K. Stol, B. Fitzgerald (2015) Theory-Oriented Software Engineering. Science of Computer Programming, vol. 101, pp. 79-98. DOI:10.1016/j.scico.2014.11.010

Theory-Oriented Software Engineering[☆]

Klaas-Jan Stol[*], Brian Fitzgerald

Lero—The Irish Software Engineering Research Centre
University of Limerick, Ireland

Abstract

There has been a growing interest in the role of theory within Software Engineering (SE) research. For several decades, researchers within the SE research community have argued that, to become a 'real' engineering science, SE needs to develop stronger theoretical foundations. However, so far, the role of theory is neither fully appreciated nor well understood in SE research. Without a good common understanding of what theory *is*, what it constitutes in SE research, and the various roles it can play in SE research, it is difficult to appreciate how theory building can help to strengthen SE research. In this paper we discuss the importance of theory and conceptualization, and review the key components that comprise a theory. We then present the Research Path Schema (RPS), which is an adaptation of an analytical framework from the social sciences. The RPS defines a research study as consisting of three components: some phenomenon, system or substance that a researcher is interested in; some technique or method to study that substance; and some form of conceptualization or theory that provides an explanation for, or abstraction of the observations made in a study. Different research studies have a different archetypical 'architecture,' depending on the selection of these three components. Consequently, the role of the conceptualization or theory will be different for each archetypical study design, or selected *research path*. We conclude this paper by outlining a number of implications for future SE research, and argue for a Theory-Oriented Software Engineering research perspective, which can complement the recent focus on Evidence Based Software Engineering.

Keywords: Theory-Oriented Software Engineering, software engineering research, theory fragment, theory building, empirical research, Research Path Schema

1. Introduction

In the last decade or so, there has been an increasing interest in Evidence-Based Software Engineering (EBSE). This paradigm gained significant traction with the seminal paper by Kitchenham et al. in 2004 with the title 'Evidence-Based Software Engineering' [2]. The premise underlying the EBSE paradigm is that SE researchers should conduct studies that generate evidence for practitioners so as to enable them to make well-informed decisions regarding software development techniques, methods and tools. There has been an increasing focus on conducting empirical studies within software engineering, a development referred to as 'empirical software engineering.' This is reflected by a number of dedicated conferences (ISESE,[1] ESEM,[2] EASE[3]) and a specialized journal (*Empirical Software Engineering*). Despite this increased focus on empirical research in software engineering, practice in software engineering is still far from 'evidence-based.'

To organize, aggregate and synthesize empirical evidence, Kitchenham et al. [2] proposed the Systematic Literature Review (SLR) in SE research, borrowing from the medical research domain where Evidence-Based Medicine is well established. In SE research, too, SLR is seeing widespread adoption; one of the SE field's prominent journals (Information and Software Technology (IST)) explicitly solicits SLR submissions. Of the 25 most-cited papers in IST up to February 2014, more than half (13) were SLRs or mapping studies. Systematic reviews can be an effective means for triangulating multiple sources of data (i.e., studies) to answer research questions with a considerable level of confidence.

However, with this strong focus on empiricism, we argue that we cannot see the "theoretical trees" through the woods of evidence. Evidence by itself must always be considered in context—findings from one study done in a certain context often do not apply in other contexts. What is missing is a clear understanding of the role of theory in SE research. We have observed a number of challenges in SE research:

- A lack of appreciation for conceptualization in software engineering research;
- A lack of agreement on what theory is, and is not, in software engineering research;

[☆]This is a revised version of the paper "Uncovering Theories in Software Engineering" presented in the 2nd SEMAT Workshop on a General Theory of Software Engineering [1].

[*]Corresponding author
Email addresses: klaas-jan.stol@lero.ie (Klaas-Jan Stol), bf@lero.ie (Brian Fitzgerald)

[1]International Symposium on Empirical Software Engineering

[2]International Symposium on Empirical Software Engineering and Measurement

[3]International Conference on Evaluation and Assessment in Software Engineering

- A lack of awareness of the purpose and goals of theory in software engineering research.

We briefly discuss each of them. Researchers may not see the need for theorizing, and consider it a task for "philosophers." One question that often arises in this discussion is whether or not software engineering is a branch of computer science or an engineering discipline [3, 4, 5, 6, 7]. The term "Software Engineering" was coined in a provocative way [8], partly because SE researchers at the time felt that the work they were doing was not 'science' but more similar to engineering. Including the term 'engineering' was one attempt to draw the interest of practitioners and involve them in the research so as to learn from them and codify their knowledge. The extent to which this goal has been achieved has been limited thus far—few practitioners read software engineering research articles [9]. We agree that Software Engineering can be considered a branch of engineering, in that software engineers must possess extensive knowledge of sound software design principles. However, such knowledge can be codified in the form of theories about what constitutes a sound design and what does not. One could, for instance, consider the set of software architecture patterns to be a form of theory: some patterns, or architectural styles, are more suitable to achieve certain architectural qualities (e.g., performance, resilience) than others. This 'theory' can inform practitioners in designing or evaluating a software system [10]. We address this point in more detail in Section 2.1.

Secondly, there is no common agreement on what theories should look like in SE [11]. Researchers may not be familiar with theorizing, perhaps due to the fact that it was not a part of their research training. As a result, researchers may not have a good understanding of what constitutes theory, its role in research studies, and how to recognize it. The SEMAT [4] initiative aims to define a *General Theory for Software Engineering* that can serve as a foundation for software engineering research. As Bourgeois [12] wrote about behavioral theory, we believe that SE research is too immature for an all-inclusive unifying general theory, and that development of so-called "middle-range" theories is an important step towards maturity of SE research. Merton [13, p. 38] referred to these as

> lying between the minor but necessary working hypotheses that evolve [...] in [...] day-to-day research and the all-inclusive systematic efforts to develop a unified theory.

Section 2.2 elaborates further on this point.

The usefulness of theories is widely recognized by other disciplines; after all, *"nothing is so practical as a good theory"* [14], but its importance has not yet been widely recognized in the SE research community. Theories provide a vocabulary for different researchers to discuss a topic of study, which helps to put research studies in context and converge towards more focused topics of research. Another important function of theory is to make explanations and understandings of how the world works explicit [15], which makes knowledge transferable. As Gregor pointed out [16, p. 613]:

> theories are practical because they allow knowledge to be accumulated in a systematic manner and this accumulated knowledge enlightens professional practice.

A number of authors have pointed at more mature and established disciplines, such as physics and the social sciences, and argued that SE research also needs to develop theories [17, 18, 19, 20, 21]. Section 2.3 discusses the role and purpose of theory in more detail.

Clearly, an improved awareness of and attention to theory in software engineering will not happen overnight. However, we believe that by gaining a better understanding of the importance and role of theory in software engineering, the community can slowly evolve to adopt, what we term, Theory-Oriented Software Engineering (TOSE), which complements EBSE. Consequently, the purpose of this paper is as follows:

> ***To increase awareness of the importance, purpose and role of theory in software engineering research.***

In Section 3 below, we present an analytical framework adapted from Brinberg and McGrath [22] which we term the Research Path Schema (RPS). The RPS defines a number of 'research paths' that represent archetypical research designs. This framework can be used to better understand the roles of theory in research in general. Using the RPS, it becomes clear that many studies in SE research do not present theories as found in other disciplines, but that the SE literature does offer many *theory fragments*, which are products of what Weick called *theorizing* [23].

The remainder of this paper proceeds as follows. Since the term theory can mean different things to different researchers [24, 23], we discuss the nature, origins, and purpose of theory in Section 2. Section 3 presents the Research Path Schema and illustrates this with some examples. Section 4 uses the RPS to analyse three topics from software engineering research that have been studied from different research perspectives. This is followed by a discussion of the implications for the practice of future software engineering research in Section 5.

2. Theory: Motivation, Definition, Purpose

The use and role of theory is not widely understood and appreciated within SE research. This section provides background information on this topic and extends the discussions of the three challenges briefly discussed in Section 1. Firstly, Section 2.1 outlines the need and importance of conceptualization in research. This is followed by a presentation of what (and what not) constitutes 'theory' in Section 2.2. Section 2.3 summarizes the purpose of science and the goals of theory, followed by an overview of efforts within SE research relating to the use of theory in Section 2.4.

2.1. The Importance of Conceptualization

Software Engineering is a multi-disciplinary field, and as such, research studies are much more varied and heterogeneous than in, say, the natural sciences such as physics. Much of the

[4] Software Engineering Method and Theory (www.semat.org)

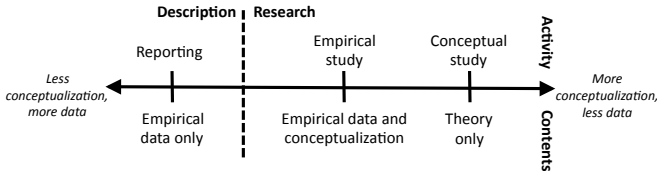

Figure 1: Continuum of conceptualization.

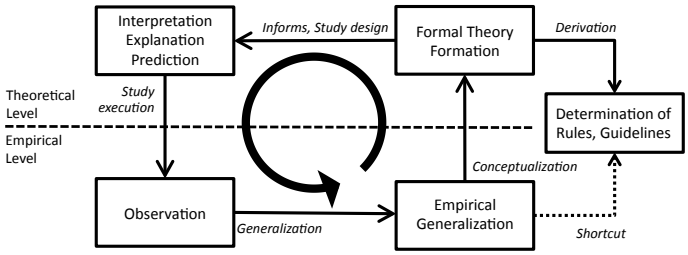

Figure 2: Research cycle.

research in physics is of a quantitative nature, with "standardized" approaches to conduct research and present results. In SE, in contrast, research studies are much more heterogeneous, with a wide range of research approaches, methods and techniques, both quantitative and qualitative. Various research methods, approaches and techniques have been imported from other disciplines, in particular the social sciences; some are more common (survey [25], case study [26], grounded theory [27]) than others (ethnography [28], repertory grid technique [29]).

