack a line of reasoning, to lay bare the assumptions that may be held within the literature, and to make explicit, with rigorous and sound argumentation, what the underlying process might be that underpins the findings. If the data support the hypothesis, then besides presenting the first empirical evidence for it, a paper would also make explicit why those findings make sense. If findings are counter-intuitive, this by itself would also be very interesting. It is thus the unpacking of a given phenomenon or topic in detail—the theorizing—that would be the contribution.

1.2 Overview of this Chapter

This chapter draws on insights from other fields, in particular Information Systems (IS), to recast the discussion from “theory in SE” to “theorizing in SE.” Specifically, this chapter is organized around 12 intermediate products of theorizing identified by Hassan et al. [47]. In previous work, I used the term “theory fragments” to refer to a similar idea [109]. I define “theory development” as a research strategy [110] that explicitly sets out to develop new theory. *Theorizing* is the process of engaging in theoretical development without the explicit goal of presenting a sufficiently developed theory.² Clearly, these activities are related; a scientific discipline needs both, but in this chapter I focus on the latter.

The goal of this chapter is twofold. First, it provides a gentle introduction to the 12 theorizing products, what they mean in general, and what they mean in a SE research context. The chapter then demonstrates how some excellent SE scholars have used these theorizing products. It is worth noting that one could fill a chapter on each of the 12 theorizing products. Many authors have presented insightful accounts on topics such as myths and metaphors; the current chapter only scratches the surface. Studying each of these is left to the reader as the proverbial exercise. Second, this chapter encourages the reader to introduce conceptual thinking into their research. In what ways can a topic of interest be elevated to a more conceptual level? What theorizing products have been used in the reader’s field of interest? How can the reader incorporate these in their own research?

Learning Objectives

The learning objectives of this chapter are:

1. To understand and appreciate the role of theorizing in SE research.
2. To understand and apply the 12 intermediate theorizing products in the context of SE research;
3. To recognize theorizing in SE papers, by reconstructing the theoretical advances made using the 12 intermediate theorizing products.

2 Theory Development in Software Engineering Research

2.1 Views on Theory in Software Engineering Research

Whereas there have been some efforts within the SE literature to pay more attention to ‘theory,’ the SE community has, by and large, ignored these. One reason for this

² Theories are arguably never complete, because they can always be extended in one way or another; hence, the phrase ‘sufficiently developed.’

is that theory development is difficult. Another reason is that there appears to be a mismatch between what researchers *believe* theory is, and the research that they do; researchers may believe that traditional notions of theory, such as a network of constructs and relationships among them, do not apply to their research.

These are good reasons, and they have to do with the fact that nobody appears to be able to define what ‘theory’ *is*. SE researchers are not alone in this; it appears fair to say that there are no objective criteria by which one can determine what is, and what is not, a theory.

In this chapter I argue that, in order to learn how to theorize, we need not be overly concerned with the traditional notion of theory as a collection of constructs and relationships, but that intermediate theorizing products [47] may also be sufficient to move the SE field forward. These outcomes can take many different shapes; the chapter seeks to explain the value of each of these types of products, what their added value is, and how they can contribute to a cumulative tradition in SE research. The chapter seeks to ‘free’ readers from thinking in fixed patterns (traditional notions of ‘theory’) and offer new ways that might be more suitable to SE.

Researchers in the SE field have varying views on theory and its role in research. This is partly due to the fact that the SE field consists of several sub-fields, many of which have their own dedicated conferences and sometimes journals, and consequently, each with different ideas and norms about what constitutes ‘research.’ For example, the MSR³ community is quite distinct from the CHASE⁴ community in the way the research is done. Some researchers have looked, seemingly with a sense of envy, at other academic disciplines where it is more common to have theories that guide and inform empirical studies, which in turn extend or dispute those theories. This envy has led some scholars to wonder: *where is the general theory of software engineering?* [57].

Others do not seem to perceive this lack of theoretical focus, and may consider the state of the art within their specific sub-field as constituting the theory that they contribute to. If we were to ask researchers active in requirements engineering or software testing, for example, to provide an overview of their theory, we’d get models, processes, and taxonomies (cf. [13, 81]). The term ‘theory’ is used here in a different way, yet it is not fundamentally incompatible with how it is used in the social sciences. In fact, the intermediate theorizing products presented in this chapter can be identified in those bodies of knowledge as well. Doing so is a challenging but excellent exercise for the reader.

2.2 What Do Theories Look Like?

Most discussions of ‘theory’ in SE provide tutorial-style explanations suggesting that a theory consists of theoretical concepts and the relationships between them

³ Mining Software Repositories

⁴ Cooperative and Human Aspects in Software Engineering

(propositions or hypotheses) (cf. [109]). This common format is a template for what is called a variance theory: if A causes B, then A and B are the concepts, and the relationship is a causal one.⁵ This means that if A goes up, then B goes up also. Figure 1 presents an example of a variance theory. Variance theories are intuitive, and they can be tested using quantitative datasets, which are relatively easy to come by, for example by mining software repositories (see Codabux et al.’s chapter in this volume) or developer surveys, notwithstanding the many challenges in both MSR and conducting surveys (see Kalinowski et al.’s chapter in this volume).

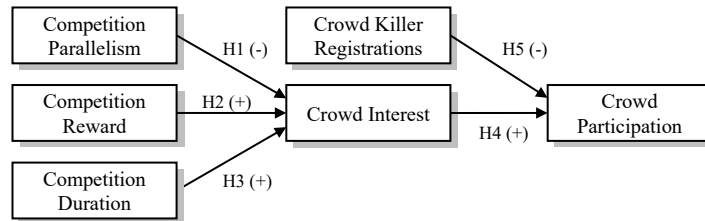


Fig. 1 Theory of crowd participation [106], an example of a variance theory

Another type of theory is called ‘process theory,’ defined by Huber and Van de Ven [53] as: “An explanation of the temporal order in which a discrete set of events occurred, based on a story or historical narrative.” Ralph defines a process theory as: “A process theory is a system of ideas intended to explain, understand (and sometimes predict) how an entity changes and develops,” and discusses four ‘ideal types’ of process theories frequently found in organizational/management research, and how they apply to SE [84]. Process theories are frequently more flexible and capture a wide range of phenomena in the real world. However, these theories are not easily tested using quantitative datasets only, but rather in comparison to an alternative process theory (cf. Ralph [83]). Figure 2 presents a process theory of large-scale agile software development. This process theory explains how tensions that arise through both internal and external perturbations require an emergent response of the organization or teams involved, so as to restore a certain ‘balance.’ As such, this can be seen as a *dialectic process theory* (one of the four ‘ideal’ types), in which “stability and change are explained by reference to the balance of power between opposing entities” [119]. In this case, the opposing entities are the tensions that arise due to a conflict between agile processes and the large-scale nature of a software project, and the emergent responses by means of adopting both plan-driven and novel agile practices that can reduce those tensions. Note that this process theory defines a number of propositions (numbered P1 to P7), and may resemble the hypotheses in

⁵ Most work tends to shy away from the verb ‘cause’ and use the more vaguely stated “is associated with” so as to not claim causality. Causality can typically not be demonstrated without conducting controlled experiments (see also Vegas and Juristo’s chapter on experimentation in this volume), though there are some techniques that can provide convincing evidence for causality.

the variance theory in Fig. 1, but that these are not easily testable using quantitative datasets.

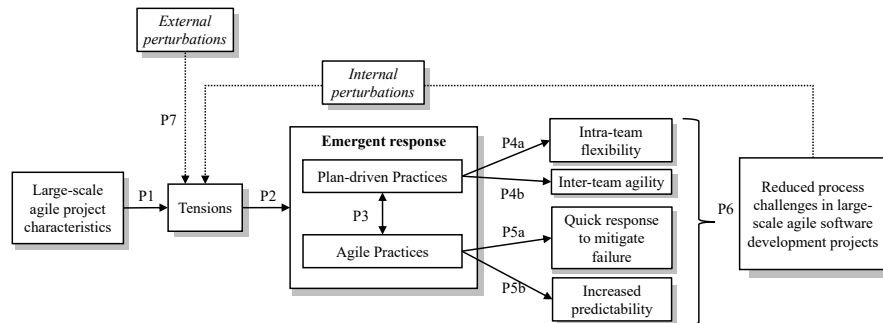


Fig. 2 Process theory of large-scale Agile software development (Rolland et al. [90])

While variance and process theories are perhaps the best known within SE literature, these are by no means the only forms. Other scientific disciplines and discourses follow other traditions. For example, Dittrich developed a *practice theory* on the use of software development methods [34], which:

“develops a conceptual base for understanding software development as social and epistemic practices, and defines methods as practice patterns that need to be related to, and integrated in, an existing development practice.”

The format of Dittrich’s presentation does not follow that of a variance or process theory, but is much more narrative. Thus, there is no clear-cut format that defines what theory is. This remains one of the major challenges in developing theory.

2.3 Challenges in Developing Theory in Software Engineering Research

As noted earlier, while much has been written over the past decades on ‘theory,’ there is no general consensus on a definition of ‘theory’ [124], and what people accept as a theory appears to be dependent on the norms of a specific field of research. What might some other reasons be why developing theory is difficult in the SE field?

- There is a general lack of knowledge and appreciation for theorizing;
- There is a mismatch between the SE field’s fast pace of development and general level of patience of researchers;
- Researchers conflate and lack understanding of terminology.

Due to a general lack of knowledge of, and appreciation for theorizing, many researchers within the SE field simply do not pay much attention to this. Related work sections are mere shopping lists of a “who’s who” in the field of interest, without a

deep engagement with the actual contributions of those authors; Nobel Prize recipient Sir Peter Medawar described it as [70]:

“a section called ‘previous work’ in which you concede, more or less graciously, that others have dimly groped towards the fundamental truths that you are now about to expound.”

In other words, the SE field lacks an effective cumulative tradition. For example, there are over 100 case studies on the use of agile methods in distributed development [33]. Through a carefully crafted introduction, many papers over-rely on ‘spotting the gap,’ i.e., identifying the study that *hasn’t* been done, without aiming to see the forest for the trees.

Another major impediment to developing theory is the time commitment that is necessary. Two related factors play a role which go against this time commitment. First, there is the SE field’s pace: new technologies and trends emerge quickly. At the time of writing, the release of ChatGPT, a publicly available LLM, immediately demanded attention from many researchers leading to papers that describe this new type of technology. Second, SE researchers are not rewarded for long-term investment; it is still common for researchers to target the International Conference on Software Engineering (ICSE) with papers that are not further developed.⁶ It appears that the community norm isn’t such that researchers are encouraged or expected to follow up, nor to develop theory. This community norm is difficult to change. Students may not be overly concerned with this, as their funding clock is ticking. Supervisors are in a difficult position, because they prefer their students to succeed and not subject them to higher standards than would be the norm within the community, while both funding agencies and tenure evaluation committees are expecting results from principal investigators.

