Towards a Theory of Software Development Expertise

Sebastian Baltes
University of Trier
Trier, Germany
research@sbaltes.com

Stephan Diehl
University of Trier
Trier, Germany
diehl@uni-trier.de

ABSTRACT

Software development includes diverse tasks such as implementing new features, analyzing requirements, and fixing bugs. Being an expert in those tasks requires a certain set of skills, knowledge, and experience. Several studies investigated individual aspects of software development expertise, but what is missing is a comprehensive theory. We present a first conceptual theory of software development expertise that is grounded in data from a mixed-methods survey with 335 software developers and in literature on expertise and expert performance. Our theory currently focuses on programming, but already provides valuable insights for researchers, developers, and employers. The theory describes important properties of software development expertise and which factors foster or hinder its formation, including how developers' performance may decline over time. Moreover, our quantitative results show that developers' expertise self-assessments are context-dependent and that experience is not necessarily related to expertise.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

software engineering, expertise, theory, psychology

ACM Reference Format:
Sebastian Baltes and Stephan Diehl. 2018. Towards a Theory of Software Development Expertise. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3236024.3236061

1 INTRODUCTION

An *expert* is, according to Merriam-Webster, someone "having or showing special skill or knowledge because of what [s/he has] been taught or what [s/he has] experienced" [74]. K. Anders Ericsson, a famous psychologist and expertise researcher, defines *expertise* as "the characteristics, skills, and knowledge that distinguish experts from novices and less experienced people" [26]. For some areas, such as playing chess, there exist representative tasks and objective criteria for identifying experts [26]. In software development, however, it is more difficult to find objective measures for quantifying expert performance [78]. Bergersen et al. proposed an instrument to measure programming skill [9], but their approach may suffer from learning effects because it is based on a fixed set of programming tasks. Furthermore, aside from programming, software development involves many other tasks such as requirements engineering, testing, and debugging [62, 96, 100], in which a software development expert is expected to be good at.

In the past, researchers investigated certain aspects of software development expertise (SDExp) such as the influence of programming experience [95], desired attributes of software engineers [63], or the time it takes for developers to become "fluent" in software projects [117]. However, there is currently no theory combining those individual aspects. Such a theory could help structuring existing knowledge about SDExp in a concise and precise way and hence facilitate its communication [44]. Despite many arguments in favor of developing and using theories [46, 56, 85, 109], theory-driven research is not very common in software engineering [97].

With this paper, we contribute a theory that describes *central properties* of SDExp and *important factors* influencing its *formation*. Our goal was to develop a *process theory*, that is a theory intended to explain and understand "how an entity changes and develops" over time [85]. In our theory, the entities are individual software developers working on different software development tasks, with the long-term goal of becoming experts in those tasks. This fits the definition of a *teleological process theory*, where an entity "constructs an envisioned end state, takes action to reach it and monitors the progress" [110]. The theory is grounded in data from a mixed-methods survey with 335 participants and in literature on expertise and expert performance. Our expertise model is task-specific, but includes the notion of transferable knowledge and expertise from related fields or tasks. On a conceptual level, the theory focuses on factors influencing the formation of SDExp over time. It is a first step towards our long-term goal to build a *variance theory* [61, 67] to be able explain and predict why and when a software developer reaches a certain level of expertise [41, 85].

The theory can help researchers, software developers as well as employers. *Researchers* can use it to design studies related to expertise and expert performance, and in particular to reflect on the complex relationship between experience and expertise (see Section 6), which is relevant for many self-report studies. *Software developers* can learn which properties are distinctive for experts in their field, which behaviors may lead to becoming a better software developer, and which contextual factors could affect expertise development. If they are already "senior", they can learn what other developers expect from good mentors or which effects age-related performance decline may have on them. Finally, *employers* can learn what typical reasons for demotivation among their employees are, hindering developers to improve, and how they can build a work environment supporting expertise development of their staff.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236061

2 RESEARCH DESIGN

To describe our research design, we follow Tashakkori and Teddlie's methodology [104]. We designed a *sequential mixed model study* (type VIII) with three phases (see Figure 1). We started with an open online survey, which we sent out to a random sample of GitHub developers (S1) to build a preliminary grounded theory of SDExp (see Section 3). In a second phase, we combined the preliminary grounded theory from the first phase with existing literature on expertise and expert performance. The result of this combination of *inductive* and *deductive* methods [41] was a preliminary conceptional theory of SDExp (see Section 4). In a third phase, we designed a focused questionnaire to collect data for building a revised conceptual theory that describes certain concepts of the preliminary theory in more detail. We sent the focused questionnaire to two additional samples of software developers (S2 and S3). Like in the first phase, we analyzed the qualitative data from open-ended questions, this time mapping the emerging codes and categories to the preliminary conceptual theory.

To complement our qualitative analysis, we conducted a quantitative analysis investigating developers' self-assessment of programming expertise and its relation to experience (see Section 6). Please note that we planned the general research design, in particular the transitions between inductive and deductive steps [61], before collecting the data. We provide all questionnaires, coding schemes, and all non-confidential survey responses as supplementary material [6].

3 PHASE 1: GROUNDED THEORY

The goal of the first phase of our research was to build a *grounded theory* (GT) of SDExp. The GT methodology, introduced by Glaser and Strauss in 1967 [36], is an approach to generate theory from qualitative data. Since its introduction, different approaches evolved: Glaser's school emphasized the inductive nature of GT, while Strauss and Corbin focused on systematic strategies and verification [13, 19]. The third and most recent school of GT, called *constructivist* GT, tried to find a middle ground between the two diverging schools by building upon the flexibility of Glaser and Strauss's original approach, combining it with constructivist epistemology [13].

All three schools rely on the process of *coding* that assigns "summative, salient, essence-capturing" words or phrases to portions of the unstructured data [91]. Those codes are iteratively and continuously compared, aggregated, and structured into higher levels of abstractions, the *categories* and *concepts*. This iterative process is called *constant comparison*. We followed Charmaz's constructivist approach, dividing the analysis process into three main phases: (1) *initial coding*, (2) *focused coding and categorizing*, and (3) *theory building*. The last step tries to draw connections between the abstract concepts that emerged from the data during the first two phases, generating a unifying theory. An important aspect of GT is that the abstractions can always be traced back to the raw data (grounding). In the first step, the initial coding, it is important to remain open and to stick closely to the data [13]. Glaser even suggests not to do a literature review before conducting GT research [68], which is a rather extreme and debatable position [105]. We decided to limited our literature review in the first phase to software engineering literature and postponed the integration of results from

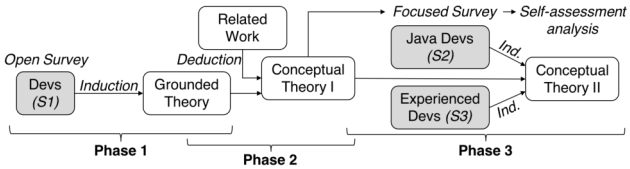

Figure 1: Research design

psychology literature to the second phase of our research. The main research questions guiding this phase were:

RQ1: Which characteristics do developers assign to novices and which to experts?

RQ2: Which challenges do developers face in their daily work?

Our main area of interest were the characteristics developers assign to novices and experts (RQ1). However, as software development experts are expected to master complex tasks efficiently [117], we included a question about challenges developers face in their daily work (RQ2) to identify such tasks.

3.1 Survey Design and Sampling

To answer our research questions, we designed an online questionnaire, which we sent to a random sample of software developers. Our goal was to start open-minded, thus we primarily relied on open-ended questions for data collection. The questionnaire contained seven open-ended and four closed-ended questions related to SDExp plus seven demographic questions. To prevent too broad and general answers, we focussed on expertise in one particular programming language. We chose Java, because at the time we designed the survey (October 2015) it was, according to various rankings, the most popular programming language [12, 106]. We analyzed all open-ended questions separately using Charmaz's grounded theory approach, performing all three constructivist GT phases (see above) on the survey answers. After deductively revising the resulting GT (see Section 4), we used theoretical sampling to collect more data on certain concepts and again performed those three GT phases, constantly comparing the new data to the data from the first iteration (see Section 5). We used the closed-ended questions to describe the samples and to analyze the relation between experience and (self-assessed) expertise (see Section 6).

Qualitative researchers often rely on convenience sampling for selecting their participants [5, 82]. However, we wanted to reach a diverse sample of novices and experts, which is hard to achieve with this sampling approach. Therefore, we drew our first sample randomly from all users who were active on both *Stack Overflow* (SO) and *GitHub* (GH) between January 1, 2014 and October 1, 2015. Both platforms are very popular among software developers and for both of them, demographic information about users is publicly available [111]. Another motivation for this sampling strategy was to be later able to correlate the self-assessments of developers with their activity on GH and SO.

We derived our sampling frame from the data dumps provided by *Stack Exchange* (August 18, 2015) [102] and *GHTorrent* (September 25, 2015) [39]. To match users on both platforms, we followed the approach of Vasilescu et al. [111], utilizing the MD5 hash value of users' email addresses. For the SO users, we retrieved the email

Table 1: Demographics of participants in samples *S*1-*S*3: Work time dedicated to sw. dev.; GE: general experience (years), GR_{sem}: general expertise rating (semantic differential from 1=novice to 6=expert), JE: Java experience (years), JR_{sem}: Java expertise.

Sample	Age			Work Time (%)			GE (years)		JE (years)		GR_{sem} (1-6)		JR_{sem} (1-6)		n
	M	*SD*	*Mdn*	*M*	*SD*	*Mdn*	*M*	*Mdn*	*M*	*Mdn*	*M*	*Mdn*	*M*	*Mdn*	
*S*1	30.4	6.4	29.0	70.2	26.3	80	11.8	10.0	5.0	3.5	4.8	5.0	3.6	4.0	122
*S*2	31.6	10.0	30.0	69.5	26.4	80	12.7	10.0	7.6	6.0	4.8	5.0	4.4	5.0	127
*S*3	59.9	4.9	59.0	68.2	32.0	80	34.1	35.0	5.7	1.5	5.3	5.0	2.8	2.0	86

hashes from an old data dump released September 10, 2013 where this information was directly available for all users. Further, for users who set a Gravatar URL in their profile, we extracted the email hash from there. In the end, we were able to retrieve the email hashes for 3.8 million SO users (75% of all users in the 2015 dataset). In the GHTorrent data dump, the email address was available for 6.6 million GH users (69% of all users in the dataset). To identify active users, we checked if they contributed to a question (asked, answered, or commented) on SO and committed to a project on GH since January 1, 2014. This resulted in a sampling frame with 71,400 unique users from which we drew a random sample of 1,000 users. In the following, *S*1 denotes this sample.