As a result of this heterogeneity, assessing an SE paper's scientific contribution can be challenging. A common remark found in referees' reports is the question *"So what?"* A paper may present interesting findings, but if these are not further interpreted or conceptualized, the scientific contribution may be insufficient. Dubin [30, p. 16] expressed this sentiment as follows:

> The distinction [between reporting and 'doing science'] lies in whether the information is gathered for its own sake, or whether it is used to measure the values associated with 'things' [...], the relationships among two or more of which is the focus of attention. The first procedure we call description; the second we call research.

In the same vein, Suddaby [31, p. 636] observed that a common problem with Grounded Theory (a research method originating from the social sciences which sees increasing uptake in SE research) is a *"a failure to 'lift' data to a conceptual level."* Nisbett [32, p. 4] described how the early Mesopotamian and Egyptian civilizations made systematic observations, but that only the Greeks made significant progress by explaining their observations in terms of the principles underpinning them. Hall et al. [33], citing Robson [34] argued that *"without theory the research may be easier and quicker, but the outcome will often be of little value."* A lack of conceptualization may also apply to *secondary* studies, such as Systematic Literature Reviews (SLR). In this context, conceptualization can be done through synthesis of findings of a set of so-called *primary* studies. Cruzes and Dybå [35] observed that synthesis of findings in SLRs is often poorly performed.

Figure 1 presents a continuum of conceptualization. The dotted line distinguishes "description" from "research"; merely reporting empirical data without conceptualization is "description" (using Dubin's terminology). Reports that present an empirical study with conceptualization, or a conceptual paper that presents concepts or theory only, are called "research." There is, of course, no clear and hard boundary between the two, as indicated by the dotted line.

It is important to consider the role of 'descriptive' studies, as the figure may suggest that these are not 'research.' Some authors have argued that 'qualitative description' can be a valued end-product on its own, whereby data can be presented without any theoretical or conceptual framework [36]. While we agree that descriptive studies can be valuable (as researchers we have published numerous descriptive studies ourselves), there is always a level of conceptualization or analysis that requires a researcher to move beyond a mere description, or, to declare a starting point (e.g., a framework) that identifies aspects of a topic that are studied. When no explicit theoretical framework or lens is declared, it remains implicit, but it nevertheless exists. Checkland [37, p. 7] captured it well:

> If you are to take part in a change process and learn from it in a research sense...it is essential to declare in advance the intellectual framework in terms of which what counts as learning will be defined...That is a condition for moving beyond the anecdotal.

A report will always present a selection of findings presented in a particular way, and decisions made in preparing such a report are by necessity influenced by an author's preferences, interests or beliefs—and are therefore subjective.

Reynolds [38, p. 43] argued that, *"unless the 'conceptualization' is explicitly described other scientists cannot understand it and probably will not adopt it."* Thus, we argue that conceptualization is an important aspect of good research studies. Conceptualization is closely related to *theorizing*, or the process of building theories [23]. We have observed an increasing level of attention for the role of theory within software engineering research [17, 18, 20]. However, conceptualization and theory building have not been recognized to be as important as empirical research within the SE research community. This lack of attention for theory building is somewhat surprising, given the various calls to do so over the years. For example, Basili and Zelkowitz [39] wrote (emphasis added):

> any future advances in the computing sciences require that the empiricism takes its place **alongside theory formation** and tool development.

In a similar vein, Broy [40, p. 19] argued that:

> engineering disciplines must be based on scientific practices and theory to justify their approaches and to give scientific evidence for why and where their methods work properly.

Figure 2 presents a research cycle (adapted from Lehman and Ramil [41]) that illustrates the role of theory in relation to empirical research, and as such it represents "the flip side" of the

empirical "coin." (Similar diagrams representing the research process are offered by Bunge [42, p. 9], Shaw [43], Carroll and Swatman [44], Lynham [15], and Endres and Rombach [45]. Wieringa et al. [46] presented an *engineering cycle* for the requirements engineering subdiscipline.)

Sutton and Staw cite Kaplan [47] in asserting that *"data describe **which** empirical patterns were observed, and theory explains **why** empirical patterns were observed or are expected to be observed"* [24, p. 374]. Theory not only *informs* the design and motivation of new research studies, but also forms the basis of "rules" and guidelines for practitioners.

What we observe in much of the SE literature, however, is that the formation of formal theory is left implicit or skipped altogether, which is indicated by the "shortcut" in the figure. Such guidelines do not provide the justifications that underpin them – in other words, there is no deep understanding as to the *why* of such guidelines. Chalofsky expressed this well when he wrote: *"Professionalization comes from theory and research: the 'why' instead of the 'how to' "* [48].

One question that may arise is where to start in this cycle of theorizing and empirical investigation. Reynolds [38] describes two approaches: "research-then-theory" and "theory-then-research." The former refers to conducting empirical studies, based on which one develops a theory (also known as the *Baconian* strategy [38, Ch. 7]); the latter starts with a theory that informs the design of a study that can subsequently be executed. Which approach to take depends, of course, on how much theory is available on a particular topic. Nascent research areas would typically take the research-then-theory approach, whereas more mature areas could rely on (and refine) existing theories to further advance the field. Glass pointed out that in software engineering, practice is often ahead of theory [49]; what we observe indeed is scientific research following developments in SE practice. Some examples are the rise of agile methods, global software development and open source software development. All three phenomena emerged in practice, after which the software engineering research community started to conduct exploratory studies, that aimed at establishing an understanding of these topics.

2.2. What Theory Is and Is Not

The question *What is Theory?* has been a topic of much discussion in other disciplines (see e.g., [50]). There are a wide variety of methods and approaches proposed for constructing theories [51, 12, 30, 15, 38, 20, 52]. Weick observed that many descriptions of theory building wrongfully suggest that it is a mechanistic process, *"with little appreciation of the often intuitive, blind, wasteful, serendipitous, creative quality of the process"* [53, p. 519]. Bourgeois argued that such steps are not discrete processes, but that the presentation of a theory may suggest a sequential ordering of thought [12] (see also [54]). We agree that constructing theories is not a linear, sequential process consisting of a number of steps, but that the various activities may occur in parallel. The research cycle depicted in Figure 2 presents a graphical representation of this sentiment, where the activities of theorizing and gathering empirical evidence is an alternating process.

The aim of this section is not to present a final answer to the question *how* theory is constructed, but rather to present a brief overview of the key components that are widely accepted in other disciplines to be a part of theories, as well as to make the reader familiar with the terminology commonly used. In what follows we briefly summarize the key components of a theory. Our discussion is by necessity incomplete—the topic of theory building and theorizing has been addressed by numerous authors in many articles and books. We focus primarily on so-called *variance* theories rather than *process* theories [55]. Variance theories focus on explaining variables and relationships among them whereas process theories explain how a sequence of events or activities result in a certain outcome. Ralph [56, 57], for instance, discusses process theories in the context of software design.

The main elements of a (variance) theory are its **constructs**. In SE research, two example constructs are *program size* and *software quality*. In order to *measure* the size of a software program, one needs to *operationalize* that construct, using a **measure** or **metric**. This can be done through a variety of empirical indicators: lines of code (LOC), memory footprint (during runtime), number of classes (in object-oriented languages), and size of the compiled object code. To operationalize "software quality," one could choose to count the number of known defects, or select a quality attribute (e.g., performance) and operationalize that with performance metrics such as start-up and response time. How well a metric represents the construct affects a study's *construct validity*: does the researcher measure what she intended to measure? Not all constructs are directly measurable; these *hypothetical constructs* [38] may still be useful to build a theory. Wieringa et al. [46] cited "gravity" and "organizational culture" as examples of hypothetical constructs.

A theory also defines the **relations** among constructs and how they interact with one another. These relations may be of different forms, of which causality is perhaps best known and arguably the most interesting. In a SE context, one relation that a researcher could suggest is between program size and software quality, such as, the bigger a software program is, the lower the quality. Theories typically have a limited scope, indicated by their **boundaries**. That is, theories are likely to be only valid under certain conditions. This is directly related to the concept of *generalizability*, or *external validity*.

A theory may have different **states**. Each state may have a different set of laws of interaction that apply only to that state. For instance, certain software development practices (e.g., peer-review) may result in high-quality code in Open Source projects, but only in *popular* projects (in other words, less popular Open Source projects may not achieve the same level of code quality). A theory can transition from one state to another. Some state transitions may be invalid. Weber [52] illustrated this with an example of a theory about human life, which has two states: alive and dead. Whereas the transition from alive to dead is "lawful," the reverse transition is generally "unlawful."

Constructs, relations, boundaries and states are all elements of a theory that must be considered in the activity of building a theory. Once constructed, a theory is put to use. Reichenbach refers to these contexts as *discovery* and *justification*, respec-

tively [58, 59]. To that end, one would define a set of **propositions**; these are concerned with making predictions about a theory's constructs. Propositions are logically derivable from a relation, whereas for the reverse one needs to make an *"inductive leap"* [30, p. 170]. One example of a "theorizing" study that presented a number of propositions is reported by Crowston et al. [60]. Based on the literature of OSS and existing theories, Crowston et al. offered a number of propositions to guide further research. A second example is by Morgan et al. [61], who derived a number of propositions related to creation and capture of value with open source software. Unlike the paper by Crowston et al. mentioned above, Morgan et al. further developed and refined their propositions through a number of empirical case studies.

Hypotheses are to propositions what measures are to constructs. That is, hypotheses (empirical level) are instantiated propositions (theoretical/conceptual level), through the replacement of constructs within these propositions by measures. For instance, to further develop the example given above, a researcher could hypothesize that, as a software program grows in terms of lines of code (*size* construct), the number of defects (*software quality* construct) will increase in a linear fashion. A proposition can therefore have different instantiations, each of which operationalizes the constructs differently.