Finally, the terminology surrounding theory, theorizing, and theory development can be rather confusing. Most of the terms discussed in this chapter are misunderstood, misused, or conflated with other terms. For example, the terms ‘model’ and ‘framework’ are frequently conflated, as are the terms ‘concept’ and ‘construct’ [102].

The remainder of this chapter addresses these three issues. By elaborating each of the 12 intermediate products of theorizing in the next three sections, it addresses the last issue of terminology. It then highlights 12 intermediate products of theorizing and demonstrates how they have appeared in SE literature, drawing attention to the role these products have played in theorizing. Finally, it draws attention to the possibility of building a cumulative tradition within SE, by demonstrating how the theorizing products can contribute to a ‘bigger picture.’

3 Theory Framing

The first set of theorizing products help in the activity of *framing* a phenomenon. The term ‘framing’ means establishing what the topic of interest is, or positioning a research problem in the real world. Framing is the activity of finding a ‘hook,’ or

⁶ Without wishing to sound cynical or overly critical, I am not the first to lament this [77, 78, 21].

problematizing prior work [5, 90] to start off the theorizing activity. This section discusses theory products to frame studies with a summary in Table 1.

3.1 Question

A starting point to frame or position a topic or phenomenon of interest is to ask questions. Hassan et al. [47] pointed out that “*a discipline is defined by the set of questions it asks.*” Different disciplines can ask related questions pertaining to the same phenomenon, but depending on the discipline, different concepts, methods, and tools are likely to be used to answer those questions. A good example to illustrate this is the question: *What is Open Source Software?* OSS emerged in the nineties as a topic of interest for several disciplines. Economists were baffled by the phenomenon that developers would work *for free* to deliver high quality products [16, 67]. Information Systems (IS) researchers tended to focus on OSS as a product, its qualities [80, 104], and how organizations could leverage open source communities to build products [4, 73]. Meanwhile, SE researchers initially focused on the development process, asking questions surrounding concepts that were of particular interest to software engineers [72]. More recently, SE researchers have focused on a wide range of other aspects of OSS, such as peer review practices [87] and corporate participation in OSS projects [126].

How is a question an intermediate product of theory? Asking a question goes together with a justification, a motivation, and a perspective that is linked to a discipline, and knowledge of a field of inquiry. Doctoral students frequently spend a considerable amount of time to develop a research proposal, and it would not be uncommon for this process to take over one year. Identifying the question to pursue is far from trivial, because questions must come with a line of reasoning—and the lack of a clear reasoning is frequently a justified reason to reject a manuscript. The ability to ask interesting questions relies on intimate knowledge of both the state of practice and the state of the art. Asking interesting questions thus helps to articulate what is yet unknown, in a way that is relevant and important to a field of inquiry, suggesting that an answer will help the field as a whole forward. Alternatively, questions can be asked by challenging what is supposedly already known: problematizing assumptions within a given body of literature can be a fruitful approach to generate new insights [90, 95, 97].

In conclusion, a question is an intermediate product of theorizing, because it conveys not only the question itself, but also an awareness and familiarity of prior literature, as well as the state of a particular field of inquiry. The question itself does not constitute a theory, but if answered, can lead to a theory.

3.2 Paradigm

Whereas questions define a research discipline, the type of questions that are being asked within a given field depends on the *paradigm* that researchers within that discipline subscribe to.⁷ A good example here is Basili’s pioneering work that has emphasized the role of experimentation in SE research [10]. Basili [9] wrote:

There is a fair amount of research being conducted in software engineering, i.e., people are building technologies, methods, tools, etc. However, unlike in other disciplines, there has been very little research in the development of models of the various components of the discipline. The modeling research that does exist has centered on the software product, specifically mathematical models of the program function.

Basili’s key point here is that there had been too much focus on building tools and technologies, rather than the main components of software engineering as a discipline. He argued for a need to make *theoretical models*, not (only) mathematical (formal) models of software products, though obviously the latter is a valid way of doing research. Basili continued:

We need research that helps establish a scientific and engineering basis for software engineering. To this end, the research methodologies required involve the need to build, analyze and evaluate models of the software processes and products as well as various aspects of the environment in which the software is being built, e.g the people, the organization, etc. It is especially important to study the interactions of these models. The goal is to develop the conceptual scientific foundations of software engineering upon which future researchers can build.

Whereas experimental SE emphasizes quantitative methods and the need to develop a conceptual scientific foundation, a next step in the evolution of empirical methods was evidence-based software engineering (EBSE), modeled after evidence-based medicine (EBM) [60].⁸ The EBSE paradigm has had a major impact on the SE field. An important part of EBSE is the Systematic Literature Review (SLR) method. This method is commonly used in medicine to collect all available evidence for a certain ‘intervention’ (for example, a medical treatment). Similarly, the SLR was envisaged to be a useful tool to gather all evidence regarding specific practices and tools, and indeed, countless SLRs are published every year. This is a good example of how the paradigm shapes the way we answer research questions.⁹

⁷ The term ‘paradigm’ is considered by some as somewhat controversial due to Thomas Kuhn’s use of the term in his influential book *“The Structure of Scientific Revolutions”* (cf. Shapere [99]). Notwithstanding, a paradigm is a way of looking at the world, and comes with a set of shared assumptions. For example, in the Object-Oriented (OO) programming paradigm, all ‘things’ are modeled as ‘objects.’ A paradigm is reflected in the way people think, write, and discuss about a topic of interest. The term is not limited to only refer to *research paradigms*, such as (post-)positivism, interpretivism, and critical realism.

⁸ Following Kitchenham et al. [60] and the emergent attention for theory in SE about a decade ago, I had suggested a new paradigm: Theory-Oriented Software Engineering [109]. It appears the paradigm has thus far not taken any root.

⁹ While the idea of the SLR is a good one, unfortunately many SLRs in the field lack synthesis and theorizing that could help readers to see the forest before the trees [28].

We can also observe a paradigm shift in how the SE community sees itself. Whereas traditionally the SE field was seen by many scholars as an engineering field [7, 21, 100, 120], there is an increasing consensus that SE is a socio-technical field [111], focusing on what has been termed Behavioral Software Engineering [66], and that software has a major impact on society; for example, since 2015, ICSE features a Software Engineering in Society (SEIS) track. In the last decade or so, there has been increasingly attention for human aspects; more recently, studies focus on themes such as burnout, developer happiness [43], and diversity and inclusion.

In sum, a paradigm is not a theory, but helps to frame a topic of interest, to direct our focus, and to inform research questions to answer. To follow a certain paradigm, a researcher adopts a specific focus on what to study, and how.

3.3 Law

A ‘law’ is a generalizing statement that holds true under certain conditions, and which provides explanatory power [36]. Newton’s law of universal gravitation in physics, for example, explains why objects fall to the ground. The general law of supply and demand in economics explains when prices of products and services go up or down (with boundaries and conditions beyond which the law may not hold, e.g., economic monopolies, price elasticity). In both cases, specific conditions may affect the extent to which these laws hold.

There are many ‘laws’ in SE [36, 96], though what exactly makes something a law is unclear. Many of them appear to be generalized statements that are based on observations in practice, though it appears that these statements have held over time. One of these is Brooks’ Law [22], which states that adding manpower to a late software project makes it later. Several researchers have taken Brooks’ Law to *frame* their research [2, 52]. Hsia et al. [52], for example, conducted a system dynamics simulation study to understand the implications of Brooks’ Law. The ‘law’ suggests a certain truism, but Hsia et al.’s simulation study suggests that “*there is a time line T for each project that if enough manpower is added before T , the project still can finish before the scheduled deadline,*” challenging the universality of Brooks’ Law.

A second example of a law in SE is Conway’s Law [26], which states that organizations produce system designs that mirror their communication structures. Herbsleb and Grinter [49] used Conway’s Law as a starting point for their study of distributed development, where good communication, in particular informal communication, is much more challenging to achieve. Conway’s Law has been used as a way to frame several other studies [20, 14].

Laws are not quite theories, but generalizations that can be used to frame a research study, to declare an initial set of theoretical expectations. The extent to which a law holds under alternative conditions can then be the focus of the study, either to confirm the law or refute it.

3.4 Framework

A framework captures the things a researcher is interested in; it describes the “map of the territory” that is of interest [71, 98].¹⁰ A framework has been described as “*a less developed form of a theory*” [93]. A framework identifies the key concepts within a study; it may also capture how the concepts of interest relate to one another, but that is not necessary. Any concept that is not part of the framework is thus not part of the ‘map,’ and thus not studied. In other words, a framework helps the researcher to focus on what is important. The conceptual framework is usually developed prior to any empirical work through a literature survey—a thorough review of literature that involves synthesis and theorizing is a study on its own [110]. As part of the empirical work, the researcher may decide to refine the conceptual framework.

An example is Stol and Fitzgerald’s framework to study crowdsourcing for software development [107, 108]. The framework is developed based on an extensive survey of prior literature on crowdsourcing to develop an understanding of the process, what it entails, and who might be involved. The framework identifies six salient themes that appear to be of particular relevance to crowdsourcing in a software development context. Further, the crowdsourcing process involves different stakeholders or perspectives (see Figure 3). This framework does not, however, capture any relationships between different concepts. Researchers can focus on one particular perspective across all six facets (one of the rows), or on one particular facet across three perspectives (one of the columns), or only on one facet from a single perspective (a single cell).

Developing the conceptual framework is a theorizing activity; there is no absolute right or wrong way of doing it, though good judgment and justification play a role. Which concepts and themes are included depends on the researcher who should justify each concept’s inclusion, typically through a careful literature review. When developing a conceptual framework, one should ensure that the map doesn’t become larger than the territory; that is, researchers may be overly diligent or mechanistic, and attempt to capture every little detail that could be relevant to their topic of interest. This would lead to a framework that is so complicated that it loses its strength; the point of a framework is to focus one’s attention to the important elements within a domain. In my experience, frameworks that capture up to 7 themes are manageable; when more themes are included, it may be useful to add some level of hierarchy to organize concepts. For example, in our study of key factors for adopting Inner Source, we identified no fewer than nine factors, and so we organized these into three categories [105].

In an attempt to keep the number of concepts limited, it can be tempting to combine concepts that appear to be related. It is important that this is done with utmost care, only when two concepts are so intertwined that keeping them separate is not practical. For example, several concepts within our framework for studying crowdsourcing

¹⁰ In this chapter I refer to theoretical or conceptual frameworks, not to be confused with software development frameworks, which are software libraries that provide functionality and may impose an architectural structure of the software that is developed. For example, Django and Flask are popular frameworks for developing Python web applications.

