Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09773-y

What distinguishes great software engineers?

Paul Luo Li[1] · **Amy J. Ko**[2] · **Andrew Begel**[3]

Published online: 03 December 2019

Abstract
Great software engineers are essential to the creation of great software. However, today, we lack an understanding of what distinguishes great engineers from ordinary ones. We address this knowledge gap by conducting one of the largest mixed-method studies of experienced engineers to date. We surveyed 1,926 expert engineers, including senior engineers, architects, and technical fellows, asking them to judge the importance of a comprehensive set of 54 attributes of great engineers. We then conducted 77 email interviews to interpret our findings and to understand the influence of contextual factors on the ratings. After synthesizing the findings, we believe that the top five distinguishing characteristics of great engineers are writing good code, adjusting behaviors to account for future value and costs, practicing informed decision-making, avoiding making others' jobs harder, and learning continuously. We relate the findings to prior work, and discuss implications for researchers, practitioners, and educators.

Keywords Software engineering · Software development management · Collaboration · Computer science education

1 Introduction

At the end of the day, to make changes [to software], it still takes a developer, a butt in a seat somewhere, to type [Source Control System] commit. — Dev Manager

Communicated by: Kelly Blincoe

✉ Paul Luo Li
paul.li@microsoft.com

Amy J. Ko
ajko@uw.edu

Andrew Begel
andrew.begel@microsoft.com

[1] Microsoft, Redmond, WA 98052, USA

[2] The Information School, University of Washington, Seattle, WA 98195, USA

[3] Microsoft Research, Redmond, WA 98052, USA

Great software engineers are essential to the creation of great software. Consequently, understanding the attributes that distinguish great engineers is foundational to our world's rapidly growing software ecosystem: companies want to hire and retain great engineers, universities want to train them, and aspiring engineers want to know what it means to be great.

Software engineering research has long aspired to develop this understanding, with many studies rigorously considering attributes of great engineers. For example, starting from the early days of computing, many researchers and practitioners recognized that some software engineers were better than others. Using metrics like lines of code produced and bug rate, early studies attempted to quantify differences between great engineers and ordinary ones in terms of productivity (Sackman et al. 1968). Often performed in laboratory or university settings (Gugerty and Olson 1986; Robillard et al. 2004), studies attempted to isolate the technical coding task. These observations led to the popular software engineering meme of the "10X developer".

Some notions of developer skill are prescriptively encoded in curriculum. For example, the ACM curricular standards define software engineers as: *people who write software to be used in earnest by others*. The curricula enumerates essential knowledge for software engineers; it focuses almost entirely of technical skills like Programming Fundamentals and Software Design (Joint Task Force on Computing Curricula 2014). The focus on skills related to coding is also true of the Software Engineering Book of Knowledge (Bourque et al. 2014), which attempts to enumerate everything a software engineer can and should know to be competent.

While much prior work insists that programming and technical knowledge is central, some evidence suggests that attributes of great engineers also lie in their ability to interact effectively with teammates. For example, many studies examining everyday activities of software engineers find that engineers spend significant time performing non-coding activities (Singer et al. 1997; Perry et al. 1994), including conflict resolution (Gobeli et al. 1998), bug triaging (Anvik et al. 2006; Aranda and Venolia 2009) and information seeking (Ko et al. 2007). Studies have also considered these factors from the perspective of new graduates entering industry, examining the gaps between abilities of new graduates and skills needed to engineer software in the real-world (Radermacher and Walia 2013). Researchers commonly find that being effective engineers requires non-coding attributes, such as confidence, eagerness to learn, and the ability to work effectively with others (Begel 2008; Hewner and Guzdial 2010).

Research on software engineering process also *imply* skills required by engineers. Various guidelines (notably, CMM Herbsleb et al. 1997), methods (e.g., the Spiral method Boehm 1988), and manifestos (e.g., Agile manifesto Beck et al. 2001), suggest many specific ways of working, planning, and deciding that are essential for productive teamwork. These are consistent with claims by luminaries like (Brooks 1995) and others from companies like Microsoft (Brechner 2003) and Google (Fitzpatrick and Collins-Sussman 2009), suggesting that great software engineers possess a collection of attributes that span the socio-technical spectrum that are supportive of organizational needs. Collectively they point to personality traits, technical abilities, and inter-personal skills combining to distinguish great engineers in real-world settings.

Research efforts have culminated recently in two comprehensive studies, synthesizing prior work into higher-level frameworks. Our prior ICSE paper (Li et al. 2015) laid the foundations by providing a comprehensive enumeration of attributes of great engineers, deriving a set of 54 attributes from 59 semi-structured interviews with experienced engineers at Microsoft. Baltes and Diehl (2018) built upon our study by contributing a comprehensive

theory that incorporates the software engineering process with the individual and social characteristics of the engineers. However, both studies lack an understanding of the relative importance of these characteristics. Thus, in this paper, we explore three research questions. First, *are some attributes viewed by experienced engineers as more important than others?* Second, *how do the perceived importance of the attributes vary by the demographic, context, and work experiences of the engineers?* Third, *how do these attributes distinguish great engineers from the rest?*

Answers to these questions require a very large sample of informants, in rich and diverse contexts, making judgments on attributes that characterize great engineers. Therefore, we conducted a large-scale mixed method study of experienced engineers. We quantitatively surveyed 1,926 expert Microsoft engineers, covering 67 countries (13% of all expert Microsoft engineers world-wide at the time of the survey). Additionally, we conducted 77 follow-up interviews to qualitatively interpret the relative importance of attributes and the effects of contextual factors. Our work is one of the largest real-world studies to date of software engineers.

In the rest of this paper, Section 2 discusses our prior interview study and the 54 attributes of great engineers it produced, which serve as the foundation for this study. Section 3 describes the methodology of this study, both the survey and the follow-up interviews. We follow in Section 4 with results. In Section 5 we first discuss prior work, then we synthesize insights and relate those insights to prior work. We discuss takeaways for researchers, educators, and practitioners, as well as threats to validity. Finally, we conclude in Section 6.

2 Attributes of Great Software Engineers

In this section, before describing our study, we review the attributes from our prior study (Li 2016), which formed the basis for our survey and interviews. Figure 1 shows an overview of these attributes and how they relate. The 54 attributes in the prior work include self-focused attributes of the software engineer's personality and of their ability to make effective decisions, as well as externally-focused attributes of the impact that great engineers have on people and products. A prior publication provides thorough descriptions of each attribute (Li et al. 2019).

As Fig. 1 shows, the first group of 18 attributes pertained to engineers' **personalities**:

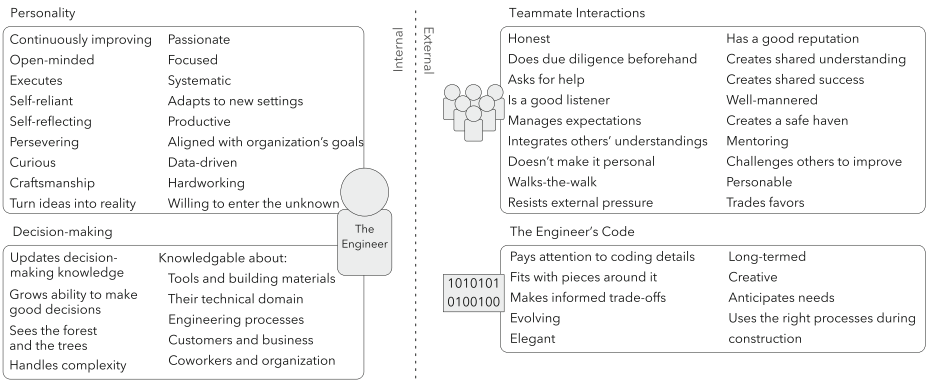

Fig. 1 Model of attributes of great software engineers

That is something that can't be taught. I think it's something a person just has to have [...] They don't need any outside motivation. They just go [...] They have just an inner desire to succeed, and I don't know why. It's not necessarily for the money, it's not necessarily for the recognition. It's just that whatever it is they do, they want to do it extremely well [...] I've seen a lot of smart people that have none of these characteristics [...] — Principal Dev Lead

Some attributes, such as passionate and curious, concerned who great engineers were as people. Informants felt that many of these attributes were intrinsic to the engineer—formed through their upbringing—and would be difficult (if not impossible) to change.

The second group consisted of 9 attributes that pertained to engineers' ability to **make decisions**. By decision-making, we mean "rational decision-making", as described by Simon (1955): assessing the current situation (i.e. understanding when/what decisions were needed), identifying alternate courses of action, gauging probabilistic outcomes, and estimating value of future states.

How do we make, what I often call, "robust decisions?" What's a decision we could make, depending on this range of potential outcomes, which we can't foresee? [...] if we can make a decision that is viable, whether A or B happens, then we don't have to fight about A or B right now. — Technical Fellow

Informants described software engineering as requiring many complex choices about "what software to build and how to build it." Furthermore, beyond book knowledge, great engineers understand how decisions play out in real-world conditions. They not only knew what should happen, but also what can and likely will happen. Combining internal knowledge, mental models that tie the knowledge together, and the cognitive ability to reason about their models, decision-making attributes were self-focused. Yet, unlike personality, the sentiment among informants was that effective decision-making could be learned.

The third group of 17 attributes pertained to engineers' **interactions with teammates**:

The way [this great software engineer] just kind of touches people, just dissolves conflicts right there [...] that magic to make people respect him. That's fun magic, I think that not everyone possesses. — Senior SDE

Most informants believed that great engineers positively influenced teammates. For many informants (whose titles contained "Lead" or "Manager"), this was an important part of their job as managers of other engineers. Attributes concerning interactions with teammates generally revolved around four concepts: being a reasonable person, influencing others, communicating effectively, and building trust. We deconstructed these four concepts into their constituent attributes.

The final group consisted of 9 attributes that pertained to the **software produced by great engineers**:

The style [...] always, an idea, and it was all clean [...] very concise. Just looking at it, you can say, "Okay, this guy, he knew what he was doing." [...] There's no extra stuff. Everything is minimally necessary and sufficient as it should be. It's well thought out off screen. — Senior SDE

Like artists appreciating masterpieces of other artists, our informants, many of whom were great engineers themselves, saw beauty in software produced by other great engineers.