There is a fine line between what is a theory, and what is not. In particular, Sutton and Staw outline a number of elements that they argue are, by themselves, not theory [24]—we suggest for each how an author could make progress in building a theory:

- **References**; Sutton and Staw argue that references (to prior literature on a topic) are sometimes used as *"a smoke screen to hide the absence of theory."* In order to develop a theory, an author could synthesize the referenced literature and identify the key concepts which can become constructs of a theory-to-be. As mentioned, while systematic reviews in software engineering research are very popular, they often lack in their quality of synthesis, which results in a mere classification of references.

- **Data**; while descriptions can be a source to build theories from, they do not constitute a theory [62]; this corresponds to Dubin's distinction between "description" and "research" discussed above, and the "continuum of conceptualization" in Figure 1. Descriptions can be very useful in new research areas, where phenomena are not well understood or defined. However, there must be some form of structuring in order to organize the presentation of the topic. A common way to do this is by means of an analytical framework. For novel research areas, existing theories or frameworks can be applied, whereas for areas that have already attracted a body of knowledge, a new framework can be derived from that literature.

- **Lists of variables or constructs**; a mere list of concepts and their definitions are what Homans [63, p. 957] described as *"a dictionary of a language that possesses no sentences."* In this case, a researcher should aim at identifying and establishing relationships between the constructs, i.e., *"forming sentences with the words in the dictionary."*

- **Diagrams**; often consisting of "boxes and arrows" [50], they can be helpful in providing structure, but *"Some verbal explication is almost always necessary"* [24]. Indeed, graphical representations are often presented to help a reader to understand a topic in one glance, in particular if the description of the topic at hand (i.e., the theory) is extensive. However, without further description, the "boxes" and "arrows" by themselves do not constitute a theory.

- **Hypotheses**; a mere set of hypotheses *without further justification or clarification* does not constitute a theory. On the other hand, a set of hypotheses (or better still, *propositions*) that are derived from either the literature or a set of empirical findings would be a sound starting point for developing a theory.

While these elements by themselves are not theories, they can be *parts* of a theory. As Weick wrote, arguing that the focus should be on the *process* (of theorizing) rather than the *product*: *"What Theory is Not, Theorizing Is"* [23]. In this context, we use the term *theory fragment*, to refer to something that can develop into a theory. We argue that, while fully developed theories in SE research may be rare, the field has many theory fragments. One of this paper's goals is to show how these can be identified.

2.3. The Purpose of Science and the Goals of Theory

Reynolds [38] discussed five purposes that science should serve: (i) to provide a method to *organize and categorize* things (i.e., to define a typology or taxonomy); (ii) to *predict* future events; (iii) to *explain* past events; (iv) to provide an *understanding* of events, and (v) to potentially *control* events. Reynolds argued that predicting future events (iii) and explaining past events (iv) differ only in a temporal perspective (that is, past v. future) but are similar otherwise.

Once constructed, theories may have different goals, independent of the degree to which a theory has been validated. Gregor [16] presented a taxonomy of theory types observed in Information Systems (IS) research. Previously, this taxonomy was used in an analysis of theory use in SE research [17]. We discuss each type briefly below.

Analysis; says *what is*; provides a description, but no explanation of causality. There is generally disagreement over whether a typology can be labeled as "theory" [24]. Some would disqualify this as theory, arguing that the primary goal of theory is to answer *how*, *when*, and *why* questions, rather than *what* questions [62]. However, on the other hand, when using the term "theory" more freely, typologies are useful in communication and education. For instance, SE students could study the "theory" of software evolution, and learn the different types of maintenance activities as identified by Swanson [64].[5] This would qualify as providing a typology, which is one of the purposes of science

[5] Adaptive, corrective or perfective maintenance.

[38]. Note that a typology differs from a "list of variables or constructs" in that the former links the various viable "values" (e.g, adaptive v. corrective maintenance), whereas the latter may consist of a set of *unrelated* constructs.

Explanation; says *what is, how, why, when, and where*. This type of theory provides explanation (insight) but has no predictive power. Though Reynolds used the word "explanation" in the context of explaining past events, what Gregor [16] meant here is what Reynolds referred to as providing an understanding [38].

Prediction; says *what is, and what will be*. This type of theory provides predictions and testable propositions, but no explanatory power.

Explanation & Prediction; combination of explanation and prediction as described above.

Design and action; says *how to do something*. This type of theory provides prescriptions for constructing artifacts (such as methods and techniques). Theories for design and action have received significant attention in the information systems field [16]. This is an area that has been studied using different labels, of which 'design science' is perhaps the best known. However, there is as of yet no agreement on the role of theory in design science—some authors exclude theory development from design science [65]. Interested readers are referred to some of the seminal papers by Nunamaker et al. [66], March and Salvatore [67], and Gregor and Jones [68].

2.4. Theory in Software Engineering

There is increasing agreement that Software Engineering is not merely a branch of Computer Science [3]. In fact, Offutt [7] wrote that "*Software Engineering is Engineering, Not Science.*" However, we agree with Broy that: *"An engineering discipline without a theory cannot work"* [40]. As pointed out by Offutt, mechanical engineering relies on physics (a traditional field with well-developed theories; a "normal" science as Kuhn would argue [69]). However, there does not seem to be a "primary" or fundamental research field with well-developed theories on which SE can depend. For instance, while mechanical engineers in designing and building structures such as buildings and bridges, can depend on well-defined theories and laws from physics, software engineers do not seem to be able to rely on such theories in SE. Interestingly, there is an increasing attention to social and human aspects in SE, so one potential fundamental field can be the social sciences that have studied team performance, for instance. However, clear laws, rules and theories about how to build reliable, resilient and high-performance software systems are not generally defined nor taught.

There have been a few studies of the use and development of theory in SE research; we summarize these next.

Hannay et al. [17] conducted an SLR on the *use* of theory in software engineering experiments. They found that of the 113 published experiments, 24 studies used a total of 40 theories. A similar study was conducted by Hall et al. [33], who investigated the use of theories in studies of software engineers' motivation. One of their findings was that many of the 92 studies they analyzed were not underpinned by the "classic" theories of motivation that originated in the social sciences.

Endres and Rombach [45] composed an extensive collection of empirical observations, laws and theories. For instance, one law is: *"Well-structured programs have fewer errors and are easier to maintain"* [45, p. 74]. While this law may have some predictive power, there is no justification or explanatory power, and as such practitioners may feel such statements are unsatisfactory.

Sjøberg et al. [20] presented a set of steps to construct theories for the domain of software engineering. In addition, they proposed a template with four archetype concepts: (i) actor, (ii) technology, (iii) activity, and (iv) software system. Furthermore, they proposed a UML-based diagrammatic notation. Shull and Feldmann [70] discussed the construction of theories from multiple and different sources of evidence. This is particularly relevant to SE given the aforementioned heterogeneous nature of research in this field.

Both Sjøberg et al. [20] and Runeson [71] argued that theories must be relevant to practitioners. We disagree with this as an extreme position however since we believe that theory plays an important role in software engineering research, and as such, one purpose of theory is to guide and support further research. So, instead of *practical utility*, a theory may also have *scientific utility* [72]. The researcher's "tools" need not be directly relevant to practitioners. Even theoretical, or "conceptual" research may, in the long run, be useful and relevant to practitioners. Conceptual papers can offer useful points of view, concepts, or analytical frameworks that can help other researchers to revisit a certain topic of study. One example of this is a paper by Jansen and Bosch [73], entitled "Software Architecture as a Set of Architectural Design Decisions," which defined *"the notion of software architecture as the composition of a set of explicit architectural design decisions."* This notion has had considerable impact on the software architecture research community—the paper has more than 300 citations per February 2014, and soon after its publication, other researchers developed tool support for capturing design decisions [74]—which clearly does have practical utility.

3. The Research Path Schema

This section presents the Research Path Schema (RPS), which is the result of our adaptation of the Validity Network Schema (VNS) proposed by Brinberg and McGrath [22]. The VNS, as the name suggests, was originally proposed to explain how the term "validity" has different meanings depending on the type of research study. This term has been a topic of much discussion in the field of consumer research [75], where epistemological considerations have received much more attention than in SE research. The VNS is a very complex and rich framework; however, for our purpose, namely that of improving the way we think about theory in software engineering research, we made a number of changes to simplify the model. In order to be able to refer to this simplified model, we termed this the Research Path Schema so as to clearly distinguish from the model that Brinberg and McGrath originally developed. Clearly, the original principles and ideas underpinning the RPS are de-

rived from the insights by Brinberg and McGrath [22]. The key differences include:

- The VNS focuses on the *validity* of studies, whereas the RPS focuses on the role of theory and conceptualization;
- The research paths in the VNS have been renamed for the RPS so as to prevent ambiguity that could arise in a software engineering context. In particular, the 'experimental' path has been renamed the 'study design' path (as the term 'experimental' may imply the use of the 'experiment' method); the 'empirical' path has been renamed the 'observational' path (as the term 'empirical' could imply that the other paths do not represent empirical research); finally, the 'theoretical' path has been renamed the 'hypothetical' path (as the term 'theoretical' could imply that the other paths may not focus on theories).
- The VNS makes a number of additional assumptions, which we are ignored in the RPS. For instance, the VNS assumes a three-step research process, with the research proper is conducted in step two. These details are not considered in the RPS.

The remainder of this section presents the RPS in more detail.

3.1. Domains of the Research Path Schema

Research designs comprise a number of building blocks, or different types of elements. Brinberg and McGrath [22, p. 14] argued that:

> research involves (a) some content that is of interest, (b) some ideas that give meaning to that content, and (c) some techniques or procedures by means of which those ideas and content can be studied.

These three aspects are referred to as the *substantive*, *conceptual*, and *methodological* domains, respectively. Examples of each domain are presented in Table 1. It should be noted that this definition does not imply or suggest any specific ontological or epistemological stance, a topic that so far has been largely ignored in SE research. The debate regarding questions such as *What is knowledge?* and *How should knowledge be acquired?* (discussed in more detail by Fitzgerald and Howcroft [76]) is irrelevant in the discussion of the RPS. Therefore, dichotomies such as positivism versus interpretivism, qualitative versus quantitative, and exploratory versus confirmatory research need not be considered in this discussion. The choice of research method, theoretical framework and topic of study are orthogonal to the RPS.

