Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

5-2019

How practitioners perceive coding proficiency

Xin XIA

Zhiyuan WAN

Pavneet S. KOCHHAR

David LO
Singapore Management University, davidlo@smu.edu.sg

Citation

How Practitioners Perceive Coding Proficiency

Xin Xia*†, Zhiyuan Wan**, Pavneet Singh Kochhar‡ and David Lo§
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†Faculty of Information Technology, Monash University, Melbourne, Australia
‡Microsoft, Vancouver, Canada
§School of Information Systems, Singapore Management University, Singapore, Singapore
xin.xia@monash.edu, wanzhiyuan@zju.edu.cn, pavneetk@microsoft.com, davidlo@smu.edu.sg

Abstract—**Coding proficiency is essential to software practitioners. Unfortunately, our understanding on coding proficiency often translates to vague stereotypes, e.g., "able to write good code". The lack of specificity hinders employers from measuring a software engineer's coding proficiency, and software engineers from improving their coding proficiency skills. This raises an important question: what skills matter to improve one's coding proficiency. To answer this question, we perform an empirical study by surveying 340 software practitioners from 33 countries across 5 continents. We first identify 38 coding proficiency skills grouped into nine categories by interviewing 15 developers from three companies. We then ask our survey respondents to rate the level of importance for these skills, and provide rationales of their ratings. Our study highlights a total of 21 important skills that receive an average rating of 4.0 and above (important and very important), along with rationales given by proponents and dissenters. We discuss implications of our findings to researchers, educators, and practitioners.**

I. INTRODUCTION

Every software system needs code. Organizations and companies want to hire people with excellent *hard* and *soft skills* to create the needed code. Previous studies have looked at *soft skills* [1]–[4]. For example, Li et al. investigated 53 *soft skills* for software engineers by interviewing 56 software engineers in Microsoft [1], and they divided the *soft skills* into four categories: personal characteristics, decision making, teammates, and software product. However, *hard skills* have not been investigated much. In this paper, we investigate what *hard skills* contribute to coding proficiency. We refer to these hard skills as *coding proficiency skills* - the skills necessary to efficiently and effectively write high-quality code. Various tests and tools have been developed to improve coding proficiency [5], [6]. Additionally, coding proficiency skills are also often assessed in a number of technical job interviews [7], [8].

A large number of *hard skills* may contribute to one's coding proficiency. Often a large investment of time is needed to acquire a new hard skill, and given the limited resources we all have (in terms of time and energy), we often need to make choices. What hard skills are necessary? What hard skill should I invest in next? These questions are often in the mind of both novices and experienced developers who need to learn new hard skills to remain relevant in the ever changing and fast paced IT industry. They may also be in the mind of recruiters who need to select competent software engineers.

In this work, we sought to figure out the hard skills that contribute to coding proficiency and estimate their importance. We first interviewed 15 software practitioners from three software companies, and derived 38 coding proficiency skills grouped into 9 categories: general coding skills, programming language and infrastructure skills, refactoring and reuse, requirement engineering, software design, understanding and learning, interacting with environments, bug prevention and fixing, and estimation. We then invited thousands of practitioners[1] from various backgrounds through emails to take our survey, including those who work in small to large companies and organizations (e.g., Google, Microsoft, LinkedIn and Intel) and those who contribute to open source projects on GitHub. In our survey, we asked respondents to rate the identified skills according to their importance and provide rating rationales. We highlighted 21 coding proficiency skills that are perceived as important and very important, along with rationales given by proponents and dissenters. We make the following contributions:

- We perform a mixed qualitative and quantitative study to investigate how practitioners perceive coding proficiency. We present our results from our 15 interviews, as well as a survey of 340 software practitioners from 33 countries across five continents.
- We derive 38 coding proficiency skills that are grouped into 9 categories and rated based on survey responses. These skills can help novices and software practitioner to be more aware of skills that others deem as (very) important.
- We highlight important coding proficiency skills and the rationales behind the importance ratings for these skills from survey responses.

The remainder of this paper is structured as follows. Section II briefly mentions related work. Section III describes our methodology. Section IV presents the results. We discuss implications and threats to validity in Section V. We conclude and present future work in Section VI.

---

*Corresponding author.

[1] The respondents of our survey are required to have experience in coding.

II. RELATED WORK

A. Studies on Developer Expertise

The closest work to ours is Li et al.'s study [1]. They interviewed 59 software engineers in Microsoft to produce a list of *soft skills* that a great software engineer should have. These skills are grouped into four categories: personal characteristics, decision making, teammates, and software product. Different from this work, coding proficiency skills go beyond soft skills, and most of them have not been studied by Li et al. We aim to fill this gap in research and complement their findings.

Prior work performed empirical studies on how to be a star engineer and conclude nine strategies [2], what employers in a game company look for in new graduates [4], daily work of 8 new hires in Microsoft [3], how developers investigate source code [9], and the relevance of computer science and software engineering education based on a survey of 168 respondents [10]. Different from these studies, our work involves a wide range of coding proficiency skills validated by 340 practitioners who come from 33 different countries and work for different companies.

B. Studies on Measuring Developer Productivity

Prior studies explored factors that contribute to the productivity of developers, including characteristics of workplace and organization [11], team size [12], better management, staffing, incentives, and component reuse [13].

Recently, researchers have also investigated developers' perceptions on productivity [14], how to summarize and measure development activity [15], correlations between personality, style and performance in computer programming [16]. Their findings include reducing context switches and setting goals could improve productivity; developers with *openness to experience* personality have a positive association with breadth-first programming style (i.e., developers incrementally build up a system by implementing *a portion* of each functionality); and developers with *conscientiousness* personality have a positive association with depth-first programming style (i.e., developers implement a specific functionality *to completion* before considering others). Our work is related to but different from the above studies: we investigate developers' perceptions on coding proficiency (particularly the importance of various hard skills).

III. RESEARCH METHODOLOGY

Our study consisted of open-ended interviews and a validation survey. In the open ended interviews, we interviewed 15 developers to get their insights into coding proficiency skills that a software engineer should have. At the end of the interviews, we finalized a collection of 38 skills that can contribute to one's coding proficiency. In the survey, we evaluated the importance of the 38 skills by surveying software practitioners from various background by means of an online survey. Each respondent spent 10-25 minutes to rate the importance of the skills and provided rationales that support the ratings.

A. Open-Ended Interviews

1) Protocol: The first author conducted face-to-face interviews with 15 software practitioners, each interview was completed within an hour. The interviews were semi-structured and divided into three parts.

<u>Part 1:</u> We asked some demographic questions such as the experience the interviewee has on software development/testing/project management, and also asked interviewees to describe the projects they have done.

<u>Part 2:</u> we asked open-ended questions to understand coding skills. The questions include: Are you satisfied with your coding proficiency? How do you improve coding proficiency? What skills would affect coding proficiency? The purpose of this part was to allow the interviewees to speak freely about coding proficiency without any bias.

