What Makes a Great Software Engineer

Paul Luo Li

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Amy J. Ko, Chair

David Hendry

Andrew Begel (Microsoft)

Charlotte P. Lee (GSR)

Program Authorized to Offer Degree:

PhD in Information Science

University of Washington

Abstract

What Makes a Great Software Engineer

Paul Luo Li

Chair of the Supervisory Committee:
Amy J. Ko Associate Professor
The Information School

Good software engineers are essential to the creation of good software. However, today, we lack a holistic, contextual, and real-world understanding of software engineering expertise. In this dissertation, we address this gap by investigating the thesis: "Experts involved in the creation of software view software engineering expertise as holistically encompassing internal personality attributes, attributes regarding engagement with others, in addition to technical capabilities in designing and writing code. Furthermore, the ability to make good decisions (e.g. choosing what software to write and how to write), which has not yet been articulated by previous research studies, is also critically important. The key aspects of being a great software engineer are: writing good code, adjusting behaviors to account for future values and costs, practicing informed decision-making, avoiding making others' jobs harder, and learning continuously." We interview 59 expert Microsoft software engineers to inductively understand what software engineering expertise entailed. We survey 1,926 more expert Microsoft software engineers to understand the relative importance of the 45 attributes of expertise derived from interviews, as well as to understand the influence of context on ratings. Finally, we interview 46 expert non-

software-engineers who have collaborated with software engineers to understand their perspectives. We collectively consider all our data to answer the question: *what makes a great software engineer?*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to thank my committee members for their time and dedication. A special thanks to Dr. Amy J. Ko, my chair and advisor, for believing in me and taking me on. I would also like to give thanks to my committee members, David Hendry, Andrew Begel, and Charlotte Lee for their service and support.

This thesis would not be possible without the generosity of the experts at Microsoft who participated in the studies. Your time and insights are greatly appreciated. Thank you!

DEDICATION

To my family and Sharon, for all their support and love.

A special thanks to my dad, whose 45-year odyssey to complete his own Ph.D. inspired me to not give up on mine.

Chapter 1. INTRODUCTION

> *At the end of the day, to make changes [to software], it still takes a dev, a butt in a seat somewhere to type [Source Depot] Commit.*
>
> *– Partner Dev Manager, Windows*

Good software engineers are essential to the creation of good software. Regardless of the advances in technologies, processes, and tools, it still takes a software engineer—a butt in a seat somewhere—to decide what software to build and how to build it. Consequently, understanding the attributes that entail software engineering expertise is foundational to our world's rapidly growing software ecosystem: companies want to hire and retrain great software engineers, universities want to train great software engineers, and young software engineers want to know what it takes to become great.

With software being ubiquitous today, software engineers are (and will be) well paid and in high demand. Data from the Bureau of Labor Statistics in 2013 (Bureau of Labor Statistics, 2015) indicate that the *median* salary for software developers was $93,350, nearly three times the national average across all occupations. US News & World Report further reports that while salaries for the average U.S. worker showed no gains between 2000 and 2013, software developers' salaries grew an astonishing 26% over the same period, more than health care practitioners, engineers, and other science professionals—7%, 6%, and 5% growth, respectively (Rothwell, 2014). Looking into the future, the demand for software engineers is forecasted to be even higher; employment is expected to grow 22% between 2012 and 2022, more than double the projected average growth across all occupations (Bureau of Labor Statistics, 2015).

Good software engineers are critical to companies, nations, and societies. Software engineering's early origin in the US Department of Defense—via the Defense Advanced Research Projects Agency (DARPA)—is well documented. Recent nation-funded software security attacks, both confirmed and rumored (e.g. North Korean attack on Sony (Wikimedia Foundation, 2015)), indicate that availability of good software engineers is of strategic importance. Commercial use of software is pervasive, though, sadly, often highlighted by costly problems (e.g. security breaches (Zetter, 2013) and failures (Bellovin, 2013)). For everyday

consumers, mobile computing devices (e.g. cell phones) and mobile applications (i.e. apps) are already ubiquitous, and software is sieving into more of our everyday activities via the internet of things (Burrus, 2013). Software is likely to grow even more essential to humanity in the future.

Understanding software engineering expertise is a critical undertaking for our increasingly software-dependent society. In order to improve software engineering, avoiding catastrophes of the past (such as the ones mentioned in the previous paragraph) and possible future failures, we need to improve the quality of software engineers—the people who produce the software. Letting great software engineers emerge through Darwinian hiring and firing processes is not a viable path, as the demand for great software engineers is likely to far outstrip the supply, as discussed previously). Therefore, we must seek to advance the discipline of software engineering as a whole, determining how to best train and educate software engineer so that every software engineer can be great. To do so effectively, we must first understand software engineering expertise—*what makes a great software engineer*?

1.1 THE KNOWLEDGE GAP

Though numerous studies have touched on the important topic of software engineering expertise, the volume of information belies a dearth of understanding. Today, we lack a holistic, contextual, and real world understanding of software engineering expertise.

By real world understanding, we mean knowledge from experts that have experienced engineering of software in practice. Real-world engineering of software is a complex and multifaceted undertaking, significantly different from software development in academic settings (Begel & Simon, 2008) and ill captured by documentation and artifact repositories (Aranda & Venolia, 2009). As with many complicated undertakings, software engineering is likely a phenomenon that can only be truly understood by doing (Schank, Berman, & Macpherson, 1999); thus, suppositions drawn from indirect and secondary data sources may be incomplete or flawed. For example, one source of information is HR professionals (Bryant, 2013); while HR professionals may be cognizant of declared technical knowledge (e.g. courses and skills on resumes) and general attributes (e.g. learning on the job), they may not understand the nuances of the application of those technical skills in the real world engineering of software (i.e. *how* the

software is engineered). To understand software engineering expertise in practice, we need first-hand information from experts that have experienced engineering of software in practice.

By holistic understanding, we generally mean examining all aspects of software engineering expertise *together*. Most research focuses—implicitly or explicitly—on only a single aspect of software engineering expertise. For example, the ACM curriculum focuses almost exclusively on technical knowledge (Shackelford et al., 2006); conversely, research on human factors in software engineering tend to ignore technical skills and abilities (Cruz, da Silva, & Capretz, 2015). Since software engineering has rarely been examined holistically, we do not know which attributes are important and to what degree—maybe there *is* a single dominant attribute that underlies all of software engineering expertise—or whether there are important attributes that yet to be uncovered.

Another facet of holistic understanding is the perspectives of experts that are involved in the engineering of software but are not software engineers, e.g. artists (Hewner & Guzdial, 2010), designers (Ivory & Hearst, 2001), and program managers (Aranda & Venolia, 2009). Existing accounts indicate that many kinds of expert non-software-engineers work together with software engineers to produce software product. These expert non-software-engineers often perform important tasks, frequently ones that software engineers are ill equipped to perform. However, we know nearly nothing about what these expert non-software-engineers believe are essential attributes of great software engineers.

By contextual understanding, we primarily mean understanding both *what* the attributes entail as well as *why/how* the attributes are important in real world situations. Clear definitions and descriptions are essential for anyone—researchers, educators, managers, young software engineers—seeking to reason about software engineering expertise. Yet, much of the research touching on software engineering expertise fails to clearly define attributes, using vague or general words—if any at all. For example, “checking your ego at the door”, the most important attribute in (Hewner & Guzdial, 2010), was never defined. Beyond clear definitions, existing research also commonly lacks understanding of *how/why* the attributes are important: how does this attribute impact the engineering of software? Why is it important? What benefits does it engender? What problems may arise if one does not have it? To make forward progress in our

understanding of software engineering expertise, we need a contextual understanding of what it entails.

Another facet of contextual understanding is knowledge about how the importance of various attributes of software engineering expertise differs by context. Various research studies *suggest* that amount of experience (Begel & Simon, 2008), gender (Margolis & Fisher, 2003), education background (Carver, Nagappan, & Page, 2008), cultural background (Borchers, 2003), type of software (Hewner & Guzdial, 2010), and the number of engineers working together (Pendharkar & Rodger, 2009) may all impact software engineering practices and perceptions. Yet no prior work has specifically examined how and why those contextual factors affect perceptions of software engineering expertise. Understanding how the importance of the attributes varies is critical for those aiming to articulate the attributes' importance, selecting among them for improvement, or seeking engineers with those attributes for their teams.

The knowledge we have today about software engineering expertise, though directionally sound, has significant gaps. We need a holistic, contextual, and real world understanding of software engineering expertise. Consequently, the questions we seek to answer in this dissertation are:

- What do expert software engineers think are attributes of great software engineers?
- Why do expert software engineers think these attributes are important for the engineering of software?
- How do these attributes relate to each other?
- How do expert software engineers rate the importance of these attributes?
- How are perceptions of importance affected by context of the software engineers?
- What do expert non-software-engineers think are the important attributes of great software engineers?
- Why do expert non-software-engineers think those attributes are important?

1.2 THESIS STATEMENT

In this dissertation, I seek to provide a holistic, contextual, and real world understanding of software engineering expertise. My thesis is:

Experts involved in the creation of software view software engineering expertise as holistically encompassing internal personality attributes, attributes regarding engagement with others, in addition to technical capabilities in designing and writing code. Furthermore, the ability to make good decisions (e.g. choosing what software to write and how to write), which has not yet been articulated by previous research studies, is also critically important. The key aspects of being a great software engineer are: writing good code, adjusting behaviors to account for future values and costs, practicing informed decision-making, avoiding making others' jobs harder, and learning continuously.

1.3 OUTLINE

This dissertation has seven parts. In Chapter 2, we provide the backdrop for our research by describing existing knowledge about software engineering expertise. In Chapter 3, to begin the research arc, we discuss the interview study of expert software engineers, aimed at deriving a holistic list of the attributes of software engineering expertise and understanding their relevance to the engineering of software. Subsequently, in Chapter 4, we discuss the mixed methods study in which we quantitatively surveyed expert software engineers about the importance of the attributes and followed-up with qualitative email interviews to understand the rankings and relationships with contextual factors. With perspectives of expert software engineers in hand, in Chapter 5, we discuss the interview study of expert non-software-engineers about their perceptions. We synthesize findings and insights from all three studies in Chapter 6. Chapter 7 relates our findings to the literature on general human expertise. We conclude in Chapter 8 with a recap of contributions and suggestions for future work.

Chapter 2. RELATED WORK

Numerous research studies and industry reports have directly or indirectly provided knowledge about software engineering expertise. Though directionally sound, prior work lacks in (one or all of) holistic, contextual, and real world understanding of software engineering expertise, as we will detail below. To facilitate understanding of these gaps, we have structured our discussions by the prior work's intent, since the intent of prior work is often the underlying cause of gaps.

2.1 COMPARING NOVICES AND EXPERTS IN WRITING AND MAINTENANCE OF CODE

This section discusses related work comparing how novice and expert software engineers write and maintain code. As a result of their focus, these studies are narrowly focused on a subset of software engineering activities.

Studies that compare novice and experienced developers generally focus on productivity differences, measured in various ways. Sackman et al., in one of the first comparisons of developer productivity in 1968 (Sackman, Erikson, & Grant, 1968)—origins of the 10X developer meme—compared developer performance on coding and debugging tasks. The authors assess differences—along various productivity dimensions—between the best and the worst programmers for an algebra and a maze problem. The differences in debug hours were 28:1 (for the algebra task) and 26:1 (for the maze task), differences in CPU time were 8:1 and 11:1, differences in code hours were 16:1 and 25:1, differences in program size were 6:1 and 5:1, and differences in runtime were 5:1 and 13:1. The authors further noted that the large differences were due to the long tail, with the worst performers taking "as much time or cost as 5, 10, or 20 good ones."

Similarly, (Valett & McGarry, 1988) investigated productivity—based on SLOC/Hour derived from personnel resource forms, interviews, and automated data collections (code repository)—at the Software Engineering Laboratory in 1988. For large projects (>20K lines of code) and small projects, the productivity of the worst, average, and best developers were 0.9, 5.4, and 7.9 for large projects and 0.5, 5.2, and 10.8 for small projects. The also authors

mentioned that most of the developers' time were spent designing (27%) and testing (28%) rather that coding (25%).

In addition to productivity, various studies also suggest that novices and experts differ in mental aspects of program recognition and comprehension. Gugerty and Olson examined differences in debugging between novices (students in their first or second programming class) and expert (graduate students in computer sciences) in 1986 (Gugerty & Olson, 1986). The authors found that experts were more likely to complete the task, complete the tasks faster, introduced fewer new bugs, and formulated and validated hypotheses about the root causes of problems faster.

Along the same lines, Robillard et al. investigated debugging approach of effective developers (recruited within the Computer Science department of the University of British Columbia) using code repository and screen capture tools (Robillard, Coelho, Murphy, & Society, 2004). They observed that successful developers were more methodical in their investigations, better comprehended constructs spanning multiple modules, better recognized relevant information, had a plan for making changes, rarely reinvestigated issues, and used structured searches. However, as we note about other studies in Section 2.4, behaviors of students may not reflect those of industry experts.

Coding is central to software engineering; findings in these studies suggest that various aspects of coding may be relevant to software engineering expertise. However, coding is only one of many activities that software engineers perform; other technical abilities may be more important (as we discuss in the next section). Furthermore, these studies examine coding in isolation. Since software engineering is collaborative, there may be non-coding activities (e.g. seeking help from experts) that directly affect coding effectiveness, but are not considered in these studies.

2.2 SOFTWARE ENGINEERING CURRICULA

This section discusses related work on software engineering curricula. The intent of these works is to prescribe knowledge and abilities that software engineering graduates should possess. Information in these works is also foundational to understanding software engineering expertise,

and distinguishes between software engineering and 'traditional' engineering as well as differences between different branches of computing (e.g. IS, IT, and software engineering).

The foremost work in this area is the ACM Computing Curricula (Shackelford et al., 2006), which stipulates areas of knowledge that graduates need in various areas of computing, including software engineering. The ACM Computing Curricula Joint Task Force, an ACM interest group that meets at technical conferences (e.g. the Conference on Software Engineering Education and Training), developed a set of core knowledge areas within computer sciences (Joint Task Force on Computing Curricula, 2014). For each knowledge area, the Task Force provides—through internal deliberation and external review—a minimum and maximum level of mastery needed for different concentrations in computing: computer science, information systems (IS), software engineering, computer engineering, and information technology (IT). The knowledge areas with the highest combined minimum and maximum levels (scale of 1-5) for software engineering are:

- Programming Fundamentals - Fundamental concepts of procedural programming (including data types, control structures, functions, arrays, files, and the mechanics of running, testing, and debugging) and object-oriented programming (including objects, classes, inheritance, and polymorphism). (Min: 5, Max: 5)

- Software Modeling and Analysis – An activity that attempts to model customer requirements and constraints with the objective of understanding what the customer actually needs and thus defining the actual problem to be solved with software. (Min: 4, Max: 5)

- Software Design - An activity that translates the requirements model into a more detailed model that represents a software solution which typically includes architectural design specifications and detailed design specifications. Alternatively, in software engineering, the process of defining the software architecture (structure), components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements. (Min: 5, Max: 5)

- Software Verification and Validation - The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements. (Min: 4, Max: 5)

- Project Management – An organizational practice and academic field of study that focuses on the management approaches, organizational structures and processes, and tools and technologies that together lead to the best possible outcomes in work that has been organized as a project. (Min 4: Max 5)

The guideline also distinguishes between software engineering and 'traditional' engineering by noting that the foundation of software engineering is primarily computer science, not natural sciences; the concentration is on abstract/logical entities instead of concrete/physical artifacts. Software maintenance primarily refers to continued development, or evolution, and not to conventional wear and tear (Joint Task Force on Computing Curricula, 2014).

These high-level areas in the Computing Curricula are divided into detailed topics in the Software Engineering (SE) specific curriculum guideline (Joint Task Force on Computing Curricula, 2014). In addition to technical topics, the SE curriculum guide also provided overall guidance about meta-issues. The SE curriculum guide dictates that graduates of SE programs should be able to demonstrate several qualities including:

- Professional Knowledge – show mastery of software engineering knowledge and skills and of the professional standards necessary to begin practice as a software engineer

- Technical Knowledge – demonstrate an understanding of and apply appropriate theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation, verification, and documentation

- Teamwork – work both individually and as part of a team to develop and deliver quality software artifacts

- End-User Awareness – demonstrate an understanding and appreciation of the importance of negotiation, effective work habits, leadership, and good communication with stakeholders in a typical software development environment

- Design Solutions in Context – design appropriate solutions in one or more application domains using software engineering approaches that integrate ethical, social, legal, and economic concerns

- Perform Trade-Offs – reconcile conflicting project objectives, finding acceptable compromises within the limitations of cost, time, knowledge, existing systems, and organizations

- Continuing Professional Development – learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development

However, the SE curriculum guide does not go into detail about what each quality entails or how/why the qualities are important in real world scenarios; instead, it focuses on approaches for instilling these attributes in students. Another relevant place where the SE curriculum provides guidance on "attributes and attitudes that should pervade the curriculum and its delivery" is Guideline 8: "students should be trained in certain personal skills that transcend the subject matter." It goes on to state that this includes: exercising critical judgment, evaluating and challenging perceived wisdom, recognizing their own limitations, communicating effectively, and behaving ethically and professionally. However, the guidance is prescriptive, without contextual understanding of the attributes' importance. The guidance for the 'Communicating Effectively' attribute is an illustrative example:

> ***Communicating effectively:*** *Students should learn to communicate well in all contexts: in writing, when giving presentations, when demonstrating (their own or others') software, and when conducting discussions with others. Students should also build listening, cooperation, and negotiation skills.*

Several research efforts have contributed to or have derived from the ACM Computing Curriculum. One of the largest is Lethbridge's survey (Lethbridge, 1998) of 168 software professionals about the relevance of 57 computer science education topics (9 mathematics, 31

software, 4 engineering, 13 miscellaneous) derived from the ACM Computing Curricula (Shackelford et al., 2006) and topics "taught in most programs." Using a Likert-like scale the author sought to understand how much the respondents learned about the topic in school, their current level of knowledge, the usefulness of the topic in their career, and usefulness of additional learnings. Overall, software topics had the highest rating, though only moderately at 2.8 average rating. The most useful subtopics were "general architecture & design" (4.3 average rating), "data structures" (4.1 average rating), "testing & quality assurance" (3.7 average rating), and "requirements gathering and analysis" (3.7 average rating). The only other subtopic with an average rating above 3.5 was 'technical writing' (3.6 average rating) within miscellaneous topics.

Addressing special needs within gaming, the International Game Developers Association (IGDA) issued their own curriculum framework (International Game Developers Association, 2008), developed through "workshops, panels at conferences and discussions." The curriculum framework prescribes the core topics of critical game studies, games and society, game design, game programming, visual design, audio design, interactive storytelling, game production, and business of gaming. The authors neither ranked these topics nor provided in-depth explanations of their importance.

Technical skills and abilities covered by works related to the software engineering curricula are extensive; however, they leave several knowledge gaps about software engineering expertise. Foremost, the curricula are prescriptive; they lack detail about how the knowledge should be applied and why this knowledge is important in real world settings. For example, for "recognizing their own limitations", the software engineering education guideline specifies that "students should be taught that professionals consult other professionals and that there is great strength in teamwork." However, *how* the engagement should proceed and the benefits of engagements (and drawbacks of its neglect) are not discussed. Similarly, for all the technical areas, contextual understanding about how or why the abilities are important for real world engineering of software is lacking. Second, interpersonal skills, like communicating effectively, are not discussed in detail—if at all—and the attributes do not have a ranking like the technical knowledge areas. Therefore, we do not know how important 'communicating effectively' ranks relative to technical areas like 'software verification and validation'. As studies and reports in

subsequent sections show, we need a holistic understanding, beyond technical skills and abilities, about software engineering expertise.

2.3 Software Development Processes and Methodologies

This section discusses related work on software development processes/methodologies. The intent of these works is to prescribe best practices for real world software engineering teams. By prescribing certain procedures as best practices and examining various aspects of software engineering projects as outcomes, these works suggest that various things may be part of software engineering expertise. However, these works generally focused on activities and outcomes at the team level; there is little direct information about *individual* expertise. The literature on software engineering development processes/methodologies is extensive. While a full review is beyond the scope this dissertation, we examine a few notable examples.

One of best known software development methodologies is Boehm's Spiral software development model, developed at TRW (B. W. Boehm, 1988). This methodology suggests that software engineers should be able to "defer detailed elaboration of low-risk software elements and avoid unnecessary breakage in their design until the high-risk elements of the design are stabilized," use "prototyping as a risk-reduction option at any stage of development, 'accommodate reworks or go-backs to earlier stages as more attractive alternatives are identified or as new risk issues need resolution." A related methodology is the Scrum software development method. As discussed by Rising and Janoff, the Scrum software development method is similar to the Spiral method, "just speeded up" (Rising & Janoff, 2000). Scrum also adheres to the Agile manifesto (Beck et al., 2001) which prescribes 'individual and interactions over processes and tools', 'working software over comprehensive documentation', 'customer collaboration over contract negotiations', and 'respond to change over following a plan'. The Scrum software development method has planned incremental development cycles (i.e. sprints), periodic reassessment, customers in the loop, and active risk management. The incremental addition to the Spiral method is the daily Scrum meeting where progress and problems are discussed. The underlying intent of the Scrum meeting is keeping others informed, getting help for problems, and keeping to a timeline are likely desired attributes of software engineers.

Carnegie Mellon University-Software Engineering Institute's Capability Maturity Model (CMM) is another area of well-known software development methodology research (Herbsleb, Zubrow, Goldenson, Hayes, & Paulk, 1997). CMM implies that various things are part of software engineering expertise. First, CMM suggests knowledge and ability to execute various processes are important, including project management processes, engineering processes and organizational support, product and process quality, and continuous process improvement. The CMM's 'maturity level'—1. Initial, 2. Repeatable, 3. Defined, 4. Managed, and 5. Optimizing—suggests that attributes at levels higher than 1 may be attributes of software engineering expertise. Finally, the indicators of success—product quality, customer satisfaction, productivity, ability to meet schedules, ability to meet budgets, and staff morale—suggest that attributes are also desirable.

In addition to overall software development methodologies, various research examines specific software engineering processes. For example, in the area of design, Beyer and Holtzblatt prescribes understanding the customer's work contexts and work flows (via participatory approaches) in order to effectively design systems that meet their needs (Beyer & Holtzblatt, 1995). Ivory and Hearst in their survey of automated usability evaluation procedures (Ivory & Hearst, 2001) require software engineers to effectively instrument, collect, and analyze data from customer usage.

Another process is negotiation between software engineers. Gobeli et al. (Gobeli, Koenig, & Bechinger, 1998) surveyed 117 managers and team members in 78 software development companies in the Pacific Northwest, ranging from 1-2 people to 300+ people, about conflicts and conflict resolution approaches within their teams. Using self-rated 'overall success', 'customer satisfaction', and 'member satisfaction' as outcomes, the authors found that the conflict resolution styles of 'confronting' (recognizing disagreement exists, and then engaging in collaborative problem solving to reach a solution to which the parties are committed) and 'give and take' (recognizing disagreement exists, and then reaching a compromise solution which the parties can accept) are higher correlated with better outcomes than 'withdrawal' (avoiding the disagreeable party or issue, or denying that any action must be taken now), 'smoothing' (recognizing disagreement exists, but then minimizing the differences while striving for harmony or a superficial solution) and 'forcing' (imposing a solution on one or more of the disagreeing

parties). In fact, ‘forcing’ was *negatively* correlated with positive outcomes. These findings suggest that the ability to execute (or to avoid) conflict resolution processes may be part of software engineering expertise.

Research on software engineering processes/methodologies cover a wide range of topics—both technical and interpersonal—that *teams* need to do or accomplish to be successful. While probably essential for teams/organizations, the topics’ relevance for *individual* expertise is unclear. For instance, for a given software engineer, we do not know whether awareness and willingness to participate in (all or any of) the activities are sufficient or whether excellence is necessary.

2.4 New Graduates and Their First Industry Jobs

This section discusses related work examining new graduates in their first industry jobs. By identifying the problems and needs of new graduates, these works aim to improve training of new software engineers.

Begel and Simon observed and interviewed eight new hires at Microsoft for four weeks over the new hires’ initial two months; Begel and Simon examined their daily tasks and the problems they encountered (Begel & Simon, 2008). The authors observed that new software developers spend a large portion of their time in communication tasks: meetings, seeking awareness of teammates, requesting help, receiving help, helping others, working with others, persuading others, coordinating with others, getting feedback, and finding people. The authors state that simply knowing *what* the new hires did was insufficient; they supplemented observation data with self-recorded video journals and predetermined question prompts that sought to understand the problems the informants faced and their feelings toward those problems. The authors found that novices needed to learn how to contribute value to the team (‘take on tasks that have an impact’), to be ‘movers’ (avoiding uncertainty and lack of self-efficacy), and to collaborate effectively in a ‘large-scale software team setting’. Furthermore, the study provided evidence that university experience (even Masters and Ph.D. programs) are different in nature from real world software development. This suggest possible validity issues for related research (e.g. those in Section 2.1) that examine only students. For example, one of

their informants stated, “[I should] get a lot of experience working on a team project with people… not just some stupid homework assignment that only lasts one week.”

Like Begel and Simon, Hewner and Guzdial interviewed and surveyed managers and artists at one small game company to learn about desired traits in new graduates (Hewner & Guzdial, 2010). The authors interviewed nine employees (a mix of developers, managers, and artists) by asking them, “Say you were going to interview some new college hires and you decided to put together a document about what was important to look for in a new hire. What would be in that document?” From all the responses, the authors selected a subset of 28 qualifications in seven group, and then surveyed 32 additional people (27 developers, 4 managers, and 1 artist):

- Programming
- Optimize
- Design
- Specifications
- People Skills
- Game Industry
- CS Education

Respondents rated the qualifications using a Likert-like scale: ‘Essential, would not hire without good skills in this area’, ‘Very Important, has a large impact on a hiring decision’, ‘Important, has an impact on the hiring decision’, ‘Sometimes useful but not required or evaluated in interviews’, and ‘Not useful’. The group with the highest overall importance was Programming, including:

- ‘Being able to solve algorithmically challenging problems’ (15.6% essential, 53.1% very important)

- ‘Proficiency with the C++ language including basic knowledge of features like templating’ (29.0% essential, 51.6% very important)

- ‘Knowledge about data structures’ (37.5% essential, 43.8% very important)

- ‘Debugging and familiarity with debugging tools’ (15.6% essential, 43.8% very important)

The most important individual qualifications were the ‘ability to work with others and check your ego at the door’ (75.0% essential, 9.4% very important) under People Skills, and ‘writing clean code’ (18.8% essential, 65.6% very important) under Design. While the approach is solid (we use a similar approach of interviews followed by a survey in our research), there are gaps in the understanding results from their focus on new graduates. As an interesting side note, some of the interviewees and survey respondents were ‘artists’, not software engineers (or managers of software engineers). These people were important within the game development organizations, yet there was no attempt to examine how their perspectives differed (or didn’t) from those of software engineers.

In 1995, Turley and Bieman (Turley & Bieman, 1995) compared competencies of exceptional and non-exceptional software engineers at “a Fortune 500 company involved in the design, manufacture, and support of single and multi-user computer systems.” The manager designated ‘exceptional’ (and ‘non-exceptional’) software engineers. The tag was tied closely to years of experience, with exceptional software engineers having an average of 9.05 years compared to 5.00 years for non-exceptional software engineers. The authors derived aspects of competence from interviews with software engineers, and then surveyed 129 managers designated ‘exceptional’ (41) and ‘non-exceptional’ (88) software engineers, asking them to self-rate themselves on those aspects. The authors found that the statistically higher aspects for ‘exceptional’ software engineers were ‘helps others’, ‘proactive role with management’, and ‘maintains big picture view’. For ‘non-exceptional’ software engineers, the statistically higher aspects were ‘seeks help from others’ and ‘willingness to confront others’. The enumerated competences covered a broad set of interesting areas. The approach of inductively deriving a set of attributes based on qualitative interviews and then conducting a qualitative survey based on

the findings, is a good approach (we use the same in this thesis). However, the comparisons may have validity issues due to social desirability biases in self-ratings and questionable managers tagging of 'exceptional' (with no clear indication that the managers had any real world software-engineering experience).

Combining findings in past and more recent studies, Radermacher and Wailia reviewed existing literature on the gaps between industry expectations and abilities of new graduates (Radermacher & Walia, 2013). The authors searched the ACM Digital Library and IEEE publications for relevant papers, selecting 38 that qualified. In addition to filtering for quality and applicability, the authors also scoped to empirical research papers published after 1995. The authors identified 14 deficiencies in four areas:

- Soft skills (oral communication, written communication, leadership, presentation)
- Software engineering practices (design, testing, requirements, software life-cycle)
- Computer science concepts (theoretical CS, data structures, programming, networking)
- Software tools (debuggers, configuration management, development tools, miscellaneous software tools)

The four deficiencies most mentioned were oral communications (11 papers), teamwork (11 papers), project communications (10 papers), and problem solving (10 papers). While covering many topics (and providing a good resource for identifying important related work), the survey had several drawbacks. First, the survey lacked clear descriptions and definitions, with little detail to help understand relevance and importance. The section on 'teamwork'—the second most mentioned topic—illustrated this:

> ***Teamwork*** *was tied with oral communication as the most identified knowledge deficiency. One paper listed the ability to get along with others and to check one's own ego as important aspects of teamwork. Another study also indicated that having experience working as part of a team or a group was important. Scott, et al. indicated that the ability to be work as part of a cross-disciplinary team was necessary in industry.*

The description neither described what teamwork entails (e.g. whether 'getting along with others' and 'check one's ego' fully encompasses teamwork) nor explained *why* it is important

(e.g. how ‘cross-disciplinary team’ impact ‘teamwork’). Second, the survey contained papers from the IT and IS fields. The authors acknowledged that those findings may not be directly applicable to software engineering and that both fields (IS and IT) had their own publications outside of their literature search (i.e. not ACM or IEEE). This appeared to affect at least one attribute, ‘ethics’, for which all supporting information were IS and IT papers.

Using findings from their 2013 survey, Radermacher et al. interviewed 23 managers and hiring professionals from companies that work with North Dakota State University’s capstone project to understand the knowledge deficiencies that prevented graduates from being hired (authors designate as *interview*) and issue/problems commonly encountered by new graduates (authors designate as *job*) (Radermacher, Walia, & Knudson, 2014). The authors used the topics from their literature review. The highest rated topics were:

- ‘Project experience’ mentioned by 13 interviewees for *interview* and 0 interviewees for *job*
- ‘Configuration management’ mentioned by 0 interviewees for *interview* and 12 interviewees for *job*
- ‘Oral communication’ mentioned by 9 interviewees for *interview* and 2 interviewees for *job*
- ‘Problem solving’ mentioned by 8 interviewees for *interview* and 3 interviewees for *job*

The lopsided ratings (high ratings for *interviews* but of no relevance for *job*) raises concerns that some the topic may be specific to novices and may not be relevant for experts. The authors explained some of the overall themes such as knowledge of tools covering ‘configuration management’ and other tools, but did not go into details for others (the topics that encompass lack of understanding of job expectations were not discussed). The authors also compared the results with topics mentioned from their literature survey. However, the use of the taxonomy from previous studies was a problem. Some of the concerns discussed by interviewees clearly spanned many topics (e.g. tools), whereas other concerns were at a much lower lever than topic headings. For example, interviewees felt that some new hires had problems estimating costs,

which the authors put with ‘project experience’; however, project experience is significantly broader than the single topic of cost estimation. The lack of clear definitions of the topics contributed to the confusion. Preconceived structured topic areas was a poor fit for their qualitative approach.