3.1.1. Substantive domain

The substantive domain is the domain of phenomena and real-world systems that can be a topic of study. This is the substance that, in the words of Brinberg and McGrath [22, p. 33], *"is 'there' prior to and independent of the intellectual enterprise we call research."* This is the content that a researcher is interested in. In SE research, elements of the substantive domain are, for instance, open source software [77] and developer motivation [78]. Each of these topics are phenomena as found in the real world, and are considered by researchers to be worthy of study. Besides 'phenomena,' that is, trends or developments that can be observed in practice, in software engineering research the substantive domain also includes real-world systems, which could be instances of phenomena. For instance, within the open source 'phenomenon,' one instance is the Linux kernel project, which has been the subject of many research studies [79].

3.1.2. Conceptual domain

Whereas the substantive domain deals with "subject matter," ("substance"), the conceptual domain deals with concepts, models, frameworks, and theories. These conceptualizations are used to describe the properties of, and relationships among the 'things' found in the substantive domain. This domain also also contains any conceptual paradigm that may underpin the research. A conceptual paradigm is a set of paradigmatic assumptions and has an important impact on what a researcher may or may not discover. Van de Ven [55, p. 19] emphasizes the importance of theory in research design as follows:

> Selecting and building a theory is perhaps the most strategic choice that is made in conducting a study. It significantly influences the research questions to ask, what concepts and events to look for, and what kind of propositions or predictions might be considered in addressing these questions.

For instance, Pfleeger [80] pointed out that the model used by nineteenth-century physics was faulty; scientists in that time never *considered* light as an electromagnetic wave, and as a result, they never observed light *particles*. In other words, following Kuhn [69], the conceptual paradigm defines what research problems are considered important to be studied, as well as any expectations with respect to the answer. Within SE research, the conceptual domain includes the models that we build to reason about software systems, or frameworks to analyze real-world systems, or even to analyze research artifacts such as analytical or comparison frameworks.

3.1.3. Methodological domain

The methodological domain of research refers to the methods and techniques to gather data about a study topic (substantive domain) or theories (conceptual domain). Such methods may be "modes of treatment," comparison techniques, or other research methods well known in SE research, such as case studies, surveys, and controlled experiments. Also included in this domain are any research techniques or approaches that a researcher may be interested in, for instance to evaluate its use in a certain setting. For example, Edwards et al. [29] discussed how the Repertory Grid Technique can be used in software engineering research.

3.2. Research Paths

Brinberg and McGrath argued that *"The research process is the identification, selection, combination, and use of elements and relations from the conceptual, methodological, and substantive domains"* [22, p. 16]. Thus, a research study consists

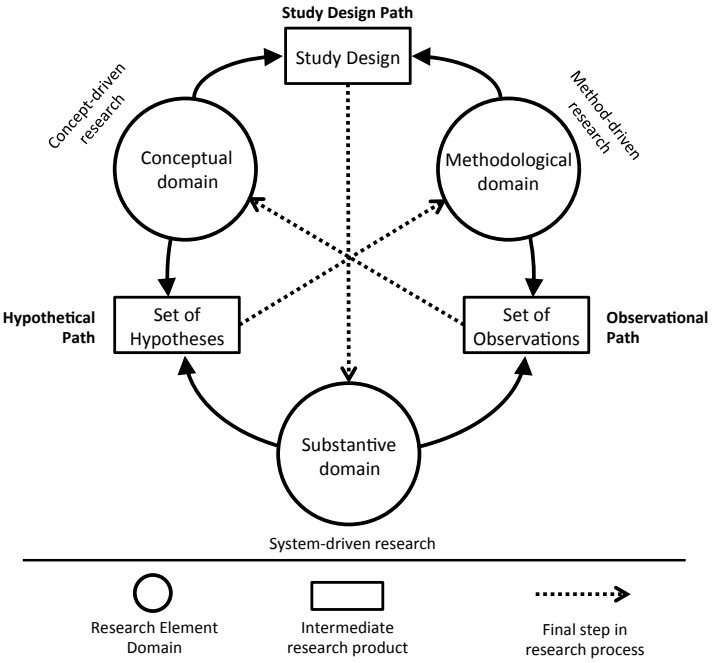

Figure 3: Research Path Schema (RPS)

Table 1: Examples for each of the three domains.

Domain	Examples
Substantive	Phenomena e.g., Open Source software development, crowdsourcing, software architecture; systems e.g., Linux, software development tools (as systems)
Conceptual	Analytical and comparative frameworks, theories, hypotheses, propositions, concepts, abstractions, (mathematical) models, design patterns, Lehman's Laws
Methodological	Case study, survey, experiment, ethnography, repertory grid technique, comparative analysis, instruments, techniques, modes of treatment, content analysis, metrics

of some phenomenon or topic of study, a research method or technique, and a set of concepts or theory. These three elements can be combined in different ways, depending on a study's goal.

Scientific research studies may have different goals; some studies attempt to *generate* new theory (e.g., using a Grounded Theory approach), whereas others attempt to *evaluate* a set of hypotheses based on an existing theory (e.g., using a controlled experiment). Others still seek to demonstrate the value of a certain method or technique within a certain domain (e.g., Tofan et al. [81] proposed using the Repertory Grid Technique to capture tacit knowledge of software architecture). As a result, a researcher designing a study will typically have a particular and primary interest in one of the three domains discussed above. The domain of the researcher's primary interest defines the type of research.

In other words, the *order* in which the elements are chosen defines which *research path* a researcher takes. Brinberg and McGrath identified three main *research paths*, which they labeled the *experimental*, the *theoretical*, and the *empirical* path, which reflect different goals of a study. As mentioned, we renamed these three paths as *study design path*, *hypothetical path* and the *observational path*, respectively, to eliminate possible ambiguity in the SE research context. Each research path has two variants, depending on which domain is of a researcher's primary interest. Figure 3, adapted from Brinberg and McGrath [22], shows the three research paths and their variants. For example, a study following the *observational* path can be either *method-driven* or *system-driven*, depending on whether a researcher's primary interest lies in the methodological domain or substantive domain, respectively.

The use of the different paths and research orientations is discussed and exemplified further below. When analyzing a research study, one is *"necessarily making presumptions about what was in the minds of the researchers"* [22, p. 61]. That is, one can never know the *actual* steps the a researcher took to undertake a certain study.[6]

3.2.1. Study Design Path

The goal of the study design path is to build a study design, and use it on one or more elements of the substantive domain. The study design is comprised of a set of concepts or a theory on the one hand, and a research method or technique on the other hand. If the primary interest is based in the conceptual domain, then the study is *concept-driven*, while if the primary interest is based in the methodological domain, then the study is *method-driven*. The last element to add to complete

[6]Clearly, the RPS should not be considered as merely a mechanism to categorise a study correctly—instead, it should be considered as a way to reflect on the role of theory in software engineering research. It can also be helpful to students of software engineering in designing their research.

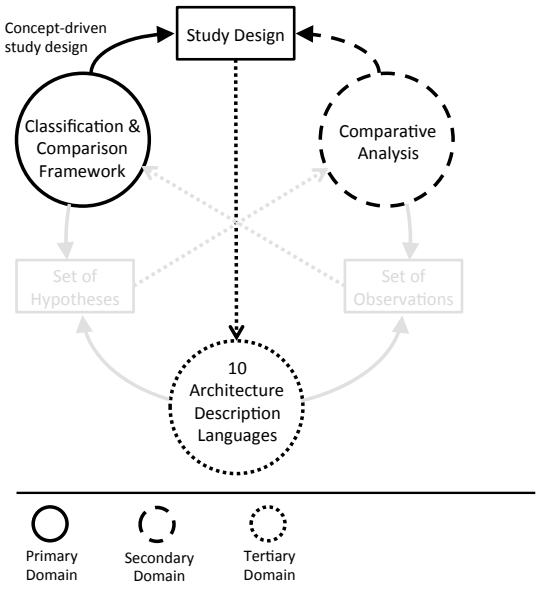

Figure 4: Research Path followed by Medvidović and Taylor [82].

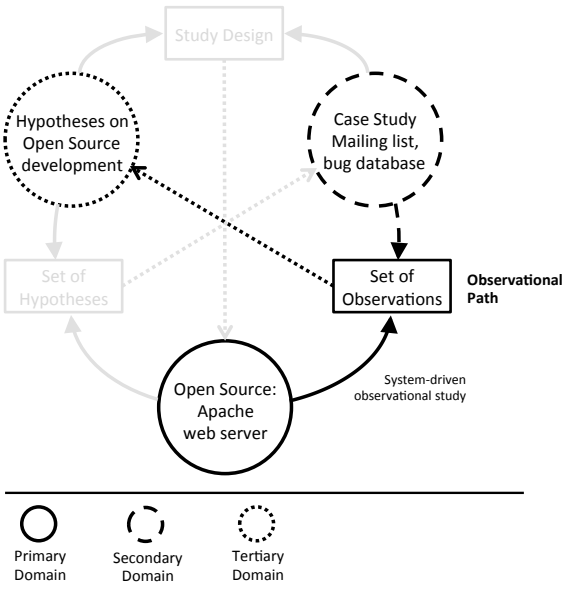

Figure 5: Research path followed by Mockus et al. [84].

the study is a real-world system or phenomenon (i.e., the substantive domain). A common scenario in SE research is that a researcher has a conceptual model or framework, and develops a technique (or tool) to implement or support this. In this scenario, the substantive domain has least "importance"; the implementation is the result, which serves as a validation of the researcher's proposed idea. One example of a study that followed the (concept-driven) study design path is by Medvidović and Taylor [82]. They developed a classification and comparison framework for architecture description languages (ADL). The framework represents a conceptual lens, as it represents a set of properties of ADLs that Medvidović and Taylor deemed important. Medvidović and Taylor speak of a taxonomy, which which could be considered a theory with an "analysis" purpose only (see Section 2.2), or a *typology* [38]. Once defined, the next step was to select a technique to compare—what in this case could be called a comparative analysis. The combination of the comparison framework and the comparative analysis is a study design, which could then be applied to instances from the substantive domain—a set of ADLs. The choice of ADLs was presumably of least interest—numerous ADLs have been defined, and Medvidović and Taylor selected ten ADLs for their comparison. The research path followed for this study is illustrated in Figure 4.