<u>Part 3:</u> We prepared candidate topics by carefully reading the table of contents of representative coding textbooks (e.g., [17], [18]), skimming the contexts of those textbooks, and referring to high-rated posts on popular Q&A websites (e.g., [19], [20]). The candidate coding proficiency skills span 7 topics, i.e., bug fixing, program comprehension, programming language, implementation, testing, tool usage, and others. We picked a list of topics that have not been explicitly mentioned in the open discussion, and asked the interviewees to further discuss those topics.

At the end of each interview, we thanked the interviewee and briefly informed him/her what we plan to do with his/her responses.

2) Participant Selection: We selected full-time employees from three IT companies in China, namely Insigma Global Service (IGS) [21], Hengtian [22], and a financial company (for confidentiality reason we can only refer to as SS). IGS and Hengtian are two outsourcing companies which have more than 500 and 2,000 employees, respectively. SS mainly builds IT solutions to support the financial service of commercial banks. The selection criteria are as follows:

- We contacted the HR departments of these three companies to get a list of employees. We then removed the employees who are interns, working at a client's company[2], doing adminstration management, or not available due to other reasons (e.g., on vacation or on leave). In the end, we have 406, 104, and 132 candidates left from Hengtian, IGS, and SS, respectively.
- To reduce the potential interference to one's work, The HR departments suggested inviting 10% of the candidates to join our interviews. We randomly selected 40, 10, and 13 employees from Hengtian, IGS, and SS, and invite them to join our interviews through emails. Out of the 63 emails, 2 received automatic replies notifying us of the absence of the receiver; 32 declined our invitation; 11 did not reply to our email; 18 accepted our invitation. Out of these 18 developers, 3 canceled their appointments before the interview.

---

[2]Since IGS and Hengtian are outsourcing companies, some of their employees need to work onsite.

TABLE I: Skills that contribute to coding proficiency, followed by the average Likert scores from the survey responses (*very unimportant* = 1, *unimportant* = 2, *neutral* = 3, *important* = 4, *very important* = 5).

	C1. General Coding Skills	
S1	Write code efficiently (i.e., clear coding task in a short amount of time)	**4.03**
S2	Write efficient code, e.g., the code can run very fast, use less memory	**4.12**
S3	Write well-documented code	**4.07**
S4	Write parallel programs that leverage multiple threads and processes	3.56
S5	Write code that embeds well with code written by others, e.g., write code for a large project team which has many developers and require much collaboration	**4.37**
	C2. Programming Language and Infrastructure	
S6	Master multiple program languages	3.61
S7	Master legacy programming languages (e.g., Cobol)	2.36
S8	Master popular programming languages (e.g., Java)	3.47
S9	Master big data infrastructure (e.g., Hadoop, elastic search)	3.15
	C3. Refactoring and Reuse	
S10	Recognize and extract reusable code from a larger code base	**4.16**
S11	Package, document and distribute a software library for others to reuse	**4.13**
S12	Able to reuse code created internally rather than reinventing the wheel	**4.19**
S13	Reuse suitable third party libraries rather than reinventing the wheel	**4.26**
S14	Refactor code by identifying and eliminating code and architecture smells	**4.28**
	C4. Requirement Engineering	
S15	Extract an abstraction or a model from a requirement (e.g., in UML format)	3.53
S16	Implement a functionality correctly according to the requirement	**4.31**
S17	Recognize mismatches between requirement and implementation	**4.29**
	C5. Software Design	
S18	Break down a complex coding task into smaller tasks that can be implemented separately	**4.45**
S19	Implement functionality following a modular design	**4.34**
S20	Implement functionality following suitable design patterns (e.g., singleton, factory)	3.71
S21	Implement functionality avoiding design anti-patterns (e.g., god class, brain class, feature envy)	3.82
	C6. Understanding and Learning	
S22	Understand existing code in a short period of time	**4.04**
S23	Understand trade-offs between different system architectures	**4.02**
S24	Read and understand most books/articles related to the programming languages the developer mastered	3.88
S25	Acquire new domain-specific knowledge or learn new programming languages fast	**4.03**
S26	Learn the usage of new tools fast	**4.04**
	C7. Interacting with Environments	
S27	Work with multiple IDEs (such as Eclipse and Visual Studio)	3.05
S28	Work with multiple OSes (such as Linux, Windows)	3.38
S29	Modify the programming environment to tailor it to the developers' personal styles	3.64
S30	Work with a version control system (e.g., SVN, Git)	**4.45**
S31	Work with a bug tracking system (e.g., Jira, Bugzilla)	3.82
S32	Work with a code review system (e.g., Gerrit)	3.64
	C8. Bug Prevention and Fixing	
S33	Locate and fix bugs fast and accurately	**4.33**
S34	Write good unit test cases to detect potential bugs	**4.18**
S35	Write good integration test cases to detect potential bugs	**4.02**
S36	Write good system test cases to detect potential bugs	3.95
	C9. Estimation	
S37	Estimate well the effort needed to implement a new functionality	3.83
S38	Estimate the space and time cost of executing a piece of code	3.92

Our 15 interviewees have varied job roles and experience. The interviewees participated in various projects located in the United States, Canada, Japan, Ireland and Germany. Table III presents the detailed information of the 15 interviewees (see Appendix A[3] [23]). In the remainder of the paper, we denote these 15 interviewees as P1 to P15.

3) Data Analysis: We processed the recorded interviews by following the steps below:

Transcribing and Coding We used a transcription service provided by a third-party company to transcribe recordings to transcripts. We then read the transcripts and coded the tran-

[3]https://goo.gl/56fZNj

scripts. During the coding process, we dropped the sentences which are not related to coding proficiency or belong to soft skills[4]. Finally, we generated a total of 805 cards for the coded sentences - 16 to 24 cards for each interview.

Open Card Sorting We performed open card sorting [24] to categorize generated cards for thematic similarity.

Iteration 1: We randomly chose 20% of the cards, and discussed the themes in these cards. The themes that emerged during the sorting were not chosen beforehand. The resulting initial classification scheme contains 34 coding proficiency skills as shown in Table I (except S7, S24, S27, and S28).

Iteration 2: Two authors independently categorized the remaining 80% cards into the initial classification scheme. They left 79 cards that cannot be categorized to be discussed later. After discussing the themes in the left cards, 4 more coding proficiency skills emerged, i.e., S7, S24, S27, and S28 in Table I. Two authors then separately categorized the left cards. We use Cohen's Kappa [25] to measure the agreement between the two labelers. The overall Kappa value was 0.83, which indicates strong agreement between the labelers. After completing the labeling process, the two labelers discussed their disagreements to reach a common decision. Finally, we get 38 coding proficiency skills grouped into 9 categories as shown in Table I.

B. Survey

1) Protocol: We designed a survey to rate the 38 skills based on the perceived importance. To support respondents from China, we translated our survey to Chinese before publishing the survey. Our survey consists of three parts:

Part 1: Demographic we asked demographic questions to understand the respondents' background (e.g., their number of years of professional experience).