The first iteration of the questionnaire was online from October 13, 2015 until November 11, 2015. Of the 1,000 contacted users, 122 responded (12.2% response rate). Of the 122 respondents, 115 identified themselves as male, one as female and six did not provide their gender. The majority of respondents (67.2%) reported their main software development role to be *software developer*, the second-largest group were *software architects* (13.9%). Most participants answered from Europe (49.2%) and North America (37.7%). Further demographic information can be found in Table 1.

3.2 Terminology

According to Sjøberg et al., the building blocks of theories are its core entities, the *constructs*, the *relationships* between these constructs, and the *scope conditions* that delineate a theory's application area [97]. To have a consistent terminology across the paper, we use the term *concepts* instead of *constructs* for the central elements of the presented theories.

The *scope* of all theories we built, including the GT, was to describe what constitutes SDExp and which factors influence its formation, focusing on individual developers. In the second phase (see Section 4), we added a task-specific notion of expertise and then revised the resulting preliminary theory in a second inductive step (see Section 5) to focus on programming-related tasks.

3.3 Concepts

Figure 2 shows the high-level concepts and relationships of the grounded theory that resulted from our qualitative analysis of all open-ended questions. Most answers regarding characteristics of experts and novices (RQ1) were either related to having a certain degree of **knowledge** in different areas or a certain amount or quality of **experience**. We marked those concepts that constitute SDExp in gray color. The factors contributing to the formation of SDExp, and the results of having a certain degree of SDExp, have a white background. Participants described typical **behaviors**, **character traits**, and **skills** of experts. Many answers mentioned properties that distinguish source code written by experts from source code written by novices. In our notion, the **quality of source code** is the result of having a certain level of knowledge and expertise and thus a measure of expert performance. When asked about challenges (RQ2), participants often named time-pressure and unrealistic demands by managers or customers. Generally, most answers related to challenges were not technical, but referred to human factors. In the GT, we summarized these factors as **work context**.

In the following, we present the most frequent sub-categories of the concepts mentioned above. The concepts are in **bold** font, the (sub-)categories are in SMALL CAPITALS. We provide a full list of all categories and subcategories as supplementary material [6].

Experience: Most statements that we assigned to this concept referred to the QUANTITY of software development experience (e.g., in terms of years), but some also described its QUALITY. Examples for the latter include having built "everything from small projects to enterprise projects" or "[experience] with many codebases". In particular, participants considered PROFESSIONAL EXPERIENCE, e.g., having "shipped a significant amount of code to production or to a customer" and working on SHARED CODE to be important factors.

Knowledge: Since we specifically asked for Java, many answers were LANGUAGE-SPECIFIC or referred to certain Java FRAMEWORKS. Experts were described as having an "intimate knowledge of the design and philosophy of the language" (DEPTH OF KNOWLEDGE), which includes knowing "the shortcomings of the language [...] or the implementation [...]." Answers also indicated the importance of having a BROAD KNOWLEDGE about algorithms, data structures, or different programming paradigms to bring "the wisdom of [...] other philosophies into Java".

Quality of source code: Regarding the quality of source code, participants named several properties that source code of experts should possess: It should be WELL-STRUCTURED and READABLE, contain "comments when necessary", be "optimized" in terms of PERFORMANCE and sustainable in terms of MAINTAINABILITY. One participant summarized the code that experts write as follows: "Every one can write Java code which a machine can read and process but the key lies in writing concise and understandable code which [...] people who have never used that piece of code before [can read]."

Behavior, character traits, and skills: For this concept, the most common category was COMMUNICATION SKILLS. Experts should be willing to "share [their] knowledge with other developers", but they should also know when to "ask for help". Some participants

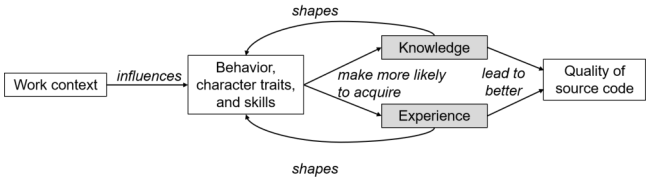

Figure 2: High-level concepts/relationships of GT (phase 1).

mentioned the role of experts as teachers, e.g. to train "younger developers". Another category was (SELF-)REFLECTION, meaning reflecting on problems ("thinks before coding") as well as on own behavior (being "aware [of] what kind of mistakes he can make"). Further, participants named PROBLEM-SOLVING SKILLS and attributes that we summarized in a category named BEING FAST.

Work context: Many participants mentioned problems related to PEOPLE affecting their work. One participant put it this way: "Computers are easy. People are hard." Salient were the comments about constant TIME PRESSURE, often caused by customers or the management. Respondents found it challenging to maintain "quality despite pressure to just make it work". One participant remarked that "sometimes non-software managers think of software like manufacturing: If 1 person works 400 parts in a day 10 should work 4000. But in software development, that analogy breaks down." There were also comments about team issues like "getting a big team of developers adopt common standards of coding, designing and unit testing." Participants also complained about the lack of well-defined REQUIREMENTS and the importance of good COMMUNICATION: "[...] User's cannot communicate what they want. [...] Project managers who talk to the users don't understand the implications by the requirements and mostly don't know enough of the business process the user lives every day. Hence, he cannot communicate the problem to the development team."

3.4 Relationships

After structuring participants' answers into concepts, categories, and sub-categories, we looked at the answers again, trying to find meaningful relationships. The result of this process is depicted in Figure 2. In our notion, certain forms of **behavior**, and an individual developer's **character traits** and general **skills** make it more likely to gain the level of **knowledge** and **experience** to be considered an expert in software development, which then manifests itself in the **quality of source code** the developer creates. However, gained knowledge and experience also affect an individual's behavior and shapes other skills. Moreover, the **work context**, meaning, for example, the office, colleagues, customers, or the application domain of a project, influence the behavior and thus the formation of knowledge and experience.

Phase 1: The grounded theory describes SDExp as a combination of a certain quantity and quality of *knowledge* and *experience*, both general and for a particular programming language. The *work context*, *behavior*, *character traits*, and *skills* influence the formation of expertise, which can be observed when experts write well-structured, readable, and maintainable *source code*.

4 PHASE 2: PRELIMINARY CONCEPTUAL THEORY

As described in our research design, the next step after *inductively* deriving a preliminary GT from the responses of the participants in our first sample was to *deductively* embed this GT in existing literature on expertise and expert performance. To this end, we reviewed psychology literature. Our main source was *The Cambridge Handbook of Expertise and Expert Performance* [28] including the

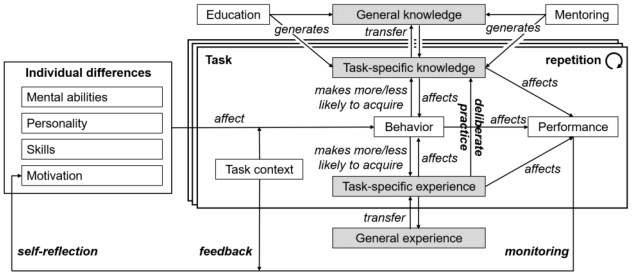

Figure 3: High-level concepts and relationships of preliminary conceptual theory (phase 2).

referenced literature. This handbook is the first [28], and to the best of our knowledge most comprehensive, book summarizing scientific knowledge on expertise and expert performance. The result of this deductive step was a task-specific conceptual theory of expertise development that is compatible with the grounded theory from the first phase. Figure 3 shows our preliminary conceptual theory, which we are going to present in this section.

Generally, *process theories* focus on events and try to find patterns among them, leading to a certain outcome—*variance theories* describe a certain outcome as a relationship between dependent and independent variables [61]. The process that we describe with our conceptual theory is the formation of SDExp, that is the path of an individual software development novice towards becoming an expert. This path consists of gradual improvements with many corrections and repetitions [27], therefore we do not describe discrete steps like, for example, the 5-stage *Dreyfus model* of skill acquisition (see Section 6.2). Instead, we focus on the repetition of individual tasks. In phase 3, we extended our conceptual theory with a focus on programming-related tasks (see Section 5), but the general structure is flexible enough to be extended towards other software development tasks as well [62, 96, 100]. Even with a focus on programming expertise, the distinction between tasks is important. For example, an excellent Java programmer is not automatically an excellent Haskell programmer. Moreover, programming itself includes diverse tasks, such as implementing new features or fixing bugs, with a varying centrality and difficulty [117].

4.1 Concepts

In the following, we will describe the concepts we deductively integrated into our grounded theory using literature on expertise and expert performance.

Individual differences and behavior: We split the GT concept *behavior, character traits, and skills* into *individual differences* and *behavior*. We modeled behavior as being relative to a certain task and as being influenced by individual differences [83] such as *mental abilities*, *personality*, and *motivation*, which have long been considered essential for general [11, 31, 43] and programming performance [21]. Even if the general intelligence is not a valid predictor for attaining expert performance in a domain [26], improvements are constrained by an individual's cognitive capacities [27]. Especially at the early stages of skill acquisition, general intelligence is in fact an important factor [58]. It is also known that mental abilities start to decline at a certain age [58]

Acquiring expertise is not exclusively a cognitive matter" [50]—developers' *personality* and *motivation* influence *behaviors* that may or may not lead to improvements of expertise [50, 101]. Generally, the term *skill* is defined as "an ability or proficiency acquired through training and practice" [2]. Thus, according to that definition, being a good software developer is also a skill. However, in the scope of our theory, we limit the term skill to fundamental skills such as communication and social skills [2].

Task context: In the GT, we described how the *work context*, including team members, managers, and customers, can influence developers' *behavior*. In the conceptual theory, we considered this context to be task-specific (e.g., communication with customers is more likely to happen during requirements analysis and communication with colleagues when refactoring an existing module). The *task context* captures all organizational, social [77], and technical constraints that are relevant for the task at hand.