In this paper, we build upon this comprehensive set of attributes of great engineers, using a mixed-method study (quantitative survey and qualitative follow-up interviews) to

understand the relative importance of the attributes and how contextual factors impact their importance. We then synthesize learnings from both studies to provide insights into what distinguishes great engineers.

3 Method

To determine the relative importance of the attributes, we employed a large-scale quantitative survey, seeking to rank the 54 attributes by *agreement* that the attribute was essential for greatness. Additionally, we considered contextual factors that impact the practice of software engineering, based on previous studies (Hewner and Guzdial 2010; Radermacher et al. 2014; Fisher and Margolis 2002; Margolis and Fisher 2003; Carver et al. 2008; Shackelford et al. 2006) (and our own previous study): amount of experience, gender, country, computer science educational background, and type of software (server-side, client-side, or both). Table 2, shown later in Section 4, shows details on how these factors were operationalized. Finally, since spurious correlations are possible with quantitative statistical tests, we used qualitative follow-up interviews to help validate and interpret the findings.

For this study, we chose to continue investigations at Microsoft, the same setting as our previous interview study (Li et al. 2015). Though there are obvious external validity limitations with Microsoft being one organization, it also offers advantages. First, continuing investigations at Microsoft helps construct and internal validity. Microsoft employees share common understandings (e.g. terms and seniority based on titles), which helps consistent interpretation of our questions as well as our interpretation of their responses. Furthermore, the common organizational structure (e.g. job titles) allows us to identify experienced engineers for the studies in a consistent manner (discussed in Section 3.1). Second, Microsoft is a conglomerate of diverse products and settings. These include games, consumer electronics, OS, productivity, search, consumer services, enterprise services, enterprise resource planning, customer relationship management, databases, developer tools, and communications, as well as regional-specific development centers around the world. This rich diversity of contexts actually benefits external validity and increases the prospects of discerning interesting contextual factors. Third, Microsoft consistently utilizes best practices and technologies, as well as employs top talent; this helps to factor out confounding deficiencies. Finally, we acknowledge expediency considerations; two authors are Microsoft employees with access to engineers and organizational permission to perform the studies. We leave further validation of our results in other contexts to future work.

3.1 Survey Method

A key decision in our methodology was determining whose subjective opinions could be considered *valid* judgments of attributes of "greatness." For this study, we made the assumption that experienced engineers were in the best positions to make these judgments, as they regularly make them when interviewing and evaluating other engineers. We included Leads and Managers of engineers in our sampling because, at Microsoft, nearly all of them had hands-on experience as engineers. As with our initial interview study (Li et al. 2015), we adopted the ACM's definition of software engineer (Shackelford et al. 2006): *people who write software to be used in earnest by others*. We operationalized this definition using the Microsoft company directory. We identified software engineers based on titles that entailed "software development" and then consolidated the list of titles, removing various address book wording anomalies. To determine "experienced," we used the approach utilized by

researchers of human expertise (Ericsson et al. 1993). We identified those having achieved some degree of *recognition* as experienced, either through hiring or promotion processes, selecting engineers at or above the Software Development Engineer Level 2 (SDE II) title.

Recognizing that insights of more experienced engineers are valuable (and more difficult to obtain), we had two sampling strata for *experience level*. First, for "experienced," we selected engineers with titles at SDE2 level up to "Senior Software Development Engineer Lead" promotion level; these engineers typically had at least 5 years of experience. Second, for "very experienced," we selected engineers above the promotion level of "Senior Software Development Engineer Lead"; these engineers typically had 10+ years of experience and were often responsible for critical technical areas.

We hosted our anonymous survey on a Microsoft Research website. We emailed engineers asking them to participate, offering a report of the findings and entry into a gift certificate raffle as incentives (two gift cards, $75 each, odds of winning proportional to number of respondents). We personalized the solicitations with the software engineer's name, described the purpose of the research, and explained why we needed their perspective; these steps help to reduce inattentive survey responses (users providing insincere or haphazard responses) (Meade and Craig 2012). Each solicitation had a separate anonymized survey link to prevent multiple submissions (e.g. via bots, which may lead to bias and spurious results). We sent reminder emails after the first week and after one month. The survey was open from Dec 2014 to Feb 2015.

In the survey, after explaining the purpose of the study and the respondent's right not to participate, we asked questions about the respondent's demographics, experience level, and current work context (using numerical input boxes, e.g. years of experiences, or radio select buttons, e.g. server- / client-side software). These are the contextual factors we later correlate with their ratings.

We then sought respondents' ratings for all of the attributes. In anticipation of respondent fatigue, we presented the questions in four groups, corresponding to the four groups from our interview study, discussed in Section 2. To address ordering bias and to enable analysis of incomplete results, we randomized the ordering of the four groups, as well as randomized (separately) the ordering of the attributes within each group. Questions about the attributes were structured and phrased in a similar manner, allowing respondents to quickly read and respond. To illustrate, Fig. 2 shows what the survey looked like for the hardworking attribute. We concluded the survey with an open-ended catch all question, aiming to detect operational problems and missing attributes.

We took several steps to ensure that respondents accurately understood the attributes. In the first iteration of survey construction, we described an engineer who possessed the attribute. We then piloted the survey to identify comprehension issues, leading to changes in wording (e.g. "software engineer" to "developer" to differentiate people that did not write code) adding supporting quotations for 37 attributes, adding clarifications for confusing attributes (e.g. "practices and techniques" was appended with "e.g. unit testing, code reviews, Scrum, etc.").

To get an absolute rating of importance, we asked "If an experienced developer — whose primary responsibility is developing software — did not have this attribute, could you still consider them a great developer?" We gave respondents six Likert-style choices (see Fig. 2): *"Cannot be a great developer if they do not have this," "Very difficult to be a great developer without this, but not impossible," "Can be a great developer without this, but having it helps," "Does not matter if they do not have this, it is irrelevant," "A great developer should not have this; it is not good,"* and *"I do not know."*

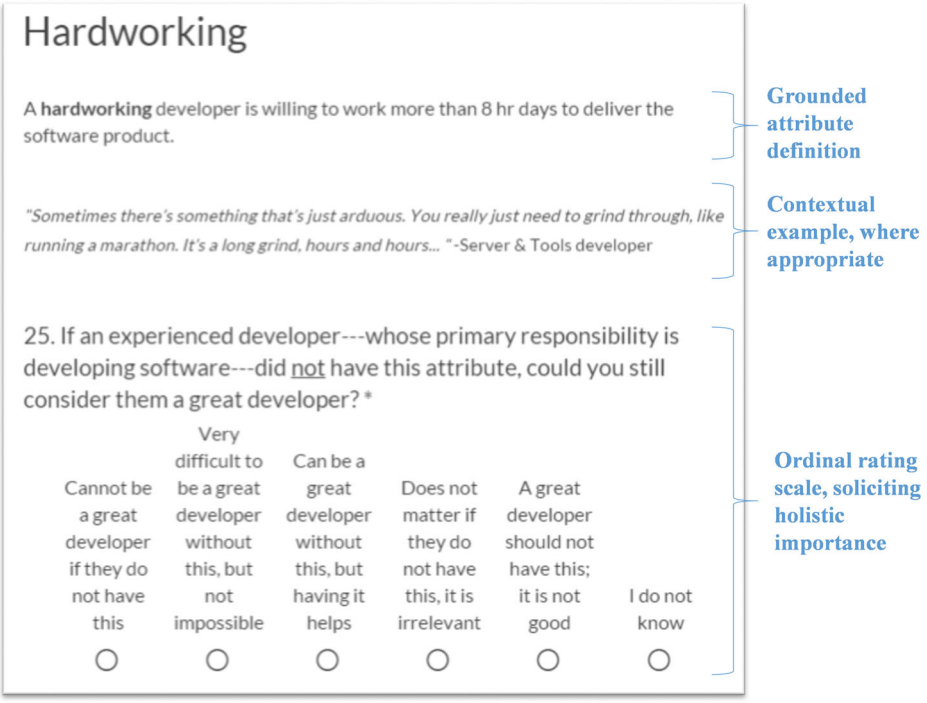

Fig. 2 Screenshot of the survey question for the attribute: "hardworking"

Piloting the survey indicated that this negative operationalization of "importance" was easier for respondents and was better at eliciting the attribute's importance. Positively phrased descriptions led to responses that lumped together because respondents could almost always imagine a situation in which an attribute can be important. The negative wording yielded more differentiation and better matched our conceptualization of importance: attributes that great engineers could not be *without*.

Once the survey construction was completed, we made an initial deployment to 200 developers (∼100 in each experience stratum) to look for additional problems, and to project response rates. After finding no issues, we deployed the survey to the full sample, aiming for 500 responses in each sampling stratum.

The survey took respondents a median of 28 minutes to complete. Overall, we obtained 1,926 survey responses, 825 responses from *experienced* engineers (∼7% of all *experienced* engineers at Microsoft, and a response rate of 46%) and 1,101 responses from *very experienced* engineers (∼35% of all *very experienced* engineers at Microsoft, and a response rate of 44%). Of the responses, 1,634 (84.3%) were complete, 292 had ratings for at least one attribute. We found no item-response bias—there was no relationship between attributes and having a response, at the $\alpha = .05$ level using Logistic regression. There were also no issues detected from the open-ended final question (e.g. no missing attributes). We used both complete and partial data in our analysis because of randomization and because ranking of each attribute was independent of other attributes (i.e. not relative ratings).

Our quantitative analysis assessed our notion of relative importance: the degree to which respondents agree that an engineer cannot be considered great without the attribute. The

two important aspects of the rankings are: the ratings and the agreement among respondents about the ratings; statistically, this means the *distribution* of ratings: both the central tendency (i.e. criticality) and the dispersion (i.e. agreement). The attributes that were more important have distributions that were more concentrated at the higher ratings. Consequently, we ranked the attributes by comparing the ratings distribution of each attribute to the ratings distribution of every other attribute, counting the number of other distributions for which an attribute's distribution was significantly higher (53 was the largest possible number). We did not use *average* ratings for three reasons: the data were ordinal (i.e. the distance between rating levels is not uniform), our response levels were not centered (four positive ratings and only one negative rating), and averages do not consider the dispersion of ratings. To rank the attributes (i.e. compare distributions), we used the Mann-Whitney rank-order test; the test can be used to compare ordinal data as well as distributions (both central tendency and dispersion), and can be used when the number of observations is not equal (Hollander et al. 2013). There were no abnormal bi-modal distributions. For each attribute, we performed 53 one-sided Mann-Whitney rank-tests, one test against every other attribute. We then calculated the number of statistically significant pairwise comparisons at level $\alpha = .05$. Finally, we ranked the attributes based on the number of statistically significant tests. For example, the ratings distribution of the most highly ranked attribute was statistically higher than every other attribute.