Part 2: Ratings of skills We then presented the skills and ask our respondents to rate each of them with one of the following ratings: *very important*, *important*, *neutral*, *unimportant*, *very unimportant*. A respondent can also specify that he/she prefers not answer or don't understand a particular skill. We included this option to reduce the possibility of respondents to provide arbitrary answers.

Part 3: Rationales for each respondent, we randomly sampled two skills that he/she has rated as important/very important, and another two skills that he/she has rated as unimportant/very unimportant, and asked the rationales of their ratings[5].

2) Respondent Selection: We aim to get a sufficient number of software practitioners from diverse backgrounds. We followed a multi-pronged strategy to recruit respondents:

[4]During our interview, our interviewees also mentioned soft skills. However, since our focus is on hard skills, we discarded them during the card sorting process. We define a skill as soft skill if it is related to personal characteristics, decision making, teammates, or software product as defined in Li et al.'s study [1], and not related to coding proficiency.

[5]Note that we use SurveyGizmo functionality to randomly sample skills based on those marked as (very) important or (very) unimportant. Due to SurveyGizmo limitation, it is not possible to include complicated logic. Still, since the process is random, if two skills have equal likelihood of being selected as (very) important or (very) unimportant, they should have equal chance to be selected for respondents to provide their rationales.

- We contacted professionals from various countries and IT companies and asked their help to disseminate our survey to some of their colleagues and friends. We sent emails to our contacts in Microsoft, Google, LinkedIn, Box.com, Infosys, Tata Consultancy Services, Hengtian, IGS, and other small to large companies from various countries to fill up the survey and disseminate it. By following this strategy, we hope to recruit respondents of professional developers in the industry from diverse organizations and backgrounds.
- We tended to recruit active practitioners in open source projects in addition to professionals working in industry. We used GitHub REST APIs to mine the commit logs of projects hosted on GitHub, and identified contributors with more than 2,500 commits. In total, we identified 1,588 email addresses and sent invitations to these addresses. Out of these emails, 156 were not delivered, 62 received automatic replies notifying us of the receiver's absence.

In the email, we emphasized that the respondents should have experience coding, but the role can vary, i.e., we welcomed all software practitioners including developers, tester, and project managers.

We received a total of 340 responses from 33 countries across five continents. The top two countries where the respondents reside are China and the United States.

The professional experience of our respondents varies from 0.4 years to 20 years, with an average of 7.36 years. For the open source practitioners, we sent 1,588 emails and got 196 response (return rate = 12%). For the industrial professionals, it is hard to estimate return rate. This is because we personally invited our industry contacts to participate and distribute survey to some colleagues. We cannot know the exact number of colleagues to whom they distribute the survey.

3) Data Analysis: We collate the ratings that our respondents provide. We drop "I don't understand" and "I prefer not to answer" ratings that form a small minority of all ratings. Next, we convert these ratings to Likert scores from 1 (very unimportant) to 5 (very important). Next, we compute the average Likert score of each skill. Furthermore, we extract comments that our survey respondents give to explain the reason why they think a particular skill is important/very important or unimportant/very unimportant.

IV. RESULTS

In this section, we describe how software practitioners rated the 38 coding proficiency skills grouped into nine categories, along with the rationales that our respondents provided to support their ratings. We consider three research questions:

- **RQ1:** How do practitioners perceive the 38 derived coding proficiency skills?
- **RQ2:** What are the highly ranked coding proficiency skills as practitioners deem important?
- **RQ3:** What are the practitioners' rationales for perceiving a particular coding proficiency skill important or unimportant?
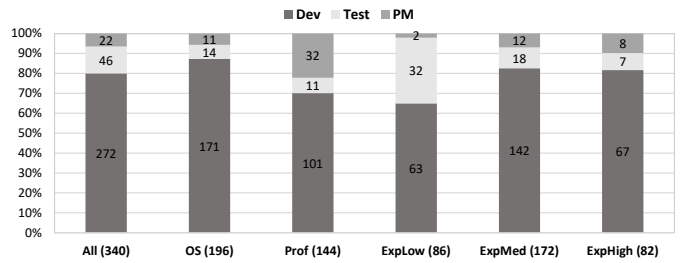

Fig. 1: Survey respondents demographics. The number indicates the count of each demographic group.

A. RQ1: Importance of the 38 Skills

Table I presents the average Likert score of the 38 skills from all respondents in the third column. We observe that 21 out of the 38 skills have an average Likert score above 4.0, indicating that the 21 skills are perceived as important and very important by our respondents.

We further investigate the ratings of various demographic groups as below:

- **Main job role**: Respondents who are developers (Dev), testers (Test) or project managers (PM).
- **Experience level**: Respondents with low experience (ExpLow), i,e., we define as the 25% with the least experience in years; with medium experience (ExpMed); or with most experience (ExpHigh), i..e, we define as the 25% with the most experience in years.
- **Education level**: Respondents with/without advanced degrees (Adv/NonAdv).
- **Open source developers vs. professionals**: Respondents who are open source developers (OS) or professionals (Prof).

The grouping of respondents on experience follows prior work, e.g., Carver et al. [26] and Lo et al. [27]. Same demographic categories are also used by previous studies [14], [15], [26]–[31]. We notice that PMs might work as developers before; Tester might require coding skills as well. In our survey, we emphasize that our respondents should have coding experience. Figure 1 presents the distributions of respondents across different demographic groups:

- Majority of the respondents (80%) are developers, 14% and 6% of the respondents are testers and PMs, respectively.
- More developers and less PMs come from open source projects, as compared to the respondents from commercial projects, e.g., 32 PMs are from commercial projects while only 11 PMs are from open source software project.
- As respondent's experience increases, the percentage of testers decreases, while the percentage of PMs increases.

Figure 2 shows the importance ratings of the 38 coding proficiency skills across various demographic groups. We observe that all demographics give more "Very Important" and "Important" ratings as compared to "Unimportant" or "Very Unimportant". Only a minority gave "Unimportant" and "Very Unimportant" ratings (less than 15%) across all demographic groups. More than 70% respondents across all demographic groups rated the 38 coding proficiency skills
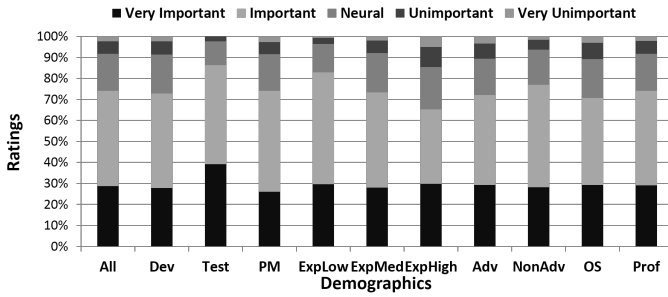
Fig. 2: Importance of the 38 coding proficiency skills to respondents of various demographic categories.

as "Very Important" or "Important", and around 28% - 39% across all demographic groups rated the 38 coding proficiency skills as "Very Important". In addition, we observe a few differences between various demographic groups (the differences are significant by using Fisher's Exact test [32] with Bonferroni correction [33] at a 95% confidence level):

- Testers consider coding proficiency skills more important than developers and project managers.
- Low-experience practitioners consider these coding proficiency skills more important than practitioners with medium and high experience.
- Professional developers perceive coding proficiency skills more important than open-source developers.
- Respondents with advanced degrees consider coding proficiency skills more important than their counterparts without advanced degrees.