3.2.2. *Observational Path*

The goal of the observational path is to collect a set of observations, and to explain them in terms of a set of meaningful concepts. In other words, a researcher starts with a topic of interest (substantive domain) and a research method (or technique). The result will be a *set of observations*. The next step is to *interpret* these observations. One goal may be to generate a set of concepts (or theory), using for instance a Grounded Theory approach [83]. Alternatively, the set of observations (resulting from a case study, for instance) may be interpreted using an existing set of concepts or theory that was developed prior to the study. However, the researcher's primary interest is in either the substantive domain (a phenomenon such as open source software) or a method or technique.

One study that followed this path is that by Mockus et al. [84] which had a primary interest in the substantive domain (see Figure 5). Their *system-driven* study focused on the open source development phenomenon which had emerged as a popular research topic in the late nineties, and in particular the Apache web server project was selected as a representative of the open source phenomenon. The second step in their research was to select appropriate methods to study this project. Based on their findings, Mockus et al. posed a number of hypotheses, representing the conceptual domain of the study. While their proposed hypotheses represent a significant contribution to the research on open source, their research design seems to have started from the substantive domain.

3.2.3. *Hypothetical Path*

The hypothetical path refers to research that seeks to *test* theory rather than build it. In particular, hypotheses can originate in the substantive domain (system-driven research) or from a theory or model (concept-driven research). In the case of evaluating a set of hypotheses to test a certain theory, a researcher's primary interest is, of course, the theory being tested (the conceptual domain). The researcher will then select an appropriate real-world situation, phenomenon, or system (from the substantive domain) to gather data. Finally, an appropriate research method or technique is selected; while the choice of a suitable method (methodological domain) is important to achieve valid results, the choice of method will be guided (restricted) by the research situation, and is therefore of "least" interest. In other words, a researcher typically will not start with a research method in mind—say, controlled experiment—when evaluating a set of hypotheses. Rather, the main interest is in investigating the hypotheses. For example, Mockus et al. followed up their study of

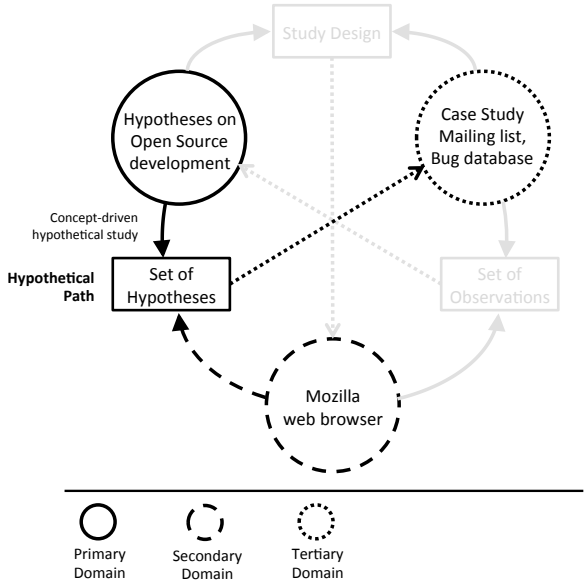

Figure 6: Research path followed by the follow-up study by Mockus et al. [85].

the Apache web server with a study of the Mozilla web browser [85]. In this follow up study (see Figure 6), they shifted their primary interest from the substantive domain (i.e., Apache) to the conceptual domain. The set of hypotheses they posed based on their findings of their initial study was now the primary focus. To see whether their hypotheses would hold (i.e., suggesting a *concept-driven* hypothetical path), they selected a second open source project to study (the Mozilla web browser). Finally, they selected appropriate methods to collect the data to test their hypotheses.

4. Alternative Research Perspectives: Three Examples

To demonstrate the use and benefits using different research orientations, we follow an approach by Brinberg and Hirschman [86] by presenting exemplar studies for each domain—a conceptual schema, a methodology or technique, and a system or phenomenon. We selected topics that are relevant in the software engineering context: Lehman's Laws (conceptual domain), object-oriented metrics (methodological domain), and software architecture (substantive domain). For each topic, we present studies that represent three different perspectives.

4.1. Lehman's Laws

Lehman's laws of software evolution [87] are among the best known 'theories' of software engineering.[7] These laws have evolved themselves over the years [88]—an extensive presentation of their history has been presented by Herraiz et al. [89]. Since their first publication, there have been many studies investigating these laws.

[7] Lehman and colleagues have argued that the term 'law' should be interpreted in the domain of the social sciences, rather than to use them to expect 'precise invariant relationships of measurable observations' [88].

Lehman's Laws originated from a study of the OS/360 operating system by Belady and Lehman [90], which is a clear exemplar of a *system-driven* observational study, whose basis lies in the substantive domain—specifically, the study's focus was the evolution of the OS/360 system. The authors stated:

> Starting with the initial release of OS/360 as a base, we have studied the interaction between management and the evolution of OS/360 by using certain independent variables of the improvement and enhancement (i.e., maintenance) process.

With the OS/360 operating system (an element of the substantive domain) as the primary focus, the second step was to identify appropriate techniques to study this. In this case, Belady and Lehman followed what they have termed "structured analysis" [90, p. 226]. The use of the statistical methods on the collected data from the OS/360 system (i.e., the combination of the OS/360 system as the topic of interest and the structured analysis as a method to study it) resulted in a set of empirical observations. Belady and Lehman then discussed how they tried to identify some underlying principles [90, p. 227]:

> Thus these first observations encouraged the search for models that represented laws that governed the dynamic behavior of the metasystem of organization, people, and program material involved in the creation and maintenance process, in the evolution of programming systems.

A different perspective is taken by Gonzalez-Barahona et al. [91]. They examined the long-term evolution of an open source project; their study has its basis in the methodological domain. This focus is clearly evident in the authors' remarks in the introduction of their article:

> Instead of coming from the laws and then trying to decide which parameters to use for the analysis, we have started with the parameters that can be extracted from the SCM [source code management] repository and have found how to use them to apply the laws.

Gonzalez-Barahona et al. did not focus on the laws themselves as the primary focus, but on the parameters to study them with. Therefore, they applied the *method-driven study design* path—their study design consisted of (1) their focus on the methodological domain (parameters to study Lehman's laws), and (2) Lehman's laws. While their choice of which open source project to study (representing the substantive domain) was well-deliberated (they did not just study *any* project), their decision to study the `glibc` project could be considered somewhat opportunistic—as becomes clear from their observations in the concluding section:

> We have been able to do it thanks to the availability of data for all this period in the current glibc git repository and to its public availability. When looking for long-lived FLOSS projects, we found that it is not common to have all their history available in their SCM repository or that it had problems (such as periods without information) that rendered them unusable for a study such as this one.

In other words, the choice to study `glibc` was based on the fact that it was an appropriate example given the availability

of its development history. However, any other project with an extensive and available development history would have sufficed.

A third perspective was taken by Lawrence [92], who conducted a research study on Lehman's Laws that has its basis in the conceptual domain. Lawrence observed:

> A significant shortcoming in this field is the scarcity of published statistical evidence in support of the laws. Indeed many of the works referenced contain little or no statistical validation of the models or results presented.

In this study, Lawrence's primary interest is to validate Lehman's Laws, and thus this is a *concept-driven* study. Concept-driven research can take one of two paths: a study design path, or a hypothetical path. In this case, Lawrence's research design was constrained by the available data:

> each law is considered in turn and statistical hypotheses based on it are analysed using the available data.

Lawrence had access to seven software systems (the substantive domain in this study), and thus this was a key constraint on what metrics (representing the methodological domain in this study) he could use. Therefore, this study followed the *concept-driven hypothetical* research path.

All three studies involve Lehman's Laws, but each study presents a different perspective. Table 2 presents an overview of these studies. The first study by Belady and Lehman had as its key contribution the laws themselves. The second study by Gonzalez-Barahona et al. added a different kind of knowledge, namely by improving our understanding on what parameters to use to test Lehman's Laws. The third study by Lawrence further expands the body of knowledge by conducting a validation study of the laws. Together, these three studies each offer different insights on Lehman's Laws, from their initial definition to ways to measure and test them, and to validate them on other software systems. The weaknesses that are inherent in one research orientation (e.g., the lack of generalizability as a result of the single case study on which the laws are based) can be compensated for by other orientations (e.g., a validation of the laws in other settings so as to gauge the generalizability of the laws). Thus, we argue that triangulating different research paths is an important strategy in establishing deep understanding of a particular topic.

4.2. Object-Oriented Metrics

There are numerous research methods and data collection techniques and metrics within software engineering research (see Table 1). One example of this are object-oriented (OO) metrics. There are a few well-known OO metric suites, such as the OO metrics proposed by Chidamber and Kemerer [93] and the MOOD[8] metric suite. Research on OO metrics has been extensive—hence, this is an useful example to illustrate how using different research orientations can contribute to a body of knowledge within software engineering.

One exemplar of research that has its basis in the conceptual domain is the original study by Chidamber and Kemerer [93] who observed that existing software metrics have been criticized for several reasons. For instance, they are *"lacking a theoretical basis"* and *"lacking in desirable measurement properties"* [93, p. 476]. To address this, Chidamber and Kemerer started with a theoretical base, namely the ontology of Bunge [94, 95].

Of course, the key contribution of this paper by Chidamber and Kemerer is the now well-known set of six OO metrics, better known as the CK metrics. One might suspect that this study is therefore grounded in the methodological domain. However, as Chidamber and Kemerer outlined, their concern was based on the fact that existing metrics did not sufficiently consider the properties and concepts of the OO paradigm:

> Therefore, given that current software metrics are subject to some general criticism and are easily seen as not supporting key OO concepts, it seems appropriate to develop a set, or suite of new metrics especially designed to measure unique aspects of the OO approach.