	Task decomposition	Coordination & communication	Planning & scheduling	Quality assurance	Knowledge management & IP	Motivation & remuneration
Customer	How do customers decompose software development tasks for crowd development?	How do customers coordinate tasks and communicate with the crowd?	How can software development tasks be planned effectively with the crowd?	How can customers ensure the quality of deliverables from the crowd?	How do customers manage project knowledge and protect their IP?	How can customers motivate the crowd, and what is an appropriate remuneration?
Crowd workers	How can crowd workers work on a single task while not knowing the rest of the system?	Do crowd workers need to coordinate among each other? How?	How do crowd workers deal with short development deadlines?	How do crowd workers test their deliverables?	What knowledge do crowd workers need to develop part of a system?	What motivates crowd workers, and what role does remuneration play?
Crowd-sourcing platform	How can platforms support both customers and crowd workers in defining tasks?	What mechanisms should platforms offer to allow effective coordination and communication?	What mechanisms should platforms offer to plan and schedule tasks?	What role can platforms play in quality assurance of deliverables?	What knowledge management solution should platforms offer to protect their clients' IP?	What guidance can platforms offer for both customers and crowd workers to come to an agreeable remuneration?

Fig. 3 Conceptual framework capturing ‘facets’ of crowdsourcing in a software development context, as well as three different perspectives from which these facets can be considered (based on Stol and Fitzgerald [107])

were combined; when discussing the topic of ‘planning,’ we could not clearly and meaningfully separate this from ‘scheduling.’ Thus, if two concepts are intertwined, it is best to combine them. If there are too many distinct concepts, it is better to add some level of hierarchy by introducing categories.

A framework is not a theory, but it certainly is an intermediate product. It helps to conceptualize important elements within a domain of interest and as such to *frame* a research study. They are reusable and extendable by others, and can provide a foundation for a whole stream of studies.

Table 1: Theory frames

Product	Description	Example in SE
Question	What questions are being asked depends on the discipline; a field of study can be defined by the set of questions it asks, the methods that are used to answer the questions, and other norms within the field [47]. Clearly, the questions being asked, the methods being used, and the norms within a field evolve over time.	The question ‘What is Open Source Software?’ has been addressed by several academic disciplines; SE research distinguishes itself from, say, Economics through the use of different theories and methods to answer the question.

Product	Description	Example in SE
Paradigm	A paradigm is a way of looking at the world. Within scientific inquiry, it defines how researchers ask questions, and how those questions are answered. Paradigms imply a set of shared assumptions, and the questions that are worth asking.	The Experimental Paradigm in Software Engineering [9]; the Evidence Based Software Engineering paradigm [60]. Software Engineering as a branch of engineering [120]; Software Engineering as a socio-technical field [56]; Software architecture as a set of design decisions [55]
Law	A law is a generalized statement that has universal applicability. Some laws are supported by both strong theory and empirical observations (e.g. Newton's universal law of gravity), others appear to be statements describing phenomena that appear to have universal application without a theoretical explanation.	Conway's Law, used to frame Herbsleb and Grinter's study "Conway's Law Revisited" [49]; Lehman's Laws of Software Evolution; Hyrum's Law ¹¹
Framework	A framework is an analytical tool that comprises the main concepts of interests, and potentially any relationships between them. Frameworks can serve many different purposes [98], and can be used to design a study, or be the outcome of a study.	Stol and Fitzgerald [107]'s research framework to study crowdsourcing, identifying six important themes within the domain of crowdsourcing when applied to SE.

Exercises

1. Consider one of the following topics:
Large Language Models, Secure Coding, Diversity and Inclusivity in SE
Identify a question to ask about this topic that is relevant to the software engineering community. Why is this question relevant to SE, and why is it important? What could be a question that is not relevant to SE?
2. Two popular paradigms within SE research are Evidence-Based Software Engineering [60] and Behavioral Software Engineering [66]. What are the assumptions underpinning each? What type of questions does each elicit? Suggest a question that is representative of each paradigm for the selected topic.
3. There are many 'laws' within SE (Brooks' Law and Conway's Law as discussed above; Lehman's Laws of Evolution, etc.) [36]. Identify a law within the SE domain, and how this could frame new research problems.
4. Given one of the topics listed above, perform a quick (ca. 1h), ad hoc literature review, and identify some themes that appear to be relevant to the picked topic. Create a table with these themes, and summarize for each

¹¹ <https://www.hyrumslaw.com/>

theme why it is relevant to the topic at hand, and identify at least one question for each theme.

4 Theory Generators

A second set of intermediate products of theorizing can help to *generate* theory; these include myths, analogies, metaphors, and models [47]. They help to generate theory because they can offer an initial foundation to capture some phenomenon of interest. This section describes these (see Table 3 for a summary).

4.1 Myth

A myth is an unquestioned belief that is not supported by evidence or fact [118], or “A *dramatic narrative of imagined events, usually used to explain origins or transformations of something*” [50]. Hassan et al. [47] argued that myths are theory generators because “*they can help uncover unquestioned assumptions within existing belief systems and theories.*” Myths help researchers to focus their research, to question whether the myth is true or not, and if not, pursuing an answer to the question what the truth then might be. Myths abound in SE research [58]. A classic example of this is the traditional waterfall approach to software development, described and critiqued by Royce [92]. In brief, the waterfall process describes a rational process, whereby first the requirements are gathered, followed by a system design, implementation, and then testing. While this approach appears to be completely rational in that *it makes sense* to the outsider, anybody who ever developed any software knows this is not possible. Parnas and Clements [79] discuss this mythical rational process, argue that it is an “idealization,” but then provide reasons as to why a description of such a process is useful. In effect, taking the myth of the rational development process, Parnas and Clements theorized why it is a myth, and continued to argue for faking it nevertheless.

Another example of this is a study by Rolland et al. [90] (briefly mentioned in Sec. 2.2), who challenged three assumptions that were widely held in relation to the scaling of agile methods. In brief, while agile methods were originally meant for small projects performed by co-located teams, it soon became clear that organizations wanted to apply agile methods at scale. A considerably body of knowledge emerged over the past decade or so, reporting on such tactics like Scrum-of-Scrums to scale up agile methods. In their study, Rolland et al. identified three assumptions, or myths, that appeared quite dominant throughout the literature, for example: “Agile and plan-driven methods are perceived to be mutually exclusive.” This set the focus of the field study, through which they demonstrated that organizations need plan-driven

practices to enable agility at scale. The analysis resulted in a series of propositions grounded in the empirical findings. Thus, the myth helped to *generate* new theory (see Fig. 2).

4.2 Analogy

An analogy can be used to explain or clarify something that is not (or less) familiar, by means of comparing to something that is known. In very simple terms, an analogy is saying *X is like Y*, whereby X is the topic at hand, and Y is the analogy. While the two things differ in some ways, using an analogy means emphasizing the common aspects, finding similarities [48]. Analogies are useful to help explain a lesser known phenomenon or thing using terminology that is more familiar.

As a simple example, consider the practice of pair programming. When we say that pair programming “is like riding a tandem bicycle,” we are using an analogy to convey that two people are working together to make progress on a task.¹² We’re not suggesting that pair programming *is* riding a tandem bike. Analogies are closely related to metaphors, discussed below; while all metaphors are analogies, not all analogies are also metaphors. Within pair programming, we rely on the metaphors ‘driver’ and ‘navigator’ to convey the meaning of the two roles engaged in pair programming.

An analogy is a very effective literary device to explain an otherwise difficult or unknown topic. To give another example: Component-Based Software Development (CBSD) was a popular approach in the nineties that attracted considerable attention from SE researchers. It can be explained by analogy as building a house from standardized, components that are pre-fabricated off-site [62]. The analogy helps in understanding that components are built ‘off-site’ (i.e. by other teams or companies); that they have some standardized interface to connect them together, and that they exhibit certain pre-defined qualities. However, the analogy may also explain why CBSD as originally envisaged has not quite become common practice and faded as a topic of research:¹³ in building construction, components that don’t quite fit well can be glued together with expanding foam, which is commonly used to seal gaps around cavities and is ideally suited to fill irregular shapes. In software development, however, we don’t have the equivalent of expanding foam: interfaces, data-formats and protocols must be strictly adhered to, otherwise the system as a whole doesn’t work (cf. Garlan et al. [39]). I would be remiss not to mention the Perl programming language, known as the “Duct Tape of the Internet” [46], but this is more a metaphor

¹² Lazy back-riders notwithstanding.

¹³ This is arguably a pessimistic view, and depends on one’s definition of ‘component.’ In the nineties, standard component architectures such as Microsoft’s COM (and later DCOM), and Java’s JavaBeans appear not to have achieved the widespread adoption that had been expected. At the same time, if we treat open source libraries as components, software reuse has become an incredible success. Software ecosystems found in many popular languages (including NPM (Node.js, JavaScript), R, and Rust) offer fantastic environments to build new software products quickly from these components.

than an analogy, because we don't *compare* the Perl language to duct tape in a structured manner (see Table 2). Metaphors are discussed next.

Table 2 Nobody makes a comparison like this to explain what the Perl language is, because duct tape is used a figure of speech. For the duct tape metaphor to work, it is important that people know what it is, else it is not a good metaphor.

Attribute	Duct Tape	Perl language
Stickiness	Ties things together	Ties different programs together.
Versatility	Easy to tear off a piece to use quickly; easy to use on any surface	Easy to fit in a Perl script into any operating system environment, including practically all Unix and Linux variants, Windows, and MacOS.
Maintainability	May not last long	Perl scripts sometimes seen as unmaintainable due to the Perl syntax that doesn't suit everybody, and which some have described as "line noise" and Perl as a "write-only language."

4.3 Metaphor

Metaphors are similar to analogies, but whereas an analogy is stating that X is *like* Y, a metaphor is stating that X *is* a Y.¹⁴ The difference is small and sometimes subtle, also because all metaphors are analogies, but not all analogies are metaphors. A few examples follow.

When we use the term 'publication pipeline,' we are using a metaphor to convey the process of producing research papers. Implied here is that a number of papers are 'in production' in various states of readiness. What we don't mean is that we have an actual pipeline in our labs into which we pour research data, and which produces papers as output.¹⁵ When we say that we don't have any 'bandwidth' to take on more research projects, we are using the term metaphorically to convey we are at capacity. To stick with the domain of publishing our research, we can treat our academic publications as a 'conversation,' whereby each publication contributes to that conversation [12]. Following this metaphor, it would only make sense to contribute to a conversation if you first listen carefully to others—that is expected in academic papers through a related work section. Thus, this metaphor helps us to theorize about how academic publishing works, or should work. As discussed in Sec. 2.3, the SE literature does not much resemble a conversation.