The most obvious drawback of these studies is the focus on novice software engineers. Several authors explicitly state that expert software engineers likely differ from novices, and that problems faced by novices may not be comprehensive or applicable to experts. For example, Begel and Simon excluded a participant from their study due to expertise (Begel & Simon, 2008). “We originally had a ninth subject in the study, whom we removed after one observation. His behaviors and actions during the observation exhibited all the signs of a fully *expert* software engineer, with no signs of hesitation, insecurity, deferment to others’ authority, etc.” The lack of clear definitions and explanation is another issue common among these studies. The problem with vague definitions is also present in other studies, such as (Radermacher et al., 2014) discussed in the previous section. For example, for the most important attribute in the Hewner and Guzdial study (Hewner & Guzdial, 2010)—‘ability work with others and check your ego at the door’—the authors only stated that “participants individually emphasized that too much ego and unwillingness to take advice was against the company culture.” It was neither clear what ‘too much ego’ meant nor why the attribute was important in practice. Finally, we note that in (Hewner & Guzdial, 2010), one of the attributes that was only mentioned ‘in passing’ in their qualitative interviews turned out to be one highest ranked attributes in their subsequent survey. This was a problem for the validity of the Hewner and Guzdial study since the authors included only attributes in the survey that the authors considered highly important or ‘provoked disagreement or interesting discussion’ in interviews; the attributes surveyed was not comprehensive. More broadly, this suggests that studies examining expertise need to consider *all* attributes together (i.e. be holistic), lest important attributes and interplay between attributes be missed.

2.5 Everyday Activities of Software Engineers

This section discusses related works that examine everyday activities of software engineers. Using mostly ethnographic approaches, these studies examine what software engineers *do*. As

the intent of the research is descriptive, it does not provide much understanding about differences between engineers in their execution of these activities, or whether they do them at all.

Ko et al. shadowed 17 Microsoft developers for 90 minutes each to examine the kinds of information that they sought (Ko, DeLine, & Venolia, 2007). The authors grouped the 20 types of information sought into work categories in which they arose: writing code, submitting a change, triaging a bug, reproducing a failure, understanding execution behavior, reasoning about design, and maintaining awareness. The most common sought information were:

- 'What have my coworkers been doing?' (work category: maintaining awareness, sought by 15 participants)
- 'What code caused this program state?' (work category: understanding execution behavior, sought by 11 participants)
- 'How have resources I depend on changed?' (work category: maintaining awareness, sought by 10 participants)

The authors also surveyed 42 different developers about needs for these types of information: 'important to making progress', 'unavailability or difficult to obtain', and 'had questionable accuracy'. For 'important to making progress', the types of information receiving the highest ratings were 'what is the program supposed to do?' (93% agree), 'what code caused this program state?' (90% agree), and 'what does the failure look like?' (88% agree). The study suggests that *having* these information is likely an important as aspect of software engineering expertise; though, the study provides little information about differences in acquisition (e.g. how quickly one acquires this information), which is likely what distinguishes great software engineers.

Similarly, Latoza et al. surveyed 28 and interviewed 13 developers at Microsoft about their activities, tools, and practices (Latoza, Venolia, & DeLine, 2006). The authors found that developers spend similar (median) amounts of times on various activities such as designing, writing, understanding, editing, unit testing, communicating, overhead (e.g. building synchronizing code or checking in changes), other code, and non-code. The authors further

reported several interesting behaviors from their interviews: personal code ownership, team code ownership and the 'moat', (integrating) new team members, and code duplication. The study suggests that these activities are likely aspects of software engineering expertise, but it does not provide much understanding about what distinguishes the execution of great software engineers.

Several older studies also provide knowledge about everyday activities of software engineers. Singer et al. surveyed (6 respondents) and shadowed (9) software engineers to learn about activities and tool use (Singer, Lethbridge, Vinson, & Anquetil, 1997). From the survey, the authors reported that the most common activities were read documentation, look at source, write code, and attend meetings. Similarly, in 1994, Perry et al. asked 13 developers at AT&T to complete time diaries to examine their activities (Perry, Staudenmeyer, & Votta, 1994). Most of the self-reported time was spent coding, followed by support, low-level test, high-level design, and planning/development. The authors further commented that nearly half of the developers' time is occupied by non-coding tasks (though coding was nearly 50% of the developers' time). The authors further investigated the number of times participants interacted with other developers, noting that much of the non-coding tasks involved interacting with others (e.g. providing code reviews). As with previous studies, these older results suggest that performing various activities well are likely aspects of software engineering expertise.

In addition to direct observational studies, numerous works have examined bug/issue reporting/tracking repositories to examine how software engineers reach decisions about what to fix/improve, how to do so, and who should do it (Anvik, Hiew, & Murphy, 2006), (Jeong, Kim, & Zimmermann, 2009), (Podgurski et al., 2003), (Runeson, Alexandersson, & Nyholm, 2007), (Bertram, Voida, Greenberg, & Walker, 2010), (Aranda & Venolia, 2009), and (Ko & Chilana, 2010). As with other research papers in this section, these studies suggest that great software engineers do various activities effectively:

- Finding owners and experts
- Determining/assigning ownership
- Pulling in relevant information (e.g. duplicate or related bugs)

- Assessing whether to address an issue (i.e. triaging)
- Negotiating the best approach for fixing a problem
- Coordinating activities (e.g. with validators/testers)
- Communicating/broadcasting status (e.g. stakeholders and customers)

Studies examining everyday activities of software engineers indicate that they engage in many activities in addition to coding; doing these activities well are likely attributes of expertise. However, much of the knowledge is about *what* software engineers do, with little understanding about *how* the execution of these activities differ between novices and experts. As studies in the next section show that *how* activities are performed may be an important aspect of software engineering expertise.

2.6 HUMAN FACTORS IN SOFTWARE ENGINEERING

This section discusses works that examine human factors in software engineering. These studies provide knowledge that *how* software engineering activities are performed are likely relevant to software engineering expertise.

In 1985, Robert E. Kelley started a 14-year study looking at successful engineers. Originating in Bell Labs, the study included engineers from many companies in many industries—Analog Devices, Air Touch, Shell Oil, and Kimberly Clark (Kelley, 1999a). Kelley started by soliciting qualities of 'star performers' from engineers at Bell Labs. Though the full list of factors is not published, the four categories investigated were:

- *Cognitive factors*, such as higher IQ, logic, reasoning, and creativity
- *Personality factors*, such as self-confidence, ambition, courage, and a feeling of personal control over one's destiny
- *Social factors*, such as interpersonal skills and leadership

- *Work and organizational factors*, such as the worker's relationship with the boss, job satisfaction, and attitudes toward pay and other rewards

Kelley and his team proceeded to measure these factors using 'standard measurement tools available' as well as surveys and interviews; however, they did not find a relationship between measurements and being 'star performers' (based on rating by managers and peers). Subsequently, Kelley postulated that anyone who was hired had the necessary abilities to succeed; therefore, *how* people did their work distinguished the star performers. Kelley. based on his own understanding, advanced nine work strategies (i.e. how to do things) for achieving success, in order of importance:

1. *Initiative* – blazing trails in the organization's white spaces
2. *Networking* – knowing who knows by plugging into the knowledge network
3. *Self-management* – managing your whole life at work
4. *Perspective* – getting the big picture
5. *Followership* – checking your ego at the door to lead in assists
6. *Leadership* – doing the small-L leadership in a big-L world
7. *Teamwork* – getting real about teams
8. *Organizational savvy* – using street smarts in the corporate power zone
9. *Show-and-tell* – persuading the right audience with the right message

Kelley's work provides excellent explanation of the work strategies with contextual understandings of their importance, supported by anecdotes and observations. Nonetheless, there are drawbacks with Kelley's work. First, the study includes non-software-engineers. The ACM distinguishes between software engineering and 'traditional' engineering, as discussed in Section 2.2; it is unclear how these differences affect the lack of relationships with the cognitive, personality, social, and organizational factors, as well as the importance of the nine work

strategies. Second, the study does not consider *technical software engineering* knowledge and abilities (e.g. coding). Understanding about software engineering expertise that does not consider developing code—the central activity in software engineering—is a significant limitation.

Ahmed et al. also examined the 'soft skills' of software engineers, analyzing 500 computer science related job advertisements worldwide (Ahmed, Capretz, & Campbell, 2012). The authors claimed that only nine soft skills were mentioned across all the advertisements:

- Communication skills
- Interpersonal skills
- Analytical and problem-solving skills
- Team-player
- Organizational skills
- Fast learner
- Ability work independently
- Innovative
- Open and adaptable to change

The authors only considered ads that mentioned one of the nine attributes and divided the ads by the position being sought: system analyst, software designer, computer programmer, and software tester. Across all positions, only communication skills (the ability to convey information so that it's well received and understood) was consistently mentioned—a part of requirements for more than 75% of advertisements for all positions. For software programmers, the frequently mentioned soft skills in postings were communication skills (90%), followed by interpersonal skills ("the ability to deal with other people through social communication and interactions under favorable and inauspicious conditions" – 65%) and team players ("someone

who can work effectively in a team environment and contribute toward the desired goal" – 62%). The authors further mentioned several regional differences. North America ads for computer programmers showed a moderate demand for independent workers (can carry out tasks with minimal supervision) while this attribute was in low demand in other regions. Australian market showed low demand for organizational skills (the ability to efficiently manage various tasks and to remain on schedule without wasting resources) while other markets showed moderate demand for this skill. Capretz further mapped the soft skills in the study to Myers-Briggs Type Indictors to conclude that most programmers are introvert, sensing, and thinking types (Capretz, 2003).

In a meta-study of human factors in software engineering, Cruz et al. reviewed 90 studies published between 1970 and 2010 to examine the research of personality (traits that can be assessed objectively using personality tests) in software engineering (Cruz et al., 2015). The authors found that the effect of personality—at least those that were objectively measured—were equivocal. Different research results showed relationships or no relationships between personality and tasks/processes of software engineering, including:

- Paired programming
- Education (mostly academic success)
- Team effectiveness
- Software process allocation (personality types and technical roles)
- Software engineer personality characteristics
- Individual performance
- Team process
- Behavior and preferences
- Leadership effectiveness

One contributing problem may have been inconsistencies in definitions, which the authors did not attempt to reconcile: “we did not investigate consistency among the operational definitions of the constructs used as outcomes in the studies.”

Related work examining human factors in software engineering indicate that software engineering expertise likely involved *how* non-coding activities are executed. However, with the exception of the Kelley study, many of the studies lack clear definition and explanations of these attributes. More problematic is the omission of technical attributes. There are likely interplays between mental and human factors with technical skills, as the studies themselves suggest; therefore, not considering technical attributes leaves an incomplete picture of software engineering expertise.

2.7 Insights From Luminaries and the Press

This section discusses opinions of luminaries as well as articles in the press. As software is becoming increasingly essential to our society, many luminaries have shared their perspectives on software engineering expertise, and the topic is popular in the press. Covering all mentions of software engineering expertise is beyond the scope of this literature review; however, we discuss some notable instances.

In his OOPSLA 2003 editorial “Things They Would Not Teach Me of in College: What Microsoft Developers Learn Later,” Brechner—a director of development training at Microsoft—discussed skills that new graduates are commonly missing when they come to Microsoft and that he’d like see taught in schools (Brechner, 2003). These skills were:

- Design analysis: design and analyze software for strong cohesion, loose coupling, clear focus (simplicity), minimal redundancy, and high testability

- Embracing diversity: write one piece of code that supports users of many nationalities as well as users who can’t see or hear

- Multidisciplinary project teaming: work with a multidisciplinary team to complete a project that satisfies a customer's imprecise list of requirements and expectation of quality, all by a fixed date

- Large-scale development: write an integrated piece of a larger project while other students simultaneously write their pieces and be accountable for every student's ability to read, modify, and debug each other's code

- Quality code that lasts: write code that withstands all forms of use, input, and attack without becoming inoperative, taxing system resources, or exposing user's data, while simultaneously recording errors that unskilled support personnel or users are able to diagnose or report

The set of attributes described by Brechner covers many technical skills and interpersonal skills; they also overlap with topics discussed in research studies at Microsoft (see studies in Section 2.4 and Section 2.5). His editorial is also notable because it is one of the few places where the need for software engineer to work well in a multidisciplinary team—including 'designers, usability engineers, and artists'—is called out explicitly.

Another notable place where software engineering expertise is discussed is in "Code Complete" (McConnell, 2004), one of the best-selling practical guides on programming. McConnell argued that effective developers, in addition to programming, also needed 'personal character':

- Humility: developers who compensate for their own mental fallacies by writing code that's easier for themselves and others to understand and that have fewer errors

- Curiosity: keeping up with changes and seeking ways of doing their job better

- Intellectual honesty: refusing to pretend you're an expert when you're not, readily admitting your mistakes, trying to understand a complier error rather than suppressing the message, clearly understanding your program—not compiling it to see if it works,

providing realistic status reports, and providing realistic schedule estimates and holding your ground when management asks you to adjust them

- Communication and cooperation: writing comprehensible code with the audience of people, not machines, in mind

- Creativity and discipline: be creative in the right places; follow disciplined practices and convention to avoid wasting time

- Laziness: some are bad (e.g. deferring unpleasant tasks, unnecessary tasks to look busy); some are good (e.g. writing a tool to do an unpleasant task)

McConnell further deems several attributes that work in other areas of life but do not work well in software development:

- Persistence: when one approach doesn't work try something else or come back to it later; no pigheadedness

- Experience: it's not the *years* of experience, but the amount of deeply reflected experience that matters

- Gonzo programming: pulling all-nighters lead to more time later to fix the bugs introduced

Finally, the author comments that 'personal character' often requires years for habit to build. Though anecdotal, McConnell clearly described various mental attributes and articulated their importance to the engineering of software; the chapters were some of the best at providing a contextual understanding of software engineering expertise. Furthermore, his opinions suggested that dissention among expert software engineers about the attributes of software engineering expertise was possible. Some attributes that he thought 'do not work well in software development' (e.g. persistence) had been suggested as positive attributes in other studies.

Brooks is another well-known luminary that commented on software engineering expertise in his well-known book *Mythical Man-Month* (Brooks, 1995). He touched upon software engineering expertise in many of his observations.

- "The second-system effect" stated that when designing a second system, software engineers should to be mindful that they are susceptible to over-engineering

- "Progress tracking" stated that software engineers need to be continuously pay attention to meeting small/intermediate milestones, lest small incremental slippage accumulate to significant project delays

- "Conceptual integrity" stated that software engineers should have programs that have conceptual integrity, preferably set by a small number of engineers

- "Project estimation" stated that software engineers should take into account additional difficulty in write *production* code as well as overhead costs (e.g. meetings) in estimating project schedules

- "The surgical team" stated that the best software engineers are 5 to 10 times more productive than mediocre software engineers

- "Communication" stated that the entire team should remain in contact as much as possible to ensure that their mental picture is complete and that all assumptions are correct

Brooks' account provided evidence that software engineering is a complex undertaking and that software engineering expertise—to be able to engineer software well—entails attributes that span many areas (e.g. technical expertise, interactions with teammates, and personality traits), with some non-technical attributes influencing technical decisions (as with 'the second-system' effect).

Various articles from Google indicate that software engineering expertise concerns successful software development organizations other than Microsoft. The New York Times

(Bryant, 2013) interviewed Brock—a vice president of people operations at Google—about the success of their hiring practices. Brock indicated that at Google, GPA and test scores are poor predictors of success. He states that significant learning and growth occur after college and that many skills needed to succeed in industry are not the same ones needed to succeed in school. Furthermore, he comments that real-world software engineering was poorly approximated by academic environments (e.g. real-world software engineering problems often lack specific answers). In a 2009 Google I/O talk, Fitzpatrick and Collins-Sussman gave an "opinionated talk" titled "The Myth of the Genius Programmer" about characteristics of good and poor software engineers based on their 'subjective experience' (Fitzpatrick & Collins-Sussman, 2009). They stated that good programmers did not "go off into your cave" and were not afraid of admitting mistakes; they were open with sharing their code and progress, even if it was not perfect. The authors cited the 'bus factor' (resiliency against the loss of a central person), quality ("many eyes make all bugs shallow"), and constant feedback about working on the right thing as important reasons for being open. The pair further pointed out that communicating with people was critical since software engineering is collaborative (i.e. no one person builds big successful software alone). Good programmers needed to be open to taking feedback, are able to give constructive feedback, and not take non-constructive feedback personally. Good programmers were also not afraid of trying and failing fast, because that was the best way (and often the only way) to learn and to iterate to a solution. The authors comment that it was better to be a 'small fish', because one had more opportunity to learn from better engineers and improve. The pair pointed out that an organizational support is needed for many of these 'good' habits to be possible, such as giving constructive feedback and learning from mistakes. In addition to reinforcing the idea of interplays between non-technical and technical attributes, reports from Google also explicitly call out 'learning' as an important aspect of software engineering expertise. Supplementing studies from Microsoft (see Section 2.4 and Section 2.5), these reports from Google indicate that having good software engineers is a problem that concerns even very successful software development organizations; having the financial resources to hire the best candidates is likely not a viable approach to getting good software engineers.

Taken as a whole the opinions of experts and articles in the press provides the best argument for needing this dissertation. These reports indicate that software engineering expertise

entail interplay among a wide variety of attributes, including technical knowledge, interpersonal skills, and mental disposition. And since the investigations were likely neither comprehensive nor rigorous, they make the case that we need a holistic, contextual, and real-world understanding of software engineering expertise.

2.8 SUMMARY AND DISCUSSION

As shown in the previous sections, related work that provides insights into software engineering expertise is extensive. Previous studies suggest that software engineering expertise is multifaceted, covering technical attributes (involving how software engineers envision, actualize, and maintain code), interpersonal attributes (related to how software engineers engage and collaborate with others), as well as mental attributes (related to how software engineers approach themselves, their work, and their craft).

As various works indicate (Section 2.1: comparing novice and expert developers, Section 2.2: software engineering curricula, Section 2.3: software development process/methodologies), expert software engineers may have a variety of technical knowledge and skills. These likely include the core ability to develop code, as well as other related technical skills like software design/architecture, testing/verification, and the familiarity with a variety of software development tools.

In addition, researchers have found that various 'soft' skills may also be attributes of expertise. Foremost is the ability to work with others; this typically entails communicating and coordinating with other software engineers as indicated by research studies on everyday activities of software engineers (Section 2.5) and in mental and human factors in software engineering (Section 2.6). 'Soft' skills may also include the execution of specific activities that facilitate success at the team level, such as 'scrum daily stand ups' that aim to maintain shared understanding and to monitor schedule slippage as well as 'confronting' and 'give and take' conflict resolution processes that aim to reach satisfactory decisions between team members with disagreements.

Various researchers suggest that the mental disposition of the software engineer may also be important, such as the willingness to perform certain activities and how the activities are

performed. From ‘systematic’ underlying effective software development (discussed in Section 2.1), to ‘being open’ underlying effective collaborations (discussed in Section 2.7), personality traits may underlie many of the externally observable actions of expert software engineers.

Enumerating every likely attribute of software engineering expertise stated, suggested, or insinuated by prior work is beyond the scope of this literature review. We do not suffer from a lack of likely attributes; however, we lack in-depth understanding of the attributes. We summarize the knowledge in related works in Table 2.1, pointing out the types of attributes described, their underlying source data, and examples of how they describe attributes of software engineering expertise; these will highlight the weakness and gaps in our understanding, which we seek to fill with this dissertation.

Table 2.1. Overview of Related Work Discussed

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
Comparing novices and experts in writing and maintenance of code: (Sackman et al., 1968)	Technical	Lab experiments on contrived programs for 12 programmers and 9 ‘trainees’ at the Advanced Research Project Agency of the DoD	‘Debugging was considered finished when the subject's program was able to process, without errors, a standard set of test inputs.’
Comparing novices and experts in writing and maintenance of code: (Valett & McGarry, 1988)	Technical	Developer self-reported data and version control data for 150+ developers at NASA’s Software Engineering Laboratory	‘Although there are obvious problems and objections to using line of code a productivity measure, the SEL used it to at least compute trends in productivity.’

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
Comparing novices and experts in writing and maintenance of code: (Gugerty & Olson, 1986)	Technical	Lab experiments on contrived programs for 10 first/second course CS students and 10 graduate CS students	'Another pattern of behavior that differentiated novices from experts was that novices frequently added bugs to their programs…'
Comparing novices and experts in writing and maintenance of code: (Robillard et al., 2004)	Technical, Mental	Lab experiments on the jEdit open source program for 5 CS students	'The successful subjects created a detailed and complete plan prior to the change whereas the unsuccessful and average subjects did not.'
Software engineering curricula: (Shackelford et al., 2006)	Technical	Opinions of the ACM Computing Curricula Joint Task Force with feedback from educators, academics, and practitioners	'Programming Fundamentals - Fundamental concepts of procedural programming (including data types, control structures, functions, arrays, files, and the mechanics of running, testing, and debugging) and object-oriented programming (including objects, classes, inheritance, and polymorphism).'
Software engineering curricula: (Joint Task Force on Computing Curricula, 2014)	Technical, Interpersonal, Mental	Opinions of the ACM Computing Curricula Joint Task Force with feedback from educators, academics, and practitioners	'Communicating effectively: Students should learn to communicate well in all contexts: in writing, when giving presentations, when demonstrating (their own or others') software, and when conducting discussions with others. Students should also build listening, cooperation, and negotiation skills.'
Software engineering curricula: (Lethbridge, 1998)	Technical	Survey of 168 software practitioners of topics commonly taught in computer science programs and in various CS curricula	'Software – General architecture & design'
Software engineering curricula: (International Game Developers Association, 2008)	Technical	'workshops, panels at conferences and discussions'	'Interactive Storytelling Traditional storytelling and the challenges of interactive narrative. Writers and designers of interactive works need a solid understanding of traditional narrative theory, character development, plot, dialogue, back-story, and world creation, as well as experimental approaches to storytelling in literature, theatre, and film with relevance to games…'

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
Software development processes and methodologies: (B. W. Boehm, 1988)	Technical	Experiences at TRW	'Defer detailed elaboration of low-risk software elements and avoid unnecessary breakage in their design until the high-risk elements of the design are stabilized'
Software development processes and methodologies: (Rising & Janoff, 2000)	Technical, Interpersonal	Experiences at AG Communication Systems	'After each sprint, all project teams meet with all stakeholders, including high-level management, customers, and customer representatives. All new information from the sprint just completed is reported. At this meeting, anything can be changed. Work can be added, eliminated, or reprioritized. Customer input shapes priority-setting activities. Items that are important to the customer have the highest priority.'
Software development processes and methodologies: (Herbsleb et al., 1997)	Technical, Interpersonal, Mental	Capability Maturity Model developed by the Software Engineering Institute	'Optimizing - Continuous process improvement is facilitated by quantitative feedback from the process and from piloting innovative ideas and technologies'
Software development processes and methodologies: (Beyer & Holtzblatt, 1995)	Technical, Mental	Projects at InContext Enterprises, Inc.	'Adjusting Focus. The designer has an idea of the scope of the system he or she might create and the kind of work requiring support... However, the designer's initial focus may be wrong or too limited. The designer may be tempted to dismiss what the customer is saying. Information that is too far out of the designer's expectations just seems wrong… Probe into the details… Probing leads to an expanded understanding of the work.'
Software development processes and methodologies: (Ivory & Hearst, 2001)	Technical	Literature review	'Automation of usability evaluation has several potential advantages over non-automated evaluation, such as the following: reducing the cost of usability evaluation… increasing consistency of the errors uncovered…'
Software development processes and methodologies: (Gobeli et al., 1998)	Interpersonal	Survey of 117 managers and team members in 78 software development companies in the Pacific Northwest	'Confrontation - Recognizing disagreement exists, and then engaging in collaborative problem-solving to reach a solution to which the parties are committed.'

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
New graduates and their first industry jobs: (Begel & Simon, 2008)	Interpersonal, Mental	Observations and video diaries of 8 newly hired developers at Microsoft	'Stoppers get stuck easily and give up. Movers experiment, tinker and keep going until a problem is solved. All of our subjects noted the importance of persistence, likely making them movers. Subject W, in particular, noted: "the attitude of not giving up here at MS… if I am given a problem I am expected to solve it. There's no going to my supervisor and saying "I can't figure this out"… Ultimately it's my responsibility.'
New graduates and their first industry jobs: (Hewner & Guzdial, 2010)	Technical, Interpersonal, Mental	Interviews with 9 employees and then survey of 32 additional people at one small game company about desired traits in new hires	'Of the people skills mentioned in interviews, the skill that was consistently ranked highest was the ability to work on a team without excessive ego… Participants individually emphasized that too much ego and unwillingness to take advice was against the company culture.'
New graduates and their first industry jobs: (Turley & Bieman, 1995)	Technical, Interpersonal, Mental	Interviews with 20 'exceptional' and 'non-exceptional' employees and then survey of 129 additional more	'Team Oriented – Definition: I value the synergy of group efforts and invest the effort required to create group solutions even at the expense of my individual results."
New graduates and their first industry jobs: (Radermacher & Walia, 2013)	Technical, Interpersonal	Literature review of studies on the gaps between industry expectations and abilities of new hires	'Teamwork was tied with oral communication as the most identified knowledge deficiency. One paper listed the ability to get along with others and to check one's own ego as important aspects of teamwork. Another study also indicated that having experience working as part of a team or a group was important. Scott, et al. indicated that the ability to be work as part of a cross-disciplinary team was necessary in industry'
New graduates and their first industry jobs: (Radermacher et al., 2014)	Technical, Interpersonal	Interviews of 23 managers and hiring professionals that work with North Dakota State University's capstone projects about deficiency of new graduates	'A lack of understanding of job expectations was the second most common problem that recent graduates were reported to experience on the job. One interviewee indicated that their new employees often seemed to be afraid of asking questions so as not to appear foolish and that they needed to understand that it was better to ask for help than to be stuck on something for extended periods of time… Multiple participants also indicated that recently graduated students occasionally lacked their professionalism. Some examples of this included dressing inappropriately or texting on their phones during meetings.'

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
Everyday activities of software engineers: (Ko et al., 2007)	Technical, Interpersonal	Observations of 17 developers at Microsoft	'Developers worked to keep track of hardware, people and information needed for their tasks:… What have my co-workers been doing? … Developers tracked their time and others', checking their calendars, glancing at schedules and asking their managers about priorities. Managers communicated to their developers about upcoming changes in informal meetings, email announcements, or planning meetings'
Everyday activities of software engineers: (Latoza et al., 2006)	Technical, Interpersonal	Interviews with 13 developers and then surveys of 28 more at Microsoft	'Developers reported spending a little less than half of their time (49% ± 39%) fixing bugs, 36% (± 37%) writing new features, and the rest (15% ± 21%) making code more maintainable. This confirmed our expectation that most developers spend much of their time fixing bugs. But the vast variability in these numbers also demonstrates that typical development activity varies greatly across teams and across the lifecycle.'
Everyday activities of software engineers: (Singer et al., 1997)	Technical, Interpersonal	Surveyed and shadowed ~ 13 employees maintaining a 'large communications system'	'Search - Using Grep, in-house search tools, or searching in an editor'
Everyday activities of software engineers: (Perry et al., 1994)	Technical, Interpersonal	Time diaries of 13 developers at AT&T	'There was much unplanned interaction with colleagues: requests to informally review code, questions about a particular tool, or general problem-solving and debriefing sessions.'
Everyday activities of software engineers: (Anvik et al., 2006), (Jeong et al., 2009), (Podgurski et al., 2003), (Runeson et al., 2007), (Bertram et al., 2010), (Aranda & Venolia, 2009), and (Ko & Chilana, 2010)	Technical, Interpersonal	Bug/Issue tracking systems for various organizations, e.g. Microsoft, Eclipse, Mozilla	'Probing for expertise - Sending emails to one or few people, not through the "shotgun" method, in the hope that they will either have the expertise to assist with a problem or can redirect to somebody that will.'

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
Human factors in software engineering: (Kelley, 1999a)	Interpersonal, Mental	Original research used standardized tests, direct observation, work diaries, focus groups, and individual interviews on 1,000+ engineers from Bell Labs, 3M, and HP. Subsequent studies at other companies	'Initiative – blazing trails in the organization's white space'
Human factors in software engineering: (Ahmed et al., 2012)	Interpersonal, Mental	Survey of job postings from online portals	'Communication skills—the ability to convey in-formation so that it's well received and understood'
Human factors in software engineering: (Cruz et al., 2015)	Interpersonal, Mental	Literature review	'Three studies found evidence of the influence of a team's personality composition on overall team performance. [One study] showed that teams with predominantly introverted members experience lower effectiveness due to communication problems'
Insights from luminaries and the press: (Brechner, 2003)	Technical, Interpersonal, Mental	Personal experiences at Microsoft	'Commercial software involves more than algorithms. It involves the user interface, online help, button and dialog box labels, money and business plans, marketing, and of course, unpredictable users. No programmer will be good at all these things and good at writing solid code. Programmers need to work with others: designers, usability engineers, and artists for the user interface, writers for online help and user interface text, planners for thinking through money issues, marketers for selling the result, and employees who think like users—testers.'
Insights from luminaries and the press: (McConnell, 2004)	Technical, Interpersonal, Mental	Personal experience at various companies, including Microsoft	'Communications and Cooperation: Truly excellent programmers learn how to work and play well with others. Writing readable code is part of being a team player. The computer probably reads your program as often as other people do, but it's a lot better at reading poor code than people are. As a readability guideline, keep the person who has to modify your code in mind. Programming is communicating with another programmer first, communication with the computer second.'

Area and Publication	*Type(s) of attributes examined*	*Source data*	*Example attribute description*
Insights from luminaries and the press: (Brooks, 1995)	Technical, Interpersonal, Mental	Personal experience on the IBM OS/360 project	'Communication – "Schedule disaster, functional misfit, and system bugs all arise because the left hand doesn't know what the right hand is doing." Team drift apart in assumptions'
Insights from luminaries and the press: (Bryant, 2013)	Mental	Experiences as a hiring manager at Google	'After two or three years, your ability to perform at Google is completely unrelated to how you performed when you were in school, because the skills you required in college are very different… You learn and grow, you think about things differently.'
Insights from luminaries and the press: (Fitzpatrick & Collins-Sussman, 2009)	Technical, Interpersonal, Mental	Experiences as developers at Google	'Well, it's hard to admit that you've made mistakes sometimes, especially publicly, right?... Why is this a problem? Why should I care about this? The primary reason is it inhibits progress and not just personal progress, but project progress, okay? It's sort of the "many eyes make all bugs shallow" quote. But if everyone's off working in a cave and just occasionally throwing code out to the project, code quality remains low, and your bus factor remains low.'

Despite numerous studies that touch on the topic, we are not aware of a single research work that provides *holistic*, *contextual*, and *real-world* understanding of software engineering expertise as we have defined in Section 1. Related works that compare how novice and expert software engineers write and maintain code generally examine only technical skills. Software engineering curricula prescribe knowledge and abilities that software engineering graduates should possess; they provide little understanding about how and why these knowledge and abilities are important in real-world engineering of software. Related works on software development processes/methodologies primarily aim to prescribe best practices for the engineering of software at the team/organization level; there is little direct information about individual expertise. Studies examining problems/needs of new graduates in their first industry jobs focus on new software engineers; while some needs of novices may overlap with needs of experts, others may not. Studies about everyday activities of software engineers generally lack information about differences in execution of those activities (i.e. what does it mean to be better or worse at those activity). Works that examine mental and human factors in software engineering generally do not consider these 'soft' factors along-side technical abilities, leaving an incomplete picture of expertise.

The insights provided by luminaries indicate the way forward in this area. Their discussions of software engineering expertise are generally holistic, incorporating technical, interpersonal, and mental attributes. Their insights are supported by real-world examples and explanations. Luminaries were neither rigorous nor complete in their investigations (nor were they aiming to be); nonetheless, the kind of understanding they provide is what we seek to provide in this dissertation.