Knowledge and experience: Knowledge can be defined as a "permanent structure of information stored in memory" [88]. Some researchers consider a developer's knowledge base as the most important aspect affecting their performance [21]. Studies with software developers suggest that "the knowledge base of experts is highly language dependent", but experts also have "abstract, transferable knowledge and skills" [100]. We modeled this aspect in our theory by dividing the central concepts *knowledge* and *experience* from the GT into a task-specific and a general part. This is a simplification of our model, because the relevance of knowledge and expertise is rather a continuum than dichotomous states [114]. However, Shneiderman and Mayer, who developed a behavioral model of software development, used a similar differentiation between general ("semantic") and specific ("syntactical") knowledge [94]. General knowledge and expertise does not only refer to technical aspects (e.g., low-level computer architecture) or general concepts (e.g., design patterns), but also to knowledge about and experience with successful strategies [57, 98, 99].

Performance, education, and monitoring: As mentioned in the introduction, it may be difficult to find objective measures for quantifying expert *performance* in software development. However, there exist many metrics and measures that can be evaluated regarding their validity and reliability for measuring expert performance. Respondents from the first sample mentioned different characteristics of experts' source code, but also the time it takes to develop a solution. This is related to the area of program comprehension where task correctness and response time are two important measures [25]. At this point, our goal is not to treat *performance* as a dependent variable that we try to explain for individual tasks, we rather consider different *performance monitoring* approaches to be a means for *feedback* and *self-reflection*. For our long-term goal to build a *variance theory* for explaining and predicting the development of expertise, it will be more important to be able to accurately measure developers' performance.

Education and *mentoring* help building knowledge and thus contribute to the development of expertise [30]. Having a teacher or mentor is particularly important for *deliberate practice* [29, 30], which is a central aspect of our theory (see below).

4.2 Relationships

The relationships in our theory are intentionally labeled with rather generic terms such as "affects" or "generates", because more research is needed to investigate them. Nevertheless, we want to point out two central groups of relationships: **deliberate practice** and the influence of **monitoring**, **feedback**, and **self-reflection**.

Deliberate practice: Having more experience with a task does not automatically lead to better performance [29]. Research has shown that once an acceptable level of performance has been attained, additional "common" experience has only a negligible effect, in many domains the performance even decreases over time [32]. The length of experience has been found to be only a weak correlate of job performance after the first two years [27]—what matters is the *quality* of the experience. According to Ericsson et al., expert performance can be explained with "prolonged efforts to improve performance while negotiating motivational and external constraints" [29]. For them, *deliberate practice*, meaning activities and experiences that are targeted at improving the own performance, are needed to become an expert. For software development, Zhou and Mockus found that developers can improve their performance over time by continuously increasing the difficulty and centrality of development tasks [117], which is in line with the concept of deliberate practice. Traditionally, research on deliberate practice concentrated on acquired knowledge and experience to explain expert performance [11, 31, 43]. However, later studies have shown that deliberate practice is necessary, but not sufficient, to achieve high levels of expert performance [11]—individual differences play an important role [43] (see above).

Monitoring, feedback, and self-reflection: A central aspect of *deliberate practice* is monitoring one's own *performance*, and getting *feedback*, for example from a teacher or coach [27]. Generally, such feedback helps individuals to *monitor* their progress towards goal achievement [64]. Moreover, as Tourish and Hargie note, "[t]he more channels of accurate and helpful feedback we have access to, the better we are likely to perform." [107]. In areas like chess or physics, studies have shown that experts have more accurate self-monitoring skills than novices [14]. In our model, the feedback relation is connected to the concept *task context* as we assumed that feedback for a software developer most likely comes from co-workers or supervisors. To close the cycle, monitoring and self-reflection influence a developer's *motivation* and consequently his/her *behavior*. In the revised conceptual theory (see Section 5), we also included mentors in this feedback cycle.

Phase 2: The preliminary conceptual theory builds upon the grounded theory. Among other changes, the theory introduces a *task-specific* view on expertise, separates *individual differences* and *behavior*, and embeds the concept of *deliberate practice*, including the relationships *monitoring*, *feedback*, and *self-reflection*. Moreover, instead of focusing on source code, it introduces the general concept of *performance* as a result of having a certain level of expertise.

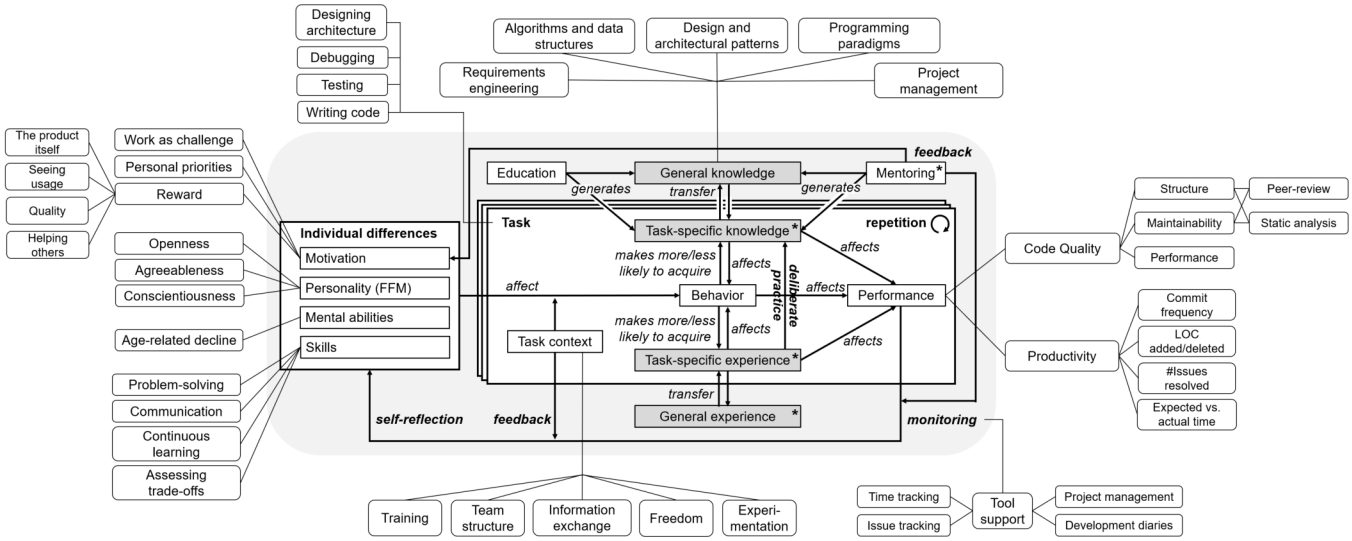

Figure 4: High-level concepts/categories of revised conceptual theory (phase 3); asterisk refers to description in the text.

5 PHASE 3: REVISED CONCEPTUAL THEORY

The goal of the third and last phase was to validate the general design of our theory and to collect more data about certain concepts, in particular the ones related to **deliberate practice**. Our focus was on programming-related tasks, but the theory can as well be extended and operationalized for other software development tasks in future work.

5.1 Survey Design

We revised the open questionnaire from phase 1 to focus on specific concepts, in fact most questions of the resulting focused questionnaire were directly related to individual concepts of the preliminary theory. We then conducted *theoretical sampling* to "elaborate and refine the categories constituting [our] theory" [13], surveying two additional samples of software developers. We tried to reach active Java developers (*S2*) and very experienced developers (*S3*). We targeted Java developers, because we wanted to compare participants' general experience and expertise with their experience and expertise in one particular programming language (see Section 6). We further targeted experienced developers, because in the first phase especially this group of participants provided well-elaborated and insightful answers. Please note that the goal of theoretical sampling is "conceptual and theoretical development", not "increasing the [...] generalizability" of results [13].

We revised and extended our two initial research questions to adjust them to our preliminary conceptual theory. Beside asking for typical *character traits* of experts (RQ1.1), we now asked in particular for traits that are supportive for becoming an expert (RQ1.2) to collect more data on factors influencing the formation of SDExp. Due to the importance of *mental abilities* in expert development and the fact that they start to decline at a certain age [58], we asked about situations where developers' performance declined over time (RQ1.3). Since our theory is task-specific, we also asked for *tasks* that an expert should be good at (RQ1.4). When we asked participants in S1 for challenges in their daily work (RQ2), they often referred to their *work context* and in particular to people-related issues. The work context may also influence developers' *motivation*, which plays an important role in expertise development (see Section 4.1). Thus, we changed RQ2 to focus more on those two aspects. Since we deductively included the concept of *deliberate practice* in our theory, we added questions about *monitoring* (RQ3.1) and *mentoring* (RQ3.2), which are important aspects of deliberate practice. We provide the research questions and the corresponding survey questions as supplementary material [6].

During the analysis of samples *S2* and *S3*, we build upon our conceptual theory, mapping the emerging codes and categories to the existing theory. This procedure is similar to what Saldaña calls *elaborative coding* [91]. Figure 4 depicts the high-level concepts and categories of our revised conceptual theory. Some categories are not shown in the figure, but are described in this section. We provide a full list of all (sub-)categories as supplementary material [6].

5.2 Sampling

As mentioned in the previous section, our preliminary conceptual theory guided the sampling (*theoretical sampling* [13, 82]). Our goal was to reach active Java developers (*S2*) as well as very experienced developers (*S3*). We retrieved the sampling frame for those samples from the *Stack Exchange Data Dump* [103] released January 1, 2016 and the *GHTorrent data dump* [39] released February 16, 2016.

For the Java sample (*S2*), we started by identifying active GH projects. We first filtered out the projects that were not deleted, not a fork, had at least two contributing users, and had at least 10 commits. Then, to select non-trivial Java projects, we only considered projects with at least 300 kB of Java source code (sum of file sizes of all Java files in the project). From the resulting 22,787 Java GH projects, we created a sampling frame with all users who contributed (committed or merged a pull request) to one of the selected projects and who pushed at least 10 commits between January 1, 2015 and December 31, 2015. From the 44,138 users who satisfied the above criteria, we contacted the ones with a public email address on their profile page ($n = 1,573$).