Metaphors are extensively used in the SE field [59]. One that has received considerable attention over a time-span of more than two decades is the term

¹⁴ This is a simplification, and does not necessarily imply that the decision whether something is a metaphor or analogy depends on the presence of the words "is like."

¹⁵ Notwithstanding the recent developments of large language models.

‘Technical Debt.’ The metaphor was first used by Cunningham [29] to describe the danger of shipping “not quite right” code, and the trade-off that is made in order to be able to ship a product fast:

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.”

Kruchten [64] explicitly recognized the role of this metaphor as a theory generator in their editorial “*Technical Debt: From Metaphor to Theory and Practice*.” Lamenting that any issues in software development could be labeled some sort of ‘debt’, Kruchten et al. argued that:

“We need a better definition of what constitutes technical debt and some perspective or viewpoints that let us reason across a wide range of technical debt. In short, we need a theoretical foundation.”

4.4 Model

Similar to the term ‘framework,’ the term ‘model’ also has different meanings in a SE research context. Whittle et al. [123] succinctly describe that “*in software engineering, a model is an abstraction of a running system.*” This approach to software engineering is better known as Model Driven Engineering (MDE).¹⁶ Renowned theorist Robert Dubin equated the term ‘model’ to mean ‘theory,’ though acknowledged that not everybody agrees with this view [35]. In this chapter, following Hassan et al. [47], I use the term ‘model’ not to refer to a theory, but as an intermediate product of theorizing.

Consistent with Whittle et al.’s description, the term ‘model’ usually implies that it represents an abstraction from something in the real world, and in the context of this chapter that is how I use the term model. Abstraction implies simplification and the leaving out of details. Curran [30], discussing statistical modeling, describes the importance of simplification:

“If I build a model 787 in my backyard, and then fly 382 people to Charles de Gaulle, it’s not a model 787, it is a freakin’ 787. It is the simplification that is the point.”

Statistical models are common in SE research, in particular multiple regression analyses. Less popular, though more powerful and flexible, are structural equation

¹⁶ One particular subset of MDE is Model Driven Development, which seeks to generate code based on a model. In MDD, the model must be exactly right so that correct code can be generated—one could say that the model can be executed; generating and compiling the code are just intermediate steps. Because details now matter and must be included for the model to be executable, it is no longer an abstraction but just a different way for a programmer to express their intention (instead of code). Therefore, the term ‘model’ in MDD diverges from how the term is used in this chapter. In contrast, in research we create models to simplify the topic of interest so as to make sense of it, to reason about it, and to analyze it.

models (SEM) which are statistical models that can be easily visualized as box-arrow diagrams, which are a common way to represent variance theories (see Sec. 2.2). Statistical models have an associated model *fit*; in multiple regression this is R^2 , indicating the variance explained by the fitted regression line, and in SEM there are a whole range of fit indicators [112]. These measures of fit indicate how well the theorized model ‘fits’ the data sample, and acknowledge that models are, in the end, simplifications that leave out details. In practice this means not only that variables are left out of a model (thus reducing the R^2), but also the fact that a quantitative analysis cannot capture the idiosyncratic, and the specific details that could provide further insight.

A well-known statistical model in SE is Boehm’s COCOMO cost estimation model [15]. But, not all models are statistical. Redwine and Riddle [86]’s technology maturity model (even though they didn’t call it a ‘model’) describes how technologies mature and get adopted over time, which can help to predict and set expectations. Models are used extensively in the literature on software processes [91], which describe processes, though models could describe other things as well. The waterfall approach described earlier (see Sec. 4.1), which was documented and criticized by Royce [92], is a model of the lifecycle of software development. The utility of such models like those described by Redwine and Riddle [86] and the waterfall model can be assessed by looking at how closely they represent reality—how well these models ‘fit’ with reality. As it turns out, the waterfall model is a rather poor fit with reality, in that most software development doesn’t actually follow such a linear process.¹⁷

Models provide the researcher to describe something in a systematic way so as to communicate the important characteristics, leaving out the idiosyncratic issues from the real world. As an abstract representation, researchers can reason about it, perform analyses, and even run computer simulations which themselves rely on simplifications [110]. Simulations are discussed in more detail in De França’s chapter in this volume.

Table 3: Theory generators

Product	Description	Example in SE
Myth	A myth is an unquestioned belief that is not supported by evidence or facts [118]. Challenging a myth can help to identify a new research direction.	The assumption that agile methods and plan-driven methods are mutually exclusive is a myth that Rolland et al. [90] challenged so as to generate new theoretical insights.
Analogy	An analogy is a comparison of a phenomenon of interest to a phenomenon that is more familiar. The goal is to draw parallels so as to explain what something is, or how or why something works.	A software system grows like a city; that is, a software system isn’t a city, but it evolves <i>like</i> a city: it is in a constant state of evolution, changing some parts.

¹⁷ Software development in safety-critical domains, such as the aerospace domain, automotive, and medical devices, is subject to regulatory guidelines or standards that prescribe the use of plan-driven methods.

Product	Description	Example in SE
Metaphor	A metaphor is a figure of speech that equates two things, thereby assigning characteristics of the one thing to the other. All metaphors are analogies, but not all analogies are metaphors.	Software Engineering; software development pipeline; software ecosystem; scaffolding code [22]; Technical Debt [64].
Model	A model is a simplified representation of a phenomenon that includes only those aspects that appear relevant and important, disregarding any details that may be idiosyncratic or incidental. A model helps to reason about the phenomenon of interest. The more detail a model has, the more realistic it is likely to be, at the cost of increased complexity.	Statistical models including structural equation models; software process models, including waterfall as a simplification of the software development life-cycle; computer simulations that are configured through a fixed number of parameters.

Exercises

1. Consider your own research topic. Attempt to find a paper that presents a “model” of in relation to the topic, and also a paper that presents a “framework” for the same topic. How are they used? How do they differ?
2. Discuss the difference between an analogy and a metaphor. How do they differ?
3. Identify a myth in the software engineering literature, and find one or two papers that discuss or investigate it. How could this myth be used as a theory generator?
4. Identify a ‘law’ in the software engineering literature, and find one or two papers that discuss or investigate it (cf. [36, 96]). Well-known examples are Brooks’ Law, Conway’s Law, Hyrum’s Law. How does the law help to frame research?

5 Components of Theory

The third set of theorizing products are actual components, or elements, of theory. These include the things we wish to reason about or measure (concepts, constructs), and the relationships between those things (statements, hypotheses). These components are commonly explained in a tutorial on ‘theory’ [109]. Because the terms concept and construct are so closely related, in defining one I make reference to the other. Table 4 presents a summary.

5.1 Concept

Concepts are “*ideas that receive names*” [47, 117], and as such, they are the building blocks to develop theories. A research discipline deliberates the things that scholars believe are important, who then go on to name them. They have been described as the “*product of the imagination that can be conveyed to others only by means of language*” [54].

The term ‘concept’ is frequently conflated with the term ‘construct’ (discussed next in Sec. 5.2); however, it can be useful to distinguish these as different things. Concepts represent sets of ideas, which are labeled to discuss them or theorize about them. Constructs, in contrast, are used to bring concepts to life, to operationalize them through some form of measurement. Hassan et al. [47] summarize that “*constructs are attempts to make concepts less ‘abstract’ even though constructs are abstractions of concepts.*”

Not all concepts are measurable, and therefore it is not always possible to define meaningful constructs for a concept. A concept can be any “idea that receives a name,” which also includes ‘software development process’ or ‘software product,’ and yet we would not seek to measure these concepts. Instead, we would measure their *properties* [35, p. 40]; for example, rather than measuring the software product, we’d seek to measure its various qualities. In other words, concepts can be much more generic ideas. Constructs are discussed in more detail below (Sec. 5.2); that section also describes one historical reason as to why the terms concept and construct may be conflated, or even considered equivalent by some.

Of all intermediate theorizing products, concepts are perhaps the most straightforward: anything that has been labeled as something worthy of study. Some examples now follow. At the time of writing, a current topic attracting attention within SE is ‘Developer Experience’ (DevEx) [37]. The DevEx concept was first defined by Fagerholm and Münch [37] as: “*how developers feel about, think about, and value their work.*” However, the question then arises as to how to operationalize this definition, and this would require us to develop one, or perhaps more *constructs*.

Another example of a concept that originated in the social sciences is the concept *social capital*, which has also been used in SE research [82, 113]. Social capital has been defined as: “*the goodwill that is engendered by the fabric of social relations and that can be mobilized to facilitate action*” [3]. Nahapiet and Ghoshal [74] argued that social capital has three dimensions: the structural dimension, which includes network ties; the relational dimension, including trust and norms, and the cognitive dimension, which includes shared language. Each of these, social capital, network ties, trust, norms, shared language, are concepts, because they have a place in theoretical reasoning. We can define, teach, and reason about these concepts. To measure these concepts quantitatively, however, we must define constructs, discussed next.

5.2 Construct

Constructs are, in short, mechanisms to allow measurement (observation) of concepts. Ironically, ‘construct’ is the term frequently used to refer to variables that cannot be observed directly. Whereas concepts are the entities that we theorize about, the term construct becomes relevant when we attempt to measure those constructs. The terms construct and concept are frequently conflated; some scholars appear to reject any distinction [88]. Suddaby [115] acknowledges that the term ‘construct’ comes with “*connotations of hypothesis testing and operationalization*,” and that for researchers following a different research paradigm (see footnote 7 in Sec. 3.2), “*concept* might be a more acceptable value-neutral term.” Whether one wishes to draw a distinction or not between ‘construct’ and ‘concept,’ it is important to achieve “construct clarity” (or conceptual clarity), i.e., clearly defined constructs (or concepts) [115].

When we adopt a quantitative research strategy, it is useful to draw the distinction between “construct” and “concept”; to clarify why, I make a brief, though technical, diversion. I do this by drawing on Rigdon’s explanation of what Bagozzi and Philips called the ‘Holistic Construal’ [6, 88].