Based on their investigation of relations of the concepts found in OO designs, they defined a number of metrics, thus the methodological domain was of secondary concern. Of least concern was the actual application of the CK metrics on some real systems. The properties and concepts of the OO approach and the subsequent development of a set of metrics to measure those concepts together formed an instrumental structure called a study design. The next step was then to apply this study design on elements of the substantive domain, as summarized in the paper's conclusion:

> In addition to the proposal and analytic test of theoretically-grounded metrics, this paper has also presented empirical data on these metrics from actual commercial systems.

This suggests that the choice of 'actual commercial systems' is not of particular interest to the researchers. Rather, of primary importance was the ability to correctly measure relationships and properties of OO designs.

Cartwright and Shepperd [96] presented an empirical investigation of an object-oriented software system and thus their primary interest lay in the substantive domain. They observed that *"the majority of object-oriented metrics research has concentrated upon defining sets of structural metrics."* Instead, Cartwright and Shepperd's focus was primarily a real-world object-oriented software system itself—an instance of the substantive domain. The second step in their research design is that of selecting an appropriate technique or method to collect data about the OO system they are studying. The approach to collect data seems of secondary importance, as they stated (p. 788):

> Initially, we had considered collecting the Chidamber and Kemerer (CK) metrics suite. Unfortunately, only two out of the six metrics were readily available from the available design documentation. These were DIT (depth of inheritance tree) and NOC (number of children). Consequently, we decided to supplement these metrics with a number of additional measures that could be easily collected at the analysis/design stage.

Thus, the choice of OO metrics was adjusted based on the selection of the system that the authors had set out to study. From this, it becomes clear that Cartwright and Shepperd followed the

[8] Metrics for Object-Oriented Design.

Table 2: Theory fragments in studies on software evolution

Study	Research Path	Theory (fragment)	Purpose
Belady & Lehman [90]	System-driven observational	Underlying principles of software evolution	Principles of software evolution identified to characterize this phenomenon.
Gonzalez-Barahona et al. [91]	Method-driven study design	Lehman's laws	Metrics were identified to test Lehman's laws and validated; Lehman's laws provided the 'background'.
Lawrence [92]	Concept-driven hypothetical	Lehman's laws	Lehman's laws were the foundation for this validating study.

system-driven observational path. The conceptual domain was presumably of least importance in this study as the authors did not pay much attention to this.

A third orientation was taken by Harrison et al. [97], who presented a study with a clearly defined research path outlined in the introduction of their article:

> In this paper, we consider a set of metrics for object-oriented design called the MOOD metrics from a measurement theory viewpoint, and then consider their empirical evaluation using three different projects.

Their study had as a primary focus the MOOD metrics, clearly an element of the methodological domain. The goal of their study was to show that the MOOD metrics are valid, *"in the sense that they accurately measure the attributes of software which they were designed to measure"* [97, p. 491]. Of secondary importance was the theoretical lens with which these are considered—in this case, the authors explicitly stated that they took a measurement theory viewpoint. The result of the select of the MOOD metrics and the theoretical viewpoint resulted in a study design, which Harrison and colleagues subsequently applied on a selected number of projects for their empirical validation.

These three studies illustrate the different perspectives that can be taken when studying elements of the methodological domain (see Table 3), in this case object-oriented metrics. The purpose of a study will affect the primary focus of a researcher. Whereas Chidamber and Kemerer focused on defining a theoretically sound set of metrics to correctly measure OO designs, thus focusing on the conceptual domain, Cartwright and Shepperd on the other hand paid very little attention to the conceptual domain and instead focus primarily on the substantive domain by studying a large-scale object-oriented software system. The purpose of the study by Harrison et al. focused on the validation of a set of metrics using measurement theory as a conceptual lens.

4.3. Software Architecture

Software architecture has been established as an important sub-field within the software engineering discipline. This topic emerged in the mid-nineties with the realization that a software system's architecture has a significant impact on its so-called quality attributes such as performance, reliability and safety [98]. This element from the substantive domain in software engineering is thus an interesting topic, as numerous studies have focused on this topic from different research orientations.

The first example is a study by Bowman et al. [99] that represents *system-driven* research. Bowman et al. presented an analysis of the software architecture of the Linux kernel. Their primary focus was on the substantive domain—software architecture—which they introduced in the beginning of the paper:

> Recent research suggests that large software systems should be designed with a documented software architecture. This architecture provides a building plan for a system at a high level of abstraction. Individual functions and even modules are not described in detail; instead, subsystems and relations between them are documented. This level of abstraction is appropriate for understanding an entire software system, and provides a good mechanism for system understanding.

The authors continued with observations that software architecture is important, and expressed a specific interest in Linux:

> Because Linux is an interesting representative of existing software systems, we chose to examine it as a case study.

The authors' second step of the research, which we would classify as having followed the observational path, is the selection of an appropriate methodology to study this topic. Within the overall case study research strategy (also an element of the methodological domain), the authors defined a process to extract the 'concrete' (implemented, as opposed to the designed) architecture. Using source code analysis and visualization tools the authors built a representation of the implementation. Thus, the combination of the system (Linux kernel) and techniques to study it (tools), resulted in a set of empirical observations.

Whereas the study by Bowman et al. is a *system-driven* observational study, the study by Petriu et al. [100] is what we would call a *method-driven* design study. Petriu et al. presented a study that has its basis in the methodological domain, as suggested in the abstract:

> This paper proposes a systematic approach to building Layered Queueing Network (LQN) performance models from a UML description of the high-level architecture of a system and more exactly from the architectural patterns used for the system.

Their primary interest is a technique to build layered queueing network (LQN) performance models. These LQN models are based on a conceptual description of a system's architecture in UML notation, specifically using architectural patterns. Architectural patterns (e.g., client/server, layers, pipes/filter) are recurring solutions to common design problems, and thus could be considered conceptualized design building blocks. The

Table 3: Theory fragments in studies on object-oriented metrics

Study	Research Path	Theory (fragment)	Purpose
Chidamber & Kemerer [93]	Concept-driven study design	Bunge's ontology	Theoretical basis for defining metrics.
Cartwright & Shepperd [96]	System-driven observational	Speculation regarding the limited use of class inheritance and polymorphism.	Suggestions for developers, managers and future research.
Harrison et al. [97]	Method-driven study design	Measurement theory	Provide theoretical basis to validate MOOD metrics.

technique that can be used to model certain properties (i.e., performance – a quality attribute) of a system architecture based on its patterns is thus a study design. The substantive domain was presumably of least interest in this study; the study design is applied to a telecommunication system, but the technique could have been applied to any system of significant size.

A third perspective on software architecture as a topic of study is given by LaMantia et al. [101]. Their research is based in the conceptual domain, and focuses on modularity, a desired property of software architectures as it supports parallel development and maintainability [102]. LaMantia et al. highlighted the current informal nature of principles of achieving modularity, and that:

> we are in need of a formal theory and models of modularity and software evolution that can capture the essence of these important but informal design principles and provide the power of description, prediction and prescription.

Thus motivated, the authors clearly described what we term a *concept-driven* hypothetical path:

> To further explore the theory's descriptive power for large and complex software systems, we examine the evolution of two software product platforms through the lens of DSM models and design rule theory.

The primary interest in this study was the 'lens of DSM models and design rule theory.' The authors then identified two software products to study. The third step of their study design, namely the methodology, received substantially less emphasis.

The variety of research perspectives as represented by a focus on a specific element of a study is an important form of triangulation, what we term "triangulation of research orientations." This form of triangulation complements other forms such as triangulation among data sources, researchers, and research methods [103]. Insights gained from different studies that have different orientations can be combined and offer rich insights into a study topic — Table 4 lists the three studies that have focused on software architecture. While the study by Bowman et al. offered unique insights into the architecture of one specific system of considerable relevance (given the importance of Linux in the software industry), the topic of software architecture has also greatly benefited from insights into modeling quality attributes such as performance (as was done in the study by Petriu et al.). A third perspective was offered by LaMantia et al. who focused on a formal theory and models of software evolution, in which case the methodology used was not as important.

5. Discussion and Conclusion

In this paper we have presented an extensive discussion of what theory is, what it looks like, and its purpose. The Research Path Schema, based on the Validity Network Schema by Brinberg and McGrath [22], is a useful analytical tool to view SE research and to better understand the role of theory (fragments). Based on our discussion and demonstration of the RPS, we suggest a number of potential implications for future SE research and education.

5.1. Stronger Focus on Theorizing

While we strongly believe in the importance of theory as both a *driver* for, and a *result* of empirical research, we also admit that not each and every study can or should present new theory. However, one of our arguments is that theory is not a luxury, to be left to 'philosophers,' but that it is an essential element of SE research and thus should be considered when presenting research results in papers. Theory *should* inform the design of new studies, which will help to converge the research literature on a particular topic (or research question). This in turn will help in linking different studies on a topic, and to focus more directly on essential questions that SE research purports to address, namely those relating to building affordable, timely and high-quality software systems. We argue that with an increased awareness of the role of theory and the theorizing process, researchers may be able to design better research studies that are grounded in theory or extend existing theory fragments. This will contribute to one of the purported benefits of theory-focused research, namely that of knowledge transfer. The RPS offers a lens to view SE research studies, which can help in locating the theory in previous studies, and to design new studies.

5.2. Theorizing and Conceptualization Vary in Shape

The RPS presumes that a research study always consists of elements of three domains: the substantive, the methodological, and the conceptual domain. It is important to emphasize that research papers may make other contributions than empirical findings. In particular, conceptual papers are important to bring the field as a whole forward, as such papers may introduce new and important perspectives on topics [59]—the paper by Jansen and Bosch [73] mentioned above is a good example of this. Conceptual contributions in empirical papers may take on a variety of forms. Important also are meta-level studies that provide guidelines to other researchers, which may pertain to the research process or the reporting of research. While such papers

Table 4: Theory fragments in studies on software architecture

Study	Research Path	Theory (fragment)	Purpose
Bowman et al. [99]	System-driven observational	'lessons learned,' suggestions and speculation about design decisions.	Draw lessons from observations.
Petriu et al. [100]	Method-driven study design	Conceptual description of software architecture (UML)	Foundation for LQN performance models.
LaMantia et al. [101]	Concept-driven hypothetical	DSM models, design rule theory	Exploration of theory's descriptive power.

may represent useful contributions, the majority of studies that will bring the software engineering discipline forward will be empirical studies that contain elements from all three domains: methodological, conceptual and substantive.