To analyze the relationship between contextual factors and ratings, we used Ordinal Logistic Regression. We assessed the first order relationships between the contextual factors and the ratings of each attribute. Due to being optional, only 1,512 respondents provided information on age. To maximize statistical power, we first fitted models with all factors to assess the effects of age and then fitted separate models without age, to assess the effects of other factors.

Since we tested for statistically significant relationships between each factor and each attribute (over one thousand tests in all), to account for performing multiple statistical tests, we used the Benjamini and Hochberg False Discovery Rate (FDR) adjustment at the standard FDR $q = 0.1$ level. We filtered to only the statistically significant relationships by removing the highest p-value relationships to achieve FDR $q = 0.1$ (all relationships had $p < .05$).

3.2 Follow-up Email Interview Method

The quantitative results tell us *what* (e.g. important attributes and statistically significant relationships), but they do not tell us *why*. To interpret the rankings and the relationships, we emailed respondents for insight into their survey responses, providing valid understandings of the quantitative results.

We performed follow-up interviews for the highest ranked attributes (the top five ranked attributes in Table 1), the potentially detrimental attributes (the two attributes with the highest percentage of *"A great developer should not have this; it is not good"* ratings, at the bottom of Table 1), as well as the attributes that were significantly affected by contextual factors (the relationships listed in Table 2). For the highest ranked attributes and positive relationships, we picked respondents with the largest positive difference between their rating of the attribute and their median ratings, aiming to avoid respondents that rated all attributes highly. For the detrimental attributes and negative relationships, we similarly picked respondents that had the largest negative rating differences.

In our survey, 771 respondents indicated that they were willing to answer follow-ups questions. We sent follow-up emails to 111, receiving replies from 77 (69.4% response rate).

Table 1 Attributes of great software engineers, along a scale of *must have*, *should have*, *nice to have*, *irrelevant*, and *should not have*

# Higher	Mode	Attribute and description
53	Must	Pays attention to coding details, such as error handling, memory, performance, style
52	Must	Mentally capable of handling complexity; can comprehend multiple interacting software components
49	Must	Continuously improving: improves themselves, their product, or their surroundings
49	Must	Honest: provide credible information and feedback that others can act on
49	Must	Open-minded: lets new information change their thinking
46	Must	Executes: knows when to stop thinking and to start doing
45	Must	Self-reliant: gets things done independently and does not get blocked easily
45	Must	Self-reflecting: recognizes when things are going wrong with a plan and pivots
43	Must	Persevering: not dissuaded by setbacks and failures
41	Must	Fits together with other pieces around it: code accounts for surrounding constraints and products.
41	Must	Knowledgeable about their technical domain, including product, platform, and competitors
39	Must	Makes informed trade-offs: code is responsive to time to market goals, critical needs of the business
39	Must	Updates their decision making knowledge: does not let their understanding stagnate
36	Must	Curious: desires to know why things happen and how things work
36	Must	Evolving: code is structured to be effectively built, delivered, and updated incrementally.
35	Should	Knowledgeable about tools and building materials: knows strengths and weaknesses of code
35	Should	Grows their ability to make good decisions: builds understanding of possible outcomes of decisions
34	Should	Sees the forest and the trees: reasons through situations at multiple levels of abstraction
31	Should	Craftsmanship: wants their output to be a reflection of their skills and abilities
30	Should	Does due diligence beforehand: examines available information before deciding
30	Should	Elegant: designs solutions that others can understand and appreciate
29	Should	Asks for help: knows the limits of their knowledge and supplements it with knowledge of others
28	Should	Desires to turn ideas into reality: takes pleasure in building software
28	Should	Long-termed: considers costs and benefits over time, not just short-term goals
25	Should	Willing to go into the unknown: can step outside of comfort zone to explore a new area
24	Should	Is a good listener: effectively obtains, comprehends, and understands others' knowledge
22	Should	Passionate: intrinsically interested in the area they are working in
22	Should	Manages expectations: clearly communicates what they are going to do and by when
22	Should	Focused: prioritizes time for the most impactful work
21	Should	Systematic: address problems in an organized, principled manner
21	Should	Adapts to new settings: continues to be valuable to the organization as environment changes
19	Should	Integrates understandings of others: can build a more complete understanding with others

Table 1 (continued)

# Higher	Mode	Attribute and description
19	Should	Does not make it personal: avoids deciding based on individual goals and feelings
19	Should	Creative: generates novel and innovative solutions based on the context and its limitations
18	Should	Walks-the-walk: acts as an exemplar for others to follow
18	Should	Knowledgeable about software engineering processes: knows the bests practices and techniques
13	Should	Anticipates needs: proactively determines potential problems and needs
13	Should	Uses the right processes during construction: uses best practices and techniques to construct software
13	Should	Resists external pressure for the good of the software product: stands firm against outside pressures
11	Should	Has a good reputation: has the belief, respect, trust, and confidence of others
11	Should	Productive: achieves the same results as others faster
10	Helps	Knowledgeable about customers and business: understands their product's value proposition
10	Helps	Creates shared understanding with others: shapes others' knowledge via effective communication
9	Helps	Creates shared success for everyone: establishes long-term goals that everyone can buy into
8	Helps	Aligned with organizational goals: takes actions for the good of the product and the organization
7	Helps	Well-mannered: treats others with respect
7	Helps	Data-driven: lets data drive actions, not solely intuition
6	Helps	Creates a safe haven for others: frees others to make decisions based on what is right, not fear
5	Helps	Mentoring: teaches, guides, and supports other developers
4	Helps	Knowledgeable about people and the organization: aware of others' responsibilities and knowledge
2	Helps	Challenges others to improve: encourages expanding capabilities and goals
2	Helps	Personable: establishes trusting, positive social relationships
1	Helps	Hardworking: is willing to work more than 8 hour days to deliver
0	Helps	Trades favors: builds personal equity with others allowing them to call upon others later

We tried to ask a single informant about multiple attributes (when the informant qualified to answer questions about multiple attributes, per the selection criteria above), in order to uncover insights that spanned multiple relationships. The email started with a salutation and a thank you for participation in the study. It followed with: "Something interesting came up when I analyzed the data, and I hope you can help me understand it better". The email then describe the attribute(s) or relationship(s), the reason for needing further understanding (e.g. is one of the most important attributes, may not be an important attribute, having X is associated with lower importance ratings for Y), the informant's rating, and then: "Why did you choose this answer? Can you help me understand your reasoning?".

We qualitatively analyzed the responses to gain understandings, selected representative quotations, and then asked the informants' permission to quote them anonymously.

Table 2 Contextual factors, distribution in survey study, and significant effects. Rows are not ordered. Some distributions listed as *[min, median, max]*

Row	Factor	Encoding	Distribution	Relationships (OLR, FDR, q=0.1)
1	Experience	Categorical by job title	Experienced (825, 43%) Very (1,101, 57%)	Executes (+) Knowledgeable about tools & materials (+)
				Makes informed tradeoffs (+)
2	Age	Numerical	[20, 39, 73]	Knowledgeable about business (+) Knowledgeable about processes (+) Aligned with organizational goals (+)
3	Years as developer	Numerical	[0, 15, 41]	Hardworking (+) Desire to turn ideas into reality (+)
4	Years at Microsoft	Numerical	[0, 10, 30]	Aligned with organizational goals (+)
5	# companies in career	Numerical	[0, 3, 10]	Continuously improving (+)
6	Years on current team	Numerical	[0, 3, 25]	—
7	Experienced as manager	Yes/No	Manager (1,028, 53%)	—
8	Is female-identified	Yes/No	Yes (149, 8%)	Uses the right processes (+)
9	Has CS degree	Yes/No	Yes (1,228, 64%)	—
10	Has non-MBA masters	Yes/No	Yes (762, 40%)	Asks for help (–)
11	Has MBA	Yes/No	Yes (49, 3%)	—
12	Has doctorate	Yes/No	Yes (49, 3%)	Walks-the-walk (+) Challenges others to improve (+)
13	Has other degree	Yes/No	Yes (103, 5%)	—
14	Has non-CS degree	Yes/No	Yes (537, 28%)	—
15	Isn't working in US	Yes/No	Yes (351, 18%)	Aligned with org. (+)
16	Work experience in non-US countries	Categorical	India (273, 14%) China (168, 9%) Canada (77, 4%) UK (63, 3%) Israel (51, 3%) Other (383, 20%) None (911, 47%)	31 attributes (+) 9 attributes (+) Hardworking (–) Hardworking (–), Aligned with org. (–)
17	Non-native English	Yes/No	Yes (926, 48%)	Passionate (+)
18	Type of customer	Categorical	Internal (791, 41%) External (270, 39% Both (865, 32%)	Persevering (+)
19	Server-side software	Categorical	Client-side (562, 29%) Server-side (754, 39%) Both (610, 32%)	—
20	# developers worked with in past year	Numerical	[0, 15, 1,000]	—

4 Results

In this section we focus on attributes and relationships that we asked about in follow-up interviews, as these were the most interesting attributes and attributes for which we have the most valid understanding. We discuss the top 5 (highest ranked) attributes, the bottom 2 (potentially detrimental) attributes, as well as overall rankings among attribute groups. For differences due to contextual factors, we discuss each statistically significant relationship.

4.1 Attribute Ranking

Table 1 shows the attributes, ranked according to level of agreement across our survey responses regarding their importance. The most important attributes are at the top and the least important attributes are at the bottom, based on their ratings distributions. We observed no abnormal bi-modal distributions and compared the distributions using the Mann-Whitney test. The number in the first column is the number of other distributions for which that distribution is higher comparatively. A subjective rating of criticality is in the second column. The third column lists and explains the attributes.