B. RQ2: Highly Ranked Skills

To further analyze the results, we applied Scott-Knott Effect Size Difference (ESD) test [34] to group the 38 skills into statistically distinct ranks according to their Likert scores. Note that we excluded 5 responses that select "I don't know" for our Scott-Knott DSD test. As shown in Tantithamthavorn et al.'s study [34], the Scott-Knott test assumed that the data is normally distributed, which might create groups that are trivially different from one another. To address the limitations of Scott-Knott test, Tantithamthavorn et al. proposed the Scott-Knott Effect Size Difference (ESD) test to correct the non-normal distribution of an input dataset, and leverage a hierarchical clustering to partition the set of treatment means (in our case: means of Likert scores) into statistically distinct groups with non-negligible effect sizes.

Table II presents the 38 skills as ranked according to the Scott-Knott ESD test in terms of means of Likert scores for all the respondents. In addition, we identify and discuss high rated coding proficiency skills across the various demographic categories (i.e., different job roles and experience levels) in Appendix B [23].

C. RQ3: Rationales

In the remainder of the paper, we use ✓ or ✗ to denote the rationale why respondents perceive a skill as important or unimportant. We carefully read all the comments from the respondents, and removed comments which do not describe

TABLE II: Skills as ranked according to the Scott-Knott ESD test (All respondents).

Group	Skill
1	**S18:** break down a complex coding task into smaller tasks
	S30: work with a version control system
	S19: implement functionality following a modular design
	S5: write code that embeds well with code written by others
	S33: locate and fix bugs fast and accurately
	S17: recognize mismatches between requirement and implementation
	S16: implement a functionality correctly according to the requirement
	S14: refactor code by identifying and eliminating code and architecture smells
2	**S13:** reuse suitable third party libraries
	S12: reuse code created internally
	S34: write good unit test cases to detect potential bugs
	S10: recognize and extract reusable code from a larger code base
	S11: package, document and distribute a software library
3	**S22, S26, S3, S23, S35, S25, S1, S36**
4	**S38, S24, S31, S37, S21**
5	**S20, S32, S29, S6, S4**
6	**S15, S8, S28**
7	**S9:** master big data infrastructure
8	**S27:** work with multiple IDEs
9	**S7:** master legacy programming languages

any rationale. For each skill, the first two authors manually extracted key phrases from the comments, and grouped comments based on the key phrases. Take two comments in S7 for example, "*Not widely used anymore in many large projects*" and "*The majority of programming jobs won't involve these at all*" both express that legacy programming languages are not widely used. Thus, we put them into one group. Note that some comments may belong to multiple groups because they listed more than one reasons. We broke these comments down and put into multiple groups.

We present the comments on the most highly ranked skills (Group 1 in Table II), and that on the most lowly ranked skills (Group 7–9 in Table II). Additionally, we list comments for selected skills whose importance is controversial in nature (i.e., receiving close number of positive and negative comments). For the complete list of the comments on 38 skills, please refer to Appendix C [23].

1) Most Highly Ranked Skills (Group 1 in Table II):

Break Down a Complex Coding Task into Smaller Tasks (S18). Respondents consider S18 as important since it: (1) helps developers to solve complex problems, (2) allows developers to create reusable and maintainable code, (3) supports effective distribution of tasks among team members, and (4) makes quality assurance checks (e.g., creation of unit tests, code review, etc.) easier.

✓ "*The more **complex problem** is it is harder to implement, breaking it into smaller tasks helps others focus on the implementation.* "

✓ "*Because if you don't you will end up with an **unmaintainable** mess.*"

✓ "*This is necessary for unit testing, as well as **dividing work among a team**.*"

Work with a Version Control System (S30). Respondents consider S30 as important since version control systems: (1) support team work, (2) facilitate reusability by tracking reusable versions of a program, and (3) allow developers to revert to older versions when latest versions are problematic.

✓ "*This is fundamental to the ability of people to **work in a team**, writing code together.*"

✓ *"Different versions of projects might correspond to different requirements from different clients, which improves the **reusability**."*

✓ *"If problems appear in a new version, version control system can **trace back** to a right older version."*

Implement Functionality Following a Modular Design (S19). Respondents explained that modular design (1) helps create understandable, reusable and maintainable code, and (2) supports collaboration among developers.

✓ *"Keeping things separated as much as possible helps improve **readability** of the code ... "*

✓ *"Modular design **contributes to the collaboration between** developers, and it is easier to share work in big teams."*

Write Code that Embeds Well with Others' Code in a Large Project Team (S5). Respondents describe well-embedded code as an important skill: (1) most software projects are challenging enough to require team work, (2) software maintenance requires one to build upon existing code often written by others, and (3) novice developers have little experience working in a large project.

✓ *"... Without being **able to work with other**, a lot of time will be lost to finish a project."*

✓ *" If I'm going to hire someone, they need to know how to do **maintenance programming** and their code should work with the team's code."*

✓ *"It is important to all developers especially for junior developers, since they are **lack of experience** on working in a large project team."*

Note that low-experience respondents list this skill as the most important skill.

Locate and Fix Bugs Fast and Accurately (S33). Due to the complexity of software systems, bugs are inevitable, and bug localization and fixing are two of the most important activities during software development and maintenance [35]–[37]. Respondents noted that (1) software quality is crucial, and (2) others may depend on a piece of code a developer is fixing:

✓ *"It doesn't matter how quickly your code runs if it does the **wrong thing**."*

✓ *"... all new code that you write will **slow down your entire team** as others will have to find and fix your bugs for you."*

Recognize Mismatches Between Requirement and Implementation (S17). Respondents state that: (1) developing the right thing considering various subtleties of a requirement is challenging and yet crucial, and (2) detecting mismatches early prevents wastage of time and resources:

✓ *"Someone who does not understand the subtleties of a requirement, and is **satisfied with an approximate implementation is a burden**. On the other hand, someone who can consistently understand requirements precisely, and provide matching implementations can be an **amazing asset to a team**."*

✓ *"Because if there is a mismatch the contract is void and the implementation needs to be completed correctly. It was a **waste of time and effort.**"*

Implement a Functionality Correctly According to Requirement (S16). Respondents consider S16 as important since it is one of the most important metric to evaluate project success:

✓ *"That is why we are implementing the software, and will **make the product successful**. At the end of the day it is the **ultimate goal of the profession**."*

In contrast, some argue that ability to collect valid requirement is even more important:

✗ *"Actually it is more important to first **confirm that the requirement is valid and useful**. "*

We notice high-experience respondents rank S16 in Group 3.

Refactor Code by Identifying and Eliminating Code and Architecture Smells (S14). Code refactoring is one of the hot topics in software engineering [38]–[40]. The purpose of code refactoring is to eliminate code and architecture smells [41]–[43] from systems to make them robust for future change and decrease maintenance cost.