Another gap in prior work is the perspectives of expert non-software-engineers. Today, the engineering of software is often an interdisciplinary undertaking involving expert non-software-engineers. Research works like Hewner and Guzdial (Hewner & Guzdial, 2010) which mentions technical artists, Aranda And Venolia (Aranda & Venolia, 2009) which mentions program managers, and Brechner (Brechner, 2003) which explicitly calls out software engineers needing to work with 'designers, usability engineers, and artists' indicate that non-software-engineers play important roles in the engineering of software. However, as can be seen from the 'source data' column in Table 2.1, prior work prioritizes perspectives of software engineers. No prior work has directly examined the perspectives of non-software-engineers.

Finally, various studies indicate that context may affect the (perceived) importance various aspects of expertise, including experience (Begel & Simon, 2008), gender (Margolis & Fisher, 2003), education background (Carver et al., 2008), cultural background (Borchers, 2003), type of software (Hewner & Guzdial, 2010), and the number of engineers working together (Pendharkar & Rodger, 2009). For example, Ahmed et al. (Ahmed et al., 2012) find that North Americans tend to value the ability to work independently more than employers in Australia, Asian, and the EU. However, no prior work has systematically examined the effects of context on software engineering expertise.

Therefore, we set out to provide a *holistic, contextual, and real-world understanding of software engineering expertise*—including perspectives of expert non-software and the effects of context—to further the growth and development of the software engineering field.

Chapter 3. INTERVIEW STUDY OF EXPERT SOFTWARE ENGINEERS

As Chapter 2 indicated, prior research has examined—explicitly and implicitly—many attributes that may be related to software engineering expertise. However, many attributes have unclear descriptions (or are without explanations), making it difficult to disambiguate and to reason about the attributes. For example, the most important attribute desired of software engineers in Hewner and Guzdial's paper (Hewner & Guzdial, 2010)—'ability work with others and check your ego at the door'—is explained as "participants individually emphasized that too much ego and unwillingness to take advice was against the company culture." However, this definition neither clarifies what "too much ego" means nor why the attribute is important in practice. This causes problems for those attempting to reason about the topic, such as in the literature review by Radermacher and Walia (Radermacher & Walia, 2013) that cites the Hewner and Guzdial study:

> *One paper [the Hewner and Guzdail study] listed the ability to get along with others and to check one's own ego as important aspects of teamwork. Another study also indicated that having experience working as part of a team or a group was important. Scott, et al. indicated that the ability to be work as part of a cross-disciplinary team was necessary in industry.*

The description above does not clarify whether 'getting along with others' and 'check one's ego' fully encompasses teamwork; it does not discuss *why* it is important, and it introduces new concepts—'cross-disciplinary team'—without any explanation. Unclear definitions and incomplete explanations leave us with unclear understanding.

In addition, since various research efforts do not specifically aim to contribute knowledge about software engineering expertise, their definitions and explanations are problematic for those seeking to understand great software engineers. For example, research into the software engineering curricula, discussed in Section 2.2., focuses on *prescribing* technical knowledge and skills that software engineering graduates should have and how they should be taught; those works contain little explanation about why those abilities are important in real-world engineering of software and how/when to apply those skills/abilities in practice. Research into software

engineering processes/methodologies is another problematic area, as much of their focus is on the team as a collective, rather than individual expertise. The description and explanation of the need to collaborate and work with customers for the Scrum software development method (Rising & Janoff, 2000) is one example:

> *After each sprint, all project teams meet with all stakeholders, including high-level management, customers, and customer representatives. All new information from the sprint just completed is reported. At this meeting, anything can be changed. Work can be added, eliminated, or reprioritized. Customer input shapes priority-setting activities. Items that are important to the customer have the highest priority.*

Teams probably should undertake this task, but it is unclear whether *everyone* in the team needs to; furthermore, it is unclear whether software engineers need to be the team member do the task.

To compound the problem of murky definitions and explanations, many studies focus on a single aspect of expertise. For example, Cruz et al. 's (Cruz et al., 2015) survey paper on 40 years of research on personality in software engineering focuses solely on personality 'traits that are assessed objectively using personality tests'. Most of the studies examining the differences between novice and expert developers, discussed in Section 2.1, focus on their technical ability in writing and maintaining code. Since software engineering expertise has rarely been examined holistically, we do not actually know whether there are other important attributes of great software engineers that have not been uncovered or whether there is actually a single attribute that underlies all facets of software engineering expertise.

A holistic understanding of software engineering expertise as well as clear definitions and explanations are foundational; they need to be examined first. In order to have meaningful conversations about software engineering expertise, we need to be able to describe it clearly. This understanding is critical for determining the different kinds of attributes that software engineering expertise entails, for assessing whether novel attributes unexplored by prior research exist, and for future work in ranking attributes and understanding the effects of context. As the first step in understanding software engineering expertise, we set out to understand:

- What do expert software engineers think are attributes of great software engineers?

- Why do expert software engineers think these attributes are important to the engineering of software?

- How do these attributes relate to each other?

3.1 CONTEXT FOR INVESTIGATIONS

For our investigations, we sought understanding from experts at Microsoft. Rather than a problematic limitation, examining Microsoft employees offered unique advantages. Rather than a monolithic company, Microsoft is a conglomerate of diverse products, cultures, and settings. These include game (e.g. serious: Halo and Forza, as well as casual: Wordament and Minesweeper), consumer electronics (e.g. Surface, Xbox, and HoloLens), OS (e.g. Windows and Windows Server), productivity (e.g. Office), search (e.g. Bing), consumer services (e.g. OneDrive), enterprise services (e.g. Azure), ERP/CRM (e.g. Dynamics), databases (e.g. SQL), developer tools (e.g. Visual Studio), and communications (e.g. Skype), as well as regional-specific development centers around the world. Microsoft also employs a wide variety of expert non-software-engineers to produce products such as artists, content developers, data scientists, design researchers, designers, electrical engineers, mechanical engineers, product planners, program managers, and service engineers (discussed more in Chapter 6). This rich diversity of contexts and perspectives benefits the external validity of this dissertation. Furthermore, Microsoft consistently utilizes best practices and technologies, as well as employs top talent; this helps to factor out confounding deficiencies. Finally, Microsoft employees share common understandings (e.g. seniority based on titles and commonly used acronyms); this helps consistent interpretation of our questions as well as analysis of informant responses. Therefore, while there are other software development organizations our study could have utilized (some potentially more interesting, discussed in Section 8.1), Microsoft is a good setting for contributing rigorous and credible knowledge for this dissertation.

3.2 METHOD

We chose face-to-face semi-structured interviews in order to holistically understand attributes that entail software engineering expertise, with detailed and contextualized explanations of their

meaning and importance. The semi-structured interview format allowed us to ask follow-up questions, going in-depth on areas of interest, and getting clarity on vague concepts.

A key decision in our method was determining whose subjective opinions of software engineering expertise could be considered valid (i.e. who are expert software engineers). We took the ACM's perspective (Shackelford et al., 2006): *people who write software to be used in earnest by others*. We operationalized this definition using the Microsoft company directory, to which I had access as a Microsoft employee. We identified software engineers based on titles that entailed 'software development', e.g. software development engineer, director of engineering, as well as several titles known to the authors as those of software engineers, e.g. architect, technical fellow, and distinguished engineer. We further used the approach utilized by researchers of human expertise (Ericsson, Krampe, & Tesch-romer, 1993), basing our definition of expertise on people having achieved some degree of recognition as software engineering experts. We selected engineers at or above the Software Development Engineer Level 2 (SDEII) title. These engineers were confirmed as experts by other engineers via the hiring or promotion processes.

We obtained a stratified random sample of software engineers across two important dimensions: *product type* (10 major divisions at Microsoft plus one for all others including Skype, Data Center Ops, and Distribution) and *experience level* ('experienced'—titles at or above SDE2—and 'very experienced'—titles at or above Senior Dev Manager, typically with 15+ years of experience). We randomly sampled software engineers in the 22 strata in a round-robin fashion, aiming for at least two informants in each stratum. Of the 152 engineers we contacted, we interviewed 59 (39%). For reporting purpose, we further anonymized the divisions using 'product type'. The number of engineers in each strata (division and feature area) and their titles are in Table 3.2.

Each semi-structured interview was about 1 hour in duration. We started by describing our study, explaining how we located the interviewee, asking permission to record the interview, informing them that all personally identifiable information will be removed, and detailing their rights to refuse to answer any question and to have their responses removed later. We began the interview with the following statement: "I want to start by learning a bit more about you. What

Table 3.2. Stratified random sample of expert software engineers at Microsoft

Division	*Product type*	*Experienced titles: SDE II, Senior SDE, Senior Dev Lead*	*Very Experienced titles: Architect, Technical Fellow, Partner Dev Manager, Partner Dev Lead, Principal Dev Lead, Senior Dev Manager, or Principal SDE*	*Totals*
Ad Platform	Web Applications	2	3	5
Bing	Web Applications	2	3	5
Corp Dev	IT	2	3	5
Dynamics	Enterprise	2	2	4
Office	Applications	2	3	5
Phone	Devices	3	2	5
Server & Tools	Enterprise	3	2	5
Windows	Windows	6	5	11
Windows Services	Web Applications	3	2	5
Xbox	Gaming	2	3	5
Other	Various	2	2	4
Totals	-	29	30	59

software products, at Microsoft and elsewhere, have you worked on?" This helped to establish rapport and facilitated reflections; this prior history was later removed during transcriptions to preserve anonymity. We then asked, "Think back to someone you've worked with that you thought was a great software engineer. What were some attributes that made the person 'great' in your mind?" We asked follow-up and clarification questions for attributes that we thought were interesting (e.g. novel, vague, or counter to prior informants).

In the second part of the interview, we asked about attributes that either lacked clarity or that we thought might vary in interpretation. As we learned more about the attributes from interviewees, we updated the set of attributes we inquired about (once every ~10 interviews). For time considerations, we limited our discussions to five attributes of interest. We closed the interview by restating the purpose of the research and asking interviewees whether they had anything else to add.

To analyze the more than 60 hours of interviews and 388,000 words of transcripts, we used an inductive approach, making multiple passes through the data. We began with open coding, reading through all the transcripts to identify and assess all excerpts that discussed attributes of great software engineers, as well as to get an overall sense of the data. Next, we produced an initial set of attributes and groupings based on my preliminary understanding. On a

second pass through the transcript, we labeled each of the excerpts with one of the attributes or created new attributes to capture the sentiment in the excerpt. Once we developed this initial set of attributes and groupings, we made a selective coding pass through the data, consolidating the attribute set. We then solicited feedback from my advisor Amy Ko. Based on her feedback, we adjusted the descriptions of the attributes as well as to their level of granularity. To validate our interpretations, we then solicited the help of a Senior Software Development Engineer at Microsoft (one of the interviewees who was interested in the study) to analyze roughly 1/3 of the interviews, developing her own attributes and groupings. We then met multiple times to consolidate the set of attributes, often going to the source excerpt to solve differences. The process was complete when the senior software engineer and we agreed on the set of attributes that covered all the excerpts. We then conferred with Amy Ko to develop the model that provided organization and structure to the attributes (see the next section). Finally, we made a last pass through all transcripts, ensuring that all insights and sentiments were captured. The entire process took ~3 months from January to May of 2013.

3.3 RESULTS

Our analysis identified a diverse set of 54 attributes of great software engineers. At a high level, our informants described great software engineers as people who are passionate about their jobs and are continuously improving, who develop and maintain practical decision-making models based on theory and experience, who grow their capability to produce software that are elegant, creative, and anticipate needs, who evaluate tradeoffs at multiple levels of abstraction, from low-level technical details to big-picture strategies, and whom teammates trust and enjoy working with.

We present a model of the 54 attributes in Figure 3.1, showing how the attributes interconnect. We organized the attributes into four areas: *internal* attributes of the software engineer's personality and ability to make effective decisions, as well as *external* attributes of the impact that great software engineers have on people and products.

By decision-making, we mean 'rational decision-making', as described in a paper by Simon (Simon, 1955), as recognizing decisions to be made, identifying alternative courses of

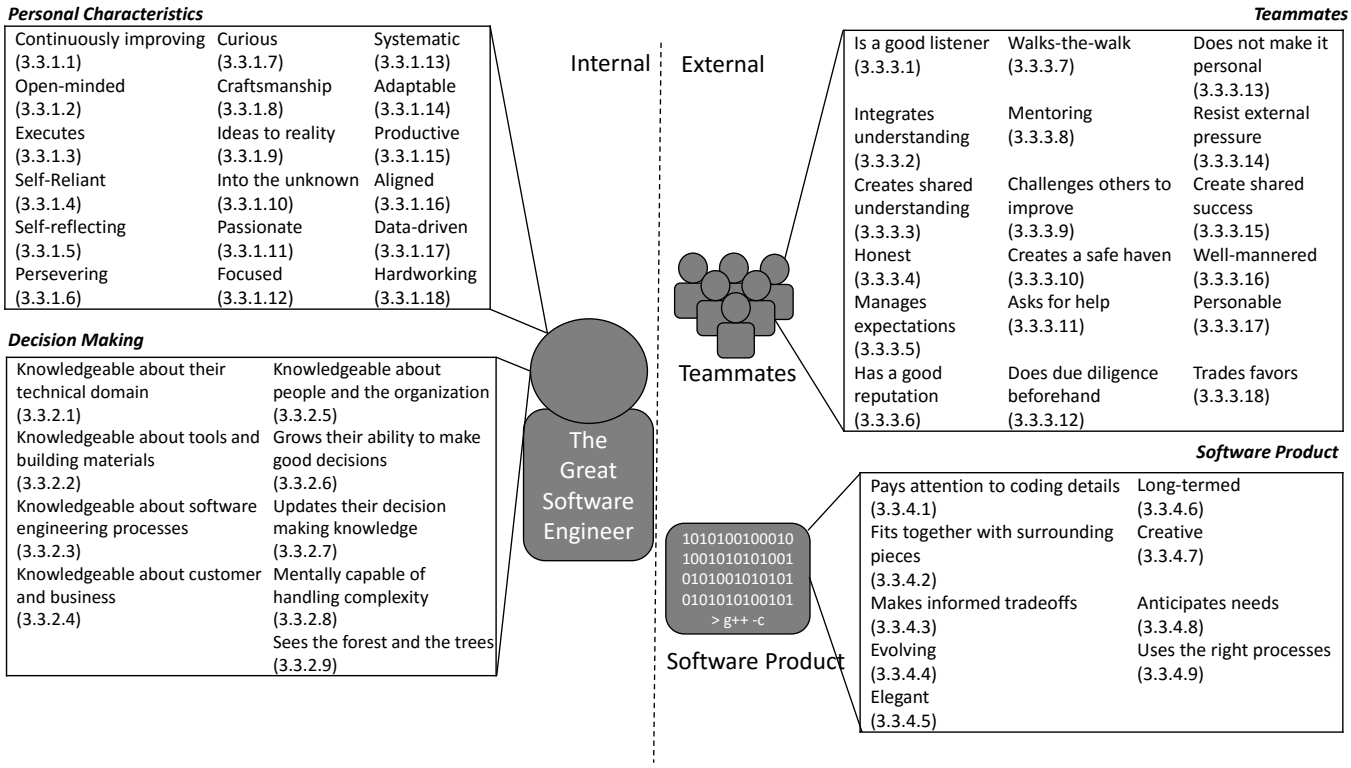


Figure 3.1. Model of attributes of great software engineers

action, assessing likely outcomes, and evaluating values of outcomes. We discuss decision-making in more detail in Section 3.3.2.

While informants generally discussed attributes of software engineers that they admired and liked, many lamented about detrimental and dysfunctional attributes of bad engineers. In an attempt to identify what makes a great engineer, this dissertation will not emphasize what makes a poor engineer." we decided to frame all of the attributes in the positive. Nevertheless, for some attributes (e.g. the well-mannered attribute described in Section 3.3.3.16), informants' sentiment was to avoid a trait—being an 'asshole'—that would inhibit a software engineer from being considered great.

While many of the attributes are applicable to many professions (some simply to being a 'good person'), our objective was to identify the attributes that expert software engineers viewed as relevant; more importantly, we aimed to provide contextualized definitions and explanations of why these attributes were important in real-world engineering of software. In the subsequent sections, we provide a description of each attribute, reasons why our informants thought it

important, and supporting quotations (including informants' title and division when this information would not reveal their identity) that capture the sentiment in interviews.

3.3.1 *Personality*

> *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, Windows*

Informants mentioned 18 attributes that we felt pertained to software engineers' personalities. With attributes like passionate and curious, these concerned who great software engineers were as people. Informants felt that many of the attributes were intrinsic to the engineer—formed through their upbringing—and would be difficult (if not impossible) to change.

3.3.1.1 Continuously improving

> *… Always looking to do something better, always looking for the next thing, studying about the newer thing… [Great software engineers will] study different articles and research papers on software development and stuff. So they're more up to date on newer technologies and newer ideas and thoughts of software architecture or software engineering in general… they are essentially continuing their education and continuing to look, to do things better, is a really big plus.*
>
> *– Senior Dev Lead, Gaming*

Many informants described great software engineers as continuously improving: constantly looking to become better, improving themselves, their product, or their surroundings. Informants felt that great software engineers desired to improve things for which they felt ownership, moving it to a state that they felt was better:

> *He was not the kind of person that would keep doing things the same way even if other people thought it was fine. He was always looking to improve.*
>
> *-Software Architect, division removed to preserve anonymity*

Generally, informants felt that continuously improving was important for two reasons. First, informants recognized that engineers did not start their careers being great; young software

engineers needed to learn and improve in order to become great. Second, informants felt that because the software field was rapidly changing and evolving, unless software engineers kept learning, they would not continue to be great. This notion of running up an infinite escalator was prevalent among our informants:

> *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 keep investigate, investment. [sic]*
>
> *– SDE2, IT*

The need to continue one's learning is closely related to 'continuing professional development' discussed in the ACM Software Engineering Curricula (Joint Task Force on Computing Curricula, 2014). Graduates are expected to continue their education even after attaining their software engineering degrees: "learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development." This edict is in the code of ethics for many professions (e.g. medicine (AMA, 2001) and 'traditional' engineering (NSPE, 2007)) and appears to be a fundamental aspect of most learned professions.

3.3.1.2 Open-minded

> *…the problem is sort of in a way the inverse of sharing, which is people not being willing to take the input of others… That I see as a big problem. You've heard of NIH – not invented here. That's a huge problem… It comes from this unwillingness to accept what other engineers are eager and willing to share.*
>
> *– Principal Dev Lead, Applications*

Most informants described great software engineers as open-minded: willing to let new information change their thinking. Informants felt that great software engineers, even if they were the experts in their area, were open to changing their thinking based on new information presented to them. Frequently, informants discussed this attribute negatively, describing some software engineers who would dismiss ideas and technologies that they did not conceive, also known as the 'not invented here' mentality. Great software engineers were not reported to be conceited about their knowledge and did not believe that they knew everything.

Informants felt that *not* being open-minded lead to suboptimal decisions, commonly in two ways. First, informants felt that outcomes in software engineering, such as user reactions and commercial success, were difficult to predict. Therefore, great software engineers needed to be open to letting real-world data change their thinking:

> *You should be open… what you think need not be the right thing tomorrow… like the Facebook explosion, when Myspace was already there, but it exploded… no one knew that Facebook would explode when it started.*
>
> *– Senior SDE, Web Applications*

Informants also felt that many software products were large, complex (e.g. extensive use of layering and abstractions), and constantly changing; therefore, it was rare for any one person to have a complete understanding of the software product and of all the implications of design choices. Therefore, even experts needed to be open to changing their understanding when provided with new information:

> *No matter how much you know, the software industry is so large… there's so many other areas… If that person has something to say that hadn't occurred to me, I'll stop everything and say, ok, explain this. What did you see, that I didn't see?*
>
> *– Senior SDE, Applications*

3.3.1.3 Executes; no analysis paralysis

> *"[Great software engineers] should not be just idealistic software designers where you can think you can do a lot of, they should not get into analysis paralysis… write the most optimal solution for the problem on hand.*
>
> *– SDE2, Devices*

Several informants described great software engineers as knowing when to execute, not having analysis paralysis: knowing when to stop thinking and to start doing. By 'analysis paralysis', informants meant taking too much time to think about alternatives or over optimizing the solution. Many informants felt that many things in software engineering, such as the variability of alternative technologies, could not be known ahead of time. Furthermore, most projects had hard deadlines. Therefore, a saturation point existed where additional thinking and debate was detrimental to the success of the software product:

…you have to not be so thorough that you don't get anything done because you're spending all your time analyzing, or researching, or prototyping, or whatever you do, you'll never deliver anything.

– SDE2, IT

Informants felt that great software engineers understood that they existed to ship products to customers in a timely manner. The product might not be successful if engineers spent too much time thinking about the problem rather than implementing the solution. The overwhelming sentiment was that 'perfect should not be the enemy of the good':

"[Some engineers who are not great] like to go very deep in the problem. For them, problem solving is the goal actually. They don't care as much about shipping. They will go for the last one percent improvement also. Then you'll be like, "there's no business value. It's 90% accurate, I'm good" They're like, "no, no I can make it 96%."… a different skill set to be successful there compared to successful here.

–Senior Dev Lead, Web Applications

Microsoft, as a for-profit company, likely influenced informants' perspectives about this attribute. Business terms like 'time to market' were commonly used by our informants in discussions. While many software development organizations are like Microsoft, software engineering research indicates that many 'open-source' software projects have a primary goal other than making money like Mozilla (Ko & Chilana, 2010) and Linux (Raymond, 2001). Whether and how this attribute manifests in the 'open source' contexts may be an interesting area for future research.

3.3.1.4 Self-reliant

Rather than looking around for somebody to solve it for them... try to figure out how they can do this on their own… get yourself unblocked attitude works really well in this company."

– Principal SDE, Windows

Informants commonly described great software engineers as self-reliant: getting things done independently (i.e. not needing to go to their lead/manager for help constantly) and removing roadblocks by leveraging their abilities and resources (e.g. asking other experts for help). Great software engineers were not expected to know everything; rather, they were expected to have the initiative and ability to seek out answers independently in order to deliver on their objectives.

For many informants, being self-reliant was a minimum requirement for working at Microsoft. Even new software engineers were expected to be able to make forward progress on complex and novel problems with limited guidance:

> *I think that engineers go through this growing up phase, but there's a key milestone where they realize that they actually don't need anyone else's help… you just need to figure out yourself… You have to be more independent.*
>
> *– SDE2, Enterprise*

However, several informants lamented that some engineers, though technically capable with high seniority, lacked the ability to reach objectives by themselves. Our informants felt that reliance on managers and leads for day-to-day guidance prevented these engineers from being considered great:

> *…there's sort of a base differentiator. I would call it effectiveness. I work with a lot of people…super smart, they have the skill sets, they're just not effective… They lack self-confidence. They come to you and ask you questions all the time and you work with them all the time and you say, just make a decision and do this on your own, you're level 63 [a senior level engineer]… you need to be able to do this on your own.*
>
> *– Principal Dev Manager, Web Applications*

This attribute closely mirrors the 'movers' attribute—avoiding uncertainty and lack of self-efficacy—discussed in Begel and Simon's paper (Begel & Simon, 2008). One of the rudimentary attributes that new hires needed in order to be effective at an organization was self-reliance. Building on the Begel et al. study, which focused on new hires, our findings indicate that the 'ability to make independent progress' may not be so basic after all; it may be an issue for new and experienced software engineers alike.

3.3.1.5 Self-reflecting

> *... a little bit of an intuition and maybe the ability to see where you're going wrong and step back so self-reflection a little bit maybe is important, being able to recognize, yeah, this ain't working, I better start over.*
>
> *– Principal Dev Lead, Gaming*

Several informants described great software engineers as self-reflecting: able to recognize when things are going wrong or when the current plan is not working, and then self-initiate corrective

actions. This attribute is likely a manifestation of the concept of metacognitive awareness in cognitive psychology (Schraw, 1998), where people have (or can learn) the ability to self-monitor and self-regulate. While it was not clear what triggered the recognition (e.g. checklist or intuition), informants felt that great software engineers were able to self-initiate corrective actions in order to avoid dead ends (i.e. wasted effort), failures to deliver, and/or bugs that harm users:

> *It turned out most of the accidents… you learned that the engineer who wrote that code, didn't have the right level of training and understanding to write it… they did something that was textbook but it didn't really apply to what they were doing.*
>
> *– Software Architect, division removed to preserve anonymity*

Informants felt that great software engineers needed to be self-reflecting because they were commonly the people who were the most knowledgeable about the area and the situation. Therefore, they were best able to recognize when the current direction or strategy was untenable:

> *[Great software engineers] should have a sense of where you should be... I understand that if it's a two-week project, I really know that by four or five days and I need to be at this point and if I'm not there, I need to make adjustments. It's surprising of how many people don't necessarily recognize that.*
>
> *–Principal Dev Lead, Enterprise*

3.3.1.6 Persevering

> *Ultimately I will never give up. I will live here day and night to make sure it happens… definitely intelligence is required but the people continuously keep hearing that, 'okay, I won't give up. I will try to find out a solution.' Those people always succeed.*
>
> *– Senior Dev Lead, Enterprise*

Many informants described great software engineers as persevering: not dissuaded by setbacks and failures; they kept on going, kept on trying. Their confidence and belief was bolstered by previous experiences overcoming setbacks to achieve success:

> *It's quite often that you face a problem, you look at it and say, 'I have no idea how to do this. This is too big.'… If you easily give up, then you will end up giving up pretty much every hard problem you touch.*
>
> *– Partner Dev Manager, IT*

Informants felt that perseverance was important because software engineers constantly encountered difficult problems during real-world engineering of software, such as seemingly impossible objectives, difficult bugs, and dead-end investigations. Therefore, it took perseverant software engineers to overcome problems and to successfully deliver software products:

> *Most coders don't know what they need to know or actually… don't know how to do something right away. There's a lot of learning on the job, right. There's a lot of figuring out, a lot of you know, doing the search for how to do this, how to do that. Following through and knowing how to do those things is very important for a coder. Like not just giving up right away and looking for someone else saying we should change our objective…*
>
> *–SDE2, Enterprise*

A side benefit of the perseverant attribute was that it often created positive feedback loops. Several informants discussed perseverant engineers—successfully delivering software products despite setbacks and failures—were often given subsequent interesting and challenging assignments by their managers because they proved their perseverance. These opportunities enabled great software engineers to grow their skills and knowledge quickly, as well as gain recognition and promotions faster:

> *They always press in to find the issue; even they facing the hardship they will aggressively to find a way to fix the issue. Maybe they get [interesting] assignments the way they handle the issue.*
>
> *– Senior SDE, Devices*

Interestingly, being persistent was explicitly called out in a paper (McConnell, 2004) as possibly being *detrimental* in software engineering. In McConnell's opinion, "Most of the time, persistence in software development is pigheadedness—it has little value. Persistence when you're stuck on a piece of new code is hardly ever a virtue." This attribute is one of many for which dissenting opinions exist; the fact that expert software engineers can have differing opinions is a motivation for our subsequent studies.

3.3.1.7 Curious

> *…the best people naturally are not satisfied until they've really figured out the problem…The best ones they just have this thing and then they just want it by themselves until they've figured it out.*
>
> *– Principal Dev Manager, Web Applications*

Many informants described great software engineers as curious: desiring to know why things happen and how things work (e.g. how the code and the context interact to produce a software behavior). Informants felt that great software engineers desired to deeply understanding how products worked end-to-end (typically their own or competitors), not satisfied with superficial 'black box' knowledge:

> *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, Gaming*

For our informants, being curious was important for three reasons. First, it motivated great software engineers to gain a more thorough understanding of their technical domains, which enabled them to derive better solutions and to make better decisions. Second, knowing the important parts of the product (i.e. where the essential difficulties lie), meant those areas received the appropriate attention during development. Third, figuring out the nuanced side effects of various actions enabled great software engineers to avoid problems when designing or coding:

> *You're doing step by step a really hard problem and then you're always curious, what's next now… when you're writing a code, you have indirectly debugging in your mind, 'Okay, I'm writing this line, this will happen, now this is going to happen, now this is going to happen, now this is going to happen.'*
>
> *– Senior Dev Lead, Enterprise*

3.3.1.8 Craftsmanship

> *Really being able to demonstrate something that you've done, that you're really proud of, and speak to it well. When you do your work that you take pride in the fact that it's quality work.*
>
> *–Principal Dev Lead, Gaming*

Several informants described great software engineers as having craftsmanship: taking pride in oneself and one's product, letting their output be a reflection of their skills and abilities. Informants believed that most software engineers knew the difference between doing something and doing something right. Great software engineers with craftsmanship did not cut corners and did things the right way:

> *It's nice to know how it works and all that kind of stuff, but actually making yourself do it that way is a task in itself. The discipline is really important to be paired with the process and all that kind of stuff, so yeah, discipline is key.*
>
> *–Principal Dev Lead, Gaming*

Informants commonly discussed two reasons for craftsmanship being important. First, even with numerous quality assurances processes in place, to test/validate all scenarios was often difficult. Unless the software engineer 'did things the right way', the shipped product would have many problems. This might have been especially important at Microsoft where, historically, other people—testers—were responsible for verifying that the code was correct and met specifications. Therefore, informants felt that great software engineers did not merely do minimum work, 'throwing it over the wall' for the tester to find the problems:

> *…in that attention to detail that willingness to, also the introspection packet to really be able to say, "Oh gee, I may not have accounted for this. Let make sure I account for that." and, "Oh, gee, this might not work here. Let me make sure I account for that. And really following through. Whereas, there's others who are just like, "Let me just do the minimum to be able to say I'm done with it and move on.*
>
> *–Principal Dev Lead, Applications*

Second, informants felt that software engineers with craftsmanship did "not stop caring once the code was checked in", extending their stewardship of their code to deployment and maintenance. Issues after deployment were common and having software engineers that

remained engaged with the product—who did not simply 'check out' or move onto the next thing—helped the long-term success of the product:

> *'I think seeing things through to the end' is like once you build something you don't immediately check out, you're not gone. It's still your baby, you still need to kind of get it walking, get it running. As people consume it they're going to find bugs and you just need to be there to fix them quick and keep people happy.*
>
> *–SDE2, Web Applications*

Overall, there was a feeling of respect for software engineers with craftsmanship among our informants. They felt that software engineering was often hard and tedious, sometimes needing to iterate many times to account for edge cases and special conditions. Consequently, software engineers were often tempted to cut corners. Our informants felt that great software engineers consistently resisted these temptations and always did things right. They took pride in doing something to the best of their abilities:

> *…willingness…to say, 'Oh gee, I may not have accounted for this. Let make sure I account for that.' and, 'Oh, gee, this might not work here. Let me make sure I account for that.' And really following through. Whereas, there's others who are just like, 'Let me just do the minimum to be able to say I'm done with it and move on.*
>
> *–Principal Dev Lead, Applications*

3.3.1.9 Desires to turn ideas in to reality

> *They feel more accomplished at the end of the day if they've actually built something whether it was with their hands, or maybe they drew something, maybe they designed something, maybe they wrote some code. I think you have to have that…. personality trait.*
>
> *–Senior SDE, Windows*

Several informants implied that great software engineers desire to turn ideas into reality: takes pleasure in building, constructing, and creating software. This can be an entire software product, a feature within a product, or even a solution to a hard problem. Informants felt that great software engineers felt joy in bringing something into existence that did not exist before.

We inferred the importance of desiring to turn ideas in to reality from our interviews. The sentiment was that software engineers with this attribute would bring new things into existence.