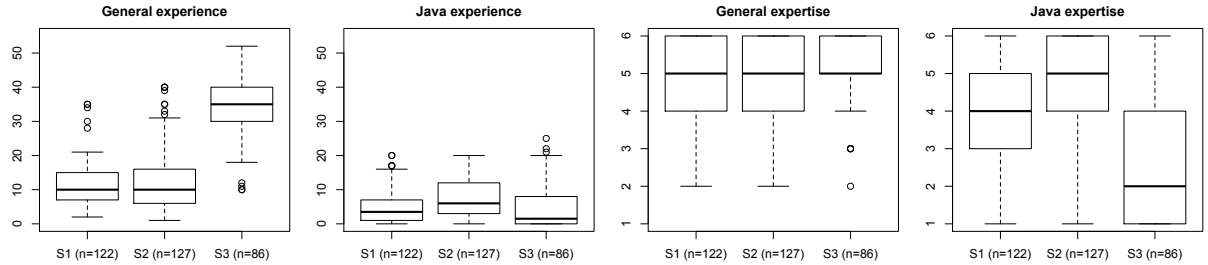

Figure 5: General/Java experience (GE, JE) and general/Java expertise rating (GR_{sem}, JR_{sem}) of participants in $S1$, $S2$, and $S3$.

With the third sample ($S3$), we wanted to reach very experienced users. Therefore, we again combined data from SO and GH. We used the age of a developer as a proxy variable for their experience. For GH users, the age was not available, but 11% of the users in the SO dump provided their age. To select experienced users, we filtered all SO users with age $\geq$ 55 years and $\leq$ 80 years and matched them with a GH account using the hash value of their email address. This resulted in a sample of 877 experienced users.

The focused questionnaire we used in the third phase contained nine open-ended and nine closed-ended questions, three of them only visible depending on previous answers, plus seven demographic questions. The full questionnaire is available as supplementary material [6]. This iteration of the questionnaire was online from February 18, 2016 until March 3, 2016 ($S2$) and from February 19, 2016 until March 4, 2016 ($S3$). Of the 1,573 contacted users in $S2$, 30 had an invalid email address and could not be reached. In the end, 127 participants filled out the questionnaire (response rate 8.2%). Of the 877 users in $S3$, 18 had an invalid email address and 91 participants completed the questionnaire (response rate 10.6%). We removed five participants from $S3$ because their answers either indicated that the age information from SO was not correct or that they were not active software developers. This lead to 86 responses available for analysis. Overall, combining $S2$ and $S3$, we had 213 valid responses in phase 3.

In $S2$, 119 respondents identified themselves as male, three as female and five did not provide their gender ($S3$: 84/1/1). The majority of respondents ($S2$: 64.6%, $S3$: 61.6%) reported their main software development role to be *software developer*, the second-largest group were *software architects* ($S2$: 13.4%, $S3$: 17.5%). In $S2$, most participants answered from Europe (47.2%) and North America (32.3%), in $S3$ the order was reversed (North America (67.4%), Europe (23.3%)). Further demographic information can be found in Table 1.

Comparing the demographics of the first two samples, we can see that $S1$ and $S2$ are quite similar, except for the fact that participants in $S2$ had more experience with Java (Mdn 3.5 vs. 6 years) and rated their Java expertise to be higher (Mdn 4 vs. 5). This indicates that our sampling approach was successful in reaching active Java developers. In $S3$, the values for the amount of professional work time dedicated to software development are quite similar to the other two samples. However, the developers in this sample are much older (M 59.9 vs. 30.4/31.6) and have much more general programming experience (Mdn 35 vs. 10/10). This indicates that our sampling approach for S3 was successful in reaching developers with a long general programming experience. However, many developers in $S3$ have little Java experience (Mdn 1.5 years) and also rated their Java expertise relatively low (Mdn 2). One reason for this could be that one quarter of the participants had a programming experience of 40 years or more ($Q_3 = 40$) and compared to this time frame, Java is a relatively young programming language (introduced 1995). The boxplots in Figure 5 visualize the differences in general/Java experience and expertise between the three samples.

5.3 Concepts

Figure 4 shows the revised conceptual theory resulting from our analysis of the closed- and open-ended answers of samples $S2$ and $S3$. In the following, we describe the most frequent (sub-)categories for the high-level concepts of our theory that emerged during the analysis and combine those qualitative results with quantitative evaluations where possible. For each concept, we indicate when there were notable differences between the answers in $S2$ and $S3$. Like before, we write the concepts in **bold** font and the (sub-)categories in SMALL CAPITALS. We also provide the number of answers we assigned to each concept or category (in brackets). We only present the most frequent categories and provide the complete coding schema as supplementary material [6].

Tasks: Since our SDExp model is task-specific, we asked our participants to name the three most important tasks that a software development expert should be good at. The three most frequently mentioned tasks were DESIGNING SOFTWARE ARCHITECTURE (95), WRITING SOURCE CODE (91), and ANALYZING AND UNDERSTANDING REQUIREMENTS (52). Many participants not only mentioned the tasks, but also certain quality attributes associated with them, for example "architecting the software in a way that allows flexibility in project requirements and future applications of the components" and "writing clean, correct, and understandable code". Other mentioned tasks include TESTING (48), COMMUNICATING (44), STAYING UP-TO-DATE (28), and DEBUGGING (28). Our theory currently focuses on tasks directly related to programming (see Figure 4), but the responses show that it is important to broaden this view in the future to include, for example, tasks related to requirements engineering (ANALYZING AND UNDERSTANDING REQUIREMENTS) or the adaption of new technologies (STAYING UP-TO-DATE).

Experience, knowledge, and performance: Like in the first phase, we asked participants about general attributes of software development experts. Aspects like having **experience** (26), a broad **general knowledge** (35) about "paradigms [...], data structures, algorithms, computational complexity, and design patterns", and an "intimate" knowledge about a certain programming language (**task-specific knowledge** (30)) were important. In particular, knowledge

about SOFTWARE ARCHITECTURE, including "modularization" and "decomposition", was frequently named (22). Interestingly, 20 of the 22 answers mentioning software architecture came from the sample of active Java developers. Also similar to the first phase, participants described properties of experts' source code such as MAINTAINABILITY (22), CLEAR STRUCTURE (12), or PERFORMANCE (9). The answers from *S2* and *S3* supported the general structure of our theory, which we derived inductively in phase 1 and deductively in phase 2. Thus, we will focus on new aspects and in particular on factors influencing the formation of SDExp in the following.

Individual differences: We asked for specific characteristics of experts and in particular for character traits that support expertise development. Regarding the **personality** of experts, participants often described three properties that are also present in the popular five factor personality model (FFM) [70]: According to our participants, experts should be OPEN-MINDED (42) and CURIOUS (35) (FFM: *openness*), be TEAM PLAYERS (37) (FFM: *agreeableness*), and be thorough and pay ATTENTION TO DETAIL (FFM: *conscientiousness*). Two other important traits were being PATIENT (26) and being SELF-REFLECTED (20). The latter is an important connection to the concept of **deliberate practice** that we introduced in the previous phase and includes understanding one's "weaknesses and strengths" and "the ability to learn from prior mistakes".

Regarding **skills** that an expert should possess, PROBLEM-SOLVING (84) was most frequently named. Sub-categories of problem solving are ABSTRACTION/DECOMPOSITION (30), ANALYTICAL THINKING (20), and LOGICAL THINKING (17). An expert can "break a giant problem into the little pieces that can be solved to add back up to the whole". Examples where an analytical approach is needed include bug fixing or "mapping the problem domain into the solution space". A second important skill was having the "drive to keep learning", which some participants described as CONTINUOUS LEARNING (55). Moreover, like in the first phase, COMMUNICATION SKILLS (42) were frequently named. In the answers of this iteration, those skills were often mentioned together with the task of understanding and implementing REQUIREMENTS (32): An expert should be "a good listener during requirement gathering", understand "a customer's desires", "work out what is really needed when the client can only say what they think they want", and should be able to "explain what he is doing to non developers". According to our participants, another important skill is being able to assess TRADE-OFFS (19) when comparing alternative solutions. Trade-offs can exist between "design, maintainability, [and] performance". Experts should be "able to discern the differences between early optimization and important design decisions for the long term goal", which is closely related to the concept of *technical debt* in software projects [59].

Mentoring: More than half the the participants in *S2* and *S3* (54.3%) had a (former) colleague or teacher whom they would describe as their mentor in becoming a better software developer. We asked those participants to describe their mentor(s). Six categories emerged during the initial and focused coding of participants' answers. One category, HAVING TIME, was only present in the answers from *S3*: Eight experienced developers named aspects such as taking time to explain things or honoring solutions that take more time in the beginning, but save time on the long run.

Regarding the mentor's ROLE, SENIOR DEVELOPER (15), PROFESSOR OR TEACHER (13) and PEER (12) were the most common answers. Two participants noted that their mentor was actually a JUNIOR DEVELOPER younger than themselves. What is important are a mentor's CHARACTER (29), SKILLS (19), his/her EXPERIENCE (16), and his/her role as a source for FEEDBACK (20) and as a MOTIVATOR (19). The most common characteristics of mentors were being GUIDING (10), PATIENT (8), and OPEN-MINDED (7). The most important aspect of a mentor's FEEDBACK were comments about CODE QUALITY (7). What participants motivated most was when mentors posed CHALLENGING tasks. In summary, we can conclude that the description of good mentors resembles the description of software development experts in general.

Monitoring and self-reflection: We asked participants if they regularly monitor their software development activities. Combining the answers from *S2* and *S3*, 38.7% of the 204 participants who answered that question said that they regularly monitor their activity. We asked those participants how they usually monitor their development activity.

In both samples, the most important monitoring activity was PEER REVIEW (16), where participants mentioned asking co-workers for feedback, doing code-review, or doing pair-programming. One participant mentioned that he tries to "take note of how often [he] win[s] technical arguments with [his] peers". Participants also mentioned TIME TRACKING (14) tools like *WakaTime* or *RescueTime*, ISSUE TRACKING (11) systems like *Jira* or *GitHub issues*, and PROJECT MANAGEMENT (14) tools like *Redmine* and *Scrum story points* as sources for feedback, comparing expected to actual results (e.g., time goals or number of features to implement). Three developers reported writing a DEVELOPMENT DIARY.

Regarding employed metrics, participants reported using simple metrics such as the COMMIT FREQUENCY, LINES OF CODE ADDED / DELETED, or number of ISSUES RESOLVED. Further, they reported to use STATIC ANALYSIS (18) tools such as *SonarQube*, *FindBugs*, and *Checkstyle*, or to use GITHUB'S ACTIVITY OVERVIEW (10). In this point, there was a difference between the answers in *S2* and *S3*: GitHub's activity overview was mentioned almost exclusively by the active Java developers (9). Three developers where doubtful regarding the usefulness of metrics. One participant noted: "I do not think that measuring commits [or] LOC [...] automatically is a good idea to rate performance. It will raise competition, yes—but not the one an employer would like. It will just get people to optimize whatever is measured." The described phenomenon is also known as *Goodhart's law* [16, 38].