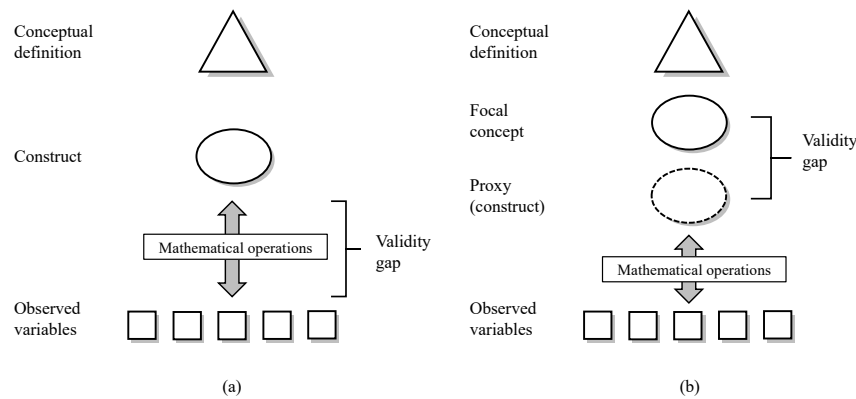


Fig. 4 (a) Simplified representation of the Holistic Construal Framework [6], adapted from Rigdon [88]. (b) Concept proxy framework, adapted from Rigdon [88] that explicitly acknowledges a distinction between concept and construct (proxy).

Figure 4(a) presents a simplified version of the Holistic Construal Framework (adapted from Rigdon [88]), which was first proposed in 1982, at a time when statistical techniques were not yet as well developed as today. The goal of the framework was to link together *theory formulation* at the level of theoretical concepts, and *theory testing* through constructs that represent those concepts.

Research involves theoretical concepts; in behavioral software engineering, such concepts include job satisfaction, trust, personality traits, and a wide range of emotions. These concepts can be clearly defined to convey to readers what the researcher means, to achieve conceptual clarity [115]. To test a theory involving these concepts, one has

to *measure* these; to that end, a researcher would define *constructs*, which are created through some mathematical operation from a set of observed (empirical) variables. At the time the Holistic Construal Framework was first proposed, the dominant way to represent constructs was as reflectively measured *common factors* (see also the chapter by Ralph et al., this volume). Figure 5 shows an example for the construct ‘trust’ (as used in Stol et al. [113]) using standard notation in SEM: ovals represent latent variables, and empirical indicators are represented by rectangles. Reflective measurement refers to the idea that a theoretical, latent variable (e.g., trust) cannot be measured directly, but only *inferred* through a set of indicators. Change in those indicators is *caused* by change in the latent variable, i.e., change in the latent variable is *reflected* in the indicators. The relationship between a common factor and its indicators is expressed as a *loading*, which indicates how much of the variance in the indicator is explained (‘caused’) by the common factor. Normally, not all variance in the indicators is explained by the common factor; the remainder is called the *residual*, and this accounts for measurement error.

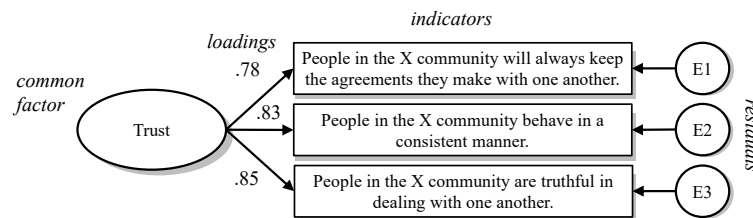


Fig. 5 Trust as a reflectively measured common factor: instead of probing respondents on ‘trust,’ which could involve misinterpretation and thus measurement error, we ask respondents to rate a set of more specific, unambiguous indicators, each of which has a certain loading on the common factor, indicating how much of its variance is explained by the common factor. Residuals explain the remaining variance in the indicators. X is a placeholder for the name of a community, not the social media platform formerly known as Twitter.

The apparent ‘natural fit’ between theoretical concepts on the one hand, and their measurement through constructs represented by common factors contributed to a “blurring” of the terms concept and construct [88]. The lack of distinction has practical implications, because this implies that constructs *are equal to* the concepts they represent. Under this assumption, ‘construct validity’ is then the extent to which the construct is properly constructed from observed variables.

However, Rigdon [88] has argued that the implications of equating concept with construct are untenable. Doing so would equate a concept with a common factor, the error terms on the indicators would perfectly capture measurement error, and that somehow, whatever conceptual definition the researcher had in mind was *exactly* captured by the common factor (i.e., the construct). Rigdon argued that the common factor is merely one approximation of whatever focal concept a researcher may have in mind, and that other representations are possible. He proposed the Concept Proxy Framework as a modified version of the Holistic Construal Framework (Figure 4(b)),

which explicitly acknowledges a distinction between the focal concept as “idealized and out of reach,” and a *proxy* that is an “empirical stand-in for the theoretical concept” [88]. Like the construct in the Holistic Construal, the proxy can be created through a mathematical operation. From this perspective, the proxy may be modeled as a common factor or as something else, for example, a composite (a weighted combination of indicators) such as those in Partial Least Squares (PLS) SEM [88, 94]. The question of validity then shifts to the extent to which the proxy is sufficiently similar to the theoretical concept that it purports to represent.

In sum, the distinction between the terms concept and construct is useful, because in quantitative studies we must operationalize the concepts we study; we do so by means of constructs. Whether the construct is statistically modeled as a common factor, or as a composite depends on many factors beyond the scope of this chapter, but whichever mathematical operation is selected, it is important to draw a distinction between the concept as something that is clearly defined and accept that it cannot be perfectly measured, and the construct that is the ‘empirical stand-in’ to represent that concept.

5.3 Statement

Statements are the fundamental building blocks of theorizing [47]. In this context, the term statement does not refer to just any sequence of grammatically ordered words [47]. Instead, it should be seen as an expression that makes sense within a given scientific discipline, i.e. it is relevant to what Hassan et al. [47], citing Foucault [38], refers to as scientific communication that makes up a *discourse*. For example, the statement “Continuous delivery improves software quality” makes only sense within the context of SE research.¹⁸ Statements are claims that may or may not be presented with evidence. It is up to the researcher to (a) identify or develop the statement, and then (b) offer evidence in support of that statement.

While concepts (Sec. 5.1) and constructs (Sec. 5.2) are building blocks that represent phenomena and things of interest, statements (and also hypotheses, discussed below) can be made to discuss those concepts and constructs in relation to one another.

A special type of statement is a proposition, which can be described as a “truth statement,” a “prediction” [35], or assertions about concepts [47]. Hassan et al. [47] distinguish different types of propositions, depending on the type of assertion. An *existential* proposition is a nonrelational proposition that simply declares the existence of a concept. This also includes typologies and taxonomies [47, 45, 84]. A *definitional* proposition goes beyond claiming the mere existence of a concept, and unpacks the concept, providing a definition and characterization. A good example of this is work by De Boer et al. [32, 31]. They observed that, while many researchers discussed architectural knowledge, it remained “a fuzzy concept.” Thus, while there was general agreement on the *existence* of the concept ‘architectural knowledge,’ they attempt

¹⁸ Note this is an example statement, not necessarily a universal fact.

to unpack *what architectural knowledge is* by drawing on different definitions and identifying its constituent elements.

Propositions should be made based on “logical links” rather than just any unjustified claim [47, 109]. This is not to say that it’s not possible to present a proposition that challenges the current state of knowledge (or assumptions held within a field); rather, such a challenge should come with a *rationale*, a line of argument, a reasoning. Thus, the researcher should offer a reasonable derivation that leads up to the proposition. At the same time, there is little point in offering trivial propositions that have already been demonstrated by others, or in other fields. While any number of propositions can be proposed, Dubin suggests to aim for parsimony by focusing on *strategic propositions* [35, p. 168]; that is, ignore the trivial propositions that do not add much insight, and focus on those propositions “*where something notable is happening*” [35, p. 169].

5.4 Hypothesis

Propositions (discussed above) are statements at the conceptual or theoretical level, but in order to test them, concepts must be operationalized using empirical indicators (and/or constructs, see Sec. 5.2) [109, 47]. The result is a hypothesis. When operationalizing a proposition, it is possible that multiple hypotheses are necessary; for example, when hypothesizing about program size, one may need to use multiple metrics to operationalize this concept. These could include lines of code, number of statements, number of classes (when using object-oriented languages), or memory footprint.

Sometimes the term ‘hypothesis’ is used in a more casual way, as if it were offered at a cocktail party. For example, closely related to Conway’s Law (discussed earlier) is the “Mirroring Hypothesis,” which can be stated as: “*Loosely-coupled organizations will tend to develop products with more modular architectures than tightly-coupled organizations*” [69, 25]. The form of this statement is more like a proposition than a hypothesis, because it only references concepts such as “loosely-coupled organizations” and “modular architectures.” To actually test this hypothesis, we’d need empirical indicators (constructs); how would one measure the modularity of an architecture? For example, as part of their study, MacCormack et al. [69] operationalize a dependency between modules as a function call; if a module A calls a function in module B, then A depends on B.

Table 4: Theory components

Product	Description	Example in SE
Concept	A name given to an idea or set of ideas so that it can be reasoned about.	Developer Experience (DevEx) first proposed as a concept by Fagerholm and Münch [37], and further studied by others [76].

Product	Description	Example in SE
Construct	A proxy that is an empirical stand-in for a theoretical concept [88]; these can be modeled as a common factor (when using covariance-based SEM), a composite (when using PLS-SEM), or other forms such as sum scores, averages, etc. In SE, it is not uncommon to use singular observed variables as a proxy for a concept.	Trust [112]; social capital [113], and social capital [82]
Statement	A description relating to one or more concepts that is derived from logical reasoning, prior theoretical insights or literature, or empirical evidence. Includes propositions, typologies, and taxonomies.	Statements include propositions (e.g. Crowston's propositions on work practices in open source software [27]), and typologies and taxonomies [84] (e.g. Lehman's SPE classification [65].)
Hypothesis	A statement that has been operationalized so that it can be empirically tested.	Many papers in the top conference in SE feature hypotheses; specific examples can be found in [113], e.g.: The more a developer trusts other members in the inner source community, the higher that developer's job satisfaction.

Exercises

1. Discuss the difference between a construct and a concept. How do they differ?
2. Discuss the difference between a statement and a hypothesis. How do they differ?
3. Consider a topic (phenomenon) of interest, for example, your current research topic. What are some of the key concepts that appear to be important that have received attention in prior work on this topic? Likewise, are there any defined constructs?
4. Continue with the same topic; what are some key relationships that appear to be important? What are some well-known assertions, propositions, or hypotheses that appear in the relevant literature?

6 Theorizing Products in Action: The Case of Software Architecture

The theorizing products presented above are drawn from Information Systems scholars [47]. One could question, or even object, that such a theoretical focus is not applicable

to a practical field like SE, a field that some perceive as an engineering field. To highlight the value of these theorizing products, I now demonstrate how theorizing takes place within the software architecture field, an area of research that is firmly rooted within the SE domain. I adopt Hassan et al.'s approach [47] to illustrate how studies move the field forward by leveraging one theorizing product, to move to another theorizing product (Fig. 6). The presentation below should not be construed as a complete and necessarily accurate presentation of how theorizing *actually* took place. Further, while the various studies all relate to the software architecture field, the studies used to illustrate the method link to different topics, including software architecture as a concept and architectural knowledge. Reflecting on how an area of research evolved using the 12 theorizing products as a lens can help to increase awareness of theorizing so that future researchers can also learn how to reflect on a field.