Great software engineers often saw potentially new products and new features based on their understanding of the technical domain; yet, it took those with a desire to turn ideas into reality to actualize those features, bring new things into existence that could substantially change the world:

> *[Great software engineers] have a sense of a potential that software has, right?... I think the people that are great are able to grasp a bigger chunk of that potential and sort of turn it into something useful … I think in this field really the limitations are all in your own head. I think there are people who are able to kind of push those limits out a little further and grab a bigger piece of what they think they can do.*
>
> *–Principal Dev Lead, IT*

This attribute overlaps somewhat with other personality attributes, e.g. executes (Section 3.3.1.3) and productive (Section 3.3.1.15); however, we felt that the desire to birth something new into existence was a distinctly separate sentiment. Whereas other attributes *might* lead to creation of new things, none focused directly on desire to 'create' as the motivation:

> *It's more like you have an urge to create. You get satisfaction from creating.*
>
> *–Senior SDE, Windows*

3.3.1.10 Willing to go into the unknown

> *People are just naturally going to gravitate towards their comfort areas and just kind of hang out there…But if you're willing to take those risks and learn about other things and then actually apply them they can help move you forward. But apply them might mean getting out of your comfort zone.*
>
> *– Senior Dev Manager, Windows*

Many informants described great software engineers as willing to go into the unknown: taking informed risks into new areas even though they may not have, at the time, knowledge or expertise (e.g. a new technology). Informants felt that it was important for software engineers to overcome inertia: try new things, gain new knowledge, and push the boundaries of their domain:

> *[Great software engineers] are willing to take the risk to try to make the product successful... if we don't do it, we won't improve our selves, if we stay wherever we have we actually just never change, never bring the new stuff to the whole company*
>
> *–Senior SDE, Devices*

Informants felt that willing to go into the unknown was important for two reasons. First, in order for a software engineer to produce a successful software product, commonly entailing 'differentiator' that distinguished it from competitors, the software engineer often needed to push the technological envelope:

> *…being bold enough to take the risk of making some mistakes… explore some new ideas or some new technologies that's not foolproof yet... So, if you have shut that door right from the beginning, and I've seen many people like that, that's not going to yield good results.*
>
> *– Senior Dev Lead, Enterprise*

Second, great software engineers commonly needed broad holistic understandings of their domain; this often required them to branch out into new areas. Willingness to go into the unknown enabled engineers to gain new knowledge and perspectives, understandings different ways of 'doing things':

> *…at Microsoft, they say, "You have to move every two releases within Microsoft." They encourage people to move, so they can broaden their knowledge and learn a lot of stuff, rather than being stuck in one spot. Those might be patterns, inertia.*
>
> *– Senior Dev Manager, Windows*

3.3.1.11 Passionate

> *[Great software engineers] are usually very interested in the area they're in. They like it. They would probably play with that even if they weren't getting paid for it. The best engineers don't see it as a job, they see it as a hobby and they just like doing the work… I don't think I've ever known a really good coder who hated the feature he was in… I firmly believe every coder who hated what they're doing some other developer paid the price later.*
>
> *– Principal SDE, Windows*

Informants described great software engineers as passionate: intrinsically interested in the area they are working in, and not just doing it for extrinsic rewards such as money. Informants felt that great software engineers did not simply view engineering software as their job, rather it was their passion; great software engineers would do what they did, even if they were not paid. Informants discussed various aspects of the software product as being potentially interesting,

from the software product itself (e.g. Xbox), to attributes of the software product (e.g. aesthetics, security, or performance), to the technology area (e.g. mobile computing or big data):

> *…knowing what people are passionate about, knowing what people are not passionate about; it's hard but it's really key to people's long term health, and desire to actually produce results… Some people love security, for instance, other people hate security. If you give somebody who hates security a security function, they're just going to not perform well, regardless.*
>
> *– Senior Dev Manager, Windows*

As indicated in the quotes above, most informants felt that software engineers will not succeed if there is a mismatch between their interests and their assigned task, and the software product will ultimately suffer. Great software engineers needed to find project and assignments that matched their passion.

> *I think that there are people who are great software engineers who are in the wrong place and aren't motivated and they end up not performing well.*
>
> *– Principal Dev Lead, Enterprise*

While this raised the possibility of task/assignments that no one wants (and consequently bad software), there was an underlying sentiment among our informants that no matter the subject matter, there would be someone with a natural affinity towards it, such that they would want to work on it:

> *I found that there's always a person who's passionate about every type of thing, you just have to find the right people… I ended up in the wrong job for six months. It was painful. People around me, they loved their work*
>
> *– Principal Dev Lead, Devices*

3.3.1.12 Focused

> *In an environment like Microsoft where there's a lot of meetings and interruptions… [this great software engineer] just figured out that when he can get away from the chaos of the day-to-day, he could come back and make very good use of that time.*
>
> *–Principal Dev Lead, Web Applications*

Several informants described great software engineers as focused: allocating and prioritizing their time for the most impactful work, not overwhelmed by daily distractions and tasks.

Informants felt that software engineers experienced many distractions daily (e.g. meetings, IMs, and emails) and were assigned many tasks. Great software engineers were able to focus on the most important tasks, often structuring their days to have sufficient time to complete priority items:

> *I think the other thing is focus. At Microsoft we have priorities every day. Everybody going to be working on different issues and different priorities… It's easy to get lost by the work… It can be always busy, but do you make the choice of the right priority? That's the challenge.*
>
> *–SDE2, Devices*

Our informants did not discuss *avoiding* disruptions, as many were resigned to interruptions and meetings—where the team aligned understandings and shared information activities—being painful but necessary parts of large scale software development:

> *There's some simpler things just in terms of raw speed and focus. In an environment like Microsoft where there's a lot of meetings and interruptions, I think it takes … A developer has to kind of figure out how to get their focus and when to get their focus.*
>
> *–Principal Dev Lead, Web Applications*

Most informants viewed the focused attribute as a software engineer's ability to deal with such disruptions. Informants commonly discussed the attribute as a mental attribute, where great software engineers were intrinsically more effective at switching quickly between contexts and recovering quickly to their previous tasks; nonetheless, in several instances, informants also discussed great software engineers devising processes of dealing with disruptions, e.g. making prioritized lists, coming in early before others arrive for uninterrupted time, and blocking time out on the calendar to focus on high priority items.

The underlying issue associated with this attribute, interruptions, is a rich research topic within software engineering. Many researchers have sought to understand the nature of varying kinds of interruptions (Dabbish, Mark, & Gonzalez, 2011), their impacts on various tasks (Czerwinski, Horvitz, & Wilhite, 2004), and approach for mitigating their negative effect (Iqbal & Horvitz, 2007). In our study, our informants largely ignored the nature and impact of interruptions, instead focusing on the software engineer's ability to make progress despite the existence of interruptions.

3.3.1.13 Systematic

> *You have to be patient and not rush to the solution. You have to go through a mental gymnastics in order to get to a solution.*
>
> *– Principal SDE Lead, Windows*

Several informants described great software engineers as systematic: not rushing or jumping to conclusions, addressing problems in a systematic and organized manner. Informants felt that great software engineers took actions in logical and ordered steps, carefully reasoning about the unbounded and complex nature of software. They decomposed problem into manageable pieces of investigation to be investigated in an orderly manner:

> *They are fairly quickly able to break any arbitrary problem down into its components… help shape the solution… it's the fully and accurate picture of the problem and understanding where the boundaries are, and the pieces are.*
>
> *– Principal Dev Manager, Web Applications*

Without being systematic, informants felt that engineers were prone to waste time and resources on fruitless investigations and strategies. Informants felt that it was common for software engineers to have an initially wrong hypothesis about the situation; therefore, great software engineers needed to be "thoughtful" and "not immediately try to project this ideas about what it might be". Great software engineers systematically approached problems to avoid "chasing down blind alleys". This was true for design tasks, but also particularly true for debugging:

> *If you're given a very humongous amount of code and there is a problem, you can't debug each and every line of the pool... step by step. You get to the root of the problem very fast.*
>
> *– Senior Dev Lead, Enterprise*

Software engineers frequently formulate and validate hypothesis about code behavior, especially during maintenance tasks, as reported by many researchers (Ko et al., 2007) (Ko, 2006). Findings about the systematic attribute in this study provide nuanced understanding that *how* the activity is performed distinguishes the *great* software engineers.

3.3.1.14 Adaptable to new settings

> *…things are going to change, what are you going to do about that? Are you going to be one of the people that are helping to change? ...everything from values to fit into the group, or the product, or the problem you're trying to solve, and I think that that is important… How are you going to take and adapt your situation to move forward, and how do you adapt to work with what you have to work with?*
>
> *– SDE2, IT*

Many informants described great software engineers as adaptable to new settings: continuing to be of value to the organization even with changes in what they do (e.g. software product and organizational objectives) and how they do it (e.g. people, processes, and technologies). Whereas willing to go into the unknown entailed self-initiated changes, informants felt that changes often occurred outside of the software engineer's control, including changes to the organization, to focus of the software product, to the competitive landscape, as well as to the task assigned to the engineer:

> *…embrace new ideas, new technologies, patterns of doing things, being adaptable to a new team, being able to adapt to a new team and their culture… [great software engineers] need to be adaptable… we need to be adaptable to accommodate change in our lives, especially professional lives.*
>
> *–Senior Dev Lead, Enterprise*

Informants felt great software engineers were able to successfully navigate and adapt to the changes around them. Regardless of the context, the organization could expect positive results from great software engineers. Many informants discussed the ever-evolving nature of software development as a contributing factor to this attribute: "the time changes and good software engineer will adapt to it":

> *Whatever feature you happen to be working on one day you guys may decide that you're heading down path A and this is how the feature is going to work and then all of a sudden you said it's going to run into this problem so we need to switch and go down path B…You do have to be flexible to change because there is a lot of change in the software. It's superfast, growing and changing industry.*
>
> *–Senior SDE, IT*

However, the notion of software engineers as ‘interchangeable parts’ was not shared by all informants. It conflicted with the notion of needing a tight fit between the interests of software engineers and the task they are assigned (passionate, Section 3.3.1.11). Furthermore, some informants felt that certain technical domains required ‘deep expertise’ such that it was rarely practical to move software engineers to/from that area:

> *…our developers tend to stay… It’s a very specialized area, and there’s not any other group within Microsoft to hop around to, so you basically give up 10 years or 20 years of education in order to move to a different group if there’s something completely different*.
>
> *– Principal Dev Manager, Applications*

3.3.1.15 Productive

> *“Some developers can do things very fast. The work takes someone else maybe half a day, [they] can take half the time required.*
>
> *– Senior Dev Lead, Web Applications*

Many informants described great software engineers as productive: achieving the same results as others faster, or taking the same amount of time as others but producing more. As discussed in the related work Section 2.1, productivity–the speed and the number of tasks completed—is often used as a measure of expertise when comparing novice and expert developers. Our informants felt similarly; great software engineers produced code faster:

> *… developer productivity is always an example. Some of the developers that are most highly regarded are the ones that are able to produce more results than others… no one's ever consider the great developer if their productivity isn't great*
>
> *– Principal Dev Manager, Enterprise*

In addition to the obvious business benefit of enabling their software products to reach the market faster, informants also discussed productive engineers enabling their teams to ‘fail fast’. Informants described scenarios in which great software engineers quickly produced a MVP—minimum viable product—to understand and to reason about the product. Since some things in software engineering were difficult to know ahead of time, productive engineers quickly provided information that enabled the organization to make better decisions (sometimes to forgo further investment in the products):

In a start-up, where you've got a deadline to actually secure your next round of funding, and doing so requires that you have the product in certain level of minimal viability. Speed is really of the essence. Being able to rapidly iterate, fail fast, that kind of thing.

– Principal Dev Lead, Web Applications

3.3.1.16 Aligned with organizational goals

A mismatch of value… their number one goal is really to learn and learn … you are paid because we are a business.

– Principal Dev Manager, Web Applications

Several informants described great software engineers as aligned with organizational goals: acting for the good of the product and the organization, not for their own self-interest. Usually discussed in the negative, informants commonly described two forms of misalignment. First, some software engineers focused on an interesting technology rather than customer needs; this commonly led to wasted efforts on software features that did not make the software product more value to customers:

… my job is to provide value to the customers so that they'll buy our product. Writing the coolest, most fun, neatest software solution, in fact often does not provide the best customer value. Sometimes the most boring, mundane, simple, brute force, least cool bit of code is exactly what's going to provide the best customer value. For me having passion about providing customer value is important. More important than writing something cool software.

– Principal Dev Lead, Applications

Second, some engineers neglected less glamourous aspects of system (e.g. usability); this neglect commonly led to poor quality.

In addition to completing their own tasks, great software engineers undertook tasks outside of their responsibility in order to help their software product to be successful, such as writing documentation, answering customer questions, or running tests. Informants felt that having great software engineers that had bought-in to the success of the organization was necessary for successful software products:

> *I will do whatever it takes. You need to run a test pass, I will do the test pass. You need somebody to write some docs, I will go write docs. You need somebody to help with customer support I can do that.*
>
> *– Principal Dev Lead, Gaming*

The attribute is close to the concept of 'signing up', described by Zachary in his account of the creation of Windows NT at Microsoft (Zachary, 1994); software engineers committed to joining a team to work a software product, implicitly indicating that they were willing to do whatever it took to make the software product successful. This concept was also discussed in *Soul of a New Machine*, Kidder's Pulitzer winning account of software development at Data General (Kidder, 2000). The notion of great software engineers committing to delivering something and then doing their best to deliver on their promises appears to be a long-standing unwritten rule within software engineering:

> *He's very dedicated to whatever thing he took on, meaning that if he promised to deliver something, he was going to do his best to deliver that, took pride in delivering what he said he would deliver.*
>
> *–Principal Dev Lead, Applications*

3.3.1.17 Data-driven

> *Look at things in a more scientific way, a more empirical sort of way… do the measurements, [great software engineers] will understand, and they'll try to break down the data… a hypothesis about what I think will make it better, and try the hypothesis and measure again, and look at look at the results.*
>
> *– Principal Dev Manager, Web Applications*

Many informants described great software engineers as data-driven: measuring their software and the outcomes of their decisions, letting actual data drive actions, not depending solely on intuition.

Informants commonly discussed two benefits. First, by creating feedback loops, informants believed that great software engineers used data to confirm or disprove understandings and expectorations, helping them to improve their future decisions:

…data driven and not instinctive driven for most of the time... collect customer data and take some of that into account while you're making the next wave of decisions.

– Senior Dev Lead, Enterprise

In some situations, such as A/B online experiments (Kohavi, Frasca, Crook, Henne, & Longbotham, 2009), informants believed that great software engineers tried various options and then made choices based on actual customer preferences instead of resorting to rhetorical or intuition-driven arguments:

Very iterative… Just try it. We have a hundred online experiments running at any time on users. When people get into debate… the way I look at it is: how do I make the system better so that I can try all these three ideas… it's very experimental.

–Senior Dev Lead, Web Applications

Overall all, informants viewed being data-driven as an effective approach of avoiding confirmation bias, leading to better software products. However, many informants lamented that simply *having data* was no panacea. They stated that software engineers frequently found ways to ignore the data or to discredit the evidence, leading to bad engineering decisions:

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, 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, Applications

3.3.1.18 Hardworking

…Incredible work ethic, like the ideal Microsoft employee, he would just work 12 plus hours a day, just unbelievable. That's proto-typical programmer that we need to hire more of.

– Principal Dev Manage, Applications

Several informants described great software engineers as hardworking: willing to work more than 8 hours days to deliver the product. This typically meant working longer days, during weekends, and/or during other free time in order to accomplish goals. Informants believed that, at a minimum, software engineers needed to be willing to work beyond normal hours immediately prior to ship dates in order for the team to successfully deliver the product:

I remember it came down to the last day. He is going on vacation and we needed to ship and he stayed late and was there all night… even delayed his vacation by a day, so that he could get it done and get it out the door so we could ship on time ... he got high praise for it from the management.

–SDE2, Web Applications

There was a hidden sentiment that engineers were *expected* to be hardworking. This may be inherent to the software engineering profession, reinforced by accounts from Microsoft (Zachary, 1994) and elsewhere. Our informants seemed to accept the fact that software engineering involved significant amounts of mundane time-consuming but necessary tasks:

I have worked in many different companies and worked in different countries, engineering, at least from my experience, it's a time-consuming job, especially schedules generally are tight… There's always issues that come up. There's always a big push, especially towards the end when you have a date for a project to be completed by. You're never where you need to be when you start getting close to that date, so you wind up working extra hours. Unfortunately, there's some people that say, "I'm not going to work extra hours," and I think that hurts them. Sad to say, but, to be honest, I think that may not define you as a good engineer.

– Senor Dev Lead, Gaming

3.3.2 *Decision Making*

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, division removed to preserve anonymity

Informants mentioned 9 attributes that we felt pertained to engineers' ability to make decisions: assess the current context (i.e. understanding when/what decisions were needed), identify alternate courses of action, gauge probabilistic outcomes, and estimate values of outcomes. As indicated in the quotation above—and numerous more in the sections that follow—engineering of software requires many choices of 'what software to build and how to build it'. Many of our informants' descriptions of great software engineers involved these engineers' making optimal decisions under difficult and complex circumstances. Beyond book knowledge, great software engineers understand how decisions play out in real-world conditions. They not only know what should happen, but also what can and likely will happen.

Combining their knowledge, their mental models that tie the knowledge together, and their mental ability to reason about their models, decision-making attributes were internal to the software engineer. We grouped these attributes together because they revolved around the important *mental* process of 'making decisions'. Furthermore, in contrast to many of the personality attributes, the underlying sentiment among our informants was that attributes concerning decision-making could be acquired.

To make optimal decisions, informants discussed great software engineers having three kinds of attributes. First, they needed knowledge of several dimensions—technical domain, customers and business, tools and building materials, and software engineering processes. Second, great software engineers needed to build and maintain decision-making models that link the knowledge together—growing their ability to make good decisions and updating their decision-making knowledge. Finally, great software engineers needed the mental dexterity to use their decision-making models under real-world conditions: mentally handling complexity and seeing the forest and the trees. Great software engineers had complex and multifaceted decision-making models that were continuously updated.

3.3.2.1 Knowledgeable about their technical domain

> *You are working in some of the most complex and intricate code bases there are up there. It takes, look it, for a lot of people, it takes several years to get the point where you can reasonably go in there and do something without doing any harm, right? If you were churning people or just had people in there working willy nilly, it wouldn't help you, right?*
>
> *– Technical Fellow, division removed to preserve anonymity*

Most informants described great software engineers as knowledgeable about their technical domain: thoroughly conversant about their software product, technology area, and competitors. The exact technical knowledge discussed was product and team specific, ranging from distributed computing in Bing, to signal processing in Skype, to encryption in Servers and Tools, and more. Informants often discussed needing domain-specific training, as well as an understanding of the solutions of others (e.g. competitors) in order to have a thorough understanding one's own product.

Informants also believed that thorough understanding included knowing the entire solution, not just a small piece of the system. This might have been especially important with large products involving many interconnected pieces, as choices for those systems are more likely to have side effects. Our informants generally viewed acquisition of this knowledge as a gradual process; starting out, even great software engineers (e.g. when an experienced engineer was transferred to another team) were usually assigned a small piece of the product and then progressively developed a broader and more holistic knowledge of the product:

> *I feel that it's like you should have a very good understanding of the entire system as well as all of the moving parts. Knowing basically, you should have a very good picture, a good big picture of how it's supposed to work… the architects behind big systems, complex systems and know it, all the gotchas, in and out. I really do look up and consider them great because they spend all this time to learn about systems.*
>
> *– Senior SDE, Web Applications*

Informants' discussions of the benefits of being knowledgeable about the technical domain concerned four areas. First, most frequently, informants felt that this attribute enabled great software engineers to avoid actions with negative consequences, i.e. do not 'break something':

> *I have a better understanding of what I'm changing... I'm not going to break something that I'm not getting into… if you had originally written the code, or if you've spent the time to gain a deep architectural understanding of the code, then it's much easier and quicker to make those changes than if you're trying to make an isolated change to something that you don't really understand.*
>
> *– Senior SDE, Windows*

Second, the knowledge enabled great software engineers to focus attention on the most important areas within the software product, commonly parts of the system that were especially error prone:

> *[This great software engineer] had a profound understanding of how the hardware actually worked and was able to just optimize the key critical paths as a result.*
>
> *– Technical Fellow, (division removed to preserve anonymity)*

Third, great software engineers leveraged their knowledge to identify important innovations to improve their products. Our informants felt that the key was discerning between important

advancements—ones worth the investment—versus non-essential changes that were unlikely to yield meaningful gains. The sentiment was that many things were touted as ‘game changers’ but few actually were; great software engineers understood their technical domain and were able to discern important changes:

> *Technology changes a lot. The actual underlying ideas don’t change all that often but the way they get expressed changes. I think being able to keep a firm grasp on that stuff is important. I see people get overwhelmed a lot with the level of detail, so being able to filter out what’s the essential things.*
>
> *– Principal Dev Lead, IT*

Finally, great software engineers knew about solutions and approaches of others (typically competitors), which they would be able to borrow and apply to their own software products. This typically led to better and more successful product:

> *[Great software engineers] are always interested in what new is out there, what they can leverage… The technology you can use, what's available, whether it's from Microsoft, whether it's from somebody else who has created something new and innovative… always looking at what else is out there…*
>
> *–Senior Dev Lead, Windows*

3.3.2.2 Knowledgeable about tools and building materials

> *They understand the why the motivation for, why we have 17 different data structures, a black tree, and this tree, and that tree and what... they really, really have a better ability to make the right choice when choosing from this tool set. Or even understanding, well, you know what? This problem is different in enough ways that’s we’ve got to maybe make a new tool right here but it’s really understanding I think the why.*
>
> *– Principal Dev Manager, Web Applications*

Many informants described great software engineers as knowledgeable about tools and building materials: knowing the strengths and limitations of technologies used to construct their software. However, opinions of what constitutes ‘tools and building materials’ varied greatly among informants; many software products were ‘building materials’ for other software products at higher levels of abstraction. No single piece of technology is universally critical. For example, while the SQL Server was the final product for several informants, the database was a ‘building

material' for the informants in Dynamics that used the database to build their CRM management product:

> *...But what's happened in the industry is the applicability of those skills has been getting less and less, to this point that the teams that rely mostly on validating algorithm and data structure skills, tend to have the least reliability in terms of accurately predicting success as a developer in the group.*
>
> *... Databases apply to so much software at this point, I mean you can't really do an online service, for example, without a database, and using a database, the data structure algorithm, you're dealing with higher level concepts.*
>
> *– Principal Dev Manager, Enterprise*

Informants' opinions about the importance of knowledge about tools and building materials also varied. Some informants felt that since these were things that software engineers used frequently, and that mastery over them was essential. Others felt that information about tools and building materials could easily be looked up; thus, engineers merely needed to be *aware* of the tools and building materials:

> *I've found it less important that you really have the entire core computer science curriculum in your head at any given moment. That's just the understanding of when you see something coming that you haven't touched in a while, you have to go freshen up on it…In practice, I sent out some code once to do a binary search on something or other, and universally the reviewer said, "We don't write that kind of stuff. We rely on standard libraries for that. It's silly that you would write one of those things."*
>
> *– Principal Dev Lead, Web Applications*

Differing opinions aside, informants felt that great software engineers with knowledge of tools and building materials produced code faster, were better at debugging, and had fewer quality issues. Informants felt that the increase in productivity and quality frequently stemmed from "not having to build one's own". Great software engineers effectively leveraged existing well-tested code; the key was knowing *which* technology to use and knowing under which conditions the choices would differ:

> *[This great software engineer's] manipulation of these is very detailed, knowing what to use under what conditions. There's no universal approach to this, so the ability to match the right technology to the right situation, is actually very difficult. To be able to do it effectively is great. It's not something everyone can do.*
>
> *– SDE2, Enterprise*

In addition, knowing the limitations of the underlying technologies also enabled great software engineers to quickly diagnose and resolve unexplained anomalies:

> *If you write in Java, you're probably not going to have to performant code. That's not your fault as a programmer. It's just the constricts you're given in Java because it consumes a lot of memory… Definitely language has a choice of tool makes a big difference in how good of an end project you're going to get.*
>
> *– Senior SDE, Windows*

Many of the 'tools and building materials' (e.g. data structures and algorithms) are elements of technical knowledge needed by software engineers prescribed by the ACM Computing Curricula (Joint Task Force on Computing Curricula, 2014). However, rather than general concepts (e.g. programming languages), the discussions and descriptions provided by our informants were all grounded in detailed knowledge about specific instantiations (e.g. memory consumption for Java).

3.3.2.3 Knowledgeable about software engineering processes

> *[Great software engineers] know how to go about developing software…how to go about software development.*
>
> *– Principal Dev Lead, Web Applications*

Several informants described great software engineers as knowledgeable about software engineering processes: knowing the practices and techniques for building a software product—purposes, how to, costs, and best situations to use the process. Informants felt that there were many ways to engineer software, with differing approaches and necessary adjustments for various situations and contexts. Great software engineers have mastery over the necessary stepwise processes—and their variants—for a team to successfully complete a software product:

> *Clearly one of the difficulties in software is it's so easy to do so many different things in so many different ways, they could all be right, but the amount of effort that it takes to get*

there or the amount of effort it takes to support it later on, really drives the overall experience of what you've done... [This great software engineer] just always struck me as someone who really stood above the rest [for knowing what process to use].

– Senior Dev Manager, Windows

Informants identified three primary benefits of being knowledgeable about software engineering processes: higher quality, more deterministic timelines (i.e. fewer surprises), and efficient allocation of time/resources. Many informants discussed great software engineers using (and enforcing) validation processes/techniques to ensure high quality, such as unit testing, test drive development, and code reviews. Interviewees believed that leveraging these processes led their software to be high quality:

[This great software engineer] had a really really high bar for kind of engineering excellence… he did a test driven development thing where you kind of write the test first and then you know, kind of write the code to match the test… the state of the art was for basically creating the best way of developing software… this one particular component that he worked on had like one bug.

– Principal Dev Manager, Web Applications

In addition to the three primary benefits, several informants also mentioned great software engineers using 'processes' to effectively grow their teams. Great software engineers established well-defined and well-reasoned processes, formalizing the team's common understandings, so that the team effectively grew in size while maintaining coherence and quality:

How many developers you can throw at a project… having good practices around how you do the code reviews and check ins and having unit tests that enforces things don't break and that kind of thing it is way way more important than the actual having a beautiful architecture

– Principal Dev Lead, Web Applications

The knowledge discussed in this section shares similarities with technical knowledge prescribed by the ACM Computing Curricula (Joint Task Force on Computing Curricula, 2014) and topics discussed in software engineering processes and methodologies research (discussed in Section 2.3). 'Software Verification and Validation' and 'Project Management' are key areas of knowledge prescribed by the computing curricula; 'quality' and 'predictability' are key outcomes discussed by processes/methodology research (e.g. CMM (Herbsleb et al., 1997)).

3.3.2.4 Knowledgeable about customers and business

> *…Really understanding the point: who is the customer, why are we doing this. There is an old phrase that says an engineer does for one dollar what any damn fool can do for ten.*
>
> *– Principal Dev Lead, Gaming*

Many informants described great software engineers as knowledgeable about customers and business: understanding the role their software product plays in the lives of their customer and the business proposition that it entails. Some informants saw the purpose of software engineering as benefiting humanity, though most were realistic and pragmatic; the purpose was to make money. Therefore, most informants felt that software engineers needed to understand what their customers needed and how their software filled that need. This understanding enabled software engineers to make software products and services that customers were willing to pay for:

> *[Some software engineers] want to solve really hard problems, [but instead]… understanding your customer, find out what they've got, find out what they already want, what they already do, what's the delta you can provide, how can you help, and then go find a simple solution to it because at the end of the day, we are a for profit company.*
>
> *– Principal Dev Lead, Gaming*

Informants generally discussed three ways in which knowledge about customers and business were important. First, informants felt that great software engineers recognized that they were not the customer; therefore, they used their knowledge to avoid choices that fitted *their* needs but did not work for customers. Second, many informants mentioned needing to 'fill in the blank' during development (i.e. making everyday engineering decisions). Written specifications were often incomplete or out-of-date; therefore, software engineers often needed to exercise their own judgment in making engineering choices. Informants felt that great software engineers effectively used their knowledge about the customer and business objectives to make optimal choices:

> *Great software architects are not religious about the technology, but they're able to understand the technology and then say, "Hey, here's how I think we can solve that business problem better." Through this use of technology and they come out from a perspective that's, "I understand what my customer wants," rather than just being like, "We should just use this technology because it's cool."*
>
> *– Principal Architect, IT*

Third, great software engineers used knowledge about customers and business to appropriately test and validate their software products, ensuring that the software worked for their customer's scenarios:

> *Basically think of all the scenarios to cover, let's say I have some feature that I think I have test cases to cover, how the customer uses it. They should be able to figure out the issues before they go to the customers.*
>
> *– SDE2, Enterprise*

In addition to benefit to customers, several informants talked about benefit to the organization. Informants felt that individual software products often needed to integrate together into broader business solutions or for larger business objectives. Therefore, knowing the overall business intent enabled decisions that fit within the long-term vision of the company:

> *…Requirements that are created by the environment, and I actually believe that understanding those things are as important as good engineering because when you miss them those are the kinds of things that can set you back for years if you don't understand the environment you're in… Within five years of him pushing, the government began to require this and had we not done that work we would have basically lost an incredible amount of sales.*
>
> *– Software Architect, Applications*

The concept of employees having sufficient business knowledge about their company is discussed in *Administrative Behavior* (Simon, 1976), Herbert Simon's seminal work on organizations. The sentiment in Simon's work is the same as those of our informants: employees needed knowledge about the organization's objectives (at the sufficient level) in order to make their own decisions effectively.

3.3.2.5 Knowledgeable about people and the organization

> *The nice thing about some of the companies like Microsoft, there's literally people here who have created a world, the technological world that we live in today. They're stars in that regard. We can learn a lot from people in these companies who have more of the resources of people, I guess. Just trying to tap into this wealth of knowledge that Microsoft brings to the table, the talent pool that's here.*
>
> *– SDE2, Gaming*

Informants described great software engineers as knowledgeable about people and the organization: informed about the people around them—responsibilities, knowledge, and tendencies. Knowing ownership (i.e. areas of responsibility), enabled great software engineers to determine key stakeholders for decisions and to align their work with the appropriate teams:

> *Make sure that you are aware of that big picture, you know where you fit in and how you interact with everyone else to optimize what you are doing.*
>
> *– Principal Dev Lead, Web Applications*

Knowing who had expertise enabled great software engineers to find the right people for help—often domain experts. For software engineers in leadership positions, this knowledge also enabled them to take corrective action to address knowledge gaps within the team (e.g. assigning a more senior person):

> *[This great software engineer] would go through his organization and looked very carefully at the tasks that were being assigned and whether people had the right level of training and understanding and if they didn't, who their supervisor and whether that person did and would demand code reviews...*
>
> *– Software Architect, division removed to preserve anonymity*

Finally, knowing people's tendencies enabled great software engineers to adapt their engagement techniques to obtain desired outcomes:

> *You have to understand people so that you can influence or impact them... You have to do that both down and up and out.*
>
> *– Principal Dev Lead, Devices*

Identification of expertise and assignment of responsibility is frequently discussed in research studies that examine everyday activities of software engineers (discussed in Section 2.5), notably in studies examining bug triage/assignment processes of software engineering teams. Multiple studies found determination of responsibility—bug assignment—as well as expertise as key steps in the bug-triaging process (Anvik et al., 2006) (Aranda & Venolia, 2009). Identifying who has technical knowledge, who has ultimate decision-making power, and the methods for locating that information (e.g. 'bug tossing' (Jeong et al., 2009)) are all important to the bug triaging/assignment process of software engineering teams.