Motivation: To assess developers' motivation, we asked our participants what the most rewarding part of being a software developer is for them. Many participants were intrinsically motivated, stating that PROBLEM SOLVING (46) is their main motivation—one participant wrote that solving problems "makes [him] feel clever, and powerful." Another participant compared problem solving to climbing a mountain: "I would equate that feeling [of getting a feature to work correctly after hours and hours of effort] to the feeling a mountain climber gets once they reach the summit of Everest." Many developers enjoy seeing the RESULT (53) of their work. They are particularly satisfied to see a solution which they consider to be of high QUALITY (22). Four participants mentioned refactoring

as a rewarding task. One answered: "The initial design is fun, but what really is more rewarding is refactoring." Others stressed the importance of CREATING SOMETHING NEW (19) and HELPING OTHERS (37). Interestingly, MONEY was only mentioned by six participants as a motivation for their work.

Work context: To investigate the influence of the work context on expertise development, we asked what employers should do in order to facilitate a continuous development of their employees' software development skills. We grouped the responses into four main categories: 1. ENCOURAGE LEARNING (70), 2. ENCOURAGE EXPERIMENTATION (61), 3. IMPROVE INFORMATION EXCHANGE (53), and 4. GRANT FREEDOM (42). To ENCOURAGE LEARNING, employers may offer in-house or pay for external TRAINING COURSES (34), pay employees to visit CONFERENCES (15), provide a good analog and/or digital LIBRARY (9), and offer MONETARY INCENTIVES for self-improvement (7). The most frequently named means to ENCOURAGE EXPERIMENTATION were motivating employees to pursue SIDE PROJECTS (29) and building a work environment that is open for NEW IDEAS AND TECHNOLOGIES (23). To IMPROVE INFORMATION EXCHANGE between development teams, between different departments, or even between different companies, participants proposed to FACILITATE MEETINGS (16) such as agile retrospectives, "Self-improvement Fridays", "lunch-and-learn sessions", or "Technical Thursday" meetings. Such meetings could explicitly target information exchange or skill development. Beside dedicated meetings, the idea of developers ROTATING (15) between teams, projects, departments, or even companies is considered to foster expertise development. To improve the information flow between developers, practices such as MENTORING (9) or CODE REVIEWS (8) were mentioned. Finally, GRANTING FREEDOM, primarily in form of LESS TIME-PRESSURE (18), would allow developers to invest in learning new technologies or skills.

Performance decline: We asked participants if they ever observed a significant decline of their own programming performance or the performance of co-workers over time. Combining the answers from $S2$ and $S3$, 41.5% of the 205 participants who answered that question actually observed such a performance decline over time. We asked those participants to describe how the decline manifested itself and to suggest possible reasons. The main categories we assigned to those answers were: 1. different reasons for DEMOTIVATION (34), 2. changes in the WORK ENVIRONMENT (32), 3. AGE-RELATED DECLINE (13), 4. CHANGES IN ATTITUDE (10), and 5. SHIFTING TOWARDS OTHER TASKS (7). The most common reason for an increased DEMOTIVATION was NON-CHALLENGING WORK (8), often caused by tasks becoming routine over time. One participant described this effect as follows: "I perceived an increasing procrastination in me and in my colleagues, by working on the same tasks over a relatively long time (let's say, 6 months or more) without innovation and environment changes." Other reasons included not seeing a clear VISION OR DIRECTION in which the project is or should be going (7) and missing REWARD for high-quality work (6). Regarding the WORK ENVIRONMENT, participants named STRESS (6) due to tight deadlines or economic pressure ("the company's economic condition deteriorated"). Moreover, bad MANAGEMENT (8) or TEAM STRUCTURE (5) were named. An example for bad management would be "[h]aving a supervisor/architect who is very poor at communicating his design goals and ideas, and refuses to accept that this is the case, even when forcibly reminded.". CHANGES IN ATTITUDE may happen due to personal issues (e.g., getting divorced) or due to shifting priorities (e.g., friends and family getting more important). When developers are being promoted to team leader or manager, they SHIFT TOWARDS OTHER TASKS, resulting in a declining programming performance.

AGE-RELATED DECLINE was described in both samples, but the more elaborate answers came from the experienced developers. We consider the investigation of age-related performance decline in software development, together with the consequences for individual developers and the organization, to be an important area for future research. To illustrate the effects that age-related decline may have, we provide four verbatim quotes by experienced developers:

> "In my experience (I started programming in 1962), new languages, systems, hardware became more complex and more diverse, programming became more complex. In my 50s I found it difficult to keep up with new paradigms and languages. So I turned to technical writing and eventually stopped programming." (software developer, age 72)

> "For myself, it's mostly the effects of aging on the brain. At age 66, I can't hold as much information short-term memory, for example. In general, I am more forgetful. I can compensate for a lot of that by writing simpler functions with clean interfaces. The results are still good, but my productivity is much slower than when I was younger." (software architect, age 66)

> "Programming ability is based on desire to achieve. In the early years, it is a sort of competition. As you age, you begin to realize that outdoing your peers isn't all that rewarding. [...] I found that I lost a significant amount of my focus as I became 40, and started using drugs such as ritalin to enhance my abilities. This is pretty common among older programmers." (software developer, age 60)

> "I've been in the software industry for 36 years. [...] It seems as if for the first half or two thirds of that time I was fortunate to be involved in areas at the forefront of the technology wave [...]. For the last 10-15 years though, I have increasingly had the feeling that waves of technology were passing me by [...]. Once I do start to get involved [...] there is a huge learning curve to overcome and I labour to deliver stories as rapidly as younger colleagues who have been immersed in the relevant technology for longer." (software developer, age 57)

5.4 Relationships

The only relationships we added are related to the concept of **mentoring**. As mentioned above, participants described mentors as an important source for FEEDBACK and as MOTIVATORS. Thus, we connected **mentoring** to the corresponding concepts **motivation** and **feedback** in the revised conceptual theory.

Phase 3: To refine and elaborate certain concepts of our preliminary conceptual theory, we conducted a second inductive step, collecting data from two additional samples of software developers. We added details about *individual differences* and *task contexts* that foster the formation of SDExp, and further investigated concepts such as *monitoring*, *mentoring* and *self-reflection*, which are related to *deliberate practice*. We also asked about *performance decline* over time and identified *age-related decline* as a problem for older software developers.

6 EXPERIENCE AND EXPERTISE

Since software developers' expertise is difficult to measure [78], researchers often rely on proxies for this abstract concept [95]. We investigated the relationship and validity of the two proxies *length of experience* and *self-assessed expertise* to provide guidance for researchers.

6.1 Programming Experience vs. Expertise

As mentioned above, we asked participants for their general and Java programming experience (years) and for a self-assessment of their general and Java expertise (semantic differential from 1=novice to 6=expert), see Table 1 and Figure 5. To explore how experience, self-assessed expertise, and other variables are related, we employed the nonparametric *Spearman's rank correlation coefficient* (ρ). Our interpretation of ρ is based on Hinkle et al.'s scheme [47]: low ($0.3 \leq |\rho| < 0.5$), moderate ($0.5 \leq |\rho| < 0.7$), high ($0.7 \leq |\rho| < 0.9$), and very high correlation ($0.9 \leq |\rho| \leq 1$). We chose this nonparametric test because not all variables we tested had interval scaling and not all of them were normally distributed.

We highlight important correlations in the following and provide the complete correlation table as supplementary material [6]. For samples $S1$ and $S2$, the general experience in years (GE) correlates at least moderately with the self-assessed general expertise rating (GR_{sem}) and the participants' age in years. Interestingly, this correlation cannot be observed for the experienced developers ($S3$). For the active Java developers ($S2$), the general experience (GE) and the Java experience (JE) have a high correlation. The Java experience (JE) has a high correlation with the self-assessed Java expertise rating (JR_{sem}) for all three samples, and a moderate correlation with the age for the active Java developers ($S2$).

From the observed correlations, we cannot draw consistent conclusions that are valid for all three samples and for both types of experience (general and Java). Our interpretation of these results is that, depending on the background of the participants, experience in years can or cannot be a valid proxy for (self-assessed) programming expertise. Generally, despite the fact that most researchers would probably agree with the definition of expertise as achieving "outstanding performance" [31], in many empirical studies programming expertise has been operationalized as years (or months) of programming experience [95, 100]. Our results, which suggest that this operationalization may not be valid, is in line with studies showing that excellent software professionals have broader but not necessarily longer experience [22, 98–100].

6.2 Validity of Expertise Self-assessments

In the previous subsection, we motivated that experience may not always be a valid proxy for expertise. We were also interested in the validity of self-assessed expertise, which is, like other self-reports, context-dependent [93]. The validity of self-assessed expertise is related to the concept of **self-reflection** in our conceptual theory, but has also methodological implications for software engineering research in general, because self-assessed programming expertise is often used in studies with software developers to differentiate between novices and experts [95]. To analyze the influence of question context on expertise self-assessments, we asked the participants in samples $S2$ and $S3$ for a second self-assessment of their Java expertise at the end of the online survey. At that point, we did not only provide a semantic differential scale like in the beginning of the survey (JR_{sem}, see Table 1), but also a description of the rating scale stages based on the 5-stage *Dreyfus model* of skill acquisition [24] (JR_{dre}), ranging from *novice* (1) to *expert* (5). This model has been applied in various contexts, but researchers also discussed its limitations [84]. We based our description of the Dreyfus model on a later description by Stuart Dreyfus [23] and an adapted version by Andy Hunt [49]. We provide the description of the five stages, which we used in the focused questionnaire, as supplementary material [6]. The goal of this setup was to investigate if providing additional context has a significant influence on developers' self-assessment compared to a semantic differential scale without context.

When designing the initial questionnaire, we chose a 6-point scale for the expertise rating such that participants have to decide whether they consider themselves to be either on the novice (1-3) or expert (4-6) side of the scale, without the option to select a middle value [35, 80]. To be able to compare the ratings, we had to adjust JR_{sem} to be in range $[1, 5]$ using the following function: $adj(x) = \frac{1}{5} + \frac{4}{5}x$. To test for significant differences between the two ratings, we applied the non-parametric two-sided *Wilcoxon signed rank test* [116] and report the corresponding p-value (p_w). To measure the effect size, we used *Cliff's delta* (δ) [17]. Our interpretation of δ is based on the guidelines by Kitchenham et al. [55]. Moreover, we report the confidence interval of δ at a 95% confidence level (CI_δ).