Respondents explain that (1) smells are common and often introduced due to tight software schedule, (2) it allows developers to improve software design, and (3) it allows developers to payoff technical debts and create maintainable code:

✓ *"More often than not, people say they're **doing it "quick" as prototype** but don't even see half the problems they've introduced. "*

✓ *"You need to be able to **understand the deficiencies in designs**; in real life you don't want to hit a screw with a hammer, why would you do it in code?"*

✓ *"Beginner programmers can write tons of bad code quickly. They need to learn how to pay down **technical debt**, otherwise it can kill a projects long term momentum."*

2) Most Lowly Ranked Skills (Group 7 – 9 in Table II):

Master Legacy Programming Languages (S7). All respondents consider S7 as unimportant. The rationales include (1) majority of programming jobs do not require the mastery of legacy programming languages, (2) many new technologies are introduced and time and effort are needed to acquire competencies of those new technologies, and (3) learning legacy languages may not help in improving one's skill in newer technologies:

✗ *"It's **rare** that that comes up in my work. It's a **niche case** that, where it matters, is important, but not part of most developers' careers."*

✗ *"it would be waste of time to learn antiquated language, it's **better to spend time learning something new**."*

✗ *"**Mastering a dead language** might be fun, but it probably **won't contribute** to your success as much as mastering a current language."*

Work with Multiple IDEs (S27). Detractors point at that (1) good developers do not need multiple IDEs and can work even with simple text editors, (2) IDEs may be in the way when coding in multiple programming languages, and (3) learning a new IDE may require substantial time investment.

✗ *"All best coders I've seen (and I believe it's a rule in general) **are perfectly fine with simple text editor**, e.g. notepad. "*

✗ *"IDEs are just helpers and good tools for uni-language programmers. **For polyglots programmer it is often something in the way**. Generally using emacs or vim is preferred. "*

✗ *'Any IDE will work and stick with one would be better because **learning IDE can also take time.**"*

Proponents argue on the value of using the best IDE for a job:

✓ *"When working with different projects or languages, the **preferred IDE can be different**."*

Note that low-experience respondents perceive this skill as the second most unimportant coding proficiency skill.

Master Big Data Infrastructure (S9). Respondents consider explain that (1) there are not many big data projects, and most of important development tasks are not 'big data' things, and (2) understanding the principle of how to run code efficiently is more important:

✗ *"This is **context specific**. So in some jobs this might be important, but it's not essential to being a good software developer at all. ."*

✗ *"It's important to have an **understanding** of **the cost of running your code and how it scales**. But it's not perse necessary to have knowledge of big data infrastructures, that's a quite specific a niche. ."*

3) Selected Controversial Skills (Group 2 – 6 in Table II):
Complete Coding Task in Short Time (S1). Respondents consider S1 as important since (1) it is a basic skill to acquire, (2) project schedule is tight, (3) it allows developers to have more time to test code, and (4) it allows one to learn more things by completing more projects:

✓ *"The **project schedule is always tight**, we have to clear a coding task in a short amount of time."*

✓ *"Having more time to **test the code**, to reduce the bugs."*

✓ *"Writing faster means you can do more experiments, which means you'll **learn faster.**"*

Nonetheless, some respondents believe that emphasis should be put on producing good design and writing good code. This may require more upfront investment in time, but benefit in the long run. Hastily written code may have many bugs. Other respondents work on the projects with flexible schedule:

✗ *"We should spend **some time on designing** before we write the code."*

✗ *"My **open source projects are developed on free time**, so it is not important how much it takes."*

Write Well-Documented Code (S3). Respondents consider well-documented code important because (1) it promotes reuse, (2) promotes inclusion of newcomers, (3) helps in program evolution tasks.

✓ *"Without documentation the code has no value and can not be **reused** by others."*

✓ *"Well documented code is essential for project maintenance, **inclusion of newcomers**, and the projects with high turnover rate."*

✓ *"The code written needs to be understandable for other team members to **be able to modify or adapt later** down the timeline or in cases where the developer is no longer active on the project."*

However, not everyone regards S3 as an important skill because they believe developers should write simple, self-explaining and clearly structured code.

✗ *"Documentation doesn't matter much, as long as it (the code) is **clearly structured and the developers are able to talk about it.**"*

Master Multiple Programming Languages (S6). Proponents note that (1) multiple languages expand the view of problems, and help collaborate with developers across a broad range of projects, (2) new programming languages are introduced over time, and (3) different programming languages have their own strengths and weaknesses.

✓ *"This helps a developer look at a problem **from different perspectives**, and **understand** other developer perspectives"*

✓ *"**The technology is changing** and what used to be COBOL, C, C++ is moving to golang, rust, node (javascript) ... "*

✓ *"Different languages have their **own strengths and weaknesses** as well as problem domains they are good at. "*

On the other hand, others consider it as unimportant because programming languages are similar, and deep mastery of one language may matter more to get a job.

✗ *"Much programming knowledge is **general**. For example, if you are proficient with Java you should be able to apply these skills to C# as it is a very similar language."*

✗ *"The opposite is really true, these days - **mastery of one language** - and one popular framework within a language is the best way to **get a job**, and it's where the most effective programmers tend to be."*

Interestingly, many practitioners hold negative opinions on mastery of multiple languages, while the balance tilts towards viewing S6 as important.

Master Popular Programming Languages (S8). Supporters argue that popular languages can serve as *lingua franca* that connects practitioners, and one needs to master these to remain relevant in the job market.

✓ *"It's always good, when talking about something specific to go back to something more general, **something everyone knows**. ".*

✓ *"If you don't, your work **will not be relevant.**"*

Others hold an opposite view arguing that popular languages may not be best languages for various problems. One should be able to pick an appropriate language given a specific problem rather than being swayed to popular languages.

✗ *"Popular languages are **not necessarily good languages**. What languages one should use **depends on situation**. Popularity has little to do with the usefulness."*

✗ *"... you need have the **correct language** for the current problem, no the popular one. Languages are just tools to **express your idea**."*

Extract an Abstraction or a Model from a Requirement (e.g., UML) (S15). Supporters state that UML separates developers from coders, helps in planning and ensuring that the end product matches requirements, works as a communication tool among developers, and helps newcomers to get started.

✓ *"Because this ability **separates developers from coders** ..."*

✓ *"If we don't have a model, when we write code for some period (e.g., two months), we may **forget what the original purposes** of the code."*

✓ *"UML can be used as a tool for **communication purpose**, and **help newcomers to understand the project**."*

Ho-Quang et al. also found that collaboration and communication is the most important motivation to use UML, and the use of UML can help new contributors to get started, but there is no evidence that the use of UML can help to attract new contributors [44].

However, a substantial population of our respondents hold an opposite opinion: (1) they do not find UML useful, (2) many experts do not use UML, (3) open source developers do not prefer to be constrained by model, and (4) software requirements change often.