5.3. Toward Theory-Oriented Software Engineering Research

The SE research field has a strong emphasis on Evidence-Based Software Engineering (EBSE) research, as advocated by Kitchenham et al. [2]. While we fully support this advocacy, we propose Theory-Oriented Software Engineering (TOSE) research, which complements EBSE with an explicit attention for the role of theory in research, so as to complete the cycle shown in Figure 2. This may either follow a *research-then-theory* or a *theory-then-research* approach [38] as described above. The studies by Mockus et al. are good examples of this; their first study of the Apache web server resulted in a number of observations, based on which they hypothesized (theorized) about OSS project governance and development. This theory fragment was then used to inform their second study (of Mozilla).

5.4. Theorizing in Software Engineering Education

Since current research in SE pays little attention to theories and theory building, PhD students get little—if any—exposure to, or training in building their own theories, or in using existing theories to guide and conduct their research. As argued above, for the SE research community to adopt a theory-focused approach to conducting research, new researchers (i.e., Ph.D. students) need to receive appropriate training. Researchers in other fields, in particular the social sciences, provided guidance in theory building, such as Kaplan [47], Reynolds [38] and Dubin [30]. While some guidance has been provided, such as by Sjøberg et al. [20], no in-depth discussions of how to theorize in SE research are available. Leshem and Trafford [104] presented an analysis of how conceptual frameworks can be used in doctoral research, which could be a sound starting point.

An important challenge here is that there is no agreement on what theories should look like in software engineering. While there are clear examples such as the theory of software evolution pioneered by Belady and Lehman, other topics may be harder to capture in theoretical propositions.

5.5. Awareness of Theory in Software Engineering Research

While theories have received limited attention, the SE research community is well familiar with the use of *frameworks*. One of the purposes of developing or using a framework is to organize existing concepts from the literature, or to assist in the development and testing of a theory [105]. As such, frameworks can be seen as theory fragments with an analysis goal. One of the examples presented in this paper, the study by Medvidović and Taylor, is a good example of this. Their framework provides a typology of ADLs, based on which researchers can design new studies. We believe that by using the RPS as a lens to analyze existing studies, researchers can increase their awareness of the role of theory or theoretical and conceptual elements of a study. Furthermore, while using the RPS as a lens, the lack of such theoretical elements can also become more clear.

5.6. Conclusion

In this paper we have discussed the need for and the role of theory in software engineering research. In order to better understand the role of theory, we adapted the Validity Network Schema (VNS), originally proposed by Brinberg and McGrath. Our adaptation, which we refer to as the Research Path Schema (RPS), is a model of software engineering research studies. To demonstrate how this model can be used as a 'guide' for designing research studies, we dissected a number of influential software engineering research papers. While we believe the RPS is an effective framework to reason about the design of a study, we would also wish to remind the reader of George Box's words, namely that *"All models are wrong, but some are useful"* [106]. Clearly, there will be research studies that do not perfectly match the structure that the RPS defines (i.e., conceptual, substantive, methodological domains). Nevertheless, we believe that the RPS as a model is "useful," and demonstrated this with a number of examples in Section 4.

Besides the RPS as a tool to dissect and design new research studies, we believe that a Theory-Oriented Software Engineering (TOSE) research philosophy could complement the Evidence-Based Software Engineering (EBSE) approach that has become popular in SE research. We believe a stronger focus on the development of theory in software engineering research can significantly help in increasing both rigor and relevance—rigor may increase as research studies will be organized around explicated theories (or theory fragments) and thus will take into account confirming or disconfirming perspectives. Relevance may increase as research studies will converge around theories that aim to explain and understand real-world phenomena in software engineering practice.

Acknowledgments

We thank the anonymous reviewers for their constructive and thoughtful feedback which has helped to improve this paper, in particular the distinction between variance and process theories, as well as Section 4 that presents more extensive illustrations of the RPS in software engineering research studies. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] K. Stol, B. Fitzgerald, Uncovering theories in software engineering, in: Proceedings of the 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE), IEEE Computer Society, 2013, pp. 5–14.

[2] B. Kitchenham, T. Dybå, M. Jørgensen, Evidence-based software engineering, in: Proceedings of the International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 273–281.

[3] L. Briand, Embracing the engineering side of software engineering, IEEE Software 29 (4) (2012) 93–96.

[4] P. J. Denning, Is computer science science?, Communications of the ACM 48 (4) (2005) 27–31.

[5] P. J. Denning, Computing is a natural science, Communications of the ACM 50 (7) (2007) 13–18.

[6] P. J. Denning, the science in computer science, Communications of the ACM 56 (5) (2013) 35–38.

[7] J. Offutt, Putting the engineering into software engineering education, IEEE Software 30 (1) (2013) 94–96.

[8] P. Naur, B. Randell (Eds.), Report on a conference sponsored by the NATO SCIENCE COMMITTEE, 1968.

[9] S. Beecham, P. O'Leary, I. Richardson, S. Baker, J. Noll, Who are we doing global software engineering research for?, in: IEEE 8th International Conference on Global Software Engineering, 2013, pp. 41–50.

[10] K. Stol, P. Avgeriou, M. A. Babar, Design and evaluation of a process for identifying architecture patterns in open source software, in: Proceedings of the European Conference on Software Architecture, Springer, 2011, pp. 147–163.

[11] P. Johnson, P. Ralph, M. Goedicke, P.-W. Ng, K. Stol, K. Smolander, I. Exman, D. E. Perry, Report on the second SEMAT workshop on general theory of software engineering (GTSE 2013), ACM Software Engineering Notes 38 (5) (2013) 47–50.

[12] L. J. Bourgeois, III, Toward a method of middle-range theorizing, Academy of Management Review 4 (3) (1979) 443–447.

[13] R. Merton, Social Theory and Social Structure, Enlarged Edition, Free Press, 1968.

[14] K. Lewin, The research centre for group dynamics at Massachusetts Institute of Technology, Sociometry 8 (1945) 126–135.

[15] S. Lynham, The general method of theory-building research in applied disciplines, Advances in Developing Human Resources 4 (3) (2002) 221–241.

[16] S. Gregor, The nature of theory in information systems, MIS Quarterly 30 (3) (2006) 611–642.

[17] J. E. Hannay, D. I. K. Sjøberg, T. Dybå, A systematic review of theory use in software engineering experiments, IEEE Transactions on Software Engineering 33 (2) (2007) 87–107.

[18] P. Johnson, M. Ekstedt, I. Jacobson, Where's the theory for software engineering?, IEEE Software 29 (55) (2012) 94–96.

[19] D. I. K. Sjøberg, Documenting theories: Working group results, in: V. Basili, D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, R. Selby (Eds.), Experimental Software Engineering Issues: Assessment and Future, Springer-Verlag, 2007, pp. 111–114, LNCS 4336.

[20] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, J. E. Hannay, Building theories in software engineering, in: Guide to Advanced Empirical Software Engineering, Springer, 2008, pp. 312–336.

[21] K. B. Zerangue, On developing a general theory of software engineering, in: Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC '93), IEEE, 1993.

[22] D. Brinberg, J. McGrath, Validity and the Research Process, SAGE Publications, 1985.

[23] K. Weick, What theory is not, theorizing is, Administrative Science Quarterly 40 (3) (1995) 385–390.

[24] R. Sutton, B. Staw, What theory is not, Administrative Science Quarterly 40.

[25] S. Pfleeger, B. Kitchenham, Principles of survey research: Part 1: Turning lemons into lemonade, ACM SIGSOFT Software Engineering Notes 26 (6) (2001) 16–18.

[26] P. Runeson, M. Höst, A. Rainer, C. Wohlin, Case Study Research in Software Engineering, Wiley, 2012.

[27] G. Coleman, R. O'Connor, Using grounded theory to understand software process improvement: A study of Irish software product companies, Information and Software Technology 49 (6) (2007) 654–667.

[28] H. Sharp, H. Robinson, An ethnographic study of XP practice, Empirical Software Engineering 9 (4) (2004) 353–375.

[29] H. Edwards, S. McDonald, S. Young, The repertory grid technique: Its place in empirical software engineering research, Information and Software Technology 51 (2009) 785–798.

[30] R. Dubin, Theory Building, Revised Edition, The Free Press, 1978.

[31] R. Suddaby, From the editors: What grounded theory is not, Academy of Management Review 49 (4) (2006) 633–642.

[32] R. Nisbett, The Geography of Thought: How Asians and Westerners Think Differently and Why, Nicholas Brealey Publishing, 2005.

[33] T. Hall, N. Baddoo, S. Beecham, H. Robinson, H. Sharp, A systematic review of theory use in studies investigating the motivations of software engineers, ACM Transactions on Software Engineering and Methodology 18 (3) (2009) Art. 10.

[34] C. Robson, Real World Research: A Resource for Social Scientists and Practitioner-Researchers, 2nd Edition, Blackwell Publishers, 2002.

[35] D. Cruzes, T. Dybå, Research synthesis in software engineering: A tertiary study, Information and Software Technology 53 (5) (2011) 440–455.

[36] M. Sandelowski, Whatever happened to qualitative description?, Research in Nursing & Health 23 (4) (2000) 334–340.

[37] P. Checkland, Notes on teaching and researching IS, Systemist (IS Special Edition Part II) 16 (1) (1994) 6–8.

[38] P. D. Reynolds, A Primer in Theory Construction, Macmillan Publishing Company, 1971.

[39] V. Basili, M. Zelkowitz, Empirical studies to build a science of computer science, Commun. ACM 50 (11) (2007) 33–37.