3.3.2.6 Grows their ability to make good decisions

> *There's a way to look at a problem and get a pretty accurate reading on how much work is involved to solve it to a certain level of satisfaction…learning where the hard parts of the problems are probably lurking and what trouble they might cause you or something like that… Maybe having a good pattern of recognition from that standpoint is important too.*
>
> *– Principal Dev Lead, IT*

Our informants' descriptions of great software engineers suggest that they grow their ability to make good decisions: building their understanding of real-world situations including alternatives, outcomes, and values of the outcomes. Great software engineers effectively identified and understood aspects of the context that impacted alternative choices and probable outcomes, which entailed the ability to identify when decisions were needed, available alternative choices (including how to search for options), *probabilistic* outcomes (including things what can go wrong), and the value of the outcomes (including identifying the dimensions of the value vector). The underlying idea was that great software engineers' experiences evolved into predictive models over time; they grew their ability to make good decisions. Our informants rarely used academic terms such as 'models', 'alternatives', 'states', or 'outcomes'; they commonly used terms like knowing 'what to do' or 'what works':

> *…Transition from being driven by intuition versus experience is kind of evaluating… The growth that you're going to experience, it's kind of like the science project, right. When you're operating on intuition, you're soon to be operating on theory about how things should work… Like a real scientist, also set expectations about what the outcome should be and measure those expectations and all that stuff. Kind of reworked that theory until you converge at something that's functional, I guess, working.*
>
> *– SDE2, Gaming*

Our informants felt that by growing their decision-making abilities, great software engineers became progressively better at making decisions, taking actions that were likely to succeed and avoiding actions that were unlikely to work. Great software engineers also became better at preparing for things that could go wrong and put appropriate contingency plans in place:

> *[Great software engineers] can predict, or they can forecast what's the future… And he also can predict, say, what's the challenge in the implementation, implemented into this design. So he can predict how much time you will use, how much developer should be involved is one, and how much tester [sic], and how long to ship it, something like that.*
>
> *– Senior Dev Lead, Web Applications*

The outward manifestation of this attribute is improvements in interactions with teammates and in engineering of their software products (discussed in Section 3.3.3 and 3.3.4). Nonetheless, the sentiment among our informants was that the underlying genesis of those improvements is the *mental* ability of great software engineers to make better decisions over time.

3.3.2.7 Updates their decision-making knowledge

> *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 in this world…*
>
> *– Technical Fellow, division removed to preserve anonymity*

Several informants described great software engineers updated their decision-making knowledge, not allowing their understanding and thinking stagnate. Informants felt that great software engineers evaluated changes in their context and updated their mental models (i.e. how they would make certain decisions), sometimes throwing away obsolete knowledge and building new mental models:

> *…it's a constant improvement and constant evolution of what you're doing by learning how your product is functioning and how it's being used. You then are able to get feedback and put it back into the product.*
>
> *– Principal Dev Lead, Web Applications*

The two areas that most commonly required updating were knowledge about tools and building materials and knowledge about the limitations/restrictions of existing technologies. Informants felt that new and evolving technologies would frequently impact both the available engineering choices as well as the expected outcomes of those decisions. Great software engineers incorporated those evolving circumstance into their decision-making models:

Doctors always need to know about the newest medical treatments, the newest drugs and interaction between the drugs. Lawyers have a similar thing, they always need to keep reading, keep understanding what new precedents have been set in the law journals and stuff. I think the same is true for us. We need to know what problems are solved. I think we are at a point in software development where we have a lot of options for implementations, those kinds of things. If we're not current, you just pick the thing you always work with and it may not be the best tool for the job. I think staying current helps you know what's the best tool for the job.

– SDE2, Web Applications

Informants felt that updating decision-making knowledge was essential for great software engineers to *continue* being great. Software engineers that failed to update their thinking would begin to make suboptimal decisions, losing the respect and confidence of their peers:

…Software engineering is one area where probably it has changed the most if you look at an engineer who started in 2000…Good [engineers] know how to keep learning because this is an area it doesn't matter how smart you are; things just change all over… back in 2000 lot of things mattered and you were doing lot of, writing a code in a way this buffer that buffer. Today it's just stupid.

– Senior Dev Lead, Web Applications

3.3.2.8 Mentally capable of handling complexity

There are engineers who are frighteningly intelligent, and smart, and they just walk around with this picture in their head all the time of how everything fits together, and they get stuff done.

– Principal SDE, Windows

Many informants described great software engineers as mentally capable of handling complexity: able to comprehend and understand complex situations, including multiple layers of technology and interacting/intertwining software. Informants felt that some software engineering problems were inherently complex, necessitating software engineers who can mentally keep track of all the considerations and implications. This might have been especially salient at Microsoft, where products were often constructed on top of multiple layers of technologies and interacted with many other components. Informants felt that the ability to build an accurate mental model of the interconnections and to be being able to reason about the various options and outcomes was critical for great software engineers, especially those in technical leadership positions.

To solve the problem, [great software engineers] have to have the ability to connect things… You are always debugging layers of stacks of code… this layer talks to some other layer in the horizontal...

– Senior SDE, Web Applications

Informants felt that great software engineers needed to be able to handle complexity because they are commonly assigned the difficult problems. Many great software engineers had to tackle complex problems where having *any* solution was an accomplishment. Informants felt that some software engineering problems were unconstrained messes—often resulting from years of engineering debt—where software engineers could struggle to simply understand the full extent of the problem, let alone come up with a solution. Great software engineers are often assigned those tasks:

The [great software engineers] who tend to move up though, you can give them a complete mess. Problem is not well defined; maybe somebody's tried to solve it six different ways. There's just all this ambiguity about it… [great software engineers that can address these issues are] high up in the chain or they will be. If you really need your problem constrained for success, you're never going to grow out of that [lesser] role.

– Senior Dev Manager, Windows

Though some informants felt that the ability to handle complexity was a natural ability, others felt that great software engineers could effectively augment their natural abilities using tools and processes (e.g. externalize their knowledge by writing it down):

Ability to capture… simulate the architecture in their head… there's probably a little bit of innate skill and cognitive ability… That said, the fact that you don't have that skill doesn't mean that there's no other ways of doing it that may be more brute force… writing things down and studying very carefully the architecture you've put down is putting the brute force time into studying a problem.

– Partner Dev Lead, Windows

The externalization of knowledge discussed by our informants differed in intent from most research on the topic, such as 'knowledge sharing' within free/open source projects (Sowe, Stamelos, & Angelis, 2008). Whereas knowledge seekers were commonly the audience of the knowledge externalization in related work, the software engineer *him/herself* was the audience of the externalized knowledge in our study. Great software engineers were helping themselves reason better about the situation by externalizing their understanding.

3.3.2.9 Sees the forest and trees

> *[A great software engineer] has to have both a very, very narrow extremely technical prospective on his code, but also know where it fits in with the bigger picture, and to be aware of how it affects even our major external customers, and the company vision.*
>
> *– Principal SDE, Windows*

Many informants described great software engineers as being able to see the forest and the trees: reasoning through situations and problems at multiple levels of abstraction, including technical details, industry trends, company vision, and customer/business needs. Informants felt that mental models could exist at various levels and that great software engineers reasoned at all levels quickly and accurately:

> *What differentiated [this great software engineer] from other people in management positions… capability to zoom into the details, and he was not just a high level guy… know the reality of the stack or the reality of the software…*
>
> *– Senior Dev Lead, Web Applications*

Our informants commonly discussed three reasons why software engineers needed to be able to see the forest and the trees. First, many informants felt that some objectives, while seemingly simple, were technically difficult (or impractical); therefore, great software engineers needed a working understanding of the implications of their decisions at multiple levels in order to make optimal choices. Second, most informants felt that engineering of software was usually in service of some higher business objective and that these objectives could be met in a variety of ways, some may not involve software. Great software engineers that saw the forest and the trees were able to make globally optimal decisions, avoiding local optimizations (e.g. focusing only on code solutions). Finally, related to the previous point, being able to see the big picture helped great software engineers avoid getting enamored with technologies: the 'if all you have a hammer, everything looks like a nail' problem:

> *It's making sure they understand both the big picture and the details at the same time. The people that are really good have enough hands-on [knowledge] to be able to identify and solve problems and see the problems and stuff, but they also have a high enough view that they're not just chasing interesting problems to solve*
>
> *– SDE2, IT*

3.3.3 *Interacting with Teammates*

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

Informants mentioned 17 attributes that we felt pertained to engineers' interactions with teammates. Most informants believed that great software engineers positively influenced teammates. For many of our informants (whose titles contained 'Lead' or 'Manager'), this was an important part of their job as managers of other software engineers.

Attributes concerning interactions with teammates generally revolved around four concepts: being a reasonable person, influencing others, communicating effectively, and building trust. These concepts are frequently mentioned in the literature, but often without clear definitions and with little contextual understanding of their importance, as evident in several survey papers involving interactions with teammates (discussed in Section 2.4 and 2.6 (Radermacher & Walia, 2013) (Cruz et al., 2015)). In our discussions, we deconstruct these four concepts into their constituent attributes and then examine each attribute separately.

3.3.3.1 Is a good listener

One of the most frequently discussed soft skills of software engineers is communication skills . Ahmed et al. found communication skills to be the most commonly cited soft skill in job advisements for software engineers (Ahmed et al., 2012). Our findings were similar; many of our informants discussed how great software engineers communicated. Within these discussions, we discerned *effective* communications as comprising of three connected attributes: is a good listener, integrates understandings of others, and creates shared context. We will explain each of these attributes in turn.

> *Being a good listener is important, that you're really hearing the other person's concerns and opinions…*
>
> *– Senior Dev Lead, Windows*

Many informants described great software engineers as being good listeners. For our informants, this entailed effectively obtaining and comprehending others' knowledge about the situation. This knowledge may include static information, e.g. people/organizations and technologies, as well as dynamic mental models about actions and consequences.

Informants discussed three reasons why software engineers needed to be good listeners. First, since software engineers needed to be continuously learning (Section 3.3.1.1)—both to become and to continue being great—acquiring knowledge from others is essential. This commonly helped software engineers avoid mistakes of the past by knowing the approaches that others had attempted. Central to that process is being a good listener:

> *[Great software engineers] don't have to make the same mistakes that other people made and you can also, you can learn from some of these mistakes by talking to people. It is a very less expensive way to pick up, you know, valuable experience and knowledge and all that stuff... engaging people and like learning from them, it's good to be like very active, like active listening and all that kind of stuff and ask them questions.*
>
> *– SDE2, Gaming*

Second, informants felt that the complexities of software systems today often exceeded the mental capacity of a single engineer or a single team. Therefore, to make decisions, software engineers needed to gather knowledge from multiple people:

> *As the company got big, that role broke down because it did get too big for being able to hold it in their head. Dave Cutler, when he came on, had that capability as well, but even today, Cutler, there are parts that he doesn't know about. So that broke down that role.*
>
> *– Partner Dev Lead, Windows*

This need was even greater when collaborating with external teams (e.g. other divisions or teams outside of Microsoft), since external people, in addition to having different technical knowledge, often had different contexts. Because of these differences, our informants felt that the ability to acquire the understand others was important:

> *[This great software engineer] really listens to other very important customers, and he's not just listening to what they're saying, but he's listening to what they're trying to say. He's trying to get a sense for what is the real big problem that they're trying to solve, and where does Microsoft fit into this…*
>
> *– Principle SDE, Windows*

Finally, informants felt that great software engineers' efforts needed to align with organizational goals (discussed in Section 3.3.1.16). Therefore, they needed to acquire directional guidance and input from their managers/leaders as well as peers:

> *[Great software engineers] need to have the connections with the right people because priority is important. Talk to manager, talk to peers, talk to whatever connection you need to find that's your priority.*
>
> *–SDE2, Devices*

Though the need to be a good listener seems obvious, many informants lamented that some software engineers—even experienced engineers—were poor listeners, thus limiting their potential for growth. Some of the causes that our informants associated with poor listening included the listener as egotistical, non-native English speakers, and 'mentally wired' in a different, inexplicable way:

> *I think what was hardest for me was the interaction with other people… Learning to...To understand what my managers or the company needed.... I don't think I could have changed what I felt but if I could acquire better skills to communicate with people. To listen to people… It does create problems I think because you can still be successful in the right field writing good software. I think you're perceived as someone that just solves those problems but not someone that can help see the bigger picture.*
>
> *– Principal Dev Lead, Web Applications*

3.3.3.2 Integrates the understanding of others

> *If they say something that doesn't really line up with your intuition, like that's another time would want to ask questions and like try to figure out, you know, where the discrepancies lie… To really get it, internalize it and connect it with the way you think about things. So I think that is when you really are benefiting from the people around you, you're not just getting good answers from them but you are also being incorporated into your own, like, mesh it with you own knowledge base.*
>
> *– SDE2, Gaming*

Many informants felt that integrating the understanding of others was another component of 'effective communications'. This entailed combining and integrating the knowledge of many people into a more complete understanding of the situation, and then noticing and asking questions about the gaps. Informants discussed an 'integration' process during which great

software engineers considered conflicting views of involved parties or gaps in actions/considerations, and rectified inconsistencies in understanding.

Informants felt that integration of understanding was an important attribute because many poor decisions resulted, not from lack of communication, but rather from lack of *clarity*. Integrating understandings was especially challenging for great software engineers in leadership positions, who need to integrate the understanding of *many* engineers, with disparate understandings and perspectives, in order to make advantageous decisions:

> *I think some of it is a willingness to ask questions and also perhaps figuring out a way to have clarity of thought… oftentimes lots of disparate ideas and pieces of information have to be collected and the ones who are able to recognize patterns and put pieces together can see the picture more clearly. You get some of that through asking good questions, but you also have a way to organize your thoughts that will help you make those connections…*
>
> *– Principal Dev Lead, Enterprise*

An interesting benefit of the integrating understanding of others attribute was that it often benefited others as well the great software engineers. Informants felt that the process of asking questions and clarifying understanding helped all involved parties gain a better understanding or new perspectives on the situation:

> *… you say 10 things, you learn two new things yourself because either people will say, "Hey, do you think about it this way" or they might just come back and say, "Hey, I also thought of this way." It's almost always whenever you share, you also get better. It gives you more clarity on what you're sharing and also makes you learn new things with what other people are basically thinking.*
>
> *– Principal Dev Lead, Web Applications*

3.3.3.3 Creates a shared understanding with others

> *An exceptional engineer will understand how to most compellingly relate the value of that abstraction as it goes to non-abstract to very abstract to each person in the communication chain: their peers, as developers, their testers, their PMs, their designers, their management or if they were to speak at a conference or do demos or interviews of that nature. It's not merely recognizing it but also being able to empathize with your audience, whether they are groups or individuals, in order to get them to get it...*
>
> *– SDE2, Windows*

For many informants, creating a shared understanding with others was the most important component of effective communication. This involved a great software engineer molding another person's understanding of the situation, tailoring communications to be relevant and comprehensible to others. Informants felt that great software engineers could effectively get others to see the situation as they saw it. Beyond simply speaking clearly, great software engineers grasped the level of understanding of others and *adjusted* their communications—often simplifying the message—so that others can understand and incorporate the information into their thinking:

> *You perceive who you are talking to, and you are able to judge on those levels that they are, or you just ask important questions. Do you know about this? And then, be able to simplify the problem to the level that they're working in, or you estimate the amount of information given to them.*
>
> *– Senior SDE, Windows*

Generally, three themes emerged to describe why creating shared understanding was important. First, as leads or as managers, great software engineers often marshalled efforts of other engineers to achieve engineering objectives, which closely aligns with the notion of establishing and maintaining 'conceptual integrity', as discussed by Brooks in the *Mythical Man Month* (Brooks, 1995). Creating shared understanding was requisite for aligning everyone toward shared objectives:

> *One person can only accomplish so much so you've always got to be working as part of the bigger group. People who can't communicate are only going to be sort of so-so effective…*
>
> *– Principal Dev Lead, IT*

Second, engineering teams (especially teams at Microsoft) needed to coordinate efforts with other engineering teams. For example, Windows application team working on the Edge internet browser had dependencies on the Windows platform. Therefore, creating shared understanding with engineers in other areas was often necessary in order to make decisions about where and how to make changes to software:

> *…bring partners, especially difficult issues when people have different opinions… It really depends on your personality and how you communicate.*
>
> *– Senior SDE, Web Applications*

This theme is close to the concept of information sharing reported in studies that examine 'negotiation' processes of software engineering teams (Sandusky & Gasser, 2005). Software engineers must be able to communicate their understanding and perspective to partner teams in order to achieve good outcomes.

Finally, great software engineers often need to communicate with important stakeholders who are not engineers, including executives, experts in other areas (e.g. marketing), and external customers. These people may not have a similar or complete understanding of the situation, but are critical to the success of the software engineering effort. Therefore, great software engineers need to adjust their messaging to fit both their audience as well as the intent of the communication:

> *Our areas where the things are inherently difficult to talk about… business partners or with a customer… When you go outside and you talk to customers, they think about things in much different terms and so in some ways you have to kind of switch gears… why you should care about it and here is how you should think about it.*
>
> *– Principal Dev Lead, IT*

This attribute is closely related to the concept of 'grounding' proposed by Clark and Brennan, which, when done successfully, requires parties to coordinate the content and the process of communication (Clark & Brennan, 1991). Since the engineering of software often involves many people, getting everyone to have a shared understanding considered essential:

> *…communicate about software design really well, and they're able to simplify their language around what needs to be accomplished in a way that makes it quick and easy to get to the heart of a particular solution… you don't speak to each other in code. You speak to each other in human language.*
>
> *– Principal Architect, IT*

3.3.3.4 Honest

Another commonly discussed concept in our interviews was 'trust'; others trusted great software engineers. Examining the discussions about 'trust', we discerned three central attributes: honest,

manages expectations, and has a good reputation. We will explain these three attributes in the next three sections.

> *The thing is everybody make mistakes. When you do make mistakes, you've got admit you made a mistake. If you try to cover up or kind of downplayed mistake, everybody will see it, it's super obvious. It affects your effectiveness, no question about that.*
>
> *– Partner Dev Manager, IT*

Informants felt that being honest was the most important aspect of 'trust'. This attribute was about great software engineers being truthful—not sugarcoating or spinning the situation to their own benefit—and providing credible information on which others can act.

Informants disdainfully viewed software engineers who presented distorted versions of the situation to suit their own benefit. Informants needed to trust the information that the software engineer provided in order to take appropriate action:

> *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, Web Applications*

Our informants did not appreciate wasting time shifting the blame for problems. Many informants discussed software engineers spending significant time avoiding responsibility for mistakes; in contrast, great software engineers accepted responsibility and focused their attention and efforts on addressing the problem:

> *Rather than thinking about how to actually fix the problem at hand, [other engineers were] more like 'How do I make sure that nobody will come back and think that maybe that happened because of something that I might have done?' [This great software engineer] has a way of kind of saying: It doesn't matter…What matters is right now. How do we actually work through it?*
>
> *– Senior SDE, Windows*

Additionally, software engineers need to 'speak the awful truth' in order to help the team forestall problems. Our informants felt that great software engineers need to be honest when they saw problems, even if the bad news might not be welcomed:

> *...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, Web Applications*

Honesty is the attribute most closely related to trust; many informants felt strongly that they would leave teams (or have left teams) that *lacked* honesty. Many informants discussed frustrating situations where they were unable to make engineering progress because they could not trust the information that was provided by team members. Furthermore, there was also a lack of respect for leaders who tolerated (or were incapable of discerning) dishonesty.

3.3.3.5 Manages expectations

> *It's really about making sure that your leads, your managers … setting expectations, they know what you're going to do, you do it...*
>
> *– SDE2, Enterprise*

This was the second attribute that contributed to 'trust'. Informants described great software engineers as managing expectations: setting forth what they are going to do and by when, updating expectations (e.g. explaining the implications of unexpected problems), and then delivering on promises. Great software engineers made sure that stakeholders—usually their managers, but also other teams and their teammates—knew what they intended do and by when. Managing expectations is related to the self-reflecting attribute (Section 3.3.1.4); great software engineers self-initiated corrective action when necessary, and then proactively notified others of changes and made them aware of the consequences:

> *[Great software engineers] take ownership of the project, whatever it is, and they state their deadlines properly. I think accountability is another aspect, like a good software developer is usually very accountable. If you slip on deadlines more than once, or that kind of stuff, I think your credibility is hurt and I think that's a big detriment to software engineers.*
>
> *– SDE2, Web Applications*

For our informants, the most important reason for managing expectations was that it enabled others to set and adjust their plans accordingly. This was especially important for teams with many interconnected components or external dependencies, since delays or changes could

have significant impact on the plans of others. Our informants' sentiments about this attribute reflect findings in the paper by Poile et al. (Poile, Begel, Nagappan, & Layman, 2009): coordination—especially involving changes in plans—was both critical and difficult for large-scale software engineering efforts at Microsoft. Our informants, many of whom were in leadership positions, appreciated software engineers who proactively made them aware of changes in expectations:

> *Some people have that awareness and a lot of people don't… this is the one that you should be done by and if we're not going to be there, what are we going to do to correct that… That'll be more about telling the managers, this is what we need to do rather than the managers saying to the individual contributor, this is what needs to happen.*
>
> *– Principal Dev Lead, Enterprise*

A rarely discussed but interesting aspect of managing expectations is maintaining direction during times of uncertainty. One informant described a great software engineer setting expectations and establishing 'north stars' during times of organizational flux. This kind of expectation management helped the team to maintain its focus and direction to deliver their software product:

> *You have to give a really clear vision of goals that what you are going to achieve, by merging projects, software or the teams… In either case, I think it's very, very important to be very clear about what is the role, by merging those projects with technology… Then those leaders must be able to communicate all the way up and all the way down, technically if necessary, and be able to complete a clean architectural view of what the future of the merging teams going to be.*
>
> *– Principal SDE, IT*

3.3.3.6 Has a good reputation

> *Well it was because of a combination of things, but one of it is because I trusted, I've seen his previous work, I knew about it, I've seen him probably make other recommendations that turned out to have good outcomes… And I think that is exactly what I tell some of my other senior people. You have to build up that reputation and that trust through your years or whatever, how long worth of good deeds essentially, so that when you make that recommendation, they go, I am going to listen to him*
>
> *– Principal Dev Manager, Web Applications*

Many informants felt that having a good reputation also contributed to 'trust'. This attribute was about great software engineers having the respect and confidence of others. Informants felt that

great software engineers needed those around them to trust and believe in them. Great software engineers that had a track record of success were entrusted to make current and future decisions. Beyond organizational imperatives, a track record of success was often seen as a "difference maker" in engagements with others; software engineers that had good reputations were treated favorably by others:

> *It wasn't like [this great software engineer] was just some guy walking off the street throwing off this confidence because that could just be ignorance, but it was...he had done, he wrote the whole...for this product was this thing… And so again, I knew he had that track record and history of doing some pretty impressive things by himself…*
>
> *– Principal Dev Manager, Web Applications*

Informants felt that the upshot of having a good reputation was that the team made better decisions. When other engineers sought out and heeded the advice of the great software engineers, the whole team benefitted from the expertise of that great software engineer.

Some informants had mixed feelings about having a good reputation, because they believed that it was often due to chance and somewhat beyond one's control. The informants felt that most engineers were competent but often lacked the opportunity to demonstrate their competence:

> *I think just the realization that it's not an ideal world... Are you visible to the right people? Are you at the right place at the right time? Are you getting the right opportunities?... There might be two people who have the same and equal talent. But if one person is at the right place at the right time happens to get that opportunity and another doesn't, tough luck. Life is not always fair.*
>
> *– Senior Dev Manager, Windows*

3.3.3.7 Walks the walk

In our interviews, we discerned four attributes contributing to the concept of 'positively influencing others': walks the walk, mentoring, challenging other to improve, and creates a safe haven. Commonly associated with great software in leadership positions, the underlying sentiment for these attributes was that great software engineers helped others to improve.

> *I would like to model myself against that behavior [of a great software engineer]. Like it inspires me to do the same thing.*
>
> *– Senior Dev Lead, Web Applications*

Informants felt that walking the walk was one way that great software engineers positively influence others. This attribute was about being an exemplar for others—being a great software engineer—letting others see their actions and inspiring other to follow. Informants discussed this attribute as passive; great software engineers did not *explicitly* try to walk the walk:

> *But [this great software engineer] was so highly competent and so thoughtful and thorough and basically excellent at everything that he did that he just attracted people to him and he attracted people through his work.*
>
> *– Principal Dev Manager, Web Applications*

While the primary benefit discussed by our informants was improving the capabilities of the team by inspiring teammates to improve, some informants also saw walking the walk as requisite for engineers in leadership positions. Great software engineers were expected practice what they preach and led their team with their own actions:

> *Then I think one weekend [this great software engineer, who was a manager of other engineers] just sat down and was like, "I'll figure it out"... He actually did figure out some things. He did not figure out everything but some of these things is also about leadership by example… you are part of it, and that also pulls the team forward.*
>
> *– Senior Dev Lead, Web Applications*

Many informants felt that *passively* walking the walk was insufficient; great software engineers also needed to *actively* pass on their knowledge and ability to others. This commonly involved mentoring and challenging others to improve:

…how a great software engineer should make other people better around them … there's different levels to that. There's the level of you're just so great at what you do that people can watch and learn from you, but you don't take the time to really help. You are a leader by example instead of actually really going out and doing the teaching and mentoring…I think it's even better… if you actually like teaching, and mentoring, leading that you spend the time to truly coach and mentor folks. I do believe that to really call yourself a master in a subject or discipline or whatever it is you're working with, it's another level to be able to teach it to someone…

– Senior Dev Lead, Windows

3.3.3.8 Mentoring

A mentor is, he's somebody that's got more experience, and he's seen stuff that you haven't seen yet, and he's willing to share his knowledge. The kind of people that horde their own knowledge; I have no time for that. It's great that they have the knowledge and they can be successful, but we're a company, we're trying to survive, let's spread some of that good knowledge around.

– Senior SDE, Applications

Informants felt that mentoring was a common way that great software engineers *actively* positively influence others. This attribute was about great software engineers teaching, guiding, and instilling knowledge into others, helping others—often new team members—improve and to be more productive. Informants often drew on their own experiences to describe great software engineers that helped them when they first joined the team:

Being helpful as a developer… You are willing to sit down with them and kind of show them how it works, maybe get them started in the code a little bit and kind of send them off on the right path.

– Senior SDE, IT

While mentoring was commonly discussed in the context of helping to integrate new team members, several informants also discussed great software engineers mentoring others as replacements so that the great software engineer could move to new teams/projects. The implied understanding was that, if their software was important/critical, then the software engineer may not be allowed to take on other challenges without a replacement. This concept was similar to the 'hand it off to a competent successor' theme discussed in *The Cathedral and the Bazaar* (Raymond, 2001). Our informants felt that, as the great software engineers grew in their career, they had succession plans in place and groomed another to take over:

Yeah. I think sharing/mentoring is very important… He took the time to teach as well as manage, and he influenced many people, more than me, because of that. There was an interesting aside from him though. I think that in return, he had an expectation of loyalty... You were going to see the project through. You weren't going to immediately hop on the next most interesting thing that came around. It took some investment on your … if he was going to invest in you, he expected you to invest in the project as well.

– Partner Dev Lead, Windows

3.3.3.9 Challenges other to improve

...the way he communicates implies that he believes that you can do it. There's this shared confidence so it's like he's done it and so you can do it…. passion lead organizations, like this guy starts, he has to be able to spark your imagination and your sense of self confidence for you to boot strap yourself up to being a productive developer.

– SDE2, Windows

Another way that great software engineers positively influenced others was by challenging them to improve. This attribute was about great software engineers challenging others to take actions to expand their limits and capabilities, such as doing something new or taking on more responsibilities. The great software engineer usually knew that the goal was achievable, having achieved similar objectives themselves, and pushed others to grow professionally:

I had never done anything quite like that. But, he was like oh yeah, we can do that, it's no problem. I ended up writing it. He didn't write it, but it was his confidence and his ability to know that we will walk into that problem and we will get it done somehow that really inspired me.

– Principal Dev Manager, Web Applications

The sentiment among informants was that great software engineers enjoyed being challenged. Many (as indicated in the quotations above) recounted growing in their capabilities and self-efficacy as a result of completing challenges. Likely necessitating the great software engineer to create a safe haven (discussed in the next section), informants felt that challenging others to improve was an effective way of improving the team:

Good developers want to work on teams with great developers and so having a great developer in your team is something that is important and that more junior developers look for and desire in a group and so they have to kind of play this role of being a positive influence to other developers. Other forms of leadership are introduction of ideas, development changes, tools change, practices change. Leaders are trying to help

lead change. Trying to help make the team better, trying to help socialize and introduce new ideas, new tools, new techniques, new ways of thinking.

– Principal Dev Manager, Enterprise

3.3.3.10 Creates a safe haven for others

I think failing is good. If you learn something from a failure, that's a wonderful sort of thing..... [but] If you're afraid of getting smacked upside the head because you made a failure, you're taking a small risk there, but most good managers don't behave that way, right. They encourage the people to experiment, possibly succeed, possibly fail.

– Senior SDE, Applications

Several informants described great software engineers as creating a safe haven for others, so that other software engineers—commonly subordinates or junior software engineers—were not afraid of making mistakes; this empowered young software engineers to do what they felt was right and learn from their actions. Informants felt that, if software engineers were afraid of mistakes, then their development would be stunted:

Chasing after a career path or something… you will deliver your best performance if you are not insecure… One of the challenges as a manager people face these days is retaining talent because there is so much attrition all over.

– Senior Dev Lead, Web Applications

Many informants saw the absence of this attribute as a major contributing factor for dysfunctional teams and talent loss. They believed that the fear of being punished for mistakes often caused software engineers to lie, causing problems for the entire team because their information could no longer be trusted (honest, Section 3.3.3.4). Our informants felt that software engineers did not want to work in environments where they felt insecure, and often avoided those teams/organizations:

If you make one mistake or don't know something and you're sort of dinged by that… and you're only judged if you say everything's perfect even if it isn't… Then you start to have this really kind of I think dysfunctional environment set up where everybody just doesn't say the truth.

– Principal Dev Manager, Web Applications

Though informants felt that having a safe haven was important, many expressed the need to balance a safe environment with feeling the pain of mistakes. Their reasoning was that the

pain from mistakes was the best teacher. If an engineer was hurt by a wrong decision, then the engineer quickly learned to avoid it in the future; informants felt that completely removing this educational mechanism was undesirable:

> *I believe in having people feel the pain of their own mistakes… dealing with the ramifications of the decisions that are being made, I guess is the best way to learn.*
>
> *– Principal Dev Lead, Applications*

3.3.3.11 Asks for help

> *Yeah. Ask for help immediately. I do that mistake. I don't ask for help sometimes because I'm just so focused on debugging or like learning some concepts and don't you forget about the big picture. Someone has to come back and come and pull me out of this. I'm like "Oh, OK, we went way too far. Just come up." If you don't ask for help, you don't know what's going on outside… it's super easy to get lost in a company like Microsoft.*
>
> *– Senior SDE, Web Applications*

Informants felt that great software engineers were willing to ask for help: willing to find and engage others with needed knowledge and information. Great software engineers know the limits of their knowledge and actively seek to supplement their own knowledge with the knowledge of others.

Informants felt that asking for help was important in three ways. First, informants felt that the willingness to ask for help led to greater productivity and faster learning. Great software engineers recognized when asking others for help allowed them to acquire the necessary information significantly faster than they could by themselves:

> *Without asking for help, you cannot navigate all the way to the bottom. If you become Nancy Drew and start looking for clues every single layer, sure you will reach there, but it's not fruitful if you reach there four days from now…*
>
> *– Senior SDE, Web Applications*

Second, informants believed that asking for help was often necessary for software engineers to correctly leverage components produced by other teams. This was especially important within Microsoft because many teams used ‘internal APIs’ or ‘internal tools’ produced by internal partner teams that were not well documented or needed to be used in specific ways.

Informants felt that, to accurately understand the detailed behaviors of other components, great software engineers sought out the owners of those components for help:

> *[This great software engineer will] take the time to go talk to all the other vested parties and get their take on something, and get their feedback on why something would or would not work. He does his homework and anything that he doesn't know he either goes and learns it, or he goes and finds a person that does know. He doesn't try to know it all himself.*
>
> *– Principal SDE, Windows*

Seeking information from other software engineers is a common activity reported in studies that examine everyday activities of software engineers (Ko et al., 2007). Software engineers commonly consult and confer with their colleagues before deciding if/how to change code.