4.1.1 Highest Ranked Attributes

The most important attribute was pays attention to coding details (ranked 1, higher ratings distribution than 53 attributes; 63.1% essential, 28.8% important, 7.5% helpful, 0.3% doesn't matter, 0.1% detrimental). Respondents explained that first and foremost, engineers judged other engineers by their code. Therefore, engineers that could not get the basics correct were not respected:

> *Another strong driver is the respect of our peers, which you won't get by writing shoddy code [...] — Principal SDE*

Second, informants felt that software could be used in many ways, often unforeseen by the engineer; therefore, engineers needed to pay attention to the details to avoid costly problems:

> *This code is performance critical, compatibility sensitive, and is used in a huge variety of contexts. If a developer fails to handle an error, some customer will hit it, and we will likely need to issue a hotfix; if a developer implements an inefficient algorithm ($O(N^2)$ is not ok) [...] consumes memory excessively in some environment [...]etc. — Principal SDE*

This may have been especially important at Microsoft, where software products are often platforms, components, and/or used in contexts unforeseen by the engineer.

This understanding also underlies mentally capable of handling complexity (ranked 2, higher ratings distribution than 52 attributes; 54.2% of respondents gave it the highest rating, 36.2% important, 20.1% helpful, 1.6% doesn't matter, 0.2% detrimental) having a high ranking. Informants felt that great engineers need to be able to think through complex situations:

> *Most useful software has to be highly tolerant of incorrect usage by the user / caller above it, and interacting with the supporting code below it [...] Developers who cannot handle complexity tend to always be fixing bugs or having to do "another" release to take into account situations they had not thought of [...] — Principal SDE*

Informants felt that continuously improving (ranked 3, higher ratings distribution than 49 attributes; 51.0% of respondents gave it the highest rating, 34.8% important, 13.5% helpful, 0.7% doesn't matter, 0.1% detrimental) and open-minded (ranked 5, higher ratings distribution than 49 attributes; 49.4% of respondents gave it the highest rating, 36.5% important, 13.2% helpful, 0.7% doesn't matter, 0.1% detrimental) were important because the software industry moves quickly; therefore, great engineers needed to be open to new ideas and also to keep learning:

> *As the technology/technique evolves and better tools come along, the open-minded developer picks up on these and is willing to apply them to be more productive / effective [...] without an effort to continuously improve [...] developers will soon find themselves lagging behind the industry and/or state-of-the-art with technology and technique. — Principal SDE Lead*

This thinking also contributed to honest (ranked 4, higher distribution than 49 attributes, 50.8% of respondents gave it the highest rating, 32.1% important, 14.3% helpful, 2.2% doesn't matter, 0.1% detrimental) being important. Informants indicated that great engineers needed to acknowledge mistakes in order to make optimal future decisions for themselves and their teams:

> *Lying to yourself is much easier in my profession than in any other profession I know [...] It's so easy to think that you know the topic and miss (subconsciously ignore) evidence that contradicts your "knowledge." Great developer [...] simultaneously knows a lot and questions everything he knows. — Principal SDE*

Informants also discussed developers' *dis*honesty as potentially detrimental to others and felt strongly that such behaviors were deleterious:

> *This has happened to me any number of times [...] a team which had such a component would "lie" to me about its availability and maturity in order to get me to be a user and justify their own existence to management [...] — Principal*

4.1.2 Lowest Ranked Attributes

Two attributes received negative ratings — *"A great developer should not have this; it is not good"* — from more than 5% of the respondents: trades favors (ranked 54, higher distribution than 0 attributes; 4.0% essential, 15.1% important, 44.1% helpful, 29.1% doesn't matter, 6.0% detrimental) and hardworking (ranked 53, higher distribution than 1 attribute, 11.0% essential, 19.9% important, 36.0% helpful, 27.8% doesn't matter, 5.0% detrimental). We did not expect the high detrimental ratings because the attributes —derived from previous interviews — were all positive.

Follow-up interviews suggested that these attributes may not be inherently bad, but likely reflected bad situations. For the hardworking attribute, informants believe that needing to work more than an 8-hour day may be indicative of poor planning or unsustainable software engineering practices:

> *[...] workload for a developer is a function of management and planning happening above that developer. Usually long working hours are needed, because the planning was not good, the decisions made during the project lifecycle were bad, the change management wasn't "agile" enough. — SDE2*

For the trades favors attribute, informants believed that needing personal favors might reflect a biased decision-making culture, where decisions were not based on reason but rather on subjective opinions of individuals:

> *They should be totally separated, else what I have seen is we tend to make biased decisions and opinions about others. — SDE2*

Furthermore, needing undocumented processes to get things done might indicate poor organizational practices, making it harder for engineers to operate effectively:

> *Once you "trade favors" you are getting into personal give and take and builds institutional memory around a couple of nodes in a people graph and possibly not visible outside of that relationship [...] — Principal SDE*

4.1.3 Overall Ranking Among Attribute Groups

Overall, attributes associated with interacting with teammates were rated the lowest (high agreement on low importance), with a median ranking of 40 (lowest among the 4 groups) and 77.8% of attributes in the bottom half of rankings; they are followed by personality attributes (median ranking of 24 and 44.4% in the bottom half). In contrast, attributes associated with decision-making were rated the highest (median ranking of 17 and 33.3% in the bottom half), followed closely by attributes of the software product (median ranking of 17.5 and 33.3% in the bottom half). This can be seen visually in Fig. 3, which plots the attributes based on their ranking (x-axis) and the percent of ratings in the top two boxes (y-axis).

In follow-up interviews, many informants believed that a developer's idea should demonstrate value by its own merits, not via the persuasive powers of its presenter. For example, the following is a quotation regarding the creates shared context with others attribute, which

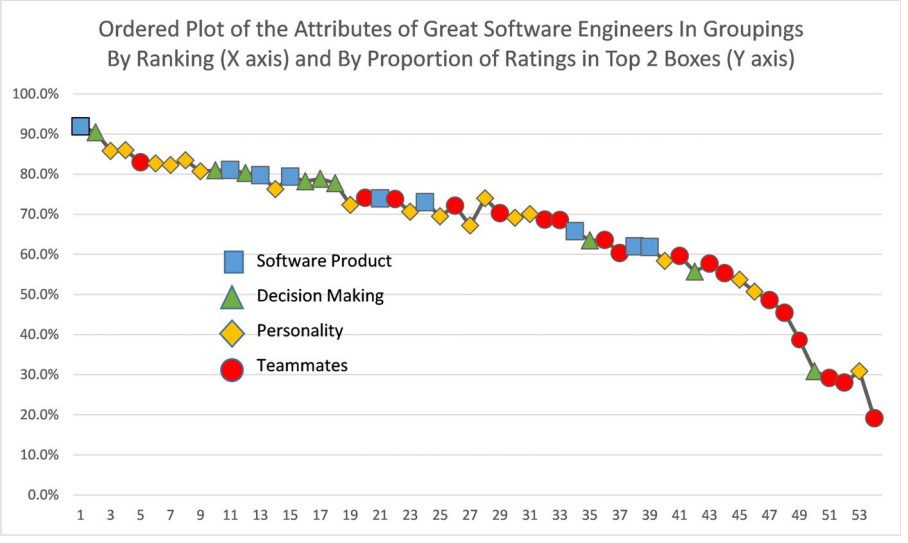

Fig. 3 Attribute rankings of the four types of attributes

was the most important component of "effective communications", but ranked 43 out of 54 attributes:

> *[...] that feels like imposing your will on someone else [...] other devs pushing their ideas through by controlling the conversation or talking over other people give me a negative gut reaction to that particular attribute. Ideas should stand on their own merits, not on how well / how strongly they're sold. — Senior SDE*

It is possible that lower relative rankings for these attributes reflect a culture of inclusiveness at Microsoft. With highest rankings for coding skills and mental capabilities (Section 4.1), the rankings may reflect beliefs that different kinds of people (personality) and different approaches (interactions with teammates) exist. As long as good software results, different ways that each engineer reaches those objectives can (and should) be tolerated. Informants felt that the primary job of the developer is to produce high-quality software, the rest is non-essential:

> *A great developer furthers the commercial interests of the company. He does this by producing software that is so bullet-proof and reliable [...] Outside of these considerations, I have no interest in that developer [...] — Principal SDE*

4.2 Influences of Context

Table 2 describes the 26 contextual factors in our survey: descriptions, descriptive statistics, and summaries of the statistically significant relationships between the factors and ratings. To facilitate statistical analysis, we split out only the top 5 countries (each with more than 51 respondents) for work experience in non-US countries; the other 61 non-US countries were combined into *Other* (see row 16 in Table 2). Statistically significant relationships, based on Ordered Logistic Regression (OLR) after FDR correction, are listed in column 5. We indicated positive relationships (the presence of the factor or higher values of the factor related to higher ratings) with (+), and negative relationship (presence of the factor or higher values of the factor related to lower ratings) with (−). Of the contextual factors, 10 did not have any statistically significant relationships (after the FDR correction), indicated by "–".

4.2.1 Level of Experience

We discuss the first five factors in Table 2 (*Is very experienced*, *Age*, *Years as a professional developer*, *Years at Microsoft*, and *Employment at software companies*) collectively as level of experience, because all the factors aim to measure the same underlying construct of "experience" and are highly correlated with each other. The relationships between *level of experience* and the eight attributes (first 4 rows in Table 2) were all *positive*.