The Wilcoxon signed rank test indicated that JR_{dre} is significantly higher than JR_{sem} for the experienced developers in $S3$ ($p_w = 0.0009$), but the difference is not significant for the active Java developers in $S2$ ($p_w = 0.47$). Cliff's δ shows only a negligible effect for $S2$ ($\delta = 0.08,\ CI_\delta = [-0.20, 0.04]$), but a small positive effect for $S3$ ($\delta = 0.17,\ CI_\delta = [0.004, 0.33]$), i.e., experienced developers tended to adjust their self-assessments to a higher rating after we provided context. A possible interpretation of this result could be found in the *Dunning-Kruger effect* [60], which is one form of the *illusory superiority bias* [48] where individuals tend to overestimate their abilities. One result of Kruger and Dunning is that participants with a high skill-level underestimate their ability and performance relative to their peers [60]. This may have happened in the sample with experienced developers ($S3$) when they assessed their Java expertise using the semantic differential scale. When we provided context in form of the Dreyfus model, they adjusted their ratings to a more adequate rating, whereas the less experienced developers ($S2$) stuck to their, possibly overestimated, ratings. We cannot conclude that the Dreyfus model in fact leads to more adequate ratings for experienced developers, because we do not have the data to assess the validity of their ratings. However, we can conclude that the way we asked developers to assess their Java programming expertise was influenced by the context we provided.

Experience and expertise: Neither developers' experience measured in years nor the self-assessed programming expertise ratings yielded consistent results across all settings. One direction for future work is to investigate and compare different expertise rating scales to provide guidance for researchers designing studies with expertise self-assessments.

7 LIMITATIONS AND THREATS TO VALIDITY

Since we conducted mixed-methods research, we assess the limitations and threats to validity of our study in terms of the typical quantitative categories *internal* and *external validity* [51], but we will also apply the qualitative evaluation criteria *credibility*, *originality*, *resonance*, and *usefulness* [13].

Internal validity: In our analysis of expertise self-assessments (see Section 6.2), we cannot rule out that a confounding factor lead to the higher self-assessments of experienced developers ($S3$). However, although we used the same questionnaire for $S2$ and $S3$, the effect was only significant and non-negligible for $S3$. Our goal was not to be able to quantify the effect of context on developers' self-assessment, but to show that it exists to motivate future research on this aspect.

External validity: The main limitation affecting external validity is our focus on Java and on open source software development, in particular on GH and SO users. Moreover, as one of three samples targeted experienced developers and only five participants identified themselves as female, our results may be biased towards experienced male developers. Nevertheless, we are confident that our theory is also valid for other developer populations, because of the abstract nature of its core concepts and their grounding in related work. Moreover, although we contacted open source developers, many of them reported on their experiences working in companies (see, e.g., the concepts *work/task context*).

Qualitative evaluation criteria: To support *credibility* of our findings, we not only inductively built our theory from surveys with 335 software developers, but also deductively included results from related work on expertise and expert performance. We constantly compared the answers between all three samples and mapped them to overarching concepts and categories. For the core concepts general/task-specific *knowledge* and *experience*, and the connection of *individual differences*, *work context*, *behavior*, and *performance*, we observed theoretical saturation in the way that those concepts were frequently named and the descriptions did not contradict the relationships we modeled. However, as we only collected data from three samples of developers, the concepts, and in particular the categories we added in phase 3, have to be validated using more data to achieve a higher level of theoretical saturation. In terms of *originality*, we not only contribute a first conceptual theory of SDExp, but also a research design for theory building that other software engineering researchers can adapt and apply. Regarding the *resonance* of our theory, the feedback, in particular from samples $S2$ and $S3$ with focused questions directly related to theory concepts, was generally positive. Participants described their participation as a "very informative experience" and a "nice opportunity to reflect". However, there was some negative feedback regarding the Java focus, especially in sample $S3$. Participants were mainly asking why we concentrated on Java, not questioning the general decision to focus on one particular programming language for some questions. To motivate the *usefulness* of our theory, we refer to Section 9, which contains short summaries of our findings targeting researchers, software developers, and their employers.

Other limitations: The qualitative analysis and theory-building was mainly conducted by the first author and was then discussed with the second author. We tried to mitigate possible biases introduced by us as authors of the theory by embedding our initial GT in related work on expertise and expert performance (see Section 4) and then again collecting data to further refine the resulting conceptual theory (see Section 5). However, when theorizing, there will always be an "uncodifiable step" that relies on the imagination of the researcher [61, 113].

8 RELATED WORK

Expertise research in **software engineering** mainly focused on expert recommendation, utilizing information such as change history [53, 71, 78], usage history [66, 112], bug reports [3], or interaction data [34, 86]. Investigated aspects of software development expertise (SDExp) include programming experience [95], age [81], developer fluency [117], and desired attributes of software engineers [63] and managers [54]. Moreover, similar to our study, Graziotin et al. observed that vision and goal-setting are related to developers' performance [40]. However, as mentioned above, up to now there was no theory combining those individual aspects.

Beside the references mentioned in the description of our theory, the psychological constructs **personality**, **motivation**, and **mental ability** provide many links to theories and instruments from the field of psychology. To assess developers' **personality**, e.g., one could employ the *International Personality Item Pool* (IPIP) [37], measuring the *big five personality traits*. There have been many studies investigating the personality of software developers [20]. Cruz et al. conclude in their systematic mapping study that the evidence from analyzed papers is conflicting, especially for the area of individual performance. Thus, more research is needed to investigate the connection between personality and expert performance. Our theory can help to identify confounding factors affecting performance, in particular the interplay between an individual's mental abilities, personality, motivation, and his/her general and task-specific knowledge and experience. The connection between mental abilities, personality, and domain knowledge in expertise development has, for example, been described by Ackerman and Beier's [1].

The concepts of **communication** and **problem-solving skills** have been thoroughly described in psychology literature [45, 64, 87]. Researchers can use this knowledge when designing studies about the influence of such skills on the formation of SDExp. The other two general skills we included in our theory, CONTINUOUS LEARNING and ASSESSING TRADE-OFFS, have also been described by Li et al. [63], who identified *continuously improving* and *effective decision-making* as critical attributes of great software engineers.

Very closely related to the concept of **deliberate practice** [29], which we included in our theory, is the concept of *self-directed learning* [73] that connects our work to educational research. Similar to our theory, motivation and self-monitoring are considered to be important aspects of self-directed learning [73]. To capture the **motivation** of developers one could adapt ideas from *self-determination theory* [90] or McClelland's theory of the *big three motives* [69]. There also exist instruments like the *Unified Motive Scales (UMS)* [92] to assess human motivation, which can be utilized in studies. Beecham et al. [8] conducted a systematic literature review of motivation in software engineering. While many studies reported that software developers' motivation differs from other

groups, the existing models diverge and "there is no clear understanding of [...] what motivates Software Engineers." Nevertheless, the authors name "problem solving, working to benefit others and technical challenge" as important job aspects that motivate developers. This is very similar to our categories WORK AS CHALLENGE and HELPING OTHERS, which we assigned to the concept **motivation** in our theory. An area related to motivation is the (perceived) productivity of individual developers [15, 75] or software development teams [42, 89]. The results from existing studies in this area can be adapted to assess the performance of developers for **monitoring**, **feedback**, and **self-reflection** [76, 108]. Beside their connection to existing software engineering research, those concepts also connect our theory to two additional areas of psychology: *metacognition* ("knowledge about one's own knowledge [... and] performance") [32] and *self-regulation* [118].

To measure **mental abilities**, test like the *WAIS-IV* [115] or the graphical *mini-q* test [7] can be employed. As motivated above, the connection between aging and expertise [58], and in particular how a (perceived) *age-related performance decline* influences individuals and how they compensate this decline, are important directions for future research. Considering the phenomenon of global population aging [65], the number of old software developers is likely to increase in the next decades. With their experience and knowledge, those developers are a valuable part of software development teams. However, as our qualitative data suggests, they may become unsatisfied with their jobs and may even drop out of software development.

To assess the **performance** of individual software developers, researchers can choose from various existing software metrics [33, 52]. Especially maintainability metrics [18] are of interest, because in our study, maintainability was the most frequently named source code property of experts. Tests about general programming **knowledge** could be derived from literature about typical programming interview questions [4, 72, 79]. To assess task-specific Java **knowledge**, one could rely on commercially available tests like the exams for Oracle's Java certification. Britto et al.[10] report on their experience measuring learning results and the associated effect on performance in a large-scale software project. Their results can help measuring the concepts **education** and **performance**.

9 SUMMARY AND FUTURE WORK

In this paper, we presented a conceptual theory of software development expertise (SDExp). The theory is grounded in the answers of an online survey with 355 software developers and in exiting literature on expertise and expert performance. Our theory describes various properties of SDExp and factors fostering or hindering its development. We classified our theory as a *teleological process theory* that views "development as a repetitive sequence of goal formulation, implementation, evaluation, and modification of goals based on what was learned" [110]. Our task-specific view of SDExp, together with the concept of deliberate practice and the related feedback cycle, fits this framing, assuming that developers' goal is to become experts in certain software development tasks.

We reached a diverse set of experienced and less experienced developers. However, due to the focus on Java and open source software, future work must investigate the applicability of our results to other developer populations. We plan to add more results from existing studies in software engineering and psychology to our theory and to conduct own studies based on our theory. In particular, we want to broaden the scope to include more tasks not directly related to programming. Nevertheless, the theory is already useful for researchers, software developers, and their employers. In the following, we will briefly summarize our findings with a focus on those target audiences.

Researchers: Researchers can use our methodological findings about (self-assessed) expertise and experience (see Section 6) when designing studies involving self-assessments. If researchers have a clear understanding what distinguishes novices and experts in their study setting, they should provide this context [93] when asking for self-assessed expertise and later report it together with their results. We motivated why we did not describe expertise development in discrete steps (see Section 4), but a direction for future work could be to at least develop a standardized description of *novice* and *expert* for certain tasks, which could then be used in semantic differential scales. To design concrete experiments measuring certain aspects of SDExp, one needs to operationalize the conceptual theory [44]. We already linked certain concepts to measurement instruments such as UMS (motivation), WAIS-IV (mental abilities), or IPIP (personality). We also mentioned static analysis tools to measure code quality and simple productivity measures such as commit frequency and number of issues closed. This enables researchers to design experiments, but also to re-evaluate results from previous experiments. There are, e.g., no coherent results about the connection of individual differences and programming performance yet. One could review studies on developers' motivation [8] and personality [20] in the context of our theory, to derive a research design for analyzing the interplay of individual differences and SDExp.