Finally, in the context of great software engineers in leadership positions, some informants felt that these engineers knew when to ask other engineers—typically experts—for help in order to ensure that an area received sufficient technical oversight. This was typically about great software engineers knowing that young/new software engineers needed oversight for tasks, while recognizing that they—the great software engineers—did not have the available time to help. Great software engineers asked other experienced engineers to provide the needed guidance, ensuring the success of the project:

> *For me, as a dev manager, if someone's having a problem, I'm not sure that they're struggling with a task, I grab a senior or a principle developer and say, "Hey, I need someone to help work with this person to get them through the task."*
>
> *– Principal Dev Manager, Enterprise*

3.3.3.12 Does due diligence beforehand

> *I don't respect people who don't do their homework... they don't read the MSDN article, they don't download the SDK, they don't read the help files, they don't read the sample code... they just shoot off an email to the distribution list…*
>
> *–Senior SDE, Windows*

Informants agreed that great software engineers did due diligence beforehand: searching for and examining available information before engaging. Informants felt that great software

engineers are prepared when they discuss situations or ask for help, not wasting other people's time.

Informants discussed two main reasons why software engineers need this attribute. First, our informants felt strongly that great software engineers did not waste other people's time. Related to the asking for help attribute (discussed in the previous section), informants expected great software engineers to do *some* preliminary investigations prior to engaging with others. This typically involved identifying the right people and formulating thoughtful questions. Furthermore, software engineers were expected to provide *justification* for seeking help from other engineer and evidence of preliminary investigations. Informants felt that this was common courtesy when asking other to for their time:

> *Yes, it's just about [software engineers] coming to me… So if an ops person walked into my office ... There's just this intuitive set of things they would have to know to convince me that they know the whole ops thing.*
>
> *–SDE2, Windows*

Informants felt that great software engineers need to be credible when engaging with others. By doing their homework ahead of time to ensure that concerns and questions of others are addressed, great software engineers were positioned to get the desired responses from others:

> *Basically he has an idea, to improve the search quality and he needs to sell his idea to the managers and he does a lot of homework to prepare all the data and he presents to the managers and he finally, the project get approved.*
>
> *– Senior Dev Lead, Web Applications*

In addition to seeking information from others (discussed in the previous section), Ko et al. also reported software engineers seeking some information by themselves (Ko et al., 2007). It appears from our findings, that there sometimes was a dependent relationship between seeking information by oneself and seeking information from others. Great software engineers might usually first seek out information by themselves, prior to seeking information from others.

3.3.3.13 Does not make it personal

> *You can have a very open and heated discussion... But it is all very professional; none of this is ever taken personally. So you can have a very good discussion. When you ask all of us being human beings, we have our moments when we are very enamored with an idea, and want to see that it sort of carries the day, but you have a very good strong debate of it and then you come to the right conclusion. There's no hard feelings, it never gets personal; oh, this is your idea, and it's good or it's bad. It's all very professional.*
>
> *– Principal Dev Lead, Enterprise*

Several informants mentioned that great software engineers did not make it personal: acted and reacted based on fact and reason, avoiding personal biases and perceived slights. Informants commonly discussed this attribute in the context of *reacting* to others. Great software engineers neither took personal offense to communications nor reacted disproportionally to affronts, avoiding *unreasonable* behaviors:

> *I think that it is not effective to try to give it right back to them. Trying to one up them often does more harm than good… Your ability to listen to others and to give useful feedback in a way that's respectful, it matters in our ability to ship the product on time with high quality.*
>
> *– Principal Dev Lead, Enterprise*

Benefits of this attribute were commonly discussed negatively; informants discussed toxic situations when software engineers *made it personal.* Some other informants discuss unpleasant work environments where software engineers took personal umbrage to feedback and discussion; the situation would typically escalate to shouting matches, causing others on the team to feel uncomfortable:

> *They think these people are after them, to show them that they're bad or stupid or not a good engineer and it's not that way at all…you get one person trying to help, another person saying "You're not helping me, you're making fun of me." Then, it gets elevated and gets ugly and production goes bad and if something like that gets so verbal or loud that it causes a mix in the entire group not just between these two people, it's not a good thing.*
>
> *- Senior Dev Lead, Gaming*

Some other informants discussed poor performing teams: some software engineers were making decisions and actions that were meant to discredit adversaries, rather than for the good of the project:

> *You try to discredit and discard his input just to prove your point. One program manager told me that "Whatever is great for Microsoft is not necessarily great for your career and whatever is good for your career is not necessarily good for Microsoft."*
>
> *–Principal Dev Lead, Devices*

3.3.3.14 Resists external pressure for the good of the product

> *[This great software engineer] will say no, if he has to. If what they're asking him to do jeopardizes something else, he'll say no. He can stand up and be brave about it.*
>
> *– Principle SDE, Windows*

Informants described great software engineers, when necessary, resisting external pressure for the good of the product: articulating and advocating actions to ultimately benefit the product. Informants felt that software engineers were frequently pressured, by external partners, by internal partners, by management, and by team members, to take action that may not be good for the software product (e.g. add features, change behaviors, go faster, won't fix bugs, etc.). Great software engineers were willing to take a stand—backed by sound reasoning—whenever those demands jeopardized the long-term success of the software product. Though this may lead to unpleasant situations including escalations, slipping schedules, and negative reviews; great software engineers would stand up for what they felt was right:

> *I think one attribute which is not always seen is like to always do the right thing. At one time you may be forced to make a decision which you feel is not right or you think is not right and just trying to stand up for that decision and be able to articulate or to try to explain to people what they may change is also I think would play a big factor.*
>
> *– Principal Dev Lead, Devices*

Interestingly, not all pressure originated from partner teams and management; many informants discussed pressure from teammates. Multiple informants discussed great software engineers demanding sound technical solutions or extra quality assurance processes, despite higher costs and tighter schedules for the team. Great software engineers sometimes advocated actions that, though painful in the short term, would be better for the product in the long run:

> *[This great software engineer] was very insistent that we have provable security... He wasn't satisfied until we had that proof because he didn't want to replace something that had been cracked by another system which wasn't theoretically secure. It took an enormous amount of work. It took about two years to generate the proof and we actually found some vulnerabilities, fixed them… The system has never been cracked.*
>
> *– Software Architect, Applications*

Interestingly, even though the benefit of this attribute was seemingly obvious—the good of the software product—some informants felt that the attribute and the derived benefit was an oxymoron. The informants felt that great software engineers produced software products that aligned with the goals/objectives of their organization, as discussed in Section 3.3.1.16. Therefore, resisting the desires/wishes of the organization could not be good for the software product.

3.3.3.15 Creates shared success for everyone

> *[Great software engineers] having the skill to be able to find the common good in a solution, be able to say, "I'm pushing for a solution but here's the value for me," and also express here's the value for you. Even though you're still accomplishing the goals you want. They're feeling like they're winning. It's a win-win situation.*
>
> *– Senior Dev Lead, Windows*

Many informants said great software engineers created shared success for everyone: win-win situations that are beneficial to everyone. This often involved great software engineers establishing common big picture or long-term goals that everyone can support. Informants felt that people and teams involved in software engineering efforts commonly had different personal motivations and organizational objectives; great software engineers could effectively align everyone toward shared goals:

> *No matter how good is our code, if our partner [sic] cannot give it a good product for us then we cannot share our greatness to the whole world. A lot of time I see our support to our client is not very well [sic]… we should have a good result combined together.*
>
> *– Senior SDE, Devices*

Informants commonly discussed creating shared success in three scenarios. First, great software engineers often needed actions by partner teams to deliver the final product. For some teams like Windows and Windows Phone, this involved working with external partners (e.g.

equipment manufacturers like Dell and HTC) to deliver a complete product; for other teams, like Office, creating shared success involved working across feature teams on interdependent features and functionalities. Great software engineers needed to establish shared objectives among the stakeholders for optimal outcomes:

> *Like integrating works from different teams, and being able to like stop and understand how these two systems interact with each other… many times it's very easy for the platform or app dev, when there's a problem, you say, "Oh, you should fix it, go to it." Really if you step back and think of whose responsibility like who's that person in terms of that code. I'm being able to say, "Yeah, you're right. This should be done by [our component], the platform, not by the app.*
>
> *– SDE2, Applications*

Second, great software engineers—frequently in leadership positions—needed to put other software engineers in positions to succeed. This generally involved assigning them projects that matched their interests and providing them the appropriate training and guidance:

> *That's individual attention from a manager to an individual contributor, especially initially that helps them get better and learn some of these things that they need to do, and that allows them to be more adventurous and figure out a number of these things themselves.*
>
> *– Principal Dev Lead, Enterprise*

Third, great software engineers needed to proactively manage up to ensure that their leaders made good decisions and that their own actions best contributed to the success of the team. Great software engineers commonly had better understanding of the 'ground truth'; the leadership often had better awareness of the higher level considerations. Therefore, for the team to be successful, great software engineers needed to proactively create shared success with their leaders:

> *It's a two-way communication… there's something going to happen down the road, this piece of code or this feature going to have some issues, need to make your manager aware.*
>
> *– SDE2, Devices*

As discussed in Perlow's work *The Time Famine* (Perlow, 1999), a time famine is when crises arise in teams due to a lack of shared understanding about status and objectives. This

attribute likely helped to avoid dysfunctional ‘time famine’ situations by establishing common objectives and priorities, software engineers were less-prone to spend their time on nonessential tasks:

> *… try to understand what other people need from you…You are really willing to make compromises sometimes, sacrifices to really collaborate with other people to succeed as a team.*
>
> *– SDE2, Enterprise*

3.3.3.16 Well-mannered

> *I think [this great software engineer] is also smart but not cocky. He’s not arrogant. He’s very down-to-earth... you know he’s the one who knows all the information. He doesn’t show it that way. He never come across that way. And the way he sort of communicates ideas and maybe proposals. People would just show respect like, “Oh wow! That is a great idea!” But then, he would never, you know, kind of like drive the conversation in a way that makes the other people seem like, “Oh, I feel so stupid.” Or, like, “I feel so belittled in the presence of you because the way you portray that pride or maybe arrogance, sometimes.”*
>
> *– Senior SDE, Web Applications*

Many informants described great software engineers as <u>well-mannered</u>: treats others with respect, not obnoxious about title, accolades, or knowledge. Informant sentiments about this attribute were rarely about specific actions, rather they were characterized an overall *feeling*. Informants felt that great software engineers made others feel respected—their ideas, opinions, and actions mattered. <u>Well-mannered</u> was the best known and easily identified attribute among our informants; even software engineers who did not discuss this attribute immediately recognized <u>well-mannered</u> when we asked them about this attribute. Though, in interviews, this attribute was discussed using the less polite but more common terminology of ‘not being an ass-hole’. The ease of recognition among informants indicated that they perceived that *many engineers may lack this attribute*:

> *Even though I was the most talented, I was also the last person that people wanted to go to for assistance, because being not humble could alienate them… Even though I had the talent, people did not want to use me as a leader because of the not being humble... Humble is a way of making a person accessible, and creating a favorable experience when people are interacting with your expertise.*
>
> *–Principal Dev Manager, Enterprise*

The consensus among informants was that no one wanted to work with ‘assholes’. This attribute is closely related to the concept of ‘psychological safety’—mutual respect and trust among team members— and contributes to effective teams in many industries (Edmondson, 1999). However, in our study, many informants indicated that if the software engineers were truly gifted, they would probably still acknowledge ‘assholes’ as great. This sentiment seemed counter-intuitive since greatness was a peer bestowed designation and promotion/review processes at Microsoft involved feedback from peers/partners; it was difficult to envision how an ‘asshole’ could be recognized as a great software engineer. One possible explanation might be that the community of software engineers does not value EQ; literature indicates that maverick geniuses may be revered, like Dave Cutler at Microsoft (Zachary, 1994):

> *… unless you’re extremely productive and extremely gifted, you generally can’t do too well at a company like Microsoft if you’re a real asshole. There are people like that, I know that are partner level, they got that from pure talent… you take your super geeks and the ones that are doing extremely well in computer science, they usually are somewhat lacking in social skills.*
>
> *– Principal Development Manager, Applications*

Another contributing factor might be that software engineers were more results- and facts-oriented, as insinuated in the does not make it personal attribute discussed in Section 3.3.3.13, such that software engineers that produced the best results—even ‘assholes’—were acknowledged. Finally, the scarcity of great software engineers, requiring employers to trade off technical ability for other qualifications.

> *…it’s okay to be an asshole if you’re really, really good… it is somewhat true in the profession. Maybe there’s a shortage of software engineers so management tolerate assholes, but that’s definitely not the way to go…*
>
> *– Senior SDE, Windows*

3.3.3.17 Personable

> *I look for in every person that I get, coder or not, but definitely if it was a coder is: can I have a beer with this guy?... That’s important, because if I can’t then we can’t really work together because there’s going to be some point where … they’re very, very stubborn and you know that you can only put them on one thing and that’s it.*
>
> *– SDE2, Enterprise*

Informants described great software engineers as personable: people with whom others enjoy interacting. This attribute is a step beyond well-mannered (discussed in the previous section) and commonly entailed social settings. Informants implied that a certain level of personal relationship and understanding was needed for successful collaborations:

> *[Great software engineers] have to be clear, you have to be respected, you have to get to know people. I think a lot of the personal relations that you can develop you spend a lot of time doing that. That's really helped me and I see that in other good managers that they're very personal. They connect to people well.*
>
> *– Principal Dev Lead, Devices*

The underlying sentiment was that social engagement helped software engineers to better understand the context of fellow software engineers. This understanding likely helped interpretations of communications and facilitated collaborations. Informants felt that teams in which coworkers enjoyed each other's company were more likely to be successful:

> *…a hobby or just be a people skill or just be networking with people or build a good relationship with friends, whichever. They all help.*
>
> *– Senior SDE, Web Applications*

3.3.3.18 Trades favors

> *It's [the great software engineer] returning a favor here and there… I've seen that through a number of cases where someone goes above and beyond to help somebody else out and then somewhere down the road that person has that extra good will to come help you out at some point.*
>
> *– Senior Dev Lead, Windows*

Several informants described great software engineer trading favors, building personal equity with others; the great software engineer can call upon others for personal favors in order to accomplish goals. The informants felt that by leveraging help of others with whom they had personal relationships, great software engineers with this attribute were able to solve problems that other software engineers could not.

The need to trade favors might be especially important within Microsoft due to the large number of teams and the interdependent nature of the software products. Software engineers commonly needed assistance from other engineers (or teams) that had no organizational

obligation to cooperate. Therefore, the ability of great software engineers to get another software engineer (or engineering team) to take action might have been critical to achieve successful outcomes:

> *You can't just sit in your office and code, you need to get out and network. It really facilitates collaborations. When you need something, they will get it done for you. Otherwise, they'll just put you off.*
>
> *– SDE2, Enterprise*

Informants also felt that the back-and-forth between teams promoted better collaboration. By doing favors for another team and having them reciprocate helped both teams to work better together:

> *We talk about trade favor… We're one team, and the core team sometime they help us to do some things, and we help them to do some other things… We help them to make their code better…we help them connect between the customer and the core team.*
>
> *– SDE2, Devices*

There was also a latent sentiment among informants that official organizational processes/policies can be circumnavigated by trading favors. While it was not clear what kinds of policies or decisions can be subverted, several informants hinted that to get things done despite managerial opposition at Microsoft sometimes required calling in favors:

> *When my management reached out to his management, they said no, you can't borrow him because we need him right now. So, I said wait a minute, and I went up the chain; ah-huh, this guy owns me a favor. So, I sent him a really nice email, and he said sure you can have him for a couple of days, and he solved our problem. We were in a real sticky position, and that worked out really rather nicely.*
>
> *– Senior SDE, Applications*

3.3.4 *Engineering the Software Product*

> *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, Windows*

Informants discussed 9 attributes that we felt pertained to the software that great software engineers produced. Like artists appreciating masterpieces of other artists, our informants, many of whom were great software engineers themselves, saw beauty in the software produced by other great software engineers.

3.3.4.1 Pays attention to coding details

> *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, Enterprise*

Many informants felt that a great software engineer paid attention to coding details: including error handling, memory consumption, performance, security, and style. Taken as a whole and considering the tone in which informants discussed this attribute—negatively when software engineers neglected to take into account something obvious leading to problems—we saw this attribute as about great software engineers *not* writing shoddy code. Informants felt that most software engineers—if they put in thought and effort—should be able to write 'good' code. The underlying sentiment was that 'greatness' was a peer-bestowed recognition and that software engineers did not respect other engineers that could not get the basics right:

> *You've got to do the best in whatever you do … you want to try your best, not just get it done, not just finish it, try your best, that's what differentiator between great software engineer and average software engineers… whether it's adaptable, maintainable, scalable all these tricks, performance, security all these. Some are tangible some are less tangible and tractable. Like what is maintainable, you need time to figure it out.*
>
> *–Principal Dev Manager, Web Applications*

Informants also felt that software engineers that paid attention to coding details produced quality software with fewer issues. Great software engineers avoided obvious problems and accounted for likely issues:

> *Attention to detail, it almost sounds cliché, but I view this much deeper than cliché in the software world. I've seen lots of software where yes it works in this scenario, but what if you introduce this thing here. Will it still work? No, we didn't really think about that…*

> *make sure that it can either handle everything that gets thrown at it or it properly recovers or reports or does something useful other than just ignore it… the good engineer will produce maybe quite similar code but will take and have handled a lot of the details and made sure that it's structured in a way that's for the future and considered a whole lot more than just getting the job done for that.*
>
> *– SDE2, IT*

A common extension of discussions of this attribute involved having code in place to localize and debug issues in case unexpected failures occurred. Informants felt that when unexpected issues arose, the code written by great software engineers handled problems gracefully, typically involving having support in place to easily diagnose the problem:

> *Graceful failure handling is crucial at that point because it's always really hard to go back after the fact, it's a natural human tendency to want to write the feature first and get results and then go back and bolt on all the things you need to actually kind of make it useable in the long term. I don't think that's a good way to approach things… designing how to handle these things so that you build them in as you write your code will make your life infinitely easier.*
>
> *– Principle SDE, Windows*

3.3.4.2 Fits together with other pieces around it

> *Because [great software engineers] understand better, interactions around you or around your code. How your code is supposed to work. Why your code should do one thing as opposed to another thing? When you're off implementing or fixing bugs, you realize if I tweaked this here I'm not going to break something else in some other part that I didn't really know about… people continue to be able to look at the entire package…*
>
> *– Senior Dev Lead, Gaming*

Informants felt that great software engineers produced software that fit together with other pieces around it, such as environmental constraints, complementary components, and other products. Beyond integration with surrounding components and meeting their own requirements, informants often discussed this attribute at inter-organizational levels. Software built by great software engineers fit with software and hardware products built by other (internal and external) organizations.

This attribute might have been especially important at Microsoft where many software products were tightly integrated as platforms (e.g. Windows, .NET) or as interconnected

offerings (e.g. SQL DB and Dynamics); furthermore, some products were consumer electronics with *physical constraints* (e.g. XBox, Windows Phone). Great software engineers made appropriate design choices based on the boarder context, assuring that their software worked well in real-world environments with other software and underlying hardware components:

> *If they're making a car part for a car, they'll say, "These are the operating requirements…"… If you have an environment where memory's stringent, it's not very appropriate to use this piece of coding. That would be something that's well documented and well understood from a code.*
>
> *– Senior SDE, Windows*

Furthermore, great software engineers ensured that their technology choices and product decisions aligned with what other partner teams were choosing and the overall direction of the organization. Their software products enhanced and built on other efforts within the organization, making the whole better:

> *…recognizing all of the pitfalls around it. It's not so hard to come up with an idea that's very forward thinking but absolutely doesn't fit anything. It doesn't fit the current dynamics of … at least, if you were to use Microsoft as an example, it doesn't fit with anything Microsoft's doing. … whatever you're doing has to be able to fit within the dynamics of whatever environment you're in… Whatever we come up with, whether it's great or not great, has to fit within that environment.*
>
> *– Principal SDE Lead, Windows*

3.3.4.3 Makes informed tradeoffs

> *[Great software engineers are] quick on pros and cons, I think. Being able to say, these are the tradeoffs. Almost no solution is perfect, but if you can list three and say here are the tradeoffs, and I'm explicitly choosing to give up on a few things in order to gain other things so you go with the solution, that's good problem solving. Relatively fast. Quick thinking in these situations because you run into it so frequently.*
>
> *– Senior Dev Lead, Web Applications*

Many informants described great software engineers making informed tradeoffs with their software (e.g. code quality for time to market), meeting critical needs of the situation. Overwhelmingly, informants felt that few software engineering decisions were black and white; informants could envision or had experienced situations where a desirable attribute—elegant (Section 3.3.4.5) or anticipates needs (Section 3.3.4.8)—was traded for more important

objectives. Great software engineers understood the situation and made effective, and sometimes difficult, tradeoffs to meet critical needs.

The most frequently discussed tradeoff was optimizing for deadlines, which was critical in many situations, such as securing continued funding for project, be first to market, fixing a critical customer problem, etc. Informants expressed willingness or having personally traded almost *anything* for time:

> *I think with a company like Microsoft versus a startup, with a company like Microsoft you've got the luxury of doing things the right way. Whereas with a startup it's the fast way. We do take time here to do design reviews and peer reviews and unit tests. They're the first things to go when you've got next Tuesday it's got to be working and it's got to be out there on the web. You don't spend all your time doing nice design documents and having a big peer review and then going back and iterating on that a couple of times to get it exactly right. You don't have the luxury.*
>
> *– Principal SDE, Gaming*

Some informants also discussed great software engineers considering the longevity of the software product. Informants often contrasted long-living software (e.g. Windows) with evolving online services, which are frequently updated and rewritten; they felt that software engineers took the lifespan of the software into consideration, enabling some attributes—especially anticipates needs (Section 3.3.4.8)—to be traded:

> *Part of answering this question requires knowing what is the longevity, what is the lifetime of the software to be developed. If you're talking about developing a system, like where we work, any system we develop lives on forever, for a long time. Relatively speaking then, there's a maintenance cost, there's a scalability cost, there's a future proofing cost.*
>
> *– SDE2, Enterprise*

3.3.4.4 Evolving

> *I really want to put ideally something out, very small changes, in front of users every couple of weeks… Starting from there, can we actually break that down into what are the individual components that would take… Just being able to have a very clean step-wise process moving forward… What are the immediate steps to that, how can we break this down so that we have really concrete deliverables on an ongoing basis?*
>
> *–Senior Dev Lead, Web Applications*

Some informants felt great software engineers produced software designs that were evolving: structured to be effectively built, delivered, and updated in pieces. This closely resembled the 'evolvability' software attribute (Myers, 2003).

Informants agreed on two common situations where the software design needed to be 'evolvable'. First, even great software engineers may not be able to predict user reactions to new software/features; therefore, great software engineers needed to be able to iteratively learn and adapt their software according to customer reactions. Second, many Microsoft product were very large, necessitating the ability to replace or update parts of the system while the entire software system continued to function. This second need was commonly compounded by tight schedules; therefore, great software engineers needed to be able to structure their software for effective incremental changes.

> *It's kind of like evolution. You start with a strong component with a good idea and slowly you move forward. Slowly adjust the system or the requirements are coming, more like a market or industry is changing. You adapt.*
>
> *– Senior SDE, Windows,*

Informants felt that evolving software designs limited risks associated with wrong design decisions and provided agility to meet changing demands. Informants felt that the designs enabled great software engineers to quickly adjust or reverse directions when decisions resulted in negative reactions from users, thus limiting the impact of problems.

> *It's a constant improvement and constant evolution of what you're doing by learning how your product is functioning and how it's being used. You then are able to get feedback and put it back into the product.*
>
> *– Principal Dev Lead, Web Applications*

Delivering updates/changes incrementally enabled great software engineers to reevaluate and adjust investments frequently, adapting to emerging needs of the users or market conditions:

> *I always believe in iterating quickly. The worst thing in the world is going in the wrong direction for a long, long time…losing lots of money for a long period of time, feels pretty bad to me. So, I try to iterate quickly all the time.*
>
> *– Senior SDE, Applications*

3.3.4.5 Elegant

> *Sometimes when you look at the code that [this great software engineer] developed, you feel, first of all, it's very easy to read his work, it's highly structured… they are simple. It's very easy to understand in a sense that it's very simple. Doing something well and in a very simple way is very very hard.*
>
> *Lot of times, it's very easy to just put down your thoughts and be done with it and then to look at his work and when you see the way he solves the problem, it's very straightforward. When I discuss with him, you see that the simplest solution is, sometimes it's not the first solution he thought. This is improved through looking at a problem closely and through a lot of optimization, eventually after you have arrived at a simpler solution. Seeking that simple solution, I think is one way just to make a better software engineer…*
>
> *– Principal Dev Lead, Web Applications*

Many informants described the software of great software engineers as elegant: intuitive software design solutions that is easily understood. Informants recognized that some problems in software were highly complex and constrained, making it difficult to have a simple solution that met the requirements. Therefore, they admired great software engineers that produced easy-to-understand solutions, elegant designs that others could easily reason about how the designs addressed requirements and constraints.

The underlying sentiment was that avoiding complexity was critical. This is the same thinking that underlies research into complex complexity metrics such as McCabe's Cyclomatic Complexity Measure (McCabe, 1976). Informants felt that complex solutions increased the likelihood of bugs and increased maintenance costs (if problems were fixable at all):

> *Is this the simplest way to do things and the most skillful way to do things as compared to making it overly complicated… It's concise and clear. How easy is it to debug? Debugging usually is harder than actually coding up those things first time around, so if you've done it in a complicated way, then you're probably not going to be able to debug it…*
>
> *– Senior Dev Lead, Web Applications*

Furthermore, complex solutions resulted in brittle code that were more costly to evolve and maintain:

> *Never complicate any things… when you simplify things it becomes easier for you to maintain, going forward for customers… You get lesser number of issues reported by a customer.*
>
> *– Senior Dev Lead, Enterprise*

Despite being simple, informants made it clear that elegant software did not equate to *terse* code. Great software engineers created software designs that were each to *comprehend*, communicating intentions clearly. Simply having fewer characters often made the software more difficult to understand:

> *[Some engineers], for whatever reason, want to type as little as possible, so their code is always terse and these sorts of things. I think once you teach them, "Look, maintainability matters and simplicity is good." And strive for that, then those things become details that they need to work on…*
>
> *– Principal Dev Lead. Gaming*

3.3.4.6 Long-termed

> *And then over time the whole health of your code base evolves because you've built in a framework to handling failures, a solid framework, you're not trying to make something up later and glue it into code that's already written… What you get if you don't do that is a lot of spaghetti code where people try to go in after the fact and add in their own error handling.*
>
> *– Principal SDE, Windows*

Several informants described great software engineers as long-termed with their software: considering long-term costs and benefits, not just short-term gratification. Commonly associated with bug fixes, informants felt that problems would arise that necessitated solutions spanning disjointed places such as component/executable, software products, teams, etc. Great software engineers would accurately recognize these situations to craft solutions that solved the problem holistically, not simply shifting the manifestation of the problem to another location.

The underlying sentiment was that ‘duct taping’ a solution together was tempting, especially in situations where the software engineer may not completely understand the software that he/she was repairing. However, these ‘kludges’ often did not address the root cause of the problem. Informants felt that great software engineers fully understood the problems and produced solutions that did not simply ‘kick the can down the road’:

They've got a bigger breadth or areas, if you've got a problem and you really have no idea what it is… They can own it and work through it and drive it and be crossing the technical boundaries in exploring it and trying to resolve it.

– Principal Dev Manager, Enterprise

3.3.4.7 Creative

[Great software engineers] can think outside the box. Being able to sort of like, hey, here's a traditional solution, but guess what ... Usually with solutions we often have constraints… Being creative is actually, I feel that, able to take these constraints, take the difficult circumstance and actually make it into something that could actually still work, but without a huge complex overhead...

– Senior SDE, Web Applications

Informants described software of great software engineers as creative: novel and innovative solutions based on understanding the context and limitations of existing solutions. Informants felt that there were two important interconnected aspects to creative solutions. First, software engineers needed to understand the unique constraints and requirements of the problem. Great software engineers comprehended how these contextual conditions affected possible solutions:

If you're looking for really an innovative …or just a solution that's outside the current norm… think through the problem…constraints that are currently imposed on the environment.

– Principal SDE Lead, Windows

Subsequently, software engineers needed to know when to apply existing solutions. Informants felt that great software engineers did not invent new solutions without reason; they used existing solutions when appropriate. Informants stressed this point because they felt that known solutions (e.g. standard libraries) were generally less costly and less error-prone:

You are now using all of your creativity to reinvent things that are already invented and that is just basically wasteful.

– Principal Dev Manager, Web Applications

Nonetheless, most informants felt that novel problems occurred frequently in software engineering, needing great software engineers with the ability to come up with innovative solutions or adapting an existing solution:

Understanding patterns and understanding how to apply something is very important so you don't recreate wheels all the time… when there isn't an obvious pattern… Are you creative enough… come up with something new?

– Senior Dev Lead, Windows

3.3.4.8 Anticipates needs

[This great software engineer] would be like, "Now, imagine that you already have that and you've built that and now you have a team that might come to you and say we'd like to maybe use it for that and that… Now, a few years later, somebody else wanted to start working with that."… examples of how people might want to use technology… How would you maybe change your design with that in mind that we might somehow have to accommodate inter-operating with that technology in the future? How might you do that?

– Senior SDE, Windows

Informants felt that great software engineers anticipated needs with their software designs: problems and needs not explicitly known at the time of creation based on their knowledge and understanding. Great software engineers accommodated possible future requirements not known at the time of inception. Informants commonly mentioned scale (more users), feasibility (technology advancing to the point where new things were possible), and integration (interoperability with additional software products). This attribute is closely related to the concept of 'extensible' designs (Krishnamurthi & Felleisen, 1998); however, while extensible designs in the literature generally involves adding *new* features and functionality, informants commonly discussed supporting the *same* requirements but at different *scales*, both smaller (e.g. an operating system that runs both PCs and Phones) and larger:

QQ, the Chinese chat program. It now has hundreds of millions of users. That system was designed fifteen years ago, when QQ only had a few million users. It still works today, that's amazing, to have a system that scales that well, to foresee all the issues it would have to face.

– SDE2, Enterprise

More than any other attribute, informants discussed the propensity to go overboard with anticipating needs. Many informants discussed software engineers attempting to anticipate needs in the face of uncertainty, incurring high costs to add unneeded flexibility. Some thought that any prediction of the future was foolish and preferred to design for current needs and being open to rewrites:

Architect something now that's going to survive well 20 years from now? Nobody is that smart to be able to predict the future that well, I will refactor towards new requirements and I constantly do that.

– Senior SDE, Applications

3.3.4.9 Uses the right processes during construction

Unit testing, of the code. Well before that was fashionable. [This great software engineer] must have been right on the leading edge of it, it was all about the code quality and he had almost no bugs ever found in the product and that was actually his track record, too.

– Senior Dev Manager, Windows

Informants described great software engineers as using the right processes during construction (e.g. unit testing and code reviews), in order to prevent potential problems. Generally, these were quality-control processes intended to discover problems before deployment; the three most commonly mentioned processes were unit testing, test-driven development, and code reviews. Informants felt that great software engineers effectively used these processes to ensure that software engineers thought through their designs. For example, several discussed software engineers who were pressured to produce high-quality code because they needed to present in front of peers in code reviews:

Like the way we enforce it, the process really makes that happen… So you really have to think through in order to stand up in front a room and defend the spec that you wrote and similarly with code reviews, you push those things out and you don't get to check in until your peers sign off on. You really can't do that without having thought through what you're doing.

– Principal Dev Lead, Web Applications

An important aspect of the using the right processes during construction attribute was knowing how and when to use these processes. Informants felt that simply executing the processes was not sufficient; software engineers needed to understand how to execute the processes effectively. For example, some processes (e.g. test-driven engineering) could be garbage-in-garbage-out if not executed correctly.