Informants in our follow-up interviews — all of whom were *very experienced* — suggested four underlying reasons for the observed positive relationships. First (and most obviously), informants felt that developers with higher *level of experience* placed more importance on contributing to "business goals" because engineers at higher levels were evaluated based on their contributions toward meta-organizational goals. This likely underlies the relationships with aligned with organizational goals and knowledgeable about the customer and business:

> *Our evaluation system(s) have always emphasized developers that deliver on the organizational goals of the company [...] more experienced developers are likely*

to understand, that alignment with the company goals delivers greater rewards. — Principal SDE Manager

Second, informants felt that developers with higher level of experience valued delivering results, encompassing the relationships with hardworking, desires to turn ideas into reality, and executes. Informants felt that, with experience, engineers gained the understanding that to make meaningful contributions, engineers needed to deliver software:

20 years of experience managing engineers in startups and big companies alike [...] No matter how talented, sharp minded and skillful one is, if they are not hardworking (i.e. willing to work long hours to meet deadlines / deliverables) they will not succeed [...] — Partner SDE Lead

Third, informants felt that developers with higher level of experience placed more emphasis on gaining knowledge and making smarter decisions because they had gone through multiple releases and experienced the pain of mistakes. This encompassed the relationships with knowledgeable about tools and building materials, knowledgeable about software engineering processes, and makes informed trade-offs:

"Knowledgeable" and "Informed" only come from experience. This is all about breadth and exposure to lots of situations that let you generalize to new ones [...] you learn to be less confident that you immediately know the best answer to a problem. You actually become more flexible and are willing to trade off among goals you might not even have considered earlier in your career [...] It takes a while for most people to really appreciate the big picture and to be able to make decisions based on a broader context [...] — Architect

Many informants further felt that this knowledge and understanding could only be gained through actual first-hand experience:

Software engineering processes are there for a reason [...] The more experienced you are, the more you see the pros and cons of process firsthand. — Principal SDE Lead

Finally, a corollary to the previous finding, informants felt that engineers with higher levels of experience understood that they needed to be continuously improving to stay ahead. Experienced software engineers recognized that if they did not continue to learn, they might become obsolete:

Nobody can stay at the top without "improving" because the next wave of technology will soon obsolete [sic] whatever was at the top. — Partner SDE

4.2.2 Gender

We observed a statistically significant positive relationship between *gender* and uses the right process during construction. We asked female informants why they rated the attribute highly and then attempted to infer the commonality in reasoning behind the responses. It appears that female informants believed that processes existed for good reasons and that good software engineers should not be attempting to "reinvent the wheel":

You cannot be great if you are constantly re-inventing the wheel or using out of date tools/processes. — Senior SDE

Furthermore, it appears that female informants felt more strongly that the engineering of software should proceed in an orderly manner, not going-off on their own:

> *Good engineers MUST know the process of execution and follow it. Each project/product/team may have a different process, but a good engineer must be aware of it and follow it, or start a discussion if he/she thinks the process should be changed [...] — Senior SDE Lead*

4.2.3 Educational Background

We found that having a Master's and/or PhD degree had *negative* relationships with asks for help, challenging others to improve, and walks the walk (for Master's, see row 11 in Table 2; for Ph.D., see row 12 in Table 2). Informants provided two interesting hypotheses. First, informants felt that a graduate degree was largely optional for success in the software industry (likely less important than hands-on experience, per the previous discussion); therefore, engineers that get those degrees may be more intrinsically motivated than others, leading them to be less inclined to give or to receive help:

> *They weren't satisfied with the bare minimum of a bachelor's degree [...] getting a master's degree doesn't really impact your paycheck very much in this industry [...] I think these people who seek knowledge [...] they want to find things out for themselves. — Principal SDE*

Second, informants also suggested that engineers with graduate degrees were often hired as *technical experts* such that they were often given novel problems to solve, rarely having the opportunity to give or to receive help:

> *[...] problems which either nobody has tried to solve before, everyone else has failed solving before, or handling some major sort of crisis [...] they operate under the assumption that there's nobody to ask help from when there's a crisis and they will need to be able to figure out the solutions themselves. — Principal SDE*

We further examined this second explanation by comparing the *number of developers worked with in the past year* (row 20 in Table 2) between engineers with and without advanced degrees. We found that the *number of engineers worked with in the past year* was statistically significantly less ($\alpha = .05$) for both engineers with a Master's degree ($p = 0.004$, with medians of 12 for those with and of 15 for those without a Master's degree) and with a Ph.D. degree ($p = 0.037$, with medians of 11 for those with and of 15 for those without a Ph.D. degree) using the Mann-Whitney rank test. These results show that engineers with advanced degrees worked with fewer engineers.

Overall, we conclude that the relationships were unlikely due to graduate schools *teaching* engineers to devalue giving and getting help. Rather, as discussed, findings were likely due to selection bias (conditions that lead an engineer to pursue a graduate degree) and survivor bias (job assignments of Masters / Ph.D. graduates).

4.2.4 Work Experience in Another Country

We found many positive relationships between attributes and work experiences in another country; qualitative follow-ups suggest four underlying themes. We asked informants about their ratings and then inferred the underlying themes. First, informants suggested that there was intense competition for well-paying software engineering jobs in some countries. This

may be the underlying reason for the 31 *positive* relationships between attributes and *having work experience in* ***India***, as well as the 9 positive relationships between attributes and *having work experience in* ***China***. Competition necessitated engineers in those countries to excel in many areas to compete:

> *I think from the culture[...] If you're not the top of your class, you're not getting in. On to your next thing, whatever. If you're not rank number one, you're not getting into the IITs. You're not ranked number whatever, you're not getting that job [...] Doesn't work that way in the Western world because [...] population. There's a lot of opportunity [...] not only one person wins, ten people can win. In the Eastern side of the world maybe not. — Senior Dev Manager*

Second, informants felt that cultural norms influenced the practice of software engineering in some countries. The most salient example is the relationship between *having work experience in China* (summarized in row 16 of Table 2) and the trades favors attribute. While the trades favors attribute was the lowest ranked attribute overall, its ratings were significantly higher for respondents in China. Follow-up interviews indicated that the higher ratings were influenced by broader cultural norms in China:

> *Culturally there is a different perception [...] ("guanxi") it's just a part of how business is done. Well of course, the best, the most successful are the ones that have those relationships. That would be a positive thing [...] any career or profession [...] even in an engineering context. — Principal SDE Manager*

Beyond trading favors, informants implied that many other attributes (e.g. hardworking and systematic) were similarly influenced by local practices and expectations:

> *Systematic, I wouldn't be surprise if that's skewed [...] Part of it is culture. There's just a daily grind of getting things done. People there would acknowledge that it doesn't make sense; it's just the way it works, why would you change it. — Principal SDE Manager*

The third theme was distance. Informants felt that some software engineers lacked visibility into company direction due to being far away from Microsoft headquarters—based in Redmond, WA, USA. This might have impacted engineers' perceptions on being aligned with organizational goals. Some informants suggested that the numerous shifts in company focus in recent years led engineers to focus on their immediate customers rather than the overall company strategy:

> *[...] organizational goals are usually generic and change quite often [...] a developer is great regardless of external happenings, conditions or events [...] a great developer should take actions for the good of the product and customer. In good companies, such actions will pay off and benefit the individual and the organization as well. — SDE2*

The fourth theme was that some attributes were likely tied to the kind of software products being engineered in those countries, as well as the state of software engineering practices in those countries. For the negative relationship with hardworking, several developers, all with non-US work experience, reported having worked in the games industry where they had to do "death marches", needing to work excessive hours to ship the software product. This may have been especially salient for respondents outside of the US,

accounting for the negative relationships between hardworking and having work experiences in the UK and Other countries (row 16 in Table 2):

> *I've definitely seen this firsthand, as people steadily become less productive over time and tend to make more short-term decisions [...] Having previously worked in both games and visual effects, where the "death march" is not uncommon — Senior SDE*

4.2.5 Type of Customer

There was a statistically significant positive relationship between having both internal and external customers and persevering (row 18 in Table 2). However, based on follow-up interviews, we believe this relationship was likely spurious (not surprising since FDR adjustment reduces, but does not eliminate, statistically significant relationships occurring by chance).

One informant tentatively offered the possible explanation that having customers with differing needs leading to conflicting requirements necessitating perseverance; though, even he felt that the relationship could be coincidence:

> *It is also frustrating to deal with two sets of customers at once, as they often have conflicting reqs. It requires persevering to being able to battle out which ones to implement and to persist in the face of conflict. I have no more thoughts vs. what I've mentioned already, so it could be coincidental. —Principal SDE Manager*

5 Discussion

In this section we synthesize and discuss learnings about what distinguishes great engineers. First, we set the context by discussing prior work, then we discuss what our learnings tell us, both about what distinguishes great engineers and about prior work. We discuss implications for researchers, educators, and practitioners. Finally, we conclude with a discussion of threats to validity and future work.

5.1 Related Work

Prior work provides extensive and valuable insights into many aspects of being a software engineer, but the literature fails to provide a holistic, developer-centric, ecologically-valid perspective on developer skills. For instance, many studies focus only on technical skills: comparing novice and expert programming skills (e.g., Sackman et al. 1968; Gugerty and Olson 1986; Robillard et al. 2004) or propose skills software engineering graduates should possess (e.g., Joint Task Force on Computing Curricula 2014; Bourque et al. 2014; Begel 2008; Hewner and Guzdial 2010). Some studies focus on human and social factors, but do not relate these factors to technical skills (e.g., Kelley 1999; Rozovsky 2015; Ahmed et al. 2012). Other prior work provides team and organizational perspectives on best practices and information needs that imply necessary skills, (e.g., Herbsleb et al. 1997; Boehm 1988; Beck et al. 2001; Ko et al. 2007; Singer et al. 1997; Perry et al. 1994), but often do not explicitly address *individual* skills. Some prior work by industry practitioners *do* provide a more holistic and experiential view of developer skills, incorporating technical, interpersonal, and mental attributes; their insights are also supported by real-world examples and explanations (Brooks 1995; Brechner 2003; Fitzpatrick and Collins-Sussman 2009). However, these works are neither rigorous nor complete in their investigations (nor were they aiming to be).

The most relevant prior work, Baltes and Diehl's comprehensive theory of software engineering expertise (Baltes and Diehl 2018), is an interesting *complement* to our study. While being "great" and being an "expert" are closely related, they are not exactly the same. Being an *expert* software engineer implies being successful at producing software; however, being a *great* software engineer may extend beyond engineering outcomes. For example, the "mentoring" attribute (see Table 1): while being a good mentor is a desirable attribute of engineers (and many of our informants discussed benefiting from having mentors) it is not clear how the attribute would make the mentor more of an "expert". Consequently, insights that our rankings provide are valuable, enabling understanding of how engineering outcomes matter.

Therefore, relative to prior work, our study attempts to be holistic, developer-centric, ecologically valid, and more scientifically rigorous than existing literature.

5.2 What Distinguishes Great Software Engineers

Relative to prior work, our research suggests that attributes that distinguish great engineers *comprehensively* encompass internal personality traits, ability to engage with others, technical capabilities, and decision-making skills. Even though opinions of experts have pointed to the "combined arms" nature of the distinguishing attributes of great engineers, few prior research efforts have examined more than one kind of attributes. The need to examine software engineers comprehensively and deeply is the key insight from our study, and one of the key contributions of our work.