Developers: Software developers can use our results to see which properties are distinctive for experts in their field, and which behaviors may lead to becoming a better software developer. For example, the concept of deliberate practice, and in particular having challenging goals, a supportive work environment, and getting feedback from peers are important factors. For "senior" developers, our results provide suggestions for being a good mentor. Mentors should know that they are considered to be an important source for feedback and motivation, and that being patient and being open-minded are desired characteristics. We also provide first results on the consequences of age-related performance decline, which is an important direction for future work.

Employers: Employers can learn what typical reasons for demotivation among their employees are, and how they can build a work environment supporting the self-improvement of their staff. Beside obvious strategies such as offering training sessions or paying for conference visits, our results suggest that employers should think carefully about how information is shared between their developers and also between the development team and other departments of the company. Facilitating meetings that explicitly target information exchange and learning new skills should be a priority of every company that cares about the development of their employees. Finally, employers should make sure to have a good mix of continuity and change in their software development process, because non-challenging work, often caused by tasks becoming routine, is an important demotivating factor for software developers.

ACKNOWLEDGMENTS

We thank the survey participants, Bernhard Baltes-Götz, Daniel Graziotin, and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Phillip L. Ackerman and Margaret E. Beier. 2006. Methods for Studying the Structure of Expertise: Psychometric Approaches. In *The Cambridge Handbook of Expertise and Expert Performance*. 147–165.
[2] American Psychological Association. 2015. *APA Dictionary of Psychology*.
[3] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who Should Fix This Bug?. In *28th International Conference on Software Engineering (ICSE 2006)*. 361–370.
[4] Adnan Aziz, Tsung-Hsien Lee, and Amit Prakash. 2012. *Elements of Programming Interviews*. CreateSpace Independent Publishing Platform.
[5] Sebastian Baltes and Stephan Diehl. 2016. Worse Than Spam: Issues In Sampling Software Developers. In *10th International Symposium on Empirical Software Engineering and Measurement (ESEM 2016)*. 52:1–52:6.
[6] Sebastian Baltes and Stephan Diehl. 2018. Towards a Theory of Software Development Expertise — Supplementary Material. (2018). https://doi.org/10.5281/zenodo.1299798
[7] Tanja Gabriele Baudson and Franzis Preckel. 2016. mini-q: Intelligenzscreening in drei Minuten. *Diagnostica* 62, 3 (2016), 182–197.
[8] Sarah Beecham, Nathan Baddoo, Tracy Hall, Hugh Robinson, and Helen Sharp. 2008. Motivation in Software Engineering: A systematic literature review. *Information and Software Technology* 50, 9 (2008), 860–878.
[9] Gunnar R. Bergersen, Dag I. K. Sjoberg, and Tore Dyba. 2014. Construction and Validation of an Instrument for Measuring Programming Skill. *IEEE Transactions on Software Engineering* 40, 12 (2014), 1163–1184.
[10] Ricardo Britto, Darja Smite, and Lars-Ola Damm. 2016. Experiences from Measuring Learning and Performance in Large-Scale Distributed Software Development. In *10th International Symposium on Empirical Software Engineering and Measurement (ESEM 2016)*. 1–6.
[11] Guillermo Campitelli and Fernand Gobet. 2011. Deliberate Practice. *Current Directions in Psychological Science* 20, 5 (2011), 280–285.
[12] Pierre Carbonnelle. 2016. PYPL PopularitY of Programming Language: March 2016. (2016). http://pypl.github.io/PYPL.html
[13] Kathy Charmaz. 2014. *Constructing grounded theory* (2nd ed.). Sage.
[14] Michelene T. H. Chi. 2006. Two Approaches to the Study of Expert's Characteristics. In *The Cambridge Handbook of Expertise and Expert Performance*.
[15] Earl Chrysler. 1978. Some Basic Determinants of Computer Programming Productivity. *Communications of the ACM* 21, 6 (1978), 472–483.
[16] K. Alec Chrystal and Paul D. Mizen. 2003. Goodhart's law: Its origins, meaning and implications for monetary policy. *Central banking, monetary theory and practice: Essays in honour of Charles Goodhart* 1 (2003), 221–243.
[17] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.
[18] D. Coleman, D. Ash, B. Lowther, and P. Oman. 1994. Using metrics to evaluate software system maintainability. *Computer* 27, 8 (1994), 44–49.
[19] Juliet Corbin and Anselm Strauss. 2008. *Basics of qualitative research* (3rd ed.). SAGE Publications.
[20] Shirley Cruz, Fabio Q. B. da Silva, and Luiz Fernando Capretz. 2015. Forty years of research on personality in software engineering: A mapping study. *Computers in Human Behavior* 46 (2015), 94–113.
[21] Bill Curtis. 1984. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *7th International Conference on Software Engineering (ICSE 1984)*.
[22] Oscar Dieste, Alejandrina M. Aranda, Fernando Uyaguari, Burak Turhan, Ayse Tosun, Davide Fucci, Markku Oivo, and Natalia Juristo. 2017. Empirical evaluation of the effects of experience on code quality and programmer productivity: An exploratory study. *Empirical Software Engineering* 22, 5 (2017), 2457–2542.
[23] Stuart E. Dreyfus. 2004. The five-stage model of adult skill acquisition. *Bulletin of science, technology & society* 24, 3 (2004), 177–181.
[24] Stuart E. Dreyfus and Hubert L. Dreyfus. 1980. A five-stage model of the mental activities involved in directed skill acquisition. *University of California, Berkeley* ORC 80-2 (1980), 1–22.
[25] Alastair Dunsmore and Marc Roper. 2000. A comparative evaluation of program comprehension measures. *Department of Computer Science, University of Strathclyde* EFoCS-35-2000 (2000), 1–7.
[26] K. Anders Ericsson. 2006. An Introduction to Cambridge Handbook of Expertise and Expert Performance: Its Development, Organization, and Content. In *The Cambridge Handbook of Expertise and Expert Performance*. 3–19.
[27] K. Anders Ericsson. 2006. The Influence of Experience and Deliberate Practice on the Development of Superior Expert Performance. In *The Cambridge Handbook of Expertise and Expert Performance*. 683–703.
[28] K. Anders Ericsson, Neil Charness, Paul J. Feltovich, and Robert R. Hoffman (Eds.). 2006. *The Cambridge Handbook of Expertise and Expert Performance*.
[29] K. Anders Ericsson, Ralf T. Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.
[30] K. Anders Ericsson, Michael J. Prietula, and Edward T. Cokely. 2007. The making of an expert. *Harvard business review* 85, 7/8 (2007), 114.
[31] K. Anders Ericsson and Jacqui Smith. 1991. Prospects and limits of the empirical study of expertise: An introduction. In *Toward a general theory of expertise: Prospects and limits*. Vol. 344. 1–38.
[32] Paul J. Feltovich, Michael J. Prietula, and K. Anders Ericsson. 2006. Studies of Expertise from Psychological Perspectives. In *The Cambridge Handbook of Expertise and Expert Performance*. 41–67.
[33] Norman Fenton and James Bieman. 2015. *Software Metrics: A Rigorous and Practical Approach*. CRC Press.
[34] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a Programmer's Activity Indicate Knowledge of Code?. In *6th European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2007)*.
[35] Ron Garland. 1991. The Mid-Point on a Rating Scale: Is it Desirable? *Marketing Bulletin* 2, Research Note 3 (1991), 66–70.
[36] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction.
[37] Lewis R. Goldberg. 1999. A broad-bandwidth, public domain, personality inventory measuring the lower-level facets of several five-factor models. *Personality psychology in Europe* 7, 1 (1999), 7–28.
[38] Charles A. E. Goodhart. 1984. *Monetary Theory and Practice: The UK Experience*. Macmillan Press.
[39] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *10th International Working Conference on Mining Software Repositories (MSR 2013)*. IEEE.
[40] Daniel Graziotin, Xiaofeng Wang, and Pekka Abrahamsson. 2015. How do you feel, developer? An explanatory theory of the impact of affects on programming performance. *PeerJ Computer Science* 1 (2015), e18.
[41] Shirley Gregor. 2006. The nature of theory in information systems. *MIS quarterly* 30, 3 (2006), 611–642.
[42] Lucas Gren. 2017. The Links Between Agile Practices, Interpersonal Conflict, and Perceived Productivity. In *21st International Conference on Evaluation and Assessment in Software Engineering (EASE 2017)*. 292–297.
[43] David Z. Hambrick and Elizabeth J. Meinz. 2011. Limits on the Predictive Power of Domain-Specific Experience and Knowledge in Skilled Performance. *Current Directions in Psychological Science* 20, 5 (2011), 275–279.
[44] Jo E. Hannay, Dag I. K. Sjoberg, and Tore Dyba. 2007. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering* 33, 2 (2007), 87–107.
[45] Owen Hargie (Ed.). 2006. *The Handbook of Communication Skills* (3 ed.). Routledge.
[46] James D. Herbsleb and Audris Mockus. 2003. Formulation and preliminary test of an empirical theory of coordination in software engineering. In *4th European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2003)*. 138–137.
[47] Dennis E. Hinkle, William Wiersma, and Stephen G. Jurs. 1979. *Applied statistics for the behavioral sciences*. Rand McNally College Publishing.
[48] Vera Hoorens. 1993. Self-enhancement and superiority biases in social comparison. *European review of social psychology* 4, 1 (1993), 113–139.
[49] Andy Hunt. 2008. *Pragmatic Thinking and Learning: Refactor Your Wetware*. Pragmatic bookshelf.
[50] Earl Hunt. 2006. Expertise, Talent, and Social Encouragement. In *The Cambridge Handbook of Expertise and Expert Performance*. 31–38.
[51] R. Burke Johnson, Anthony J. Onwuegbuzie, and Lisa A. Turner. 2007. Toward a definition of mixed methods research. *Journal of mixed methods research* 1, 2 (2007), 112–133.
[52] Capers Jones. 2008. *Applied Software Measurement: Global Analysis of Productivity and Quality* (3 ed.). McGraw-Hill Education.
[53] Huzefa H. Kagdi, Maen Hammad, and Jonathan I. Maletic. 2008. Who can help me with this source code change?. In *24th IEEE International Conference on Software Maintenance (ICSM 2008)*. 157–166.
[54] Eirini Kalliamvakou, Christian Bird, Thomas Zimmermann, Andrew Begel, Robert DeLine, and Daniel M. German. 2017. What Makes a Great Manager of Software Engineers? *IEEE Transactions on Software Engineering* Early Access Articles, 1 (2017), 1.
[55] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart M. Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust Statistical Methods for Empirical Software Engineering. *Empirical Software Engineering* 22, 2 (2017), 579–630.
[56] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734.
[57] Andrew J. Ko and Bob Uttl. 2003. Individual differences in program comprehension strategies in unfamiliar programming systems. In *11th International*