[40] M. Broy, Can practitioners neglect theory and theoreticians neglect practice?, IEEE Comput. 44 (10) (2011) 19–24.

[41] M. Lehman, J. Ramil, An approach to a theory of software evolution, in: Proceedings of the International Workshop on Principles of Software Evolution, 2001, pp. 70–74.

[42] M. Bunge, Scientific Research I: The Search for System, Springer-Verlag New York Inc., 1967.

[43] M. Shaw, Prospects for an engineering discipline of software, IEEE Software 7 (6) (1990) 15–24.

[44] J. Carroll, P. Swatman, Structured-case: a methodological framework for building theory in information systems research, European Journal of Information Systems 9 (4) (2000) 235–242.

[45] A. Endres, D. Rombach, A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories, Pearson Education Ltd., 2003.

[46] R. Wieringa, M. Daneva, N. Condori-Fernandez, The structure of design theories, and an analysis of their use in software engineering experiments, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 295–304.

[47] A. Kaplan, The Conduct of Inquiry: Methodology for Behavioral Science, Crowell, 1964.

[48] N. Chalofsky, Professionalization comes from theory and research: The "why" instead of the "how to", in: New Directions for Adult and Continuing Education, 1996, pp. 51–56.

[49] R. L. Glass, The relationship between theory and practice in software engineering, Communications of the ACM 39 (11) (1996) 11–13.

[50] D. Whetten, What constitutes a theoretical contribution?, Academy of Management Review 14 (4) (1989) 490–495.

[51] H. M. Blalock, Jr., Theory Construction: From Verbal to Mathematical Formulations, Prentice-Hall, Inc., 1969.

[52] R. Weber, Theoretically speaking, MIS Quarterly 27 (3) (2003) iii–xii.

[53] K. Weick, Theory construction as disciplined imagination, Academy of Management Review 14 (4) (1989) 516–531.

[54] P. B. Medawar, Is the scientific paper a fraud?, Saturday Review (1964)

42–43.
[55] A. Van de Ven, Engaged scholarship: a guide for organizational and social research, Oxford University Press, 2007.
[56] P. Ralph, Y. Wand, A teleological process theory of software development, in: Proceedings of JAIS Theory Development Workshop, 2008.
[57] P. Ralph, Comparing two software design process theories, in: Global Perspectives on Design Science Research, Springer, 2010, pp. 139–153.
[58] H. Reichenbach, Experience and Prediction, University of Chicago Press, 1938.
[59] M. Yadav, The decline of conceptual articles and implications for knowledge development, Journal of Marketing 74 (2010) 1–19.
[60] K. Crowston, H. Annabi, J. Howison, C. Masango, Effective work practices for software engineering: Free/libre open source software development, in: WISER, 2004, pp. 18–26.
[61] L. Morgan, J. Feller, P. Finnegan, Exploring value networks: theorising the creation and capture of value with open source software, European Journal of Information Systems 22 (2013) 569–588.
[62] S. Bacharach, Organizational theories: Some criteria for evaluation, Academy of Management Review 14 (4) (1989) 496–515.
[63] G. Homans, Contemporary theory in sociology, in: Handbook of modern sociology, Rand McNally, 1964, pp. 951–977.
[64] E. Swanson, The dimensions of maintenance, in: 2nd International Conference on Software Engineering, 1976, pp. 492–497.
[65] J. R. Venable, The role of theory and theorising in design science research, in: Proceedings of the First International Conference on Design Science Research in Information Systems and Technology, 2006, pp. 1–18.
[66] J. F. Nunamaker, M. Chen, T. D. M. Purdin, Systems development in information systems research, Journal of Management Information Systems 7 (3) (1991) 631–640.
[67] S. March, G. F. Smith, Design and natural science research on information technology, Decision Support Systems 15 (4) (1995) 251–266.
[68] D. Jones, S. Gregor, The anatomy of a design theory, Journal of the Association for Information Systems 8 (5) (2007) Art. 1.
[69] T. S. Kuhn, The Structure of Scientific Revolutions, 4th Edition, The University of Chicago Press, 2012.
[70] F. Shull, R. Feldmann, Building theories from multiple evidence sources, in: Guide to Advanced Empirical Software Engineering, Springer, 2008, pp. 337–364.
[71] P. Runeson, Theory building attempts in software engineering, in: Proceedings of the SEMAT Workshop on a General Theory of Software Engineering, 2012.
[72] K. Corley, D. Gioia, Building theory about theory building: What constitutes a theoretical contribution?, Academy of Management Review 36 (1) (2011) 12–32.
[73] A. Jansen, J. Bosch, Software architecture as a set of architectural design decisions, in: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, 2005, pp. 109–120.
[74] R. Capilla, F. Nava, S. Peréz, J. C. Duenas, A web-based tool for managing architectural design decisions, ACM SIGSOFT Software Engineering Notes 31 (5) (2006) Art. 4.
[75] J. McGrath, D. Brinberg, External validity and the research process: A comment on the calder/lynch dialogue, Journal of Consumer Research 10 (1) (1983) 115–124.
[76] B. Fitzgerald, D. Howcroft, Towards dissolution of the is research debate: from polarization to polarity, Journal of Information Technology 13 (1998) 313–326.
[77] J. Feller, B. Fitzgerald, S. A. Hissam, K. R. Lakhani (Eds.), Perspectives on Free and Open Source Software, MIT Press, 2005.
[78] S. Beecham, N. Baddoo, T. Hall, H. Robinson, H. Sharp, Motivation in software engineering: A systematic literature review, Information and Software Technology 50 (2008) 860–878.
[79] K. Stol, M. A. Babar, B. Russo, B. Fitzgerald, The use of empirical methods in open source software research: Facts, trends and future directions, in: Proceedings of the 2nd Workshop on Emerging Trends in FLOSS Research and Development, collocated with ICSE, IEEE Computer Society, 2009, pp. 19–24.
[80] S. Pfleeger, Albert Einstein and Empirical Software Engineering, IEEE Computer 32 (10) (1999) 32–38.
[81] D. Tofan, M. Galster, P. Avgeriou, Capturing tacit architectural knowledge using the repertory grid technique, in: International Conference on Software Engineering, 2011, pp. 916–919.
[82] N. Medvidović, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.
[83] B. Glaser, A. Strauss, The discovery of grounded theory: strategies for qualitative research, Aldine Transaction, 1967.
[84] A. Mockus, R. Fielding, J. Herbsleb, A case study of open source software development: The apache server, in: Proceedings of the International Conference on Software Engineering, 2000, pp. 263–272.
[85] A. Mockus, R. Fielding, J. Herbsleb, Two case studies of open source software development: Apache and mozilla, ACM Transactions on Software Engineering and Methodology 11 (3) (2002) 309–346.
[86] D. Brinberg, E. C. Hirschman, Multiple orientations for the conduct of marketing research: An analysis of the academic/practitioner distinction, Journal of Marketing 50 (4) (1986) 161–173.
[87] M. M. Lehman, Programs, cities, students: Limits to growth?, in: M. M. Lehman, L. A. Belady (Eds.), Program Evolution. Processes of Software Change, Academic Press, 1985, pp. 133–164.
[88] M. W. Godfrey, D. M. German, On the evolution of Lehmans laws, Journal of Software: Evolution and Process in press.
[89] I. Herraiz, D. Rodriguez, G. Robles, J. Gonzalez-Barahona, The evolution of the laws of software evolution: A discussion based on a systematic literature review, ACM Computing Surveys 46 (2) (2013) Art. 28.
[90] L. A. Belady, M. M. Lehman, A model of large program development, IBM Systems Journal 15 (3) (1976) 225–252.
[91] J. M. Gonzalez-Barahona, G. Robles, I. Herraiz, F. Ortega, Studying the laws of software evolution in a long-lived floss project, Journal of Software: Evolution and Process in press.
[92] M. J. Lawrence, An examination of evolution dynamics, in: Proceedings of the 6th International Conference on Software Engineering, 1982, pp. 188–196.
[93] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[94] M. Bunge, Treatise on Basic Philosophy: Ontology I: The Furniture of the World, Riedel, 1977.
[95] Y. Wand, V. C. Storey, R. Weber, An ontological analysis of the relationship construct in conceptual modeling, ACM Transactions on Database Systems 24 (4) (1999) 494–528.
[96] M. Cartwright, M. Shepperd, An empirical investigation of an object-oriented software system, IEEE Transactions on Software Engineering 26 (8) (2000) 786–796.
[97] R. Harrison, S. J. Counsell, R. V. Nithi, An evaluation of the MOOD set of object-oriented software metrics, IEEE Transactions on Software Engineering 24 (6) (1998) 491–496.
[98] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 3rd Edition, Addison-Wesley Professional, 2012.
[99] I. T. Bowman, R. C. Holt, N. V. Brewster, Linux as a case study: its extracted software architecture, in: International Conference on Software Engineering, ACM, 1999, pp. 555–563.
[100] D. Petriu, C. Shousha, A. Jalnapurkar, Architecture-based performance analysis applied to a telecommunication system, IEEE Transactions on Software Engineering 26 (11) (2000) 1049–1065.
[101] M. LaMantia, Y. Cai, A. MacCormack, Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases, in: Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture, 2008, pp. 83–92.
[102] K. Stol, P. Avgeriou, M. Babar, Y. Lucas, B. Fitzgerald, Key factors for adopting inner source, ACM Transactions on Software Engineering and Methodology 23 (2) (2014) Art. 18.
[103] J. Creswell, D. Miller, Determining validity in qualitative inquiry, Theory into Practice 39 (3) (2000) 124–130.
[104] S. Leshem, V. Trafford, Overlooking the conceptual framework, Innovations in Education and Teaching International 44 (1) (2007) 93–105.
[105] A. Schwarz, M. Mehta, N. Johnson, W. Chin, Understanding frameworks and reviews: A commentary to assist us in moving our field forward by analyzing our past, The DATA BASE for Advances in Information Systems 38 (3) (2007) 29–50.
[106] G. Box, N. Draper, Empirical Model-Building and Response Surfaces, Wiley, 1987.