[Great software engineers] have to know the test cases, so you have to know how your code is going to be used. … Those are all the areas and a good developer will know

those. That's why I say they need to know how to write their own specs, so that they can design the right outcomes, implement it well, and then actually test their work.

– Principal Development Manager, Applications

This attribute appeared to be the manifestation of the knowledgeable about software engineering processes attribute discussed in Section 3.3.2.3. Whereas knowledge was internal to the software engineer, this attribute captured the effect on the software resulting from great software engineers appropriately applying those processes.

3.4 DISCUSSION

In this study, we have sought a holistic, contextual, and real-world understanding of what software engineering expertise entails. We looked for definitions and explanations from interviews with 59 expert software engineers across various divisions within Microsoft. In this section, we will highlight key insights and conclude with a discussion of the threats to validity.

3.4.1 *Nuanced Understanding of Software Engineering Expertise*

Overall, we found that software engineering expertise entailed a holistic set of attributes, including personality, engagement with others, and technical abilities in designing and writing code. These results suggest that productivity is only one criterion for excellence. How software engineers go about engineering their software relative to management (managing expectations, Section 3.3.3.5), subordinates (creating a safe haven, Section 3.3.3.10), teammates (asking for help, Section 3.3.3.11), partners (creating shared success, Section 3.3.3.3), and even oneself (perseverant, Section 3.3.1.6), are all important considerations. This reinforces the perspective that software engineering is a *socio*technical undertaking, and not solely a technical one.

Furthermore, simply delivering the code is insufficient. With attributes like elegant, creative, long-termed, and seeing the forest and the trees, our findings indicate that great software engineers need to take into account complex, experience-driven, contextual *technical* considerations. In addition, many mental attributes are also important, especially attributes associated with learning. We found that the ability to learn new technical skills is likely more important than any individual technical skills. Informants, even those in the same division, used

diverse technologies—sometimes project specific tools (e.g. Cosmos, a Microsoft version of Hadoop). There was no consensus on which specific technical topic (e.g. architecture) was essential. Rather, most informants stressed the importance of learning new skills—manifested in personality attributes like curious and continuously improving—as requisite for great software engineers.

We also identified important attributes of software engineering expertise that had not been studied in detail in the software engineering literature until now. Our findings indicate that effective *decision-making* is an essential part of software engineering expertise. Our informants felt that there were usually myriad options—not all good—for what to do and how to do it. As software engineers grow in their careers, they are tasked with making decisions in increasingly complex and ambiguous situations, often with significant ramifications for themselves and their teams. Therefore, the ability to make good decisions and the mental development that it entails are an important attribute of expert software engineers.

Several prior works have hinted at *deciding* as an important skill. For example, key work activities of Microsoft software engineers observed in Ko's paper (Ko et al., 2007) included 'reasoning about design' and 'what are the implications of this change?' Many studies have examined bug triage processes of software engineering teams (Anvik et al., 2006) (Guo, Zimmermann, Nagappan, & Murphy, 2011) (Jeong et al., 2009); these processes are essentially software engineers *making decisions* about which actions to take in response to bugs. Nevertheless, our study is the first to explicitly identify 'decision-making' (an area of research with its own extensive research literature) as an important topic within software engineering. Making effective decisions, using attributes in Section 3.3.2, is an important skill for engineers to develop.

3.4.2 *Threats to Validity*

As with any empirical study, our results are subject to various threats to validity. Our sampling method contains threats to *external* validity. Though our 59 interviews yielded rich insights, it was a small sample, even for Microsoft, which employs tens of thousands of engineers. This small sample size led to some natural biases, such as underrepresentation of women; we had only three among our 59 informants. In addition, we only sampled engineers in

Seattle, USA; findings may not generalize to other cultures. The size of the organization may also affect generalizability, especially for attributes related to people and organizations. Microsoft also had an established set of practices, tools, and products; findings may not generalize to other contexts (e.g. startups). Finally, Microsoft is a software-centric company; informants discussed unfavorable conditions in non-software centric industries, like finance and retail. It is unclear whether the same attributes (or their standards) generalize. Nonetheless, Microsoft is a good place to start, as we discussed in Section 3.1.

There are threats to the *construct* validity from the lack of a clear and shared definition of a software engineer. Though, in general, informants understood that we meant people who wrote code to be used by customers, and we clarified the definition whenever there was confusion.

Our interview and analysis processes also contain threats to *internal* validity. Informants could generally only mention a few salient attributes unprompted; given more time to think, informants may have produced more attributes. Moreover, while our analysis was systematic, other researchers may discern different attributes, definitions, or models than ours.

Though nearly all of the attributes of great software engineers we uncovered have been mentioned to some degree in prior work and many attributes overlap with ones important to other professions, our study is the first to produce a holistic set of attributes of *software engineering expertise*, with definitions and explanations. This foundational knowledge enables us to build toward additional understanding about software engineering expertise, detailed in the next chapter.

Chapter 4. SURVEY STUDY OF EXPERT SOFTWARE ENGINEERS

Our initial interview study of expert software engineers has provided a foundation of definitions and explanations of attributes of software engineering expertise. However, our prior study, described in Chapter 3, was largely *qualitative*; we still lack *quantitative* knowledge about the relative ranking of the attributes as well as how those rankings are affected by the context of the software engineer. This knowledge enriches our understanding of software engineering expertise, helping practitioners and educators prioritize their improvement and pedagogical efforts as well as assisting researchers focus future investigations.

Findings from our interview study suggested that expert software engineers varied in their opinions of the importance of the attributes of software engineering expertise. For example, the continuously improving attribute (Section 3.1.1.1) was frequently mentioned and commonly deemed essential by our informants; while other attributes, like mentoring (Section 3.3.3.8) and well-mannered (Section 3.3.3.16) were mentioned less often or were deemed *unimportant* by some informants:

> *I think great software engineers can get stuff done without being humble… I've worked with some software engineers, good software engineers who aren't particularly humble. They still get a lot of respect just because they are great software engineers.*
>
> *– Senior Dev Lead, Web Applications*

Multiple prior research efforts have attempted to determine the relative importance of attributes of software engineering expertise. The ACM Computing Curricula (Shackelford et al., 2006) (Section 2.2) provides rankings—minimum and maximum levels of knowledge—for various *technical skills* that software engineers should possess; however, the rankings did not consider interactions with others or personality traits, which our findings and others have found to be important aspects of expertise. Several studies have ranked attributes that new graduates need in order to get their first industry job (Hewner & Guzdial, 2010) (Radermacher et al., 2014) (Section 2.4). Attributes that ranked highly included technical skills (e.g. 'proficiency with the C++ language'), interactions with teammates (e.g. 'ability to work with others and check your

ego at the door'), and mental abilities (e.g. 'problem solving'); though, the authors acknowledge that some needs may be specific to new graduates and may not be applicable to *expert* software engineers. Kelley's 14-year study looking at successful engineers—not *software engineers* specifically—identified and ranked nine successful 'work strategies', covering various approaches to working with teammates (Kelley, 1999a), described in detail in Section 2.6. However, not considering *technical skills* specific to software engineers was a major limitation. The SEI's Capability Maturity Model (CMM) prescribes activities that teams should possess at higher levels of maturity, (e.g. 'quantitative process management' and 'software quality management') at the 'managed' CMM level (Herbsleb et al., 1997). However, CMM focuses on *teams*, not individuals; it is unclear which (and how much) of these apply to *individual* software engineering expertise. The volume of prior work indicates that knowledge about the relative importance of attributes is likely important; however, for various reasons (as we have discussed) prior work has fallen short of a holistic, contextual, and real-world understanding.

Furthermore, our interview study and other prior work suggested that importance of the attributes likely vary across different contexts. For example, several informants in our interview study felt that various "how easily the software can be updated" affects importance of attributes:

> *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…. I will take the risk… I don't think probably [it] could apply to the product when you are shipping something by a floppy disk.*
>
> *– Senior Dev Lead, Web Applications*

Some informants discussed expectations of software engineers differing depending on the country where the software engineering takes place:

> *Asian [sic]… being successful…here, people are actually more focused on your working [sic], your outcome. In Taiwan, social [sic] is very big piece of being successful. Sometimes the communication [sic] is not about you are right or not [sic], but also about that relationship. The people has the better relationship, they would listen more than with other relationship [sic].*
>
> *– Senior SDE, Devices*

One informant also mentioned that the number of software engineers working on the project affects the importance of attributes related to interacting with teammates:

Well, if I'm a sole person working on a product…It doesn't matter if I'm open to ideas... If it's 10 people all working on disparate parts, we have to agree maybe on a common interface somewhere, but if it's several people working on the same code, they're maintaining the same thing, they all have to agree on a concise style or the proper vision, the direction… it makes more of a difference.

– Senior SDE, Windows

In addition to our findings, various research studies also indicated that contextual factors may affect perceived importance of various attributes of expertise. As discussed above and in Section 2.4 on 'new graduates in their first industry jobs', the amount of experience may affect perceptions. Gender may affect perceptions due to cultural convention differences and educational hardships (A. Fisher & Margolis, 2002) (Margolis & Fisher, 2003). Carver et al. found those non-computer science degrees were more effective in conducting code reviews (Carver et al., 2008), suggesting that educational background may impact perceptions. Ahmed et al. examined job postings for software engineers and found the demand for independent workers ("can carry out tasks with minimal supervision") to be higher in North America relative to other regions of the world (Ahmed et al., 2012), suggesting that there may be cultural differences in perceptions. The existence of different curricula for different types of software engineering efforts (e.g. games (Hewner & Guzdial, 2010) and embedded systems (Shackelford et al., 2006)), indicates that the type of software may affect perceptions. Various studies examining the size of software engineering projects (Brooks, 1995) (Pendharkar & Rodger, 2009) discuss that more software engineers working together necessitates better communication, suggesting that size of the engineering team may affect perception. However, despite numerous studies hinting at the influence of various contextual factors, no prior study has examined these factors in tandem quantitatively—statistically testing, quantifying, and explaining their effects (if any).

In this study, we expanded our understanding of software engineering expertise by conducting a worldwide quantitative survey of experienced Microsoft software engineers, along with qualitative follow-up interviews, to answer the following questions:

- How do experienced software engineers rate the importance of these attributes?
- How are perceptions of importance affected by context of the software engineers?

4.1 METHOD

This study proceeded in two parts. First, we constructed and deployed a large-scale survey to assess the relative ratings of the 54 attributes we identified in our interview study and to examine the relationships between the ratings and the contextual characteristics of the respondents. Then, we performed follow-up qualitative email interviews to understand the rankings and relationships found in the survey.

4.1.1 *Survey*

To ensure that we obtained information from the most experienced software engineers we created two sampling strata, based on their titles in the company address book. We selected *experienced* software engineers: employees in the software engineering role with titles at the 'Software Development Engineer [Level] II' promotion level up to 'Senior Software Development Engineer Lead' promotion level. These software engineers typically had at least 5 years of working experience. We also selected *very experienced* software engineers: employees in the software engineering role above the promotion level of "Senior Software Development Engineer Lead". These software engineers typically had 10+ years of working experience and were often responsible for critical technical areas within Microsoft. The titles of the software engineers in our survey are in Table 4.3. We included Leads and Managers of engineers in our sampling because, at Microsoft, nearly all of them had hands-on experience as software engineers. We consolidated the list of titles, removing various address book anomalies: expanding abbreviations (e.g. manager and MGR), reconciling numberings (e.g. 2 and II), and consolidating wording variations (e.g. senior software engineer and senior software development engineer).

The anonymous survey was hosted 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. We personalized the solicitations with the software engineer's name, briefly described the purpose of the research, and explained why we needed their perspectives; these steps help to reduce inattentive survey responses (users providing insincere or haphazard responses) (Meade & Craig, 2012). Each solicitation had a separate anonymized

Table 4.3. Titles of expert Microsoft software engineers studied

Titles in 'experienced' sampling strata	*Titles in 'very experienced' sampling strata*
Software Development Engineer II	Principal Software Development Engineer
IT Software Development Engineer II	Senior Software Development Engineer Manager
Senior Software Development Engineer	Principal IT Software Development Engineer
Senior IT Software Development Engineer	Principal Software Development Engineer Lead
Senior Research Software Development Engineer	Principal Software Development Engineer Manager
Software Development Lead II	Principal IT Software Development Software Engineer Manager
Senior Software Development Engineer Lead	Application Development Manager
	Senior Application Development Manager
	Principal Software Architect
	Partner Software Development Engineer
	Partner Software Development Engineer Lead
	Partner Software Development Engineer Manager
	Architect
	Software Architect
	Senior Software Architect
	Partner Software Architect
	Architect Manager
	Director of Engineering
	Distinguished Engineer
	Technical Fellow

survey link to prevent multiple submissions (e.g. via bots, which may introduce bias and lead to spurious rankings/relationships). The solicitation email is in Appendix A and the full survey is in Appendix B. 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 respondents' right not to participate, we asked questions about the respondents' demographics, experience level, and current work context. We focused on contextual factors discussed in prior work and mentioned in our interview study, as described in the introduction. Appendix B contains the demographic questions and the type of response solicited. Table 4.4 (Section 4.2.2) lists the contextual factors and their distributions within the sample.

We sought respondents' ratings for all of the 54 attributes: 18 on personality, 9 on decision-making, 18 on interacting with teammates, and 9 on the software produced. In anticipation of respondent fatigue, we presented the questions in four groups, corresponding to the four groups above (from our interview study discussed in Section 3.3). 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. Figure 4.1 shows what the survey looked like for the hardworking attribute.

We took several steps to ensure that respondents accurately understood the attributes. We presented each attribute by describing a software engineer who possessed the attribute. We then piloted the survey with five software engineers using the think-aloud protocol to identify comprehension issues. This led to several changes to match the thinking and understanding of Microsoft software engineers. We changed 'software engineer' to 'developer' to differentiate people on engineering teams that did not write code; supporting quotations were added for 37 of the attributes; several clarifications were added for potentially confusing attributes (e.g. 'practices and techniques for building a software product' was appended with 'e.g. unit testing, code reviews, Scrum, etc.').

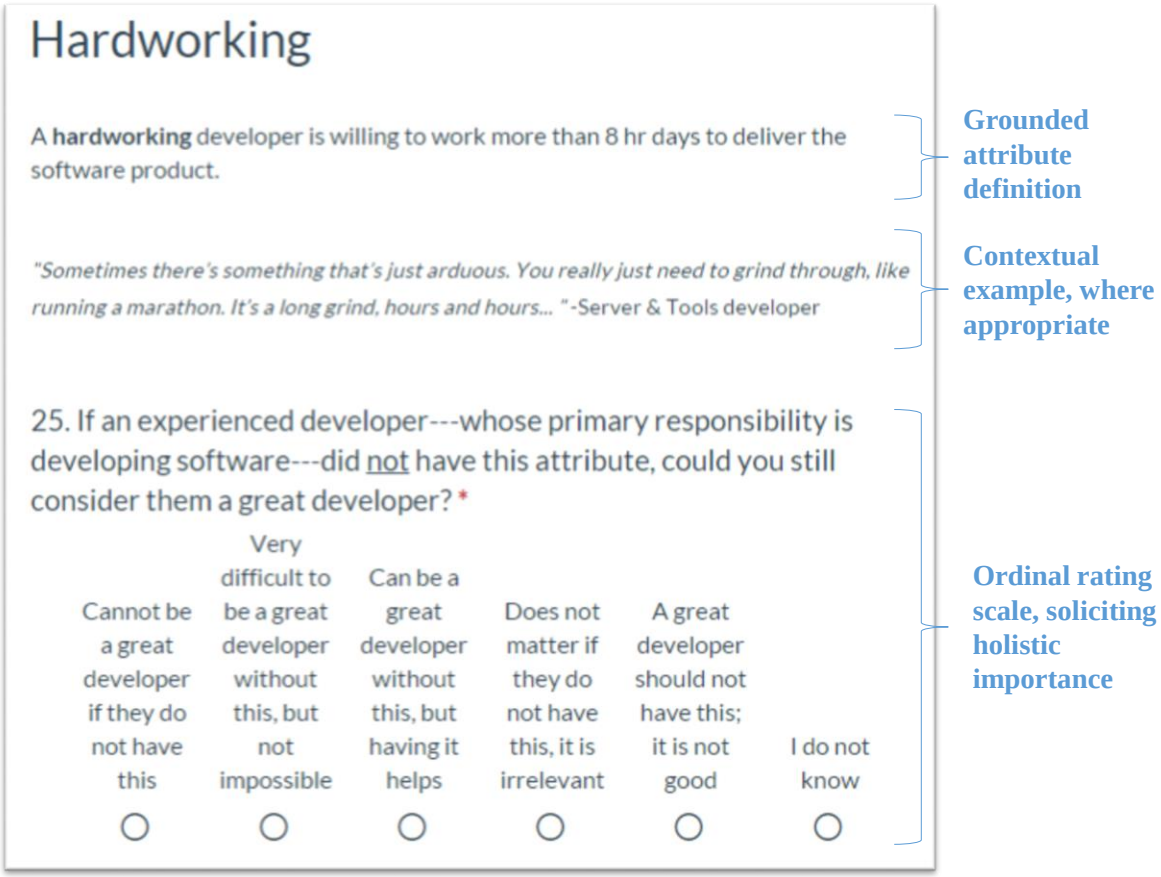


Figure 4.1. Survey question for the hardworking attribute

To get a holistic and 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 Figure 4.1):

- 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
- I do not know

Pre-testing showed that this negative operationalization of the notion of 'importance' was easier for respondents to answer and was better at eliciting the attribute's holistic importance. Positively phrased variants led to responses that lumped together because respondents could almost always imagine a situation in which an attribute *can be* important; by asking respondents to think about situations in which an attribute would *not* be important, we observed more differentiation. The highly-rated attributes were the attributes that our informants felt great software engineers could not be without. This better matched our conceptualization of importance.

We deployed the survey in two waves, sending the initial set to 200 developers (~100 in each experience strata) to look for problems. We examined questions with high rates of 'I don't know' and high median response times, as well as complaints in the closing open-ended question. We also assessed the expected response rate based on this initial wave. After finding no issues and assessing the response rate, we deployed the survey to the larger sample, aiming for 500 responses in each sampling strata.

The survey took respondents a median of 28 minutes to complete, with some outliers due to respondents leaving and returning to the survey at a later time. The minimum, 25th percentile, 75th percentile, and maximum completion times were 7 minutes, 17 minutes, 77 minutes, and 44 days, respectively. Overall, we obtained 1,926 survey responses. We obtained 825 responses from *experienced* software engineers (~7% of all *experienced* software engineers at Microsoft); of the 1,802 software engineers we solicited in the strata, this was a response rate of 46%. We obtained responses from 1,101 responses from *very experienced* software engineers (~35% of the all very experienced software engineers at Microsoft); of the 2,496 software engineers solicited in the strata, this was a response rate of 44%. Of the respondents, 1,634 (84.3%) completed the survey, with an additional 292 providing ratings for at least one attribute. We found no item-response bias—there was no relationship between attributes and having a response, at the α=.05 level using Logistic regression. We used both complete and partial data in our analysis because, due to randomization, each attribute had an equal chance of being seen and the assessment of importance of each attribute was independent of other attributes due to how we asked about importance.

Our notion of importance for an attribute was the degree to which expert software engineers believe that a software engineer cannot be considered great without the attribute. There are two aspects to this conceptualization: the importance rating and the agreement among respondents. Statistically, this means that the distribution of ratings: both the central tendency (i.e. criticality) and the dispersion (i.e. agreement). The attributes that are deemed more important have distributions that are more concentrated at the higher ratings (Table 4.4 has the ratings distributions for the attributes).

To determine the most and least important attributes, we ranked the attributes by comparing the rating distribution of each attribute to the rating distribution of every other attribute, counting the number of 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 and thus should not be averaged), our response levels were not centered (four positive ratings and only one negative rating), and averages do not consider the dispersion of ratings. To compare distributions, we used the Mann-Whitney rank-order test (Hollander, Wolfe, & Chicken, 2013).

The test can be used to compare ordinal data and distributions (i.e. both central tendency as well as dispersion), and can be used when the number of observations is not equal. 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 α=.05 level. 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 all 53 other attributes. See Table 4.4 in Section 4.2.1 for each attribute's ratings distribution.

To analyze the relationship between contextual factors and the attribute ratings, we used Ordinal Logistic Regression. We assessed the first order relationships between the contextual factors and the ratings of each attribute. To account for performing multiple statistical tests, we used the Benjamini & Hochberg False Discovery Rate (FDR) adjustment at the q=0.1 level. 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.

4.1.2 *Follow-up Email Interview*

To help interpret the importance ranking and relationship to contextual factors, we emailed respondents to ask for further insight into their responses. We asked about the highest ranked attributes (the top five ranked attributes in Table 4.3), 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 4.3), as well as the attributes that were significantly affected by context (the relationships listed in Table 4.4). 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 differences.

In our survey, 771 respondents indicated that they were willing to answer follow-ups questions. We sent follow-up emails to 111 of these engineers, receiving replies from 77 informants (69.4% response rate). When reasonable, we tried to ask a single informant about multiple attributes, in order to uncover insights that spanned multiple relationships. We

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

4.2 RESULTS

We focused on the attributes and relationships that we asked about in the follow-up email interviews (for which we have the most credible understanding). Regarding the essential attributes of software engineering expertise, we discuss the top 5 (highest ranked) attributes, the bottom 2 (potentially detrimental) attributes, as well as the surprisingly low rankings for attributes associated with 'interacting with teammates'. Regarding differences in perceptions due to contextual factors, we discuss each of the statistically significant relationships.

4.2.1 *Essential Attributes of Software Engineering Expertise*

The ordered list of attributes in Table 4.4 shows the most important attributes at the top and the least important attributes at the bottom, based on their ratings distribution. The number in the first column is the number of other distributions for which that distribution is more to the left comparatively (based on the Mann-Whitney rank-order test). The ratings distribution is in the second column: the more to the left—right skewed—the better. The third column lists and explains the attributes.

4.2.1.1 Highest ranked attributes

The most important attribute was pays attention to coding details (ranked 1, higher ratings distribution than 53 attributes; 63.1% of respondents gave it the highest rating, 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:

Table 4.4. Attributes of great software engineers, ranked and with ratings distributions

# Higher	Ratings distribution	Attribute and description
53		Pays attention to coding details, including error handling, memory consumption, performance, and style
52		Mentally capable of handling complexity: able to comprehend and understand complex situations, including multiple layers of technology and interacting/intertwining software
49		Continuously improving: constantly looking to become better: improving themselves, their product, or their surroundings
49		Honest: truthful, not sugar coating or spinning the situation for their own benefit. They provide credible information and feedback that others can act on
49		Open-minded: willing to let new information change their thinking: do not believe they know everything and will consider new information
46		Executes: does not have analysis paralysis; knows when to stop thinking and to start doing
45		Self-reliant: gets things done independently and does not get blocked easily; they get around problems by themselves
45		Self-reflecting: recognize when things are going wrong or when their current plan is not going to work, and then self-initiate corrective actions.
43		Persevering: not dissuaded by setbacks and failures; keeps on going, keeps on trying
41		Fits together with other pieces around it: such as environmental constraints, complementary components, and other products.
41		Knowledgeable about their technical domain: thoroughly conversant about their software product, technology area, and competitors
39		Makes informed trade-offs: in their software (e.g. code quality for time to market), meeting critical needs of the situation
39		Updates their decision making knowledge: does not let their understanding and thinking stagnate
36		Curious: desires to know why things happen and how things work
36		Evolving: structured to be effectively built, delivered, and updated in pieces.
35		Knowledgeable about tools and building materials: knows the strengths and limitations of technologies used to construct their software
35		Grows their ability to make good decisions: builds their understanding of real world situations, including alternative, outcomes, and values of the outcomes.
34		Sees the forest and the trees: reasons through situations and problems at multiple levels of abstraction
31		Craftsmanship: takes pride in their work; wants their output to be a reflection of their skills and abilities
30		Does due diligence beforehand: searches for and examines available information before engaging. They are prepared when they discuss situations and do not waste others' time.
30		Elegant: intuitive software:(i.e. minimum complexity) design solutions that others can understand
29		Asks for help: will find and engage others with needed knowledge and information. They know the limits of their knowledge and supplement their knowledge with the knowledge of others
28		Desires to turn ideas into reality: takes pleasure in building, constructing, and creating software
28		Long-termed: considers long-term costs and benefits in producing software and designs, not just short-term gratification
25		Willing to go into the unknown: willing to step outside of comfort zone to explore a new area, even when risky and benefits uncertain
24		Is a good listener: effectively obtains, comprehends, and understands others' knowledge about the situation
22		Passionate: intrinsically interested in the area they are working in; not just in it for a pay check
22		Manages expectations: sets forth what they are going to do and by when, updates expectations (e.g. explaining impacts and implications of unexpected problems), and then delivers on promises
22		Focused: allocates and prioritizes time for the most impactful work; is not overwhelmed by daily distractions and tasks
21		Systematic: does not rush to conclusions or jump to conclusions; address problems in a systematic and organized manner
21		Adapts to new settings: continues to be valuable to the organization even with changes in their environment
19		Integrates understandings of others: can combine and integrate the knowledge of others into a more complete understanding, noticing and asking questions about the gaps
19		Does not make it personal: avoids personal biases. They act and react based on fact and reason, avoiding dysfunctional behaviors based on personal feelings and perceived slights
19		Creative: novel and innovative solutions based on understanding the context and limitations of existing solutions.
18		Walks-the-walk: is an exemplar for others: being a great developer themselves, letting others see their actions, and inspiring others to follow.
18		Knowledgeable about software engineering processes: knows the practices and techniques for building a software product: purposes, how to, costs, and when best to use
13		Anticipates needs—problems and needs not explicitly known at the time of creation—based on their knowledge and understanding.
13		Uses the right processes during construction: using the right processes (e.g. unit testing and code reviews) to construct their software and designs, in order to deal with potential problems
13		Resists external pressure for the good of the software product: will articulate and advocate actions that are for the good of software product, being firm against outside pressures
11		Has a good reputation: has the belief, respect, and confidence of others. They have a track-record of success such that they are trusted with current and future decisions
11		Productive: achieves the same results as others faster, or takes the same amount of time as others but produces more
10		Knowledgeable about customers and business: understands the role their software product plays in the lives of their customers and the business proposition that it entails
10		Creates shared understanding with others: molds another person's thinking of the situation: tailoring the communication to be relevant and comprehensible
9		Creates shared success for everyone: win-win situations that is beneficial to everyone, commonly involving establishing a common big picture or long-term goals that everyone can buy into
8		Aligned with organizational goals: takes actions for the good of the product and the organization, not just what interests them
7		Well-mannered: developer treats others with respect: not obnoxious about titles, accolades, or knowledge
7		Data-driven: measures their software and the outcomes of decisions; let actual data drive actions, not depending solely on intuition
6		Creates a safe haven for others: where others are not afraid of being blamed for mistakes, empowering others to do what they feel is right, and to learn and grow
5		Mentoring: teaches, guides, and instills knowledge to others, helping others—often new team members—to improve and to be more productive.
4		Knowledgeable about people and the organization: informed about the people around them: responsibilities, knowledge, and tendencies.
2		Challenges others to improve: challenges others to take action (e.g. doing something new or taking on more responsibilities), expanding others' limits and capabilities
2		Personable: others enjoy interacting with; they establish good personal relationships with others
1		Hardworking: is willing to work more than 8 hr days to deliver the software product
0		Trades favors: builds personal equity with others, such that the developer can call upon others to do them personal favors

> *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 software engineer; therefore, software 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 (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) as a necessary attribute. Informants felt that great software engineers need to be able to think through complex situations to produce their software products:

> *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 software engineers need to not only be open to new ideas but 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) as important. Informants indicated that great software engineers needed to acknowledge mistakes in order to make optimal 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*

Regarding the honest attribute, informants also discussed developers' *dis*honesty was 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 SDE*

4.2.1.2 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: trading favors (ranked 54, higher distribution than 0 attributes; 4.0% essential, 15.1% important, 44.1% helpful, 29.1% doesn't matter, 6.0% of respondents rated it 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% of respondents rated it detrimental). These results were unexpected because none of the attributes were expected to be detrimental; all of the attributes were from the interview study, which focused exclusively on positive attributes of great software engineers.

Follow-up suggested that these attributes were not inherently bad, but likely reflected *bad situations*. For the hardworking attribute, informants believe that needing to work more than an 8 hours a 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 to do 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 software engineers to operate effectively. Informants indicated that they disliked not understanding how to achieve their goals:

> *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*

4.2.1.3 Low rankings for attributes associated with interacting with teammates

Attributes associated with interacting with teammates were rated the lowest, relative to attributes in the other three groups. The attributes had a median ranking of 40 (lowest among the 4 groups) and 77.8% of the attributes were in the bottom half of the rankings. Attributes associated with decision-making were rated the highest with a median ranking of 17 and 33.3% of the attributes in the bottom half of the rankings; it was followed closely by attributes of the software product with a median ranking of 17.5 and 33.3% of the attributes in the bottom half of the rankings. The next lowest group, personality attributes, had a median ranking of 24, with 44.4% of attributes

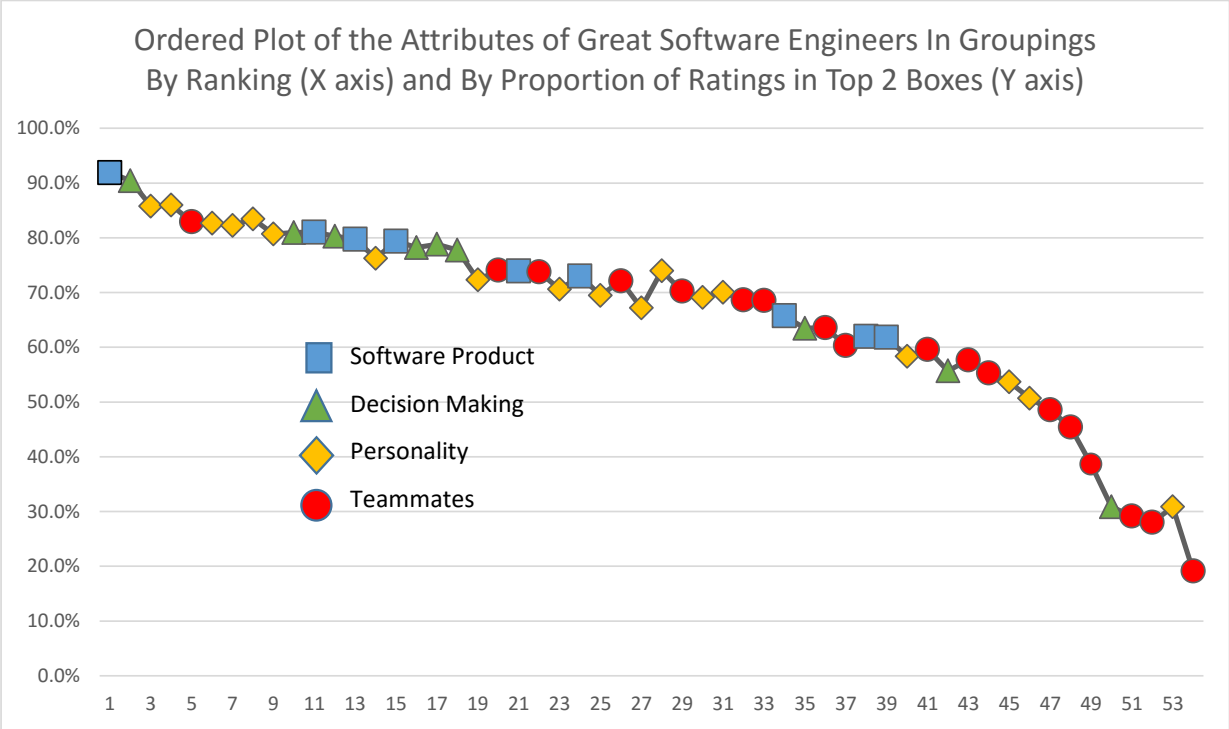


Figure 4.2. Attributes rankings of the four types of attributes

ranked in the bottom half of the rankings. This can be seen visually in Figure 4.2, which plots the attributes grouping based on their ranking (x-axis) and the percent of ratings in the top two boxes (y-axis).