✗ *"I've **not found UML diagrams to be useful** for anything but filler in giant specification documents."*

✗ *"I've **never known anyone who I have respected that uses UML format**. For example, Donald Knuth, Ken Thompson (Unix/Go), ... These are high-profile names, but just about anyone I've worked with has never used UML."*

✗ *"Opensource developers usually do what they like (or love) to do. They love imaging and creating. There is **no point in telling them what to code so precisely** ... "*

✗ *"**Requirements change all the time**, and only as soon as you implement it you see all consequences and come across unexpected side effects. It's better not to overthink requirements. "*

Implement Functionality Following Design Patterns (S20). Proponents argue that design patterns give the code additional structure to make it easy to understand and communicate, and help the improve the ability to apply abstract design ideas to concrete code implementations.

✓ *"Design patterns give the code additional structure that **makes it easier to understand and to communicate.**"*

✓ *" Relying on these patterns indicates a depth of knowledge and ability to **apply abstract design ideas to concrete code implementations.**"*

In contrast, detractors state that: (1) writing understandable code does not necessarily need design patterns, (2) they mostly apply to OO (object-oriented) languages, which have conflict with other good coding practices, (3) being overly pattern driven leads to overly abstract solutions, (4) design patterns are often improperly used and become to anti-patterns, and (5) design patterns make code hard to understand and maintain.

✗ *"Code should be obvious. Design patterns are a tool which can help, not an end."*

✗ *"Patterns tend to be routed in certain paradigms, say OO. They also tend to fall out of other good coding practices."*

✗ *"Being overly pattern driven trends to lead to overly abstract solutions ... "*

✗ *"For the most part, design patterns are now followed slavishly and became anti patterns. I don't WANT to have AbstractFactoryFactoryGenerator classes in my codebase."*

✗ *"Sometime the code optimized by using design patterns are hard to understand and maintain ... "*

Implement Functionality Avoiding Design Anti-patterns (S21).

Proponents argue that avoiding anti-patterns (1) helps avoid common mistakes, (2) makes code easy to understand, and (3) helps avoid overcomplicate solutions.

✓ *"Shows proficiency with different kinds of problems and understanding of what mistakes are common. To avoid running into design problems later on when working on a bigger project."*

✓ *"... Bad patterns lead to unmaintainable code, hard-to-find and hard-to-fix bugs, and make it difficult to scale a developer team to work on multiple parts of the codebase."*

✓ *"Design patterns do not solve everything, and people tend to overcomplicate solutions with them when there are simpler solutions that don't necessarily have a name."*

On the other hand, detractors are more pragmatic and opine. They argue that what matters more is getting work done rather than actively caring for anti-patterns.

✗ *"I think that the goal of code is to get work done, not to be perfect. Avoiding an anti-pattern makes sense, but sometimes it's ok to do those things to get the job done."*

Understand Existing Code in a Short Period of Time (S22).

Proponents mention that understanding code (1) is needed by developers when interacting with existing systems, (2) is important for code review, (3) allows developers to identify reuse opportunities, and (4) help get up to speed quickly when joining a new team or helping out in a new task.

✓ *"More often than not developers will be faced with interacting with existing systems."*

✓ *"In open source we review peers code a lot ... This provides another developers philosophy of their design. "*

✓ *"If they take too much time to understand the code, either they may take far too much time to write their own (and be quickly discouraged) or they may miss useful piece of code that could save them a lot of time."*

✓ *"For newcomers, if they can understand the code quickly, they will join the development or bug fix task fast. "*

Detractors state that "good things take time" and rushing through things may not result in good comprehension.

✗ *"Most projects are complicated, and take a long time to get to know intimately. Being able to understand patterns in projects, and eventually fully comprehend them, is more important than picking them up in a short period of time. Good things take time."*

Write Good Unit Test Cases to Detect Potential Bugs (S34).

Proponents argue that unit tests can serve as good documentation and communicate intent, and prevent regression bugs.

✓ *"Using code to validate code tends to help with communicating intent, and maintaining that documentation over time ."*

✓ *"Because tests are critical to prevent regression bugs."*

Detractors view unit tests require much time to create and maintain with little return, and some view other forms of testing to be more important.

✗ *"Usually unit tests take lots of time with no or small return. They require lots of maintenance and usually influence on module design. "*

✗ *"Unit tests, ... fail to detect structural (inter-unit) misdesigns and give a false sense of security."*

Write Good Integration Test Cases to Detect Potential Bugs (S35).

Proponents argue that integration tests (1) are valuable to ensure that code changes do not introduce adverse impact with high degree of confidence, and (2) help simulate what a user may do and detect problems early.

✓ *"Excellent integration tests are the proof of writing code for the right purpose, and also enable very fast refactorings with high degrees of confidence."*

✓ *"The ability to write the integration test cases, is more a capability to spot potential issues by imagining what a user would do ... "*

A minority of respondents hold a negative opinion since integration tests may not be suitable for some software projects.

✗ *"Most of the software I write is not well suited to typical testing methods. Integration tests are as far as I go and even that only occasionally."*

Estimate Well the Effort Needed to Implement a New Functionality (S37).

Supporters state that (1) estimation helps in planning - a software engineer can also attract others to join his/her effort if estimation works well, (2) good understanding of a problem is an essential first step towards creating a good solution, and (3) estimation helps determine project feasibility.

✓ *"So you can plan better. If estimate well, a software engineer can also attract others to join his/her effort (especially in open source development setting)"*

✓ *"It's important to understand a problem before attempting to implement it, but prototype development can be an important part of the learning process."*

✓ *"Because it allows you to make accurate guesses / estimations on the feasibility (and chance for a successful implementation) of (requested) new features. "*

Detractors note that estimation are often inaccurate; Many software engineers do not care about estimates.

✗ *"Estimates are unlikely to bear much relationship to reality. Try it and see."*

V. DISCUSSION

A. Implications

1) For Researchers: **Skill Measurement, Impact Analysis, and Automated Tutoring** Our work highlights the skills that are perceived important by our respondents. As a next step, future research could develop measurements to operationalize these skills. Such measurements can be defined based on artifacts that a software engineer has created, GitHub profile, CV, and a set of tests or questionnaires. One can then investigate how accurate such measurements are against for example peer assessments or self-declarations. Such measurements can also help managers (assess software engineers), educators (assess students), software engineers and students (assess themselves and make improvements). It would also be highly interesting to investigate the impact of varying different skills to the project outcomes or project teams. Furthermore, it would be worthwhile to develop automated tutoring tools to help software engineers improve their coding proficiency skills.

UML 3.0 Looking at ratings and comments that we receive for S15, we note a substantial push back on modelling among our respondents. Numerous respondents find UML diagrams, which often require much time to create, to be not useful

for their work, especially due to the high rate of requirement changes. Many prefer writing code than drawing UML diagrams. This poses a challenge as well as an opportunity for researchers to create a new modelling technique that is easy to use by practitioners, can deal with the high rate of requirement changes, and can be nicely linked with source code that developers write.

Our findings are consistent with Petre's [45]. Petre investigated developers from 50 companies, and 35 of them did not use UML at all. Moreover, Ho-Quang et al. found that collaboration and communication is the most important motivation to use UML [44]; thus, would be interesting to investigate how to build even more effective models that can be adopted more widely to communicate various pieces of information.