5.2.1 Being a Competent Coder

Our results suggest that, at least at Microsoft, the most important aspect of being a great engineer is being a competent coder. While numerous studies about great engineers tout various "soft skills" (Kelley 1999), the experienced engineers in our study rate the technical ability to write good code as the most essential. The understanding of how this aspect distinguishes great engineers is straightforward: without code, there is no software; therefore, great *software* engineers need to be able to write good code. As ACM's definition of a software engineer (Shackelford et al. 2006) suggests, writing code is at the core of being a software engineer.

Our survey results indicate that engineering of production software at its highest levels (at Microsoft) can be a complicated and complex technical undertaking. Informants indicated that uncertainty and complexity afflict their software, including underlying dependencies, system states, external callers, and / or partner components. Pays attention to coding details and mentally capable of handling complexity being atop the rankings likely reflects this sentiment (see Section 4.1.1), as well as overall high rankings for other product and decision-making attributes (see Section 4.1.3). This reinforces sentiments in Brooks' *The Mythical Man-Month* (Brooks 1995); writing "production" code is hard.

Our findings align with the ACM's Computing Curriculum for Software Engineering (Shackelford et al. 2006) and the SWEBOK (Bourque et al. 2014), which focused largely on technical coding skills. These findings also align with observations of everyday activities for engineers; though engineers spend time doing other activities, much of their time is still spend coding (Ko et al. 2007; Latoza et al. 2006; Singer et al. 1997; Perry et al. 1994). In addition, our findings largely justify research efforts aimed at understanding and closing the gap between novice and expert coders (Sackman et al. 1968; Valett and McGarry 1988; Gugerty and Olson 1986; Robillard et al. 2004; Baltes and Diehl 2018). Even though

focusing on coding may be myopic, since it is the essential skill of software engineers, ensuring that novices are competent coders first is likely a good starting point:

> *But when we talk about the quality of the code, performance, space, and how many bugs it has — how robust it is — and how it handles exceptions [code of great software engineers] will have great differences [...] For example, when I used to make games back in China, I worked on a board partitioning program that [...] took about 3 hours. Then my CTO took the program to optimize. When he was finished with it, the program took 10 minutes to run. That's the amount of difference it can be between people [...] — SDE2*

Conversely, our findings suggest that not considering technical skills is a major limitation of research that focus solely on soft skills of engineers (e.g. Kelley 1999; Rozovsky 2015; Ahmed et al. 2012). The lack of consistent relationships between various human factors and engineering outcomes, as discussed by Cruz et al. (2015), may be due to omission of technical skills. After all, if a software engineer cannot develop software, then all other attributes are probably moot.

5.2.2 Maximizing Current Value of your Work

The economic concept of "risk and expected returns" may explain numerous seemingly contradictory attributes and sentiments in our study. When applying an economic lens (Ventures and Knowledge 2000), considering probabilistic future value (possibly negative) and the time available for actions, a coherent theme emerges. Great engineers distinguish themselves from others by considering the context of their software product, maximizing the value of their current actions, adjusted for probable future value and costs.

The first area of (apparent) contradiction was that many informants discussed great engineers designing their software with the future in mind, e.g. long-termed and anticipates needs, taking time and effort to ensure that their software was resilient to future changes (see Table 1). However, other informants disagreed, believing that predicting the future was futile. Experimentation, faster iterations, and a willingness to make changes were better. In their view, long-termed and anticipates needs were detrimental attributes that wasted effort on futures that may not occur. These opinions can be reconciled when viewed through an economic lens. Engineering decisions may incur "technical debt" (Kruchten et al. 2012); therefore, the current value of efforts needs to take into consideration probable future costs to service and repair. For software with long lifespans and high repair costs, engineers need to think ahead. However, in other situations the software may have a short lifespan or have low repair / update costs (e.g. online services compared to boxed software); in those situations, great engineers may rightly defer future costs:

> *[...] you are writing software for the client? [...] for the cloud? It's a different ballgame. If you are writing software for the cloud [...] the cost for the bug is not that high. I'll fix it. I don't have to ship the fix to you; I'll fix it on my server [...] the point is that I will take the risk. — Senior Dev Lead*

The second area of (apparent) contradiction was expecting great engineers to take the time to thoroughly think through the problem. The systematic attribute entailed not jumping to conclusions and not acting too quickly; the elegant attribute involved thinking deeply to coming up with simple solutions; the fits together with other pieces around it attribute entailed accounting for the relationships with surrounding components (see Table 1). However, many informants also expected great engineers to just go ahead and "do it". The

willingness to go into the unknown attribute was about the willingness to take action with incomplete information, and the executes with no analysis paralysis attribute was explicitly about the need to stop thinking (see Table 1). From an economic perspective, software has value (e.g. makes money) only after deployment, and for some products, timing greatly affects these future benefits. Products like games and consumer electronics have market conditions that can incur significant revenue penalties for missing deadlines (e.g. the holiday season); defects in deployed software may be actively harming customers. Through this lens, contradicting opinions about speed of actions makes economic sense. High-quality code saves on future repair and maintenance costs; however, those savings must be weighed against possible forfeiting of revenue due to inaction. Therefore, while high-quality code is generally good, there may be situations, especially close to "shipping deadlines" or facing high priority bugs, where producing a "hack" makes more economic sense than having a complete solution that takes more time:

> *[...] there's not even a non-magic bullet. People sometimes, even I do this under, say, shipping deadline, you will do something quick and dirty, and unfortunately it always happens [...] Often, when you do this, you are aware of this [...] — Principal SDE*

The importance of risks and expected economic returns may be context specific, as Microsoft is a for-profit organization. Nonetheless, related concepts — "regression" or "reopen" — are often discussed in research literature on bug triaging for open source software projects (Anvik et al. 2006; Ko and Chilana 2011).

Value maximization, as an economic concept, is broadly relevant to many aspects of human behavior (Ventures and Knowledge 2000). However, interestingly, the software engineering education literature (even those focusing on human factors in software engineering Cruz et al. 2015) is largely silent on the application of economic thinking in the engineering of software. In our survey, influences of level of experience suggest that engineers need real-world experience to understand when and how to do certain tasks. Informants felt that many things sound good in theory or in isolation, but become unimportant when put into real-world contexts, amid competing concerns and hard deadlines (see Section 4.2.1). In contrast, ACM's curriculum (Shackelford et al. 2006) and the SWEBOK (IEEE Computer Society et al. 2014) prescribes a set of skills with little information about when or whether to use those skills. Topics like software architecture and software verification are great in theory; however, our findings suggest that, given the economics of real-world software engineering, the optimal solution may sometimes be "quick and dirty."

5.2.3 Practicing Informed Decision-Making

Engineers face myriad decisions about what software to build and how to build it; consequently, effective decision-making is a critical attribute of great engineers. As engineers grow in their careers, they are tasked with increasingly complex and ambiguous situations, often with significant ramifications for themselves and their organizations. Rather than outcomes (which were often confounded by future uncertainties and outside factors), we believe the process of acquiring needed information to be the most important aspect of effective decision-making. Great engineers differentiated themselves from others by going through the right processes for making informed decisions.

Collectively, decision-making attributes (Section 4.1.3) ranked the highest among the four groups of attributes; we found attributes associated with "information gathering" to be the most important. Engineers often did not have the information they needed to make decisions; great engineers distinguished themselves by effectively acquiring the

necessary information and then making an informed decision. Viewed within the rational decision-making framework, the systematic attribute, described *actually undertaking* the "information gathering" activity, the asks for help attribute concerned seeking out those with the best information, and the open-minded and data-driven attributes both describe great engineers' willingness to let new information influence their decisions:

> *Unlearning. That's like, the things that I used to do five years ago that make me successful don't matter anymore; in fact, they can get me into trouble right now [...] I start to get to a point where I would assess [an engineer's] ability to unlearn. After a while, like two thirds or three quarters of what you know is still valuable, quarter to a third is the wrong thing [...] — Technical Fellow*

Conversely, many negative attributes of bad engineers discussed by our experts were symptoms of not gathering or not using the right information to make decisions. For example, in discussing the data-driven attribute, informants lamented that some engineers had confirmation bias, selecting only the information that confirmed their initial understanding:

> *One thing that surprises me [...] even though we are driven by data, at least we try to believe we are [...] Some data gets shown to us, we figure out some ways to ignore it. So, maybe everybody thinks that they're data driven, but I've seen people come up with excuses for why the data doesn't apply to them. I've seen that a million times. — Senior SDE*

Decision-making is an everyday activity for everyone; however, the process of making good decision has only recently received attention in Baltes and Diehl's theory of software development expertise (Baltes and Diehl 2018). It is not mentioned in the ACM curricula. Nonetheless, aspects of decision-making — good and bad — are sprinkled throughout the software engineering literature. Bug triaging, examined by many researchers (Anvik and Murphy 2007; Jeong et al. 2009; Podgurski et al. 2003; Runeson et al. 2007; Bertram et al. 2010) is effectively a decision-making process. Gobeli et al., who examined effective (and ineffective) conflict resolution approaches within software engineering teams touched on making decisions (Gobeli et al. 1998). Consulting with team members to decide how best to implement a feature or to fix a bug is mentioned in nearly all studies that examine everyday activities of engineers (Ko et al. 2007; Latoza et al. 2006). It is time for software engineering educators and researchers to pay more attention to decision-making within their education and research efforts.

5.2.4 Enabling Others to make Decisions Efficiently

Shrouded in polite descriptions like creates shared understanding with others and creates shared success for everyone, we saw a theme in our findings as: *please don't make my job any harder*. Great engineers distinguished themselves by making others' jobs easier, helping them to make their decisions more efficiently (or, at minimum, they did not make them worse). This aspect is an organizational issue generally applicable to most teams (Simon 1973), and may be especially important to information-driven and coordination-intensive professions like software engineering. It also corresponds with Baltes and Diehl's inclusion of the personality traits agreeableness and conscientiousness (Baltes and Diehl 2018).