The low rankings were unexpected since numerous prior studies indicated that interacting with teammates is a large part of engineers' everyday activities (Ko et al., 2007) (Latoza et al., 2006), as discussed in Section 2.6. Nonetheless, informants—in follow-up interviews—felt that that the primary job of the developer is to produce high-quality software, the rest, while helpful, 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*

Another contributing factor to the low rankings was the concept of 'truth in code'. 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 was the most important component of 'effective communications' from the interview study (Section 3.3.3.3), but ranked 43 out of 54 attributes in the survey:

> *…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*

4.2.2 *Influence of Contextual Factors*

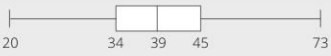
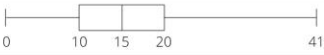
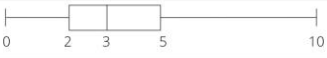
Next we sought to understand whether the ratings varied depending on the context of the software engineer, and, if so, why.

Findings about the influence of contextual factors are in Table 4.5. The table describes the 26 contextual factors, provides descriptive information and descriptive statistics, and summarizes the statistically significant relationships between the factors and attribute 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; rest of the 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 at the q=0.1 level, are listed in column 5 of the table. 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, and do not have attribute relationships listed, indicated' by '–'.

4.2.2.1 Level of experience

We discuss the first five factors in Table 4.5 (*Is very experienced, Age, Years as a professional developer, Years at Microsoft,* and *Employment at software companies*) collectively as *level of experience*. This is reasonable because the factors all aim to measure the same underlying construct of 'experience' and are highly correlated with each other. The statistically significant

Table 4.5. Contextual factors, distribution in survey study, and significant effects. Rows are not ordered.

Row	Contextual factor	Encoding	Distribution of data Box plots are min, 25th, median, 75th, max	Significant relationships with attributes (OLR, FDR, q=0.1)
1	Is very experienced	*Categorical* Based on title	Very experienced (1,101, 57.2%) Experienced (825, 42.8%)	Executes (+) Knowledgable about tools and building materials (+) Makes informed tradeoffs (+)
2	Age	*Numerical* Optional response 1,512 responses		Knowledgeable about customers and business (+) Knowledgable about software engineering processes (+) Aligned with organizational goals (+)
3	Years as a professional developer	*Numerical*		Hardworking (+) Desires to turn ideas into reality (+)
4	Years at Microsoft	*Numerical*		Aligned with organizational goals (+)
5	Number of software companies in career	*Numerical*		Continuously improving (+)
6	Years on current team	*Numerical*		—
7	Experience as manager	*Yes/No*	Manager (1,028, 53.4%)	—
8	Is female	*Yes/No*	Female (149, 7.7%)	Uses the right processes during construction (_)
9	Has bachelors/ associates degree	*Yes/No*	Yes (1,228, 63.8%)	—
10	Has non-MBA masters	*Yes/No*	Yes (762, 39.6%)	Asks for help (–)
11	Has MBA	*Yes/No*	Yes (49, 2.5%)	—
12	Has doctorate	*Yes/No*	Yes (49, 2.5%)	Walks-the-walk (+) Challenges others to improve (+)
13	Has other degree	*Yes/No*	Yes (103, 5.3%)	—
14	Has non-CS degree	*Yes/No* Based on self-reported degree	Yes (537, 27.9%)	—
15	Is not currently working in the US	*Yes/No*	Yes (351, 18.2%)	Aligned with organizational goals (+)
16	Work experiences in non-US countries	*Categorical*	Country Count % India 273 14.2% China 168 8.7% Canada 77 4.0% UK 63 3.3% Israel 51 2.6% Other 383 19.9% None 911 47.3%	31 attributes (+) 9 attributes (+) Hardworking (–) Hardworking (–), Aligned with organizational goals (–)
17	Non-native English speaker	*Yes/No* Based on first language	Yes (926, 48.1%)	Passionate (+)
18	Type of customer	*Categorical*	Internal (791, 41.1%) External (270, 39.1%) Both (865, 31.75%)	Persevering (+)
19	Server-side software	*Categorical*	Client-side (562, 29.2%) Server-side (754, 39.1%) Both (610, 31.7%)	—
20	Developers worked with in past year	*Numerical*	Min (0), 25th (8), median (15), 75th (25), max (1,000)	—

relationships between *level of experience* and eight attributes (first 4 rows in Table 4.4) were all *positive* (i.e. higher *level of experience* corresponded to higher ratings).

Informants in our follow-up interviews—all of whom were in the *very experienced* sampling strata—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 software engineers at higher levels were evaluated based on the contributions they made toward progressively higher 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, software engineers gained the understanding that to make meaningful contributions, software 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 than the one they naturally work in.*
>
> *– Architect*

Many informants further felt that this knowledge and understanding could only be gained through actual *firsthand* experience:

Software engineering processes are there for a reason… The more experienced you are, the more you saw the pros and cons of process firsthand.

– Principal SDE Lead

Finally, a natural 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

4.2.2.2 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 the 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 the female informants felt more strongly that the engineering of software should proceed in an orderly manner. Developers should adhere to the agreed upon process or change the process; they should not go 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…A different process was used each time.

– Senior SDE Lead

4.2.2.3 Educational background

Having a Master's and/or PhD degree had unexpected *negative* relationships with asks for help, challenging others to improve, and walks the walk (for Master's, see row 10 in Table 4.5; for Ph.D., see row 12 in Table 4.5). Informants provided two interesting hypotheses. First, informants felt that a graduate degree was largely optional for success in the software industry; therefore, software engineers that get those degrees may be more intrinsically motivated than others. This might have led 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 suggested that engineers with graduate degrees were often hired as *technical experts* such that they were often given the difficult problems to solve; thus, they 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 4.5) between engineers with and without advanced degrees. We found that *the number of engineers worked with in the past year* was statistically significantly *less* (α=.05) for both engineers with a Master's degree (p-value=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-value =0.037, with medians of 11 for those with and of 15 for those with a Ph.D. degree) using the Mann-Whitney rank test. These results support the hypothesis that software engineers with advanced degrees worked with fewer other software engineers.

4.2.2.4 Work experience in another country

We found many positive relationships between attributes and work experience in another country; qualitative follow-ups suggest five underlying themes. We asked informants about their ratings and then inferred the underlying themes based on their work experiences.

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*. The competitive context necessitated software engineers in those countries to excel in many areas in order to compete effectively:

> *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 4.5) and the trading favors attribute. While the trading favors attribute was the lowest ranked attributes overall, its ratings were significantly higher for respondents in China. Follow-up interviews indicated that the higher ratings were influenced by the broader cultural norms in China:

> *Culturally there is a different perception… 'guanxi' [关系] 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 the 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 being engineered in their country as well as the state of software engineering practices in their country. 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 in order 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 4.5):

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

4.2.2.5 Type of customer

There was a statistically significant *positive* relationship between having both internal and external customers and persevering (row 18 in Table 4.5). However, based on follow-up

interviews, we believe this relationship was likely spurious (which was not a complete surprise since the 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 necessitates perseverance to work through; though, even he felt that 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*

4.3 DISCUSSION

In this study, we have sought a holistic, contextual, and real-world understanding of the relative importance of attributes of software engineering expertise and the effects of context on those rankings. We surveyed 1,926 expert software engineers to derive rankings and to assess relationships; we then conducted follow-up email interviews with 77 expert software engineers to interpret the results. In this section, we will examine overall insights and conclude with a discussion of the threats to validity.

4.3.1 *The Essential Attributes*

Ranking indicates that engineering of software at its highest levels (at Microsoft) is a complicated and complex *technical* undertaking. Teams need experienced engineers who are smart, technically savvy, and dedicated to finding and implementing solutions. Informants indicated that uncertainty and complexity afflict their software from underlying dependencies, system states, external callers, and/or partner components. Pays attention to coding details and mentally capable of handling complexity topped the rankings, reflecting this sentiment, as well as high rankings for other product- and decision-making attributes (see Section 4.2.1 and Table 4.5).

Our analyses also indicate that the field of software engineering is changing constantly. Even foundational concepts can change over time, such that those who do not grow and evolve risk becoming obsolete. **Consequently, it is not a specific set of knowledge but rather the desire, ability, and capacity to learn that defines the best software engineers.** The theme of constant learning was prevalent throughout the survey and follow-up interviews; informants frequently indicated that greatness was *attained* and *maintained* over time. This contributed to multiple related attributes—honesty, open-minded, and continuously improving— to top the rankings, as well as high rankings for numerous personality attributes related to learning and improving (see Section 4.2.1 and Table 4.5).

4.3.2 *Relationship with Contextual Factors*

Influences of *level of experience* (see Section 4.2.2.1, Table 4.5) suggest that engineers need real-world experience to become experts. Informants felt that many attributes sound good in theory or in isolation, but become unimportant when put into real-world contexts, amid competing concerns and hard deadlines. This sentiment underlies relationships with aligned with organizational goals and knowledgeable about the customer and business—corresponding to identifying key objectives—as well as hardworking, desires to turn ideas into reality, and executes—corresponding to actually delivering the software product. Informants felt that software engineers needed to experience real-world consequences of their actions to appreciate the cost and benefit of their decisions. This sentiment supports the relationships with knowledgeable about tools and building materials and software engineering processes, as well as makes informed trade-offs.

Results from analyzing the negative relationship between having an advanced degree (*Master's* and *Ph.D.*) and attributes associated with giving and getting help (see Section 4.2.2.3, Table 4.5) indicate that the relationships were likely not due to graduate school education (i.e. graduate schools do not *teach* software engineers to devalue giving and getting help). Rather findings were likely due to the conditions that would lead a software engineer to pursue a graduate degree (a selection bias) and the job assignments of graduates (a survivorship bias). Informants indicated that a graduate degree was generally not seen as advantageous to a software engineer's career (probably compared to hands-on experience, per the previous discussion).

Examining the differences in ratings for those with work experiences in other countries revealed that many facets of local culture affect perceptions of software engineering expertise. Contrary to overall rankings, for those working in China—even Americans working in China—trading favors and '*guanxi*'—关系（the building of a network of mutually beneficial relationships, commonly found in Chinese business culture) is a positive attribute (see Section 4.2.2.4).

Though we also found relationships between *gender* and *type of customer* with rankings, those relationships were difficult to explain. We believe that the relationship with *type of customer* may be accidental; informants in our follow-up interviews did not have good explanations for why the relationships existed. The relationship between uses the right process during construction and *gender* may be because female engineers have stronger preferences for proceeding in an orderly and agreed upon manner. This area may be interesting for further research.

4.3.3 *Threats to Validity*

As with any empirical study, our study has various threats to validity. *Construct* validity issues may arise from engineers interpreting the attributes and the contextual factors differently. While some amount of personal variation is unavoidable, we sought to limit issues by conducting pre-tests—adjusting and clarifying survey questions—and examining Microsoft engineers, who share common understanding (e.g. common Microsoft terms). Ambiguity of the term "software engineer" is a potential construct validity issue. Based on feedback in pre-tests, we switched to using the term 'developer' to best capture the ACM's notion of 'someone who develops software to be used by others'.

Internal validity issues may arise from several sources. First, use of the FDR adjustment and the numerous significant relationships with *having work experience in India and China* may have hidden other interesting relationships. Second, our analysis examined only the first-order relationships between ratings and contextual variables. While second-order relationships may exist, we feel that our choice was appropriate given little prior research to support investigating

second order relationships. Finally, the follow-up interviewees were a self-selected subset of the respondents; other interpretations of the attributes and relationships may exist.

External validity issues may also exist. We explicitly over-sampled very experienced engineers, who may exhibit thinking and perspectives that were particularly well-suited to the Microsoft environment. This might decrease the importance of some attributes that Microsoft engineers 'take for granted' (e.g. hardworking). We also studied only Microsoft engineers. We felt that they were an interesting, important, and relevant population that may actually strengthen the external validity of our findings (see the Section 3.1), but they were nonetheless from one organization. Finally, we did not sample for other interesting attributes, specifically gender and non-US software engineers; however, we note that we received many responses from both female respondents (149 responses, 7.7%) and non-US respondents (351 responses, 18.2%). This should have allowed us to detect significant systematic differences due to these contextual factors.

In this study, we developed a robust understanding of software engineering expertise through combining knowledge from our interview study with a survey study of expert software engineers; however, an important knowledge gap remains. Today, the engineering of software is interdisciplinary, involving many expert non-software-engineers performing critical tasks. Yet, we know almost nothing about the perspectives of these expert non-software-engineers on software engineering expertise. We will address this knowledge gap in the next chapter.

Chapter 5. INTERIVEW STUDY OF EXPERT NON-SOFTWARE ENGINEERS

Software engineers do not produce software alone. Today, the engineering of software commonly entails software engineers collaborating not only with other software engineers but also expert non-software-engineers. Many roles, like artists (Hewner & Guzdial, 2010), data scientists (Begel & Zimmermann, 2014), designers (Beyer & Holtzblatt, 1995), writers (Mehlenbacher, 2000), program managers (Aranda & Venolia, 2009), and others, are important to the engineering of software products. Consequently, a holistic, contextual, and real-world understanding of software engineering expertise would be incomplete without insights from these expert non-software-engineers.

Effectively working with non-software-engineers may be even more critical than working with other software engineers, since non-software-engineers often perform essential tasks that software engineers are ill-equipped to perform. Furthermore, effective collaborations with non-software-engineers may be especially challenging since they often belong to different communities of practice (E. Wenger, 1999) with different vocabulary, culture, norms, and processes. For example, the ACM (Joint Task Force on Computing Curricula, 2014) distinguishes between ‘traditional’ engineering and software engineering in the following ways:

- The foundation of software engineering is primarily in computer science, not natural science,
- The concentration of software engineering is on abstract/logical entities instead of concrete/physical artifacts
- Software maintenance primarily refers to continued development, or evolution, and not conventional wear and tear

Despite these distinguishing factors, ‘traditional’ engineers and software engineers collaborate to develop many products, e.g. Xbox and Microsoft Surface.

Though many software engineering studies mention or involve non-software-engineers, none directly examine the perspective of non-software-engineers on software engineering expertise. Trifonova et al. surveyed more than 50 research publications about software development projects with *artist* participation (Trifonova, Ahmed, & Jaccheri, 2009). They found four kinds of topics:

- Requirements/needs for software and software functionality within the artist community, evaluation in art projects, software tools, software development methods, collaboration issues, business model
- Artists' mastery of computer skills, multidisciplinary collaboration between art and computer science students, art in computer science curricula
- Aesthetic of the code, aesthetic in software art, aesthetics in user interfaces
- Social and cultural implications of software/technology on art

The survey indicates that artists are involved in the engineering of software and that they have needs (e.g. tools, education, and engagement methods) and concerns (e.g. aesthetics) that differ from software engineers. However, the survey also indicates that studies have not examined artists' perspectives on software engineering expertise.

Begel and Zimmermann surveyed Microsoft software engineers about the questions they would like data scientists to answer (e.g. how do users typically use my application and what parts of a software product are most used and/or loved by customers) (Begel & Zimmermann, 2014). Fisher et al. examined challenges analyzing 'big data' at Microsoft (D. Fisher, DeLine, Czerwinski, & Drucker, 2012). These reports indicate data scientist help software engineers perform important and challenging functions.

Numerous studies (Barry W. Boehm, 1991) (Ropponen & Lyytinen, 2000) indicate that project managers help teams manage risks in software engineering projects. Some of the top issues like scheduling and timing risk indirectly implied that great software engineers managed, mitigated, or avoided these issues.

Lee and Mehlenbacher (in follow-up to a 1991 study that interviewed software engineers at DEC about attributes they wanted in technical writers (Walkowski, 1991)) surveyed 31 technical writers, including 16 that worked for software companies, about working with subject matter experts (SMEs) (Mehlenbacher, 2000). Among other questions, the authors asked, "what do you dislike about working with SMEs?" The most commonly reported issues were 'time and accessibility', 'respect for the documentation process', and 'communication skills'. While these are likely attributes of good software engineers, the authors were not examining software engineering expertise directly.

In the well-known book *The Inmates Are Running the Asylum*, Cooper describes software engineers ('the inmates') as making too many engineering decisions that impact usability of the product and providing biased information to others (e.g. over estimating costs of features) to further derail projects ('running the asylum') (Cooper, 1999). The author calls for designers to use a disciplined approach (e.g. using personas) to ensure better products. Cooper's insights are relevant—good software engineer should avoid those behaviors—however, his focus is on usability and design rather than all aspects of software engineering.

Existing literature indicates that perspective on software engineering expertise from expert non-software-engineers is an important knowledge gap, one that neither existing literature nor our previous studies of software engineers have addressed directly. As with our previous studies, not only do we need to know which attributes are important from the non-software-engineer perspective, we also need contextual understanding of *why* those attributes are important for the real-world engineering of software:

- What do expert non-software-engineers think are the important attribute of great software engineers?

- Why do expert non-software-engineers think those attributes are important for the engineering of their products?

5.1 METHOD

Since little is directly known about the perspectives of non-software-engineers on software engineering expertise, we sought to balance depth of understanding, breath of perspectives, and relevance of insights in this initial investigation. We conducted semi-structured interviews with 46 senior-level employees across 10 roles—non-software-engineers—in engineering teams at Microsoft. We chose to conduct the study at Microsoft because it allowed us to leverage our knowledge and learnings from the previous studies presented in this report. We chose to conduct semi-structured interviews because it afforded us the ability to explore insights in an open-ended fashion, digging into details and opinions where necessary.

Though many non-software-engineers work with software engineers at Microsoft, we wanted to focus on the non-software-engineers who were the most likely be essential to successful engineering efforts. We examined the 20 listed professions on the Careers at Microsoft site: Business Development and Strategy, Business Program & Operations, Engineering, Evangelism, Field Business Leadership, Finance, Hardware Engineering, Hardware Manufacturing Engineering, Human Resources, IT Operations, Legal & Corporate Affairs, Marketing, Product Manufacturing Operations, Research, Retail, Sales, Services, Supply Chain & Operations Management, Technical Sales, and Unassigned. We chose to focus on roles within the 'Engineering' and 'Hardware Engineering' professions (which included roles other than software engineering that support the engineering efforts) because they were the most relevant to the engineering of products containing software at Microsoft. Though, we note that other professions may have interactions with software engineers and may be interesting areas for future research. Almost all of the software engineers at Microsoft belonged to these two professions; also, as indicated by the organization arrangement, the non-software-engineers within these two professions were likely (or expected to be) collaborating closely with software engineers.

To ensure that we obtained in-depth and relevant knowledge, we chose to focus on full-time employees at the 'senior' level or above, based on their titles in the company directory. In order to reach the 'senior' level, one must have extensive industry experience—typically 5+ years—at Microsoft or elsewhere. Furthermore, one must have demonstrated expertise in their

field; within Microsoft, to reach the 'senior' level, starting as a new college hire, required at least three promotions. Therefore, via the hiring and/or promotion processes, these individuals have been affirmed by their peers as experts in their field. In addition to being experts in their own technical areas, these expert non-software-engineers—due to their extensive work experience—were also more likely to have worked with software engineers to ship products; thus, they were better able to provide valid and interesting insight into software engineering expertise.

We organized senior non-software- engineers in the 'Engineering' and 'Hardware Engineering' professions into 12 roles based on our understanding of their address book titles. We then pruned down the roles to 10—requiring at least 50 'senior' level employees working in the Seattle area—to ensure populations large enough for us to sample for face-to-face interviews. The 10 roles (with number of 'senior' level individuals in the Seattle area in parenthesis) were: Artists, Content Developers, Data Scientists, Designers, Design Researchers, Electrical Engineers, Hardware/Mechanical Engineers, Product Planners, Program Managers, and Service Engineers. Two other roles had fewer than 50 individuals at the 'senior' level *worldwide*: Audio Engineers and International Managers.

For the 10 roles, we solicited interviewees, via email, by stating that we aimed to understand software engineering expertise and wanted perspectives of non-developers; an example solicitation email is in Appendix C. We conducted interviews in a round-robin fashion among the roles. This process facilitated identification of cross-cutting themes and enabled us to think of interesting drill down questions for subsequent interviews. Of the 102 people recruited, we interviewed 46 (a response rate of 45%). The interviews were approximately an hour in length and were generally conducted in the office of the interviewee or in a near-by meeting room; in two instances, the interviews were conducted in my own Microsoft office. We interviewed at least 4 people in each role and a total of 46 people; the roles and titles of the expert non-software-engineers we interviewed are in Table 5.6.

After explaining the study, the participants' rights (e.g. to not participate and to ask questions later), and obtaining consent, in our scripted semi-structured interview, we first asked about the background of the informant and his/her role: "What is your background? And how did you come to be a <role> at Microsoft?", "What—in your opinion—is the function of <role> in

Table 5.6. Expert Microsoft non-software-engineers interviewed

Roles	*Titles of interviewees*
Artists	Art Director (x2) Technical Artist Technical Art Director Cinematic Animator
Content Developers	Senior Content Developer (x2) Senior Publishing Manager Senior Content Engineer Senior Producer
Data Scientists	Senior Data Scientist Principle Applied Science Manager Principal Data Scientist Principal Data Scientist Manager
Design Researchers	Senior User Researcher Senior Design Researcher Principal Design Researcher Senior Design Research Manager
Designers	Senior UX Designer (x2) User Exp Visual Designer Principal Creative Director Senior Design Lead
Electrical Engineers	Senior Architect Director Electrical Engineering Senior EE Design Engineer Senior Component Engineer
Hardware/Mechanical Engineer	Senior Design Verification Eng (x2) Dir, Electrical Eng Senior Engineer Senior Mechanical Engineer (x3)
Product Planners	Principal Product Planning Mgr (x2) Principal Product Planner (x2)
Program Managers	Senior Program Manager Lead Principal PM Manager Principal Security Program Mgr Senior Program Manager
Service Engineers	Senior Eng Service Engineer Senior Service Engineering Manager Principal Service Eng Manager Senior Service Engineer

engineering teams?" Then we asked about their engagements with software engineers: "How have you engaged with developers?", "How does that engagement vary in the various phases of development?" Subsequently, we asked about positive and negative attributes of the software engineers the interviewee had worked with: "What are the positive attributes of good developers you've worked with that you believe contributed to successful outcomes?", "What are negative attributes that you've seen contribute to less than successful outcomes?".

Subsequently, we transitioned to asking the interviewees about the ranked list of 54 attributes of software engineering expertise from our previous studies of expert software engineers (Table 4.4 in Section 4.2.1). We explained that the table was derived from interviewing and surveying software engineers, as well as the ordering in the table. We then asked the interviewee, "Please read through the list of attributes and note any attribute that stood out as too high or too low, and tell us why. Then we'd like your top five attributes, from the perspective of successfully collaborations with <role>." Examining the list of attributes helped to overcome saliency effects; numerous interviewees amended, added, or clarified their perspectives on software engineering expertise after looking over the attributes.

For each attribute mentioned, as appropriate, we asked clarifying questions—requesting explanations, more details, etc.—to better understand the attribute from the non-software-engineer perspective. We also asked clarifying questions to better understand why that attribute was important for successful real-world engineering projects. We concluded by asking informants, "Ideally, how would you like to see people in your role and developers collaborating together to engineer software products?"

To analyze the over 38 hours of interviews and over 350,000 words of transcriptions, I used an inductive approach, making three passes through the data. First, I read through the entire transcript to gain an overall understanding of the data and to tag all relevant discussions about software engineering expertise; various side conversations about me, the purpose of my study, and my degree program were omitted. I made a second pass through the transcript to identify key themes, noting them in comments, as well as to highlight key excerpts. Third, I analyzed all the informants in each role and organized them alphabetically. I summarized descriptions of the role when possible, and their engagements with software engineers; I extracted attributes of great software engineers that informants found to be important and analyzed the attributes' importance within the context of collaborating with the informants in that role. I aimed to identify *what* were important the attributes as well as understand *why* the attributes were important. Finally, for each section I reflected on the findings for that role.

Our choice of *separately* analyzing each role inductively, was influenced by the work of Wenger on 'communities of practice" (E. C. Wenger & Snyder, 2000) (E. Wenger, 1999) and

Tuckman on ‘developmental sequence in small groups’ (Tuckman, 1965). Rather than ‘communities of practice’, according the classification of Wegner, Microsoft teams are ‘project teams’, assigned by management to specific projects to accomplish specific tasks (E. C. Wenger & Snyder, 2000). Their tasks and power within project teams, in addition to attributes of other team members (e.g. software engineers), influenced the experiences and perceptions of expert non-software-engineers of other roles differently. Furthermore, though the project teams are not ‘communities of practice’ (and not well suited to be analyzed using that framework), many of the roles clearly were ‘communities of practice’ (e.g. artists, electrical engineers, mechanical engineers). Those experts had their own distinct ‘domain’, ‘community’ and ‘practice’, separate from software engineers as well as experts in other roles. Consequently, we analyzed each role separately, aiming to understand their distinct perspectives, before examining overall themes.

5.2 RESULTS

At a broad level, our expert non-software-engineers described great software engineers as masters of their own technical domain, open-minded to the input of others, proactively informed everyone, and saw the big picture of how all the pieces (even non-code-related attributes) contributed to the experience of customers.

In the 10 subsequent sections, we will discuss the perspectives of expert non-software-engineers in each role separately (ordered alphabetically by role), since each role had different functions within Microsoft and interacted with software engineers differently. We will describe the functions of the role within Microsoft, our expert informants, and the interactions between our informants and software engineers; for many roles, the interactions directly affect perspectives. We will then characterize their perceptions of great software engineers; we will identify and explain the attributes that our informants considered important, supported by contextual examples. Finally, for each role, we will provide a discussion section about the perspectives of the experts in that role. Two roles, Data Scientists (Section 5.2.3) and Service Engineers (5.2.10), were ill-defined within Microsoft; we will discuss the opinions of the informants within those roles individually.

5.2.1 *Artists*

At Microsoft, expert artists were concentrated within teams that develop games. These included software products that were wholly published and produced by Microsoft (i.e. 1st party games, like Halo and Forza) as well as games that were published by Microsoft but contracted out to other game studios to produce (i.e. 3rd party games). All our informants had training as artists, with industry experiences in games (e.g. Ubisoft, Bungee) or entertainment (e.g. Disney, Industrial Light and Magic) companies prior to joining Microsoft.

In discussing their experiences, several informants commented that their role had grown significantly more technical over the years. With advances in 3D modeling and game engines, artists were working more independently than ever before, inserting artistic elements (e.g. textures and figures) into games without the help of software engineers. However, our informants felt that, for most games, there were still technical limits to what artists can do independently, necessitating collaborations with software engineers to bring a game to life:

> *…Art authoring tools that we have, Maya, Max. This is where we build our 3D models and environments in… As long as the assets are already brought in, it's a way to pull everything together. So I don't need to give it to a coder and he puts it together…there's a level of sophistication to understand the technology that's happening with artists. There're certain artists that have a certain level of technical savvy…*
>
> *[Some things] still requires a coder, an engineer, all these people to come to bring this together because really the engines are a bare bones shell that could be easily [extended] with extra code written to go one way or the other…. That allows for coders and artists to collaborate on…*
>
> *-Art Director, Gaming*

While some of our informants focused on visual aspects of the product—creation of artistic assets—others were *technical artists* acting as the glue between artists and software engineers. Technical artists straddle the fence between engineering and art; they wrote scripts and automations that enable artists to plug in their assets into the game engine, and they managed processes for integrating artistic assets into the game. Several informants discussed this increasingly prevalent and important role. Software engineers focused on ‘game engines’ that enabled artists to plug in their assets (e.g. textures and figures) into the game, and technical artists facilitating and managing that integration process. With background in art, technical artists

facilitated the collaboration between pure engineering and pure art, commonly acting as translators between the two communities of practice:

> *During the meeting room with artists and programmers, "Okay, here's the project," discuss those things. It's hard to do it because the language is different, the thought process is different, so we have a role, technical artists…*
>
> *Those technical artists translate engineers and artists [sic]… that person understands script on the engineers' side, and also a little bit about the artist background… "Hey, what is the problem?" "This is my problem." He collects the information, and if he can fix it, he fixes. If he can't, he has to report to the graphic engineers, and they can fix it. Because he understands most of my language, because artists' daily vocabulary is different than the engineers. So he understands those things, and he understands the engineers' side so he can help communicate with the engineers. That kind of helps, and I think it leads to success in the game industry.*
>
> *– Cinematic Animator, Gaming*

5.2.1.1 Making great entertainment products

Our expert artists emphasized that games were *entertainment* products, necessitating software engineers and artists to collaborate on a holistic experience for users, including both technical game play as well as 'look and feel'. Our informants felt that this mindset was important for games because it influenced the objectives and priorities of the team, which in turn, changed decision-making—more importantly, decision *makers*—within the team. Several informants discussed changes in engagements between artists and software engineers in recent years; the informants felt that, in years past, games were commonly engineer-constrained (i.e. teams built what was technically possible), leading to engineering dictating the direction of projects, often influencing into artistic choices. The underlying sentiment that decisions about 'look and feel' were often inappropriately being made by software engineering leads. However, with technology advancement and industry maturation, informants felt that game development has grown increasingly even-handed, with artists having an equal voice in team decisions. Artists viewed the current engineering approach at Microsoft as balanced between artists and software engineers:

> *[At Microsoft] we're all making the game so a large amount of it does hinge on the infrastructure of the game, the bare bone–that's why engineers are critical to that…but at the end of the day, somebody else is going to hang something off this. I've got to hang artwork off this, someone is going to hang gameplay off this…*

A lot of the times back in the day and this is going back well over 10 years. [The engineers] were almost like a semi-lead for the project because of how they architected the engine… really determined a lot of [the whole team's] outcomes, what kind of gameplay could happen, what kind of art could happen… "This is what you got. This is how you are going to do it."

…It's become much more collaborative now…

–Art Director, Gaming

The primary challenge faced by our informants and their teams was the need to 'push the envelope' under technical constraints while shipping on time. Perhaps due to the competitive nature of the gaming industry, informants discussed needing to offer something outstanding for each game and/or each release. However, this was frequently checked by limits of the underlying technology (e.g. how many shapes can be painted and how fast) as well as how fast the game had to be completed (e.g. yearly refresh cycles). Consequently, many attributes of great software engineers emerge from collaborations between artists and software engineers to overcome these challenges:

Early on it's important for us to understand with engineers and tech artists to understand budgets… How big can their rig be, their memory, and budget allocation to this? ...If we're doing a game where it's a single character and he is being overrun by thousands of zombies, that's a different thing again because now you have that many characters on the screen, something has got to give, right?

–Art Director, Gaming

For software engineers and artists working on games, these needs led to interesting engagements. First, since entertainment products had a significant 'look and feel' component, artists and software engineers worked together on technical compromises that actually *enhance* the 'look and feel' of the game. For example, an artist described working with software engineers to creatively reduce the number of polygons that needed to be drawn in order to fit the technical budget, while enhancing the 'creepy factor' for a zombie game, by hiding much of the background in mist. Good art does not always equate to technical perfection for games (e.g. as in the 'uncanny valley', discussed in the next section); therefore, some technical limitations are phantom problems. Artists worked with software engineers to understand which technical problems actually needed solving. Finally, artists generally have less knowledge of technical advances such as new gameplay capabilities with the latest updates to the Xbox hardware.

Therefore, when deciding on features and making cuts, artists typically wanted help from software engineers to understand new possibilities and new alternatives. Overall, our informants felt that artists and software engineers collaborated continuously throughout each phase to achieve lofty goals while avoiding catastrophic problems at the end:

> *…if it's too easy, you're not