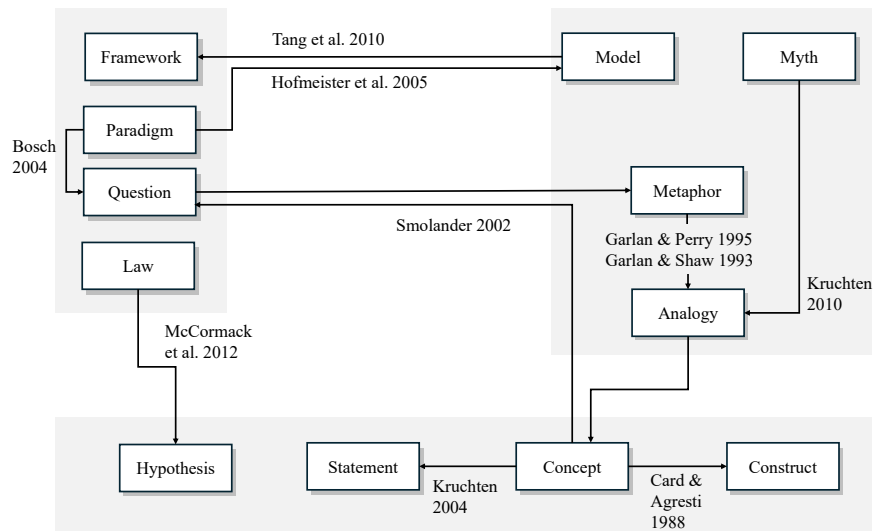


Fig. 6 Example map of products of theorizing in the study of software architecture. This map is only one illustrative example of the author's (possibly naive) interpretation of how studies relate to one another. The author acknowledges this map is incomplete. Further, in creating a map, any theorizing product can be linked to any other. Creating a more complete map can be a useful yet time-consuming way to learn about a research area

6.1 From Law to Hypothesis

Among the early literature concerned with the structure and decomposition of software, years before the term software architecture was coined, is an essay by Conway who suggested that the structure of a system will follow the communication structures

that exist within the organization that produces that system [26]. This has been labeled Conway’s Law in the context of software systems, but similar observations were made by others, and this has become more generally known as the “mirroring hypothesis.” MacCormack et al. phrased this as [69]: “*Loosely-coupled organizations will tend to develop products with more modular architectures than tightly-coupled organizations.*”

6.2 From Metaphor to Analogy to Concept

The software architecture sub-field has deep roots dating back to the very early days of software engineering, but started gaining traction in earnest in the mid to late 1980s. Garlan and Perry [40] noted that one of the trends that prompted the attention for software architecture:

For example, the box and line diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a “pipeline,” a “blackboard-oriented design,” or a “client-server system.”

Pipeline, blackboard, and client-server are all metaphors that convey a particular architectural style, the overall structure of a system. There was an increasing recognition that there is such a thing as software design at the system level, rather than at the detailed level. Up to that point, detailed design would have been captured using dataflow diagrams [125], for example. One of the fundamental papers introduced software architecture as follows [41]:

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. This is the software architecture level of design.”

Thus, an analogy was drawn to the high-level structure of a system, as suggested by the metaphors representing the architectural styles, similar to the idea that one can observe different architectural styles in buildings throughout the ages, and across the world. The term ‘architecture’ is a metaphor¹⁹ for high-level design of a software system, and quickly found its way through into software companies, for example, in the naming of the role software architect.²⁰ By now, the term ‘architecture’ (in the context of software systems) has become so common, it is a ‘dead’ metaphor [42].

¹⁹ ‘Architecture’ is the metaphor, not the term ‘software architecture,’ because prior to the coining of the term, it didn’t exist. The same applies to ‘technical debt’: debt is the metaphor.

²⁰ The analogy between building systems and construction has not gone without critique [8]. I agree that the metaphor is not perfect: software architects would normally have a strong development background, whereas builders don’t usually become architects. Another point where the analogy with construction breaks down is the fact that architects in construction do, in fact, focus on details, aesthetics, and elegance. For example, architects would design ‘hidden’ gutters and downpipes as these essential elements would take away from the aesthetics of a building when visible.

6.3 Deriving Constructs from Concept

Card and Agresti [24] suggest that “*architectural design complexity derives from two sources: structural (or intermodule) complexity and local (or intramodule) complexity.*” In effect, Card and Agresti derived two constructs from the concept of architectural design complexity, namely the structure of the system as a whole, and the complexity of individual components; each of these constructs is measured using specific metrics. While the definition of constructs is not a theory, defining constructs is essential to make any ‘theory’ (in the broadest sense of the word) of software architecture testable.

6.4 Exploring Concept by Metaphor

While software architecture had emerged as a concept, mostly as a level of design that went beyond algorithms and data structures, the question ‘What is Software Architecture’ did not have a clear answer. Whereas there have been numerous working definitions, Smolander [103] observed in 2002 that the meaning of the concept remained ambiguous and that academic and industry perspectives diverged [17]. Around that time, the Software Engineering Institute (SEI) maintained a list of close to a hundred different definitions of software architecture. Smolander investigated how software companies perceived software architecture, and identified four metaphors: architecture as a blueprint, architecture as literature, architecture as language, and architecture as decision, representing the different roles that software architecture plays in practice. The last definition, architecture as decision, forebode a paradigm shift, discussed next.

6.5 Shift in Paradigm Leads to Shift in Questions

The software architecture as a topic of interest has emerged and evolved rapidly over the past few decades. Initially the focus was on technical artifacts, considering architecture as a collection of components and connectors that exhibited a certain structure and behavior (e.g., performance, reliability, and other quality attributes) [23]. Researchers studied how to capture this using different views, using architecture design methods and architecture description languages. Bosch [18], among others, pointed out that this view on software architecture lacks attention for capturing design decisions, and consequently the knowledge about ‘what and how’ is not captured in the software architecture product. This focus on design decisions heralded a paradigm shift; rather than focusing on software architecture as the end-product, the focus was turned to the architectural design process, including capturing design decisions and managing architectural knowledge [23].

As a result, researchers started asking different types of questions. Instead of asking how we can capture different architectural styles or patterns in UML, for example, researchers now were focusing on defining architectural knowledge [32], and how it could be managed.

This second paradigm of software architecture is not a theory, but helped to frame the research done in this field, the questions that researchers ask, and ultimately the theorizing products that come out of that inquiry.

6.6 From Concept to Statement

The ‘design decision’ concept gained a lot of traction within the SA domain as it opened up a new view on what should be captured. Numerous templates have been proposed to capture various attributes of design decisions. Several researchers then started to study the concept of design decisions in more detail by not only proposing templates that capture the various attributes of a design decision, but also taxonomies and ontologies. For example, Kruchten [61] presented an ontology of design decisions, that defines different types of design decisions, and different types of relationships between those decisions. In so doing, Kruchten theorized convincingly, using examples from an air traffic control system, that there are ‘existence’ decisions (a system will have a certain element or artifact), ‘property’ decisions (a system will have a certain enduring trait or quality), and ‘executive’ decisions (decisions about the process, technology, or tools to be used). While the ontology is not a theory, it did help advance the SA field.²¹

6.7 From Model to Framework

A variety of different architecture design methods emerged over time, mostly independently as they were created by different organizations in practice. Observing that these methods vary in terminology and focus, Hofmeister et al. [51] distilled a general model of architecture design. This model very much aligned with the first paradigm of software architecture research, focusing on the software architecture as an end-product. Tang et al. [116] extended this model to capture the full life-cycle of architecture, which includes architecture implementation and maintenance, in recognition of the fact that the architectural knowledge created in the early phases is used later, including when a system has been deployed and must be maintained.

Whereas the extended model captures the full life-cycle of architecture design during which architectural knowledge (AK) can be created and used, Tang et al. [116] then turned their focus on the activities that produce and consume AK. In so doing, they identify 12 different types of activities through which producers and consumers generate and use AK.

²¹ A crude measure of impact is that Kruchten’s paper was cited 420 times at the time of writing.

6.8 Debunking Myth by Analogy

In the earlier days during the rise of agile methods in industry, there was an apparent conflict between elements of the Agile Manifesto and software architecture as a ‘Big Up-Front Design’ [63]. This myth is rooted in a false dichotomy between agile methods and architecture [1], which could be the result from a superficial reading of the Agile Manifesto. Kruchten [63] approached to resolve this mythical conflict by drawing an analogy the ethno-sociological concept of culture and intercultural conflicts, emphasizing that the conflict can be reconciled by improving mutual understanding and foregoing “*a shallow caricatural view of the ‘other culture.’*”

7 Teaching Theorizing: A Practical Guide

This chapter presents a wide range of theoretical concepts (pun intended), and in so doing, scratches only the surface of each of the 12 intermediate theorizing products. Table 5 offers a practical guide to teaching this material; it should be noted that a 12-week course could focus on any of the 12 topics, or just a subset of these, informed by the lecturer’s preferences. It is assumed there are 2 contact hours per week, for a total of 24 hours for the whole semester. This might seem overindulgent given there are so many other topics to teach, but this is, in the end, complicated material that requires considerable engagement lest the effort become meaningless. When faced with less time, rather than attempting to include too much material, I would suggest to focus on a select few themes, and discuss these in-depth.

Whatever the focus and scope is of one or a series of lectures or tutorials, it is important (a) to read one or two papers that set the stage (Table 5 offers useful starting points, but these recommended readings can easily be swapped for others); (b) to apply the material from the paper in one way or another. An effective question is: “Given the reading assignment, what elements could be applied to *my* situation or research interest?” or “what would this look like when applied to *my* research topic?” I have attempted to do this for the Software Architecture field in Section 6.²²

All 12 topics are “rabbit holes”²³, i.e., it is easy to get lost. This is both the magic, but also the curse²⁴ of doing meta-science; it is magical because there is so much more to learn, but it is a curse because there doesn’t appear to be a final answer. Welcome to theorizing!

²² The astute reader will notice that the map has considerable room for improvement.

²³ Question to the reader: a metaphor or analogy?

²⁴ Metaphor or analogy?

Table 5: Sample program for a 12-week course to teach theorizing in SE. Listed activities are examples only to get started, but lecturers are recommended to add or replace contents as they see fit.