Workshop on Program Comprehension (IWPC 2003). 175–184.
[58] Ralf Th. Krampe and Neil Charness. 2006. Aging and Expertise. In *The Cambridge Handbook of Expertise and Expert Performance*. 723–742.
[59] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 6 (2012), 18–21.
[60] Justin Kruger and David Dunning. 1999. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology* 77, 6 (1999), 1121.
[61] Ann Langley. 1999. Strategies for theorizing from process data. *Academy of management review* 24, 4 (1999), 691–710.
[62] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: A study of developer work habits. In *28th International Conference on Software Engineering (ICSE 2006)*. 492–501.
[63] Paul Luo Li, Andrew J. Ko, and Jiamin Zhu. 2015. What Makes A Great Software Engineer?. In *37th International Conference on Software Engineering (ICSE 2015)*.
[64] Edwin A. Locke, Gary P. Latham, Ken J. Smith, and Robert E. Wood. 1990. *A Theory of Goal Setting & Task Performance* (1 ed.). Prentice Hall.
[65] Wolfgang Lutz, Warren Sanderson, and Sergei Scherbov. 2008. The coming acceleration of global population ageing. *Nature* 451, 7179 (2008), 716–719.
[66] David Ma, David Schuler, Thomas Zimmermann, and Jonathan Sillito. 2009. Expert Recommendation with Usage Expertise. In *25th IEEE International Conference on Software Maintenance (ICSM 2009)*. 535–538.
[67] M. Lynne Markus and Daniel Robey. 1988. Information technology and organizational change: Causal structure in theory and research. *Management science* 34, 5 (1988), 583–598.
[68] Antoinette McCallin. 2003. Grappling with the literature in a grounded theory study. *Contemporary Nurse* 15, 1-2 (2003), 61–69.
[69] David C. McClelland. 1987. *Human motivation*. Cambridge University Press.
[70] Robert R. McCrae and Oliver P. John. 1992. An Introduction to the Five-Factor Model and Its Applications. *Journal of Personality* 60, 2 (1992), 175–215.
[71] David W. McDonald and Mark S. Ackerman. 2000. Expertise recommender: A flexible recommendation system and architecture. In *Proceeding on the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW 2000)*. 231–240.
[72] Gayle Laakmann McDowell. 2014. *Cracking the Coding Interview* (5th ed.).
[73] Sharan B. Merriam, Rosemary S. Caffarella, and Lisa M. Baumgartner. 2007. *Learning in Adulthood: A Comprehensive Guide* (3 ed.). John Wiley & Sons.
[74] Merriam-Webster.com. 2018. expert. (2018). http://www.merriam-webster.com/dictionary/expert
[75] Andre N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. The Work Life of Developers: Activities, Switches and Perceived Productivity. *IEEE Transactions on Software Engineering* 43, 12 (2017).
[76] Andre N. Meyer, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. Design Recommendations for Self-Monitoring in the Workplace: Studies in Software Development. *Proceedings of the ACM on Human-Computer Interaction* 1, CSCW (2017).
[77] Harald A. Mieg. 2006. Social and Sociological Factors in the Development of Expertise. In *The Cambridge Handbook of Expertise and Expert Performance*.
[78] Audris Mockus and James D. Herbsleb. 2002. Expertise browser: A quantitative approach to identifying expertise. In *24th International Conference on Software Engineering (ICSE 2002)*. 503–512.
[79] John Mongan, Eric Gigure, and Noah Kindler. 2013. *Programming interviews exposed* (3rd ed.). John Wiley & Sons.
[80] Guy Moors. 2008. Exploring the effect of a middle response category on response style in attitude measurement. *Quality & quantity* 42, 6 (2008), 779–794.
[81] Patrick Morrison and Emerson Murphy-Hill. 2013. Is programming knowledge related to age? An exploration of Stack Overflow. In *10th International Working Conference on Mining Software Repositories (MSR 2013)*. 69–72.
[82] Janice M. Morse. 2007. Sampling in grounded theory. In *The SAGE Handbook of Grounded Theory*. 229–244.
[83] Stephan J. Motowidlo, Walter C. Borman, and Mark J. Schmit. 1997. A Theory of Individual Differences in Task and Contextual Performance. *Human Performance* 10, 2 (1997), 71–83.
[84] Adolfo Pena. 2010. The Dreyfus model of clinical problem-solving skills acquisition: A critical perspective. *Medical Education Online* 15 (2010), 1–11.
[85] Paul Ralph. 2018. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE TSE* Early Access (2018).
[86] Romain Robbes and David Rothlisberger. 2013. Using Developer Interaction Data to Compare Expertise Metrics. In *10th International Working Conference on Mining Software Repositories (MSR 2013)*. 297–300.
[87] S. Ian Robertson. 2016. *Problem Solving: Perspectives from Cognition and Neuroscience* (2 ed.). Routledge.
[88] Pierre N. Robillard. 1999. The Role of Knowledge in Software Development. *Communications of the ACM* 42, 1 (1999), 87–92.
[89] D. Rodríguez, M. A. Sicilia, E. García, and R. Harrison. 2012. Empirical findings on team size and productivity in software development. *Journal of Systems and Software* 85, 3 (2012), 562–570.
[90] Richard M. Ryan and Edward L. Deci. 2000. Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American Psychologist* 55, 1 (2000), 68–78.
[91] Johnny Saldana. 2015. *The coding manual for qualitative researchers*. Sage.
[92] Felix D. Schonbrodt and Friederike X. R. Gerstenberg. 2012. An IRT analysis of motive questionnaires: The unified motive scales. *Journal of Research in Personality* 46, 6 (2012), 725–742.
[93] Norbert Schwarz and Daphna Oyserman. 2001. Asking questions about behavior: Cognition, communication, and questionnaire construction. *American Journal of Evaluation* 22, 2 (2001), 127–160.
[94] Ben Shneiderman and Richard Mayer. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences* 8, 3 (1979), 219–238.
[95] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *36th International Conference on Software Engineering (ICSE 2014)*. 378–389.
[96] Janice Singer, Timothy C. Lethbridge, Norman G. Vinson, and Nicolas Anquetil. 1997. An examination of software engineering work practices. In *1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1997)*.
[97] Dag I. K. Sjoberg, Tore Dyba, Bente C. D. Anda, and Jo E. Hannay. 2008. Building theories in software engineering. In *Guide to Advanced Empirical Software Engineering*. 312–336.
[98] Sabine Sonnentag. 1995. Excellent software professionals: Experience, work activities, and perception by peers. *Behaviour & Information Technology* 14, 5 (1995), 289–299.
[99] Sabine Sonnentag. 1998. Expertise in professional software design: A process study. *Journal of Applied Psychology* 83, 5 (1998), 703–715.
[100] Sabine Sonnentag, Cornelia Niessen, and Judith Volmer. 2006. Expertise in Software Design. In *The Cambridge Handbook of Expertise and Expert Performance*.
[101] Lauren A. Sosniak. 2006. Retrospective Interviews in the Study of Expertise and Expert Performance. In *The Cambridge Handbook of Expertise and Expert Performance*. 287–301.
[102] Stack Exchange Inc. 2015. Stack Exchange Data Dump: August 18, 2015. (2015). https://archive.org/details/stackexchange/
[103] Stack Exchange Inc. 2016. 2015 Developer Survey. (2016). http://stackoverflow.com/research/developer-survey-2015
[104] Abbas Tashakkori and Charles Teddlie. 1998. *Mixed methodology: Combining qualitative and quantitative approaches*. Sage.
[105] Robert Thornberg. 2012. Informed grounded theory. *Scandinavian Journal of Educational Research* 56, 3 (2012), 243–259.
[106] TIOBE software BV. 2016. TIOBE Index: March 2016. (2016). http://www.tiobe.com/tiobe_index
[107] Dennis Tourish and Owen Hargie. 2003. Motivating critical upward communication: A key challenge for management decision making. In *Key Issues in Organizational Communication*. Routledge, 188–204.
[108] Christoph Treude, Fernando Figueira Filho, and Uirá Kulesza. 2015. Summarizing and measuring development activity. In *10th European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2015)*. 625–636.
[109] Andrew H. van de Ven. 1989. Nothing is quite so practical as a good theory. *Academy of management review* 14, 4 (1989), 486–489.
[110] Andrew H. van de Ven and Marshall Scott Poole. 1995. Explaining development and change in organizations. *Academy of management review* 20, 3 (1995).
[111] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. 2013. StackOverflow and GitHub: Associations between Software Development and Crowd-sourced Knowledge. In *2013 International Conference on Social Computing (SocialCom 2013)*. 188–195.
[112] Adriana Santarosa Vivacqua and Henry Lieberman. 2000. Agents to assist in finding help. In *2000 Conference on Human factors in computing systems (CHI 2000)*. 65–72.
[113] Karl E. Weick. 1989. Theory Construction as Disciplined Imagination. *Academy of management review* 14, 4 (1989), 516–531.
[114] Robert W. Weisberg. 2006. Modes of Expertise in Creative Thinking: Evidence from Case Studies. In *The Cambridge Handbook of Expertise and Expert Performance*. 761–787.
[115] Lawrence G. Weiss, Donald H. Saklofske, Diane Coalson, and Susan Engi Raiford. 2010. *WAIS-IV clinical use and interpretation: Scientist-practitioner perspectives*. Academic Press.
[116] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics* 1, 6 (1945), 80–83.
[117] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010)*. 137–146.
[118] Barry J. Zimmerman. 2006. Development and Adaption of Expertise: The Role of Self-Regulatory Processes and Beliefs. In *The Cambridge Handbook of Expertise and Expert Performance*. 705–722.