Design Patterns: Validation, New Patterns, and Recommendation Systems Similar to modelling, a substantial number of our respondents push back on design patterns. Detractors argue that design patterns mostly apply for OO, which can be improperly or overly used. This leads to very abstract solutions impairing understanding, may conflict with other good practices, and many good patterns are not listed as design patterns. This may give opportunities to researchers to: (1) validate claims made by developers through a controlled experiment or field study, or by looking into historical data stored in software repositories (e.g., by correlating design pattern use and number of bugs, etc.), (2) create or curate additional design patterns that apply to many different languages beyond OO languages by interviewing practitioners for best practices, or by mining software repositories, (3) design tools that can give advice on design pattern misuse and recommend suitable actions to make applying design patterns a net benefit. Our findings are consistent with prior studies [46]–[48]. Jaafar et al. investigated the relationship between design patterns and faults by performing an empirical study on six design patterns and 10 anti-patterns in ArgoUML, JFreeChart, and XercesJ, and they found classes which have dependencies with anti-patterns are more fault-prone than others [46]. Wendorff et al. [47] investigated the misuse of design pattern in a large industrial project, and he found that developers misuse design patterns since (1) they do not understand the rationale behind the patterns, or (2) the implementation does not match the project requirement. Scanniello et al. setup four controlled experiments with 88 participants to investigate the relationship between design patterns and program comprehension, and they found that documenting design-pattern instances can help to improve the efficiency of source code understanding for respondents with adequate level of experience [48].

2) For Educators: Twenty one skills are rated as 4 (important) or above by the majority of our survey respondents. We make the following recommendations based on high-rated skills and our respondent comments:

General Coding Skills Educators should emphasize on abilities to code efficiently, write efficient code, document well, and embed one's code well to other's code. "Document well" is often less emphasized in standard computer science curriculum but it is highly valued by practitioners. Also, note that, as

compared to the above skills, advanced topics like parallel programming receive less support from our respondents.

Programming Language and Infrastructure Educators should put less emphasis on specific programming languages but more on programming paradigms.

Refactoring and Reuse The skills under this category are highly valued by our respondents. Thus, educators should emphasize more on these skills, e.g., recognizing and extracting reusable code from a larger code base, and refactoring code by identifying and eliminating code and architecture smells. Often these skills are neglected in many standard CS courses which focus on ability to write code from scratch.

Requirement Engineering and Software Design Educators should teach more problem solving skills (e.g., divide and conquer), basic design skills (e.g., modular design), and provide project-based courses, to help students decompose and solve complex problems, write well-modularized program, and train students to implement requirements correctly and detect mismatches. These skills are highly valued by practitioners. Note that basic design skills (e.g., modular design) are more valued by our respondents than skills that are often viewed as more advanced (e.g., UML modeling and design patterns).

Understanding and Learning Educators should put more emphasis on program comprehension skills, and ability to appreciate trade-offs. The earlier is often lacking in many basic CS courses as students are often not exposed to read and understand large amount of code. Educators can also introduce more domain concepts which go beyond traditional CS. For example, they can encourage students to take classes on specialized domains (e.g., accountancy and finance). Moreover, educators should train students on acquiring new skills and tools fast, i.e., the ability of learning to learn. These skills are highly valued by practitioners.

Interacting with Environments Educators should at least introduce version control systems which receive a much higher rating than other development tools.

Bug Prevention and Fixing Educators should put emphasis on debugging and testing skills as they are highly valued by practitioners.

3) For Practitioners: Many practitioners are often not clear on how to improve their coding proficiency [3]. Our findings conclude a number of skills that our respondents aspire to achieve. The average ratings of all skills except one are above 3.0, all except five are above 3.5, and 21 skills are above 4.0. We also provide rationales on why each skill is perceived as important or not. For many skills, our respondents have different opinions, and we highlight all sides of the argument to help practitioners decide upon which set of skills to improve based on their circumstances.

Besides, our findings may help practitioners better present themselves to prospective employers. For the skills that experienced practitioners value, practitioners can demonstrate to prospective employers that they have or can develop these skills. On the other side of the coin, managers and recruiters could identify suitable candidates by checking their proficien-

cy on the listed skills, especially those that are marked as important or very important by many practitioners.

B. Threats to Validity

Construct Validity. Our interviewees come from three companies. Projects done in these companies cover a wide range of domains. Developers working there have also worked in other companies. However, they may not be representative of all developers. To mitigate this potential bias, we have carefully chosen questions and topics for the interviews, and performed a survey that involve a large population of developers from various companies around the world. Our survey respondents complete the survey based on their belief and perception. It is possible that they conflate the skills that are very important and the skills that are very relevant to their projects or industrial contexts. To mitigate this threat, we have tried to survey many people; a total of 340 practitioners from various companies in 33 countries across 5 continents participated.

Internal Validity. It is possible that some of our survey respondents do not understand some of the 38 skills well. To reduce this threat to validity, we provide "I don't understand / I prefer not to answer" option in our survey, and we find that the number of respondents who choose this option to be small (2.3%). We also translate our survey to Chinese to ensure that respondents from China can understand our survey well. It is also possible that we draw wrong conclusions about respondents' perceptions from their comments. To minimize this threat, we read transcripts many times, and checked the survey results and the corresponding comments several times.

The selection of skills produced at the end of the interview may not be comprehensive and may be biased to the background of experts – who may not be able to articulate their own skills, focus more on skills that involve public demonstrations, and who may praise skills that have no actual benefit to performance – whom we interviewed. To mitigate this bias, we have taken the following steps:

- Aside from asking direct questions of what skills they deem important, we also ask them to discuss topics that they have not explicitly mentioned. The topics were selected from software engineering text books online resources; they include: bug fixing, program comprehension, programming language, implementation, testing, tool usage, and others.
- We have performed a survey to check whether the interviewees' opinion is perceived to be correct by many practitioners.

External Validity. To improve the generalizability of our findings, we have interviewed 15 respondents from 3 companies, and surveyed 340 respondents from 33 countries across 5 continents working for various companies (including Microsoft, Google, LinkedIn, Box.com, Infosys, Tata Consultancy Services, Hengtian, IGS and other various small to large companies) or contributing to open source projects hosted on GitHub. Still, our findings may not generalize to represent the perception of all software engineers. For example, most of the respondents are from Asia, North America, and Europe, and there are only two respondents from South America, and none

of the respondents are from Africa. Moreover, considering most of the users on GitHub are male, our findings might not be generalized to female developers. In the future, it would be interesting to perform another study to investigate how female developers perceive coding proficiency. Moreover, each respondent rates each skill, thus each skill receives 340 responses. This is substantial considering prior studies include similar number of respondents (i.e., 357 [14], 364 [49], and 512 [27] respondents) to rate many statements (i.e., 23 [14], 28 [49], and 571 [27] statements).