This theme surfaced during explanations of the benefits of engineers having — though more commonly, downsides of not having — various attributes. This sentiment was most apparent for the honest attribute (see Section 4.1.3); in almost every instance, informants

described negative situations where engineers lacking honesty caused problems for other team members:

> *Influence comes to someone else trusting you, part of that trust is that they go, "You know what? I know that this person always speaks the truth." As a result of that, when they say something is good, I will totally believe them because they are not trying to kind of misrepresent something or make them look better or whatever. — Principal Dev Manager*

The manages expectations attribute contained discussions about engineers derailing a project by not speaking up about potential delays. The self-reflecting attribute entailed engineers proactively changing plans when they realized current plans were untenable, and the same sentiment underlies the asks for help attribute (see Table 1). In addition, for many informants, the creates shared understanding attribute was about great engineers helping them understand the reasoning — commonly, pitfalls and potential problems — behind various options so they can make appropriate decisions. For these attributes, informants discussed engineers without the attributes preventing others from taking corrective actions to avoid bad outcomes for the team and ultimately making everyone's jobs harder:

> *[...] you really want to have [great software engineers] have a lot more input. If someone disagrees with the tradeoffs that we're making, have a voice [...] They really do participate and give their opinion. — Principal Dev Manager*

There is little direct mention of *don't make my job harder* in the research literature, even though there are hints in various studies of software engineering efforts. For example, Ko et al. found *maintaining awareness* to be an important concern for software engineers (Ko et al. 2007), Latoza et al. found *team code ownership* and *the moat* (which facilitated understanding within the team and limited outside perturbations) to be a common theme (Latoza et al. 2006), and stand-ups mandated in Scrum development processes (Rising and Janoff 2000) are likely meant to enforce information sharing. Latent sentiments like this may be especially difficult to detect using research methods that do not dig deeper into the reasoning behind answers. Consequently, various research methods like meta-analysis (Radermacher and Walia 2013) and secondary analysis (Ahmed et al. 2012) may leave gaps in understanding when used to study engineers.

5.2.5 Continuously Learning

Our findings suggest that because the field of software engineering is changing constantly, those who do not grow and evolve risk becoming obsolete (see Section 4.1.3). Consequently, we believe it is not a specific set of knowledge but rather the desire, ability, and capacity to learn that distinguishes great engineers.

The theme of constant learning was prevalent throughout our study; informants frequently indicated that greatness was attained and maintained over time. This contributed to multiple related attributes — honesty, open-minded, and continuously improving — to being atop the rankings (see Section 4.1.3), as well as high rankings for numerous personality attributes related to learning and improving. Informants also explained how numerous attributes contributing to learning. The curious attribute — wanting to know how things work — was a motivating factor behind learning:

> *A curiosity [...] how things work, why things work, the way they work, having that curiosity is probably a good trait that a good engineer would have. Wanting to tear*

something apart, figure out how it works, and understand the why's. — Principal Dev Lead

Both grows their ability to make good decisions and updates their decision-making knowledge involved learning and continuously re-learning how to make the best decisions. Asks for help and integrates understanding of others both involved effectively learning from others (see Table 1).

We found that the ability to learn new technical skills may be as important (if not more so) than any individual technical skill. Informants, even those in the same division, used diverse technologies. There was no consensus on which specific technical topic (e.g. architecture) was essential. Rather, most informants stressed the importance of *learning* new skills and technologies as requisite.

Greatness is not a one-time designation; it is an ongoing progress. This aligns with sentiments in related work. The need to continue learning is closely related to "continuous learning" in Baltes and Diehl's software expertise theory (Baltes and Diehl 2018), and to "continuing professional development" discussed in the ACM Curricula (Joint Task Force on Computing Curricula 2014). This edict is also in the code of ethics for many professions like medicine (AMA 2001) and "traditional" engineering (NSPE 2007). Though continuously learning appears to be a fundamental aspect of all learned professions, this may be the most important and the most difficult aspect for software engineers. As a relatively new field, the pace of innovation and change is rapid:

Computer technology, compared to other sciences or technology, it's pretty young. Every year there's some new technology, new ideas. If you are only satisfied with things you already learned, then you probably find out in a few years, you're out of date [...] good software engineer [sic], he keeps investigate, investment. [sic] — SDE2

Unlike most professions (e.g. medicine), fundamental underpinnings of computing can change; requiring software engineers to vigilantly keep pace to avoid obsolescence:

So, way back in the day, if you wanted to performance optimize something you counted instructions. Processors got faster and faster, but memory references didn't. There became a day when it made more sense to count memory references than it did to count instructions. Unless you're conscious of when those things will intersect, you'll be on the wrong side of history and be frustrated. — Technical Fellow

5.3 Implications for Research and Practice

Our learnings about the distinguishing attributes of great software engineers can have wide-ranging implications for software engineering research, practice, and training. We note considerations that may be unique to software engineering; however, software engineers are engineers, are employees, are people. Many of our insights are likely broadly applicable / beneficial to many people.

5.3.1 Researchers

Our findings may have several implications for researchers. Foremost, to better understand and leverage attributes discussed in this paper, we would benefit from metrics that operationalize the attributes. These are essential for enabling rigorous science to better understand how the attributes vary and their effects on teams and outcomes. Such metrics may also

form a foundation for managers to identify and cultivate talent, for novices to improve, and for educators to assess learning outcomes.

Second, our findings identify several pain-points that software engineering methodology researchers may want to address. For example, we discussed problems with engineers missing needed information (see Section 5.2.3). Better processes that address these issues (e.g. by enforcing information sharing) may help software engineering teams.

Third, researchers may also want to look deeper into cultural variations and the impact on effective software engineering. Since many software development organizations are multinational, researchers may want to examine the conditions in which organizations should (or should not) adapt to local cultural norms (versus instituting organizational standards).

Finally, our results suggest several new directions for tools research. For example, we are not aware of any tools that help engineers be more well-mannered in emails or evaluate trade-offs when making decisions. Tools research may also explore training engineers, especially novices, in desirable attributes.

5.3.2 Aspiring Software Engineers

Our findings have several possible implications for new software engineers. Obviously, our findings enumerate a prioritized set of attributes that new engineers may consider improving through training, practice, mentoring, or self-reflection. Furthermore, this information may also help engineers better present themselves to employers. Whether or not the attributes we identified actually lead to greatness, our findings indicate that experienced engineers (including managers) value these attributes; therefore, aspiring engineers may consider demonstrating them to employers, whether in resumes or during interviews.

5.3.3 Software Engineering Leaders

Our findings also have possible implications for leaders of engineers. Our findings enumerate multiple attributes that are important for engineers in senior and leadership positions, such as mentoring, raising challenges, and walking the walk. Therefore, engineers in leadership positions (or seeking to become leaders), may want to improve those areas.

Beyond improving themselves, our findings may also help managers make more effective hiring decisions. They may better identify candidates — with desired attributes — that fit the team. Furthermore, our findings also suggest that current hiring practices — typically, one-day interviews — can be improved. Some important distinguishing attributes of great engineers, such as the "desire, ability, and capacity to learn," require more time to assess. Approaches like internship programs — over several months and based on real-world projects — may allow managers to better assess applicants' abilities and growth potential.

Finally, our findings suggest that managers of software engineers may want to cultivate desirable attributes within their teams, building a culture that is conducive to attracting, producing, and retaining great engineers.

5.3.4 Educators

Lastly, our findings may have various implications for educators. Foremost, educators may consider adding courses on topics not found in their current curricula. While decision-making is not a part of the ACM's Curricula (Shackelford et al. 2006), we found attributes related to effective decision-making to be key distinguishing attributes of great engineers.

A course specifically about decision-making (e.g. Simon's model of rational choice (Simon 1955) or case studies of software engineering decisions) may be valuable.

Educators may also want to reexamine their teaching methods. Most distinguishing attributes of great engineers involve *how* rather than *what*, whereas most instructions in software engineering focus on knowledge (the *what*), such as techniques for testing and analysis. Educators may consider improving *how* software engineering goals are attained. For example, existing project-based courses can consider evaluating behaviors and non-functional attributes of the code, such as elegance, anticipates needs, and creative. Educators may also consider teaching *when* various skills should be used, since our results indicate that real-world conditions exist when eschewing best-practices may makes the most economic sense.

Finally, educators may consider explicitly discussing what students will not learn in school, allowing them to be aware of potential knowledge gaps and empowering them to seek out opportunities outside of the academic setting. For example, attributes like self-reliant may not be reasonable to teach in an academic setting and might be better learned through mentorships/internships.

5.4 Threats to Validity

As with any empirical study, our results are subject to various threats to validity. First, there are some threats to *construct* validity. The obvious issue is differing interpretations of terms by participant. We reduced this risk at the outset, by choosing to conduct our research at a single organization, where employees are more likely to have common understandings. We further mitigated risks by conducting pre-tests, adjusting and clarifying survey question as needed (e.g. adding supporting quotes and switching to using the term "developer"), as well as having an open-ended conclusion question to catch problems. Non-existence of bi-modal distributions indicate no obvious issues, and our large sample sizes and statistical tests reduce the impact of noise due to occasional misinterpretations (unavoidable given the diverse contexts of respondents, including many non-native English-speakers). Another issue is whether the attributes actually distinguish great engineers or are simply attributes that makes engineers more attractive to other experienced engineers: while advantageous for career advancement, the attributes may not yield good software.

Second, there are several threats to *internal* validity. The use of the FDR adjustment and the numerous significant relationships with *having work experience in India and China* may have hidden other interesting relationships. Also, our analysis examined only the first-order relationships between ratings and contextual variables. Though, while second-order relationships may exist, we feel that our choice was appropriate given little prior research to support investigating second-order relationships. Participants in our study are a self-selected group of engineers (particularly for the follow-up interviews); they may be biased in their opinions and other interpretations of the rankings and relationships may exist. Finally, even though the authors are experienced researchers with real-world software engineering experience, other valid interpretations of the data may exist.

Third, there are various interesting *external* validity questions that future work may consider investigating. Other than experience, we did not sample for other characteristics (e.g. gender and non-US software engineers). Even though we received many responses from both female (149 responses, 7.7%) and non-US respondents (351 responses, 18.2%), which should have allowed us to detect large systematic differences, more in-depth studies of interesting sub-populations are worthwhile. Comprehensiveness of attributes is another

external validity concern. We used a set of attributes derived from the interview study of Microsoft engineers. Furthermore, even though we had an open-ended question at the end of the survey asking about anything we may have missed (finding no missing attributes), it is questionable how well respondents can identify missing after seeing 54 attributes in rapid succession. Therefore, prior to repeating our survey at other organizations, repeating the interview study to identify possible missing attributes is advisable. Microsoft itself presents some issues; the obvious one being that it is a single organization. We also explicitly over-sampled very experienced engineers who may exhibit thinking and perspectives that were particularly well-suited to the Microsoft environment. Therefore, even though our study is a solid starting point, in the future, researchers may want to examine other contexts. For example, informants discussed unfavorable conditions in non-software-centric industries such as finance and retail, which many impact perspectives; replicating this study at other successful software-centric organizations (e.g. Google) may also yield interesting findings.