Week	Topic	Readings, Questions, Activities
Week 1	Introduction; Metaphor & Analogy	Read Weick on theorizing [121]. Find a few papers in your field of interest that use metaphors or analogies. Which analogies are also metaphors, and why? Which analogies are not, and why?
Week 2	Questions	Read Sandberg and Alvesson [97]. Identify a topic of interest, and identify two scientific disciplines (or sub-disciplines) that study it, for example: SE, IS, CSCW/CHI, management, economics, psychology, etc. Attempt to identify questions asked in each discipline on this topic.
Week 3	Concepts	Identify a concept that has attracted interest in the past several years; examples include productivity, developer experience. What definitions are provided, if any?
Week 4	Constructs	Read Suddaby [115], Graziotin et al. [44], Sjöberg and Bergersen [101], Ralph and Tempero [85]. Advanced reading is Rigdon's work on measurement, e.g. [88]. Discuss the differences between concepts and constructs; discuss construct validity; consider some concept within SE, and find an associated construct to measure it; is the construct a good representative of the concept?
Week 5	Frameworks	Read Leshem and Trafford [68] and Schwarz et al. [98]. Given a substantive research topic or question, attempt to create a (initial) framework to help answer this question.
Week 6	Laws	Read Conway's short article [26], and discuss the strength of evidence that Conway proposed; is the term 'law' appropriate? Did later studies provide support? Identify some other 'law' in SE: attempt to find the origin of the law, and papers that have cited the law to frame a study.
Week 7	Myths	Identify a myth in SE, for example, see Bossavit [19]. Attempt to identify the origin, and a paper that refutes the myth.
Week 8	Models	Consider the model of architectural knowledge by Hofmeister et al. [51] and the extension by Tang et al. [116]; attempt to identify a model within a different field. Discuss the difference between a framework and a model.
Week 9	Paradigms	Read Basili [10] and Kitchenham et al. [60], and compare the two viewpoints; how do Basili and Kitchenham et al. converge or diverge? Consider a research topic/field, and attempt to identify trends over the past 20-30 years, e.g. sampling papers every 4-5 years; what appear to be assumptions within the field? Are there any 'shifts' in focus when comparing earlier versus recent research? Is there a shift in methods used? Attempt to compare early vs. current papers using a table. Can you speak of a paradigm shift?
Week 10	Statements & Hypotheses	Read Crowston et al. [27]. How did they develop the propositions from prior literature? Are these propositions 'strategic' and 'interesting'? If so, why; if not, why not? Read Ralph for more guidance on taxonomies [84].

Week	Topic	Readings, Questions, Activities
Week 11-12	Theorizing Map; wrap-up	Build a theorizing map (see Sec. 6, Figure 6) on a topic of choice, ideally linked to the same concepts. This is time-consuming and challenging, but is an excellent activity to explore historical developments within a field of interest.

Acknowledgments

This work is supported with funding by Science Foundation grant 13/RC/2094-P2 to Lero, the SFI Research Centre for Software.

References

1. Abrahamsson, P., Babar, M.A., Kruchten, P.: Agility and architecture: Can they coexist? *IEEE Software* **27**(2), 16–22 (2010)
2. Adams, P.J., Capiluppi, A., Boldyreff, C.: Coordination and productivity issues in free software: The role of Brooks' Law. In: 2009 IEEE International Conference on Software Maintenance, pp. 319–328. IEEE (2009)
3. Adler, P.S., Kwon, S.W.: Social capital: Prospects for a new concept. *Academy of Management Review* **27**(1), 17–40 (2002)
4. Ågerfalk, P.J., Fitzgerald, B.: Outsourcing to an unknown workforce: Exploring opensourcing as a global sourcing strategy. *MIS Quarterly* **32**(2), 385–409 (2008)
5. Alvesson, M., Sandberg, J.: Generating research questions through problematization. *Academy of Management Review* **36**(2), 247–271 (2011)
6. Bagozzi, R.P., Phillips, L.W.: Representing and testing organizational theories: A holistic construal. *Administrative science quarterly* pp. 459–489 (1982)
7. Baragry, J.: Understanding software engineering: From analogies with other disciplines to philosophical foundations. Ph.D. thesis, School of Engineering (2000)
8. Baragry, J., Reed, K.: Why we need a different view of software architecture. In: Proceedings Working IEEE/IFIP Conference on Software Architecture, pp. 125–134. IEEE (2001)
9. Basili, V.R.: The experimental paradigm in software engineering. In: H.D. Rombach, V.R. Basili, R. Selby (eds.) *Proceedings of Dagstuhl-Workshop, Lecture Notes in Computer Science*, pp. 1–12. Springer-Verlag (1993)
10. Basili, V.R., Zelkowitz, M.V.: The software engineering laboratory: Objectives. In: Proceedings of the fifteenth annual SIGCPR conference, pp. 256–269 (1977)
11. Beecham, S., OLeary, P., Richardson, I., Baker, S., Noll, J.: Who are we doing global software engineering research for? In: 2013 IEEE 8th international conference on global software engineering, pp. 41–50. IEEE (2013)
12. Berente, N., Recker, J.: The blank page problem. In: *This IS Research Podcast* (2024). 17 April 2024
13. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: *Future of Software Engineering (FOSE'07)*, pp. 85–103. IEEE (2007)
14. Bird, C.: Conway's law. *Making Software: What Really Works, and Why We Believe It* p. 187 (2010)
15. Boehm, B.W.: Software engineering economics. *IEEE Transactions on Software Engineering* **SE-10**(1), 4–21 (1984)

16. Bonaccorsi, A., Rossi, C.: Why open source software can succeed. *Research policy* **32**(7), 1243–1258 (2003)
17. Bosch, J.: *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education (2000)
18. Bosch, J.: Software architecture: The next step. In: *Software Architecture: First European Workshop, EWSA 2004*, St Andrews, UK, May 21–22, pp. 194–199. Springer (2004)
19. Bossavit, L.: *The Leprechauns of software engineering*. Lulu (2015)
20. Bowman, I.T., Holt, R.C.: Software architecture recovery using conway’s law. In: *CASCON*, vol. 98, p. 6 (1998)
21. Briand, L.: Embracing the engineering side of software engineering. *IEEE software* **29**(4), 96–96 (2012)
22. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. *IEEE computer* **20**(4), 10–19 (1987)
23. Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Babar, M.A.: 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software* **116**, 191–205 (2016)
24. Card, D.N., Agresti, W.W.: Measuring software design complexity. *Journal of Systems and Software* **8**(3), 185–197 (1988)
25. Colfer, L.J., Baldwin, C.Y.: The mirroring hypothesis: theory, evidence, and exceptions. *Industrial and Corporate Change* **25**(5), 709–738 (2016)
26. Conway, M.E.: How do committees invent. *Datamation* **14**(4), 28–31 (1968)
27. Crowston, K., Annabi, H., Howison, J., Masango, C.: Effective work practices for software engineering: free/libre open source software development. In: *Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research*, pp. 18–26 (2004)
28. Cruzes, D.S., Dybå, T.: Research synthesis in software engineering: A tertiary study. *Information and Software Technology* **53**(5), 440–455 (2011)
29. Cunningham, W.: The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger* **4**(2), 29–30 (1992)
30. Curran, P.: Model based thinking. *Quantitude: A Useful Podcast* **S2**(E05) (2020)
31. De Boer, R.C., Farenhorst, R.: In search of ‘architectural knowledge’. In: *Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge*, pp. 71–78 (2008)
32. De Boer, R.C., Farenhorst, R., Lago, P., van Vliet, H., Clerc, V., Jansen, A.: Architectural knowledge: Getting to the core. In: *Software Architectures, Components, and Applications: Third International Conference on Quality of Software Architectures, QoSA 2007*, Medford, MA, USA, July 11–23, pp. 197–214. Springer (2007)
33. De Souza Santos, R., Ralph, P., Arham, A., Stol, K.J.: Distributed scrum: A case meta-analysis. *ACM Computing Surveys* **56**(4) (2024)
34. Dittrich, Y.: What does it mean to use a method? towards a practice theory for software engineering. *Information and Software Technology* **70**, 220–231 (2016)
35. Dubin, R.: *Theory building*, revised edn. The Free Press (1978)
36. Endres, A., Rombach, H.D.: *A handbook of software and systems engineering: Empirical observations, laws, and theories*. Pearson Education (2003)
37. Fagerholm, F., Münch, J.: Developer experience: Concept and definition. In: *International Conference on Software and System Process (ICSSP)*, pp. 73–77. IEEE (2012)
38. Foucault, M.: *The Archaeology of Knowledge and the Discourse on Language*. 1971. Pantheon Books (1972)
39. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it’s hard to build systems out of existing parts. In: *Proceedings of the 17th international conference on Software engineering*, pp. 179–185 (1995)
40. Garlan, D., Perry, D.E.: Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering* **21**(4), 269–274 (1995)
41. Garlan, D., Shaw, M.: An introduction to software architecture. In: *Advances in software engineering and knowledge engineering*, pp. 1–39. World Scientific (1993)

42. Giles, T.D.: The missing metaphor. *Journal of technical writing and communication* **31**(4), 373–390 (2001)
43. Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P.: What happens when software developers are (un) happy. *Journal of Systems and Software* **140**, 32–47 (2018)
44. Graziotin, D., Lenberg, P., Feldt, R., Wagner, S.: Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Transactions on Software Engineering and Methodology* **31**(1), 1–36 (2021)
45. Gregor, S.: The nature of theory in information systems. *MIS Quarterly* **30**(3), 611–642 (2006)
46. Hall, J.N.: Perl: Internet duct tape. *IEEE Internet Computing* **3**(4), 95–96 (1999)
47. Hassan, N.R., Lowry, P.B., Mathiassen, L.: Useful products in information systems theorizing: A discursive formation perspective. *Journal of the Association for Information Systems* **23**(2) (2022)
48. Hassan, N.R., Mathiassen, L., Lowry, P.B.: The process of information systems theorizing as a discursive practice. *Journal of Information Technology* **34**(3), 198–220 (2019)
49. Herbsleb, J.D., Grinter, R.E.: Splitting the organization and integrating the code: Conway’s law revisited. In: *Proceedings of the 21st international conference on Software engineering*, pp. 85–95 (1999)
50. Hirschheim, R., Newman, M.: Symbolism and information systems development: myth, metaphor and magic. *Information Systems Research* **2**(1), 29–62 (1991)
51. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: Generalizing a model of software architecture design from five industrial approaches. In: *5th Working IEEE/IFIP Conference on Software Architecture*, pp. 77–88. IEEE (2005)
52. Hsia, P., Hsu, C.t., Kung, D.C.: Brooks’ law revisited: A system dynamics approach. In: *Proceedings Twenty-Third Annual International Computer Software and Applications Conference*, pp. 370–375. IEEE (1999)
53. Huber, G.P., Van De Ven, A.: *Longitudinal field research methods: Studying processes of organizational change*, vol. 1. Sage (1995)
54. Jackendoff, R.: What is a concept, that a person may grasp it? 1. *Mind & language* **4**(1-2), 68–102 (1989)
55. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: *5th Working IEEE/IFIP Conference on Software Architecture*, pp. 109–120. IEEE (2005)
56. Joblin, M., Apel, S.: How do successful and failed projects differ? a socio-technical analysis. *ACM Transactions on Software Engineering and Methodology* **31**(4), 1–24 (2022)
57. Johnson, P., Ekstedt, M., Jacobson, I.: Where’s the theory for software engineering? *IEEE software* **29**(5), 96–96 (2012)
58. Jørgensen, M.: Myths and over-simplifications in software engineering. *Lecture Notes on Software Engineering* **1**(1), 7 (2013)
59. Kendall, J.E., Kendall, K.E.: Metaphors and methodologies: Living beyond the systems machine. *MIS Quarterly* pp. 149–171 (1993)
60. Kitchenham, B.A., Dyba, T., Jorgensen, M.: Evidence-based software engineering. In: *Proceedings. 26th International Conference on Software Engineering*, pp. 273–281. IEEE (2004)
61. Kruchten, P.: An ontology of architectural design decisions in software intensive systems. In: *2nd Groningen workshop on software variability*, pp. 54–61. Groningen, The Netherlands (2004)
62. Kruchten, P.: Putting the “engineering” into “software engineering”. In: *2004 Australian Software Engineering Conference*, pp. 2–8. IEEE (2004)
63. Kruchten, P.: Software architecture and agile software development: a clash of two cultures? In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ACM (2010)
64. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. *IEEE Software* **29**(6), 18–21 (2012)
65. Lehman, M.: Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* **68**(9), 1060–1076 (1980). DOI 10.1109/proc.1980.11805. URL <http://dx.doi.org/10.1109/PROC.1980.11805>