Another threat related to the completeness of our 38 coding proficiency skills. In this paper, we finalized these skills based on the open-ended interviews of 15 interviewees. Also, in the end of our survey, we also asked the respondents to provide the additional coding proficiency skills. Among the 340 responses, 144 respondents provided comments for additional skills. We also manually analyze these additional skills, and removed 76 comments which are related to *soft skills* (e.g., "good working behavior", "effective communication", and "the key is good communication and social skills"), general skills for all engineers (e.g., "understanding of grammar and math"), or noises (e.g., "N/A", "yes", or "no"). Next, we applied closed card sorting to categorize the remaining 68 comments, i.e., we tried to categorize them into the 38 skills, and we left the comments which belong to none of these skills. We notice most of the comments provided supplementary explanation to our 38 skills, e.g., "ability to pick the right level of abstraction" is an explanation for S15 (extract an abstraction or a model from a requirement), "rapid prototyping" is an explanation for S1 (write code efficiently). Moreover, we only found two additional skills that are not covered by our survey, i.e., "write awesome SQL", and "use of (agile) methodologies like Scrum, Kanban or GTD". This threat could be removed by including these two skills, and inviting more respondents.

VI. Conclusion and Future Work

In this work, we survey 340 practitioners from diverse backgrounds on their perceptions of coding proficiency. We derive a total of 38 skills which are grouped into 9 categories from our interviews, and ask survey respondents to rate the importance of these skills and provide the rationales of the ratings. We find that all but 1 of the skills receive an average rating of more than 3.0 (neutral), all but 5 receive an average rating of more than 3.5 (between neutral and important), and 21 receive an average rating of 4.0 and above (important and very important). Our findings can help practitioners by highlighting important skills to acquire, and educators by recommending important skills to include in the curriculum. The findings also highlight opportunities that software engineering researchers can work on to better help practitioners in their tasks. Future work could work on some of these opportunities and expand our study to address some of the threats to validity.

Replication package: https://goo.gl/qXCMSm

REFERENCES

[1] P. L. Li, A. J. Ko, and J. Zhu, "What makes a great software engineer?" in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 700–710.

[2] R. E. Kelley, "How to be a star engineer," *Spectrum, IEEE*, vol. 36, no. 10, pp. 51–58, 1999.

[3] A. Begel and B. Simon, "Novice software developers, all over again," in *Proceedings of the Fourth international Workshop on Computing Education Research*. ACM, 2008, pp. 3–14.

[4] M. Hewner and M. Guzdial, "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*. ACM, 2010, pp. 275–279.

[5] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux, "Code hunt: Experience with coding contests at scale," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 398–407.

[6] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop, "Pex4fun: A web-based environment for educational gaming via automated test generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 730–733.

[7] A. Aziz, T.-H. Lee, and A. Prakash, *Elements of Programming Interviews: The Insiders' Guide*. CreateSpace, 2012.

[8] J. Mongan, N. Kindler, and E. Giguere, *Programming Interviews Exposed: Secrets to Landing Your Next Job (2nd ed.)*. Wrox, 2007.

[9] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *Software Engineering, IEEE Transactions on*, vol. 30, no. 12, pp. 889–903, 2004.

[10] T. C. Lethbridge, "A survey of the relevance of computer science and software engineering education," in *Software Engineering Education, 1998. Proceedings., 11th Conference on*. IEEE, 1998, pp. 56–66.

[11] T. DeMarco and T. Lister, "Programmer performance and the effects of the workplace," in *Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 1985, pp. 268–272.

[12] J. D. Blackburn, G. D. Scudder, and L. N. Van Wassenhove, "Improving speed and productivity of software development: a global survey of software developers," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, pp. 875–885, 1996.

[13] B. W. Boehm, "Improving software productivity," in *Computer*. Citeseer, 1987.

[14] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 19–29.

[15] C. Treude, F. Figueira Filho, and U. Kulesza, "Summarizing and measuring development activity," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 625–636.

[16] Z. Karimi, A. Baraani-Dastjerdi, N. Ghasem-Aghaee, and S. Wagner, "Links between the personalities, styles and performance in computer programming," *Journal of Systems and Software*, vol. 111, pp. 228–241, 2016.

[17] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[18] J. Bentley, *Programming Pearls, 2/E*. Pearson Education India, 2000.

[19] Quora, "What skills are required to be proficient at coding," *https://www.quora.com/What-skills-are-required-to-be-proficient-at-coding*, 2017.

[20] Stackechange, "As a student, how should programming language familiarity be described on a cv/resume," *http://workplace.stackexchange.com/questions/1209/*, 2017.

[21] IGS, "Insigma Global Service," *http://www.insigmaservice.com/*, 2017.

[22] Hengtian, "Hengtian," *http://www.hengtiansoft.com/*, 2017.

[23] "How practitioners perceive coding proficiency: Appendix," *https://goo.gl/56fZNj*, 2018.

[24] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[25] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[26] J. C. Carver, O. Dieste, N. A. Kraft, D. Lo, and T. Zimmermann, "How practitioners perceive the relevance of esem research," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 56.

[27] D. Lo, N. Nagappan, and T. Zimmermann, "How practitioners perceive the relevance of software engineering research," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 415–425.

[28] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann, "Quantifying developers' adoption of security tools," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 260–271.

[29] E. K. Smith, C. Bird, and T. Zimmermann, "Build it yourself!: home-grown tools in a large software company," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 369–379.

[30] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, 2015.

[31] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 108–119.

[32] R. A. Fisher, "On the interpretation of $\chi 2$ from contingency tables, and the calculation of p," *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. 87–94, 1922.

[33] R. A. Armstrong, "When to use the bonferroni correction," *Ophthalmic and Physiological Optics*, vol. 34, no. 5, pp. 502–508, 2014.

[34] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An Empirical Comparison of Model Validation Techniques for Defect Prediction Model," *IEEE Transactions on Software Engineering (TSE)*, 2016.

[35] D. Lo, X. Xia *et al.*, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 127–138.

[36] H. Cleve and A. Zeller, "Locating causes of program failures," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 342–351.

[37] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[38] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[39] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[40] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 392–402.

[41] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 403–414.

[42] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia, and C. Sant'Anna, "On the effectiveness of concern metrics to detect code smells: An empirical study," in *International Conference on Advanced Information Systems Engineering*. Springer, 2014, pp. 656–671.

[43] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[44] T. Ho-Quang, R. Hebig, G. Robles, M. R. Chaudron, and M. A. Fernandez, "Practices and perceptions of uml use in open source projects," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 203–212.

[45] M. Petre, "Uml in practice," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 722–731.

[46] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine, "Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults," *Empirical Software Engineering*, vol. 21, no. 3, pp. 896–931, 2016.

[47] P. Wendorff, "Assessment of design patterns during software reengineering: Lessons learned from a large commercial project," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. IEEE, 2001, pp. 77–84.

[48] G. Scanniello, C. Gravino, M. Risi, G. Tortora, and G. Dodero, "Documenting design-pattern instances: a family of experiments on source-code comprehensibility," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, p. 14, 2015.

[49] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1–11.