5.5 Future Work

All the issues discussed in this section are interesting topics and justifications for future studies. Investigations are unlikely to be experimental (i.e. we cannot assign attributes to engineers and withhold them from others), and no single observational study can establish causal relationships. The path forward is to have many studies, conducted by many researchers in many contexts, triangulating the way forward towards us deepening our understanding of what makes great software engineers.

6 Conclusion

In this paper, we have contributed holistic, developer-centric, ecologically valid, and scientifically rigorous insights into what distinguishes great software engineers. As our society grows increasingly software dependent, studies like ours and others that our work may inspire will be critical. After all, great software cannot exist without great software engineers — a butt in a seat somewhere — to type "Commit."

Acknowledgements The authors wish to thank the Microsoft software engineers who participated in our research. This work was supported in part by Microsoft, Google, and National Science Foundation (NSF) Grants CCF-0952733, CNS-1240786, and IIS-1314399.

References

Ahmed F, Capretz LF, Campbell P (2012) Evaluating the demand for soft skills in software development. IT Prof 14(1):44–49

AMA (2001) American Medial Association Principles of Medical Ethics. http://www.ama-assn.org/ama/pub/physician-resources/medical-ethics/code-medical-ethics/principles-medical-ethics.page?

Anvik J, Murphy GC (2007) Determining Implementation Expertise from Bug Reports. In: Proceedings of the Fourth International Workshop on Mining Software Repositories, Minneapolis, pp 298–308, https://doi.org/10.1109/MSR.2007.7. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4228639

Anvik J, Hiew L, Murphy GC (2006) Who Should Fix This Bug? In: Proceedings of the 28th International Conference on Software Engineering, pp 361–370

Aranda J, Venolia G (2009) The secret life of bugs: going past the errors and omissions in software repositories. In: Proceedings of the IEEE 31st International Conference on Software Engineering, pp 298–308

Baltes S, Diehl S (2018) Towards a theory of software development expertise. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018. ACM, New York, pp 187–200. https://doi.org/10.1145/3236024.3236061

Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin RC, Mellor S, Schwaber K, Sutherland J, Thomas D (2001) Manifesto for Agile Software Development. http://www.agilemanifesto.org/

Begel A (2008) Pair Programming: What's in it for me? In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp 120–128

Bertram D, Voida A, Greenberg S, Walker R (2010) Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In: Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, pp 291–300

Boehm BW (1988) A spiral model of software development and enhancement. IEEE Comput 21(5):61–72

Bourque P, Fairley RE et al (2014) Guide to the software engineering body of knowledge: Version 3.0. IEEE Computer Society Press

Brechner E (2003) Things they would not teach me of in college: what Microsoft developers learn later. In: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, pp 134–136

Brooks FP (1995) The Mythical Man-Month: Essays on Software Engineering, 2nd edn. Addison-Wesley Professional, Reading

Carver JC, Nagappan N, Page A (2008) The impact of educational background on the effectiveness of requirements inspections: an empirical study. IEEE Trans Softw Eng 34(6):800–812

Cruz S, da Silva FQ, Capretz LF (2015) Forty years of research on personality in software engineering: a mapping study. Comput Hum Behav 46:94–113

Ericsson KA, Krampe RT, Tesch-romer C (1993) The role of deliberate practice in the acquisition of expert performance. Psychol Rev 100(3):363–406

Fisher A, Margolis J (2002) Unlocking the clubhouse: the carnegie mellon experience. ACM SIGCSE Bullet 34(2):79–83

Fitzpatrick B, Collins-Sussman B (2009) The Myth of the Genius Programmer

Gobeli DH, Koenig HF, Bechinger I (1998) Managing conflict in software development teams: a multilevel analysis. J Prod Innov Manag 15:423–435

Gugerty L, Olson GM (1986) Debugging by skilled and novice programmers. ACM SIGCHI Bull 17(4):171–174

Herbsleb J, Zubrow D, Goldenson D, Hayes W, Paulk M (1997) Software quality and the Capability Maturity Model. Commun ACM 40(6):31–40

Hewner M, Guzdial M (2010) What game developers look for in a new graduate: interviews and surveys at one game company. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education, pp 275–279

Hollander M, Wolfe DA, Chicken E (2013) Nonparametric Statistical Methods, 3rd edn. Wiley, New York

IEEE Computer Society, Bourque P, Fairley RE (2014) Guide to the Software Engineering Body of Knowledge (SWEBOK), 3rd edn. IEEE Computer Society Press, Los Alamitos

Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp 111–120

Joint Task Force on Computing Curricula (2014) Software Engineering 2014: Curriculum guidelines for undergraduate degree programs in software engineering. Technical report. ACM

Kelley RE (1999) How to Be a Star at Work: 9 Breakthrough Strategies You Need to Succeed. Crown Buisness

Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: Proceedings of the 29th International Conference on Software Engineering, pp 344–353

Ko AJ, Chilana PK (2011) Design, discussion, and dissent in open bug reports. Proceedings of the 2011 iConference, pp 106–113

Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: From metaphor to theory and practice. IEEE Softw 29(6):18–21

Latoza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th International Conference on Software Engineering, pp 492–501

Li PL, Ko AJ, Zhu J (2015) What Makes A Great Software Engineer? In: Proceedings of the 37th International Conference on Software Engineering

Li PL (2016) What Makes a Great Software Engineer. PhD thesis, University of Washington. https://digital.lib.washington.edu/researchworks/handle/1773/37160

Li PL, Ko AJ, Zhu J (2019) Appendix to What Makes a Great Software Engineer? Technical Report MSR-TR-2019-8, Microsoft. https://www.microsoft.com/en-us/research/publication/appendix-to-what-makes-a-great-software-engineer/

Margolis J, Fisher A (2003) Unlocking the Clubhouse: Women in Computing. The MIT Press, Cambridge

Meade AW, Craig SB (2012) Identifying careless responses in survey data. Psychol Methods 17(3):437–455

NSPE (2007) National Society of Professional Engineers Code of Ethics for Engineers. http://www.nspe.org/resources/ethics/code-ethics

Perry DE, Staudenmeyer NA, Votta LG (1994) People, organizations, and process improvement. IEEE Softw 11(4):36–45

Podgurski A, Leon D, Francis P, Masri W, Minch M, Sun J, Wang B (2003) Automated support for classifying software failure reports. In: Proceedings of the 25th International Conference on Software Engineering, pp 465–475

Radermacher A, Walia GS (2013) Gaps between industry expectations and the abilities of graduates: systematic literature review findings. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education, pp 525–530

Radermacher A, Walia G, Knudson D (2014) Investigating the skill gap between graduating students and industry expectations. In: Proceedings of the 28th International Conference on Software engineering, pp 291–300

Rising L, Janoff NS (2000) The Scrum software development process for small teams. IEEE Softw 17(4):26–32

Robillard MP, Coelho W, Murphy GC, Society IC (2004) How effective developers investigate source code : an exploratory study. IEEE Trans Softw Eng 30(12):889–903

Rozovsky J (2015) The five keys to a successful Google Team. re:Work p 1. https://rework.withgoogle.com/blog/five-keys-to-a-successful-google-team/

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: Proceedings of the 29th International Conference on Software Engineering, pp 499–510

Sackman H, Erikson W, Grant E (1968) Exploratory experimental studies comparing online and offline programmmg performance. Commun ACM 11(1):3–11

Shackelford R, McGettrick A, Sloan R, Topi H, Davies G, Kamali R, Cross J, Impagliazzo J, LeBlanc R, Lunt B (2006) Computing curricula 2005: The Overview Report. SIGCSE Bullet 38(1):456–457

Simon H (1955) A behavioral model of rational choice. Q J Econ 69:99–188

Simon H (1973) Applying information technology to organizational design. Public Adm Rev 33(3):268–278

Singer J, Lethbridge T, Vinson N, Anquetil N (1997) An examination of software engineering work practices. In: Proceedings of the 1997 Conference of the Centre for Advanced Studies onf Collaborative Research, pp 174—-188

Valett JD, McGarry FE (1988) A summary of software measurement experiences in the software engineering laboratory. In: Proceedings of the 21st Annual Hawaii International Conference on System Sciences, pp 293–301

Ventures CBS, Knowledge CUD (2000) Risk and return: expected return. http://ci.columbia.edu/ci/premba_test/c0332/s6/s6_3.html

Paul Luo Li is a Principal Data Scientist in the Core Operating System & Intelligent Edge organization at Microsoft in Redmond, WA, USA. He received his Ph.D. in Information Sciences from the University of Washington in 2016, and his Masters in Software Engineering from Carnegie Mellon University in 2007. In addition to his research on "what makes great software engineers", Paul also has research publications on software reliability engineering, large-scale experimentation systems, statistical analysis of local differential private data, and numerous other areas of AI and ML.

Amy J. Ko is an Associate Professor at the University of Washington Information School, where she studies human aspects of programming. Her earliest work included techniques for automatically answering questions about program behavior to support debugging, program understanding, and reuse. Her later work studied interactions between developers and users, and techniques for web-scale aggregation of user intent through help systems; she co-founded AnswerDash to commercialize these ideas. Her latest work investigates programming skills and new methods for learning them, including the programming language knowledge, APIs knowledge, and programming strategies. She received her Ph.D. at the Human-Computer Interaction Institute at Carnegie Mellon University in 2008, and degrees in Computer Science and Psychology with Honors from Oregon State University in 2002.

Andrew Begel is a Principal Researcher in the Ability group at Microsoft Research in Redmond, WA, USA. He received his Ph.D. in Computer Science from the University of California, Berkeley in 2005. Andrew focuses on studying and helping people on the autism spectrum achieve employment and facilitate social interaction. Andrew also explores evolving job roles in the software industry and studies the growing impact of AI technologies on software engineering.