66. Lenberg, P., Feldt, R., Wallgren, L.G.: Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and software* **107**, 15–37 (2015)
67. Lerner, J., Tirole, J.: Some simple economics of open source. *The Journal of Industrial Economics* **50**(2), 197–234 (2002)
68. Leshem, S., Trafford, V.: Overlooking the conceptual framework. *Innovations in education and Teaching International* **44**(1), 93–105 (2007)
69. MacCormack, A., Baldwin, C., Rusnak, J.: Exploring the duality between product and organizational architectures: A test of the “mirroring” hypothesis. *Research policy* **41**(8), 1309–1324 (2012)
70. Medawar, P.: Is the scientific paper a fraud. PB Medawar, The threat and the glory. New York: Harper Collins (1990)
71. Miles, M.B., Huberman, A.M.: Qualitative data analysis: An expanded sourcebook. Sage (1994)
72. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology* **11**(3), 309–346 (2002)
73. Morgan, L., Finnegan, P.: Beyond free software: An exploration of the business value of strategic open source. *The Journal of Strategic Information Systems* **23**(3), 226–238 (2014)
74. Nahapiet, J., Ghoshal, S.: Social capital, intellectual capital, and the organizational advantage. *Academy of management review* **23**(2), 242–266 (1998)
75. Nisbett, R.: The geography of thought: How Asians and Westerners think differently... and why. Simon and Schuster (2004)
76. Noda, A., Storey, M.A., Forsgren, N., Greiler, M.: Devex: What actually drives productivity? *Communications of the ACM* **66**(11), 44–49 (2023)
77. Offutt, J.: What is the purpose of publishing? *Software Testing: Verification & Reliability* **21**(4) (2011)
78. Parnas, D.L.: Stop the numbers game. *Communications of the ACM* **50**(11), 19–21 (2007)
79. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering* **SE-12**(2), 251–257 (1986)
80. Payne, C.: On the security of open source software. *Information systems journal* **12**(1), 61–78 (2002)
81. Pohl, K.: Requirements engineering: An overview. Citeseer (1996)
82. Qiu, H.S., Nolte, A., Brown, A., Serebrenik, A., Vasilescu, B.: Going farther together: The impact of social capital on sustained participation in open source. In: 2019 IEEE/ACM 41st international conference on software engineering (icse), pp. 688–699. IEEE (2019)
83. Ralph, P.: Software engineering process theory: A multi-method comparison of sensemaking–coevolution–implementation theory and function–behavior–structure theory. *Information and Software Technology* **70**, 232–250 (2016)
84. Ralph, P.: Toward methodological guidelines for process theories and taxonomies in software engineering. *IEEE Transactions on Software Engineering* **45**(7), 712–735 (2018)
85. Ralph, P., Tempero, E.: Construct validity in software engineering research and software metrics. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, pp. 13–23 (2018)
86. Redwine, S.T., Riddle, W.E.: Software technology maturation. In: International Conference on Software Engineering, pp. 189–200 (1985)
87. Rigby, P.C., German, D.M., Cowen, L., Storey, M.A.: Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology* **23**(4), 1–33 (2014)
88. Rigdon, E.E.: Rethinking partial least squares path modeling: In praise of simple methods. *Long range planning* **45**(5–6), 341–358 (2012)
89. Robson, C.: Real World Research. Blackwell Oxford (2002)
90. Rolland, K.H., Fitzgerald, B., Dingsøyr, T., Stol, K.J.: Acrobats and safety-nets: Problematising large-scale agile software development. *ACM Transactions on Software Engineering and Methodology* **33**(2) (2024)

91. Rombach, H.D., Verlage, M.: *Directions in Software Process Research*, p. 1–63. Elsevier (1995)
92. Royce, W.W.: Managing the development of large software systems: concepts and techniques. In: *Proceedings of the 9th international conference on Software Engineering*, pp. 328–338 (1987)
93. Rudestam, K.E., Newton, R.R.: *Surviving your dissertation*. Sage (1992)
94. Russo, D., Stol, K.J.: PLS-SEM for software engineering research: An introduction and survey. *ACM Computing Surveys* **54**(4) (2021)
95. Ryan, I., Roedig, U., Stol, K.J.: Unhelpful assumptions in software security research. In: *ACM Conference on Computer and Communications Security*. ACM (2023)
96. Sachs, S.: *Laws of software*. URL www.laws-of-software.com
97. Sandberg, J., Alvesson, M.: Ways of constructing research questions: gap-spotting or problematization? *Organization* **18**(1), 23–44 (2011)
98. Schwarz, A., Mehta, M., Johnson, N., Chin, W.W.: Understanding frameworks and reviews: a commentary to assist us in moving our field forward by analyzing our past. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* **38**(3), 29–50 (2007)
99. Shapere, D.: The paradigm concept. *Science* **172**, 706–709 (1971)
100. Shaw, M.: Prospects for an engineering discipline of software. *IEEE Software* **7**(6), 15–24 (1990)
101. Sjøberg, D.I., Bergersen, G.R.: Construct validity in software engineering. *IEEE Transactions on Software Engineering* **49**(3), 1374–1396 (2022)
102. Sjøberg, D.I.K., Dybå, T., Anda, B.C.D., Hannay, J.E.: *Building Theories in Software Engineering*, p. 312–336. Springer London (2008)
103. Smolander, K.: Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In: *Proceedings International Symposium on Empirical Software Engineering*, pp. 211–221. IEEE (2002)
104. Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L.: Code quality analysis in open source software development. *Information systems journal* **12**(1), 43–60 (2002)
105. Stol, K.J., Avgeriou, P., Babar, M.A., Lucas, Y., Fitzgerald, B.: Key factors for adopting inner source. *ACM Transactions on Software Engineering and Methodology* **23**(2) (2014)
106. Stol, K.J., Caglayan, B., Fitzgerald, B.: Competition-based crowdsourcing software development: A multi-method study from a customer perspective. *IEEE Transactions on Software Engineering* **45**(3), 237–260 (2019)
107. Stol, K.J., Fitzgerald, B.: Researching crowdsourcing software development: perspectives and concerns. In: *Proceedings of the 1st International Workshop on CrowdSourcing in Software Engineering*, pp. 7–10 (2014)
108. Stol, K.J., Fitzgerald, B.: Two’s company, three’s a crowd: a case study of crowdsourcing software development. In: *Proceedings of the 36th International Conference on Software Engineering*, pp. 187–198 (2014)
109. Stol, K.J., Fitzgerald, B.: Theory-oriented software engineering. *Science of Computer Programming* **101**, 79–98 (2015)
110. Stol, K.J., Fitzgerald, B.: The ABC of software engineering research. *ACM Transactions on Software Engineering and Methodology* **27**(3), 1–51 (2018)
111. Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: a critical review and guidelines. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 120–131. IEEE (2016)
112. Stol, K.J., Schaarschmidt, M., Goldblit, S.: Gamification in software engineering: The mediating role of developer engagement and job satisfaction. *Empirical Software Engineering* (2021)
113. Stol, K.J., Schaarschmidt, M., Morgan, L.: Does adopting inner source increase job satisfaction? a social capital perspective using a mixed-methods approach. *Journal of Strategic Information Systems* **33**(1) (2024)
114. Suddaby, R.: From the editors: What grounded theory is not. *Academy of Management Journal* **49**(4), 633–642 (2006)

115. Suddaby, R.: Editor's comments: Construct clarity in theories of management and organization (2010)
116. Tang, A., Avgeriou, P., Jansen, A., Capilla, R., Babar, M.A.: A comparative study of architecture knowledge management tools. *Journal of Systems and Software* **83**(3), 352–370 (2010)
117. Thomson, G.P.: *The inspiration of science*. Oxford University Press (1961)
118. Trice, H.M., Beyer, J.M.: Studying organizational cultures through rites and ceremonials. *Academy of Management Review* **9**(4), 653–669 (1984)
119. Van de Ven, A.H., Poole, M.S.: Explaining development and change in organizations. *Academy of management review* **20**(3), 510–540 (1995)
120. Wasserman, A.I.: Toward a discipline of software engineering. *IEEE Software* **13**(6), 23–31 (1996)
121. Weick, K.E.: What theory is not, theorizing is. *Administrative science quarterly* **40**(3), 385–390 (1995)
122. Weyuker, E.J.: Empirical software engineering research-the good, the bad, the ugly. In: 2011 International Symposium on Empirical Software Engineering and Measurement, pp. 1–9. IEEE (2011)
123. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. *IEEE Software* **31**(3), 79–85 (2013)
124. Wieringa, R., Daneva, M.: Six strategies for generalizing software engineering theories. *Science of Computer Programming* **101**, 136–152 (2015)
125. Yourdon, E.: *Managing the Structured Techniques*, 4 edn. Yourdon Press (1989)
126. Zhang, Y., Stol, K.J., Liu, H., Zhou, M.: Corporate dominance in open source ecosystems: a case study of openstack. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1048–1060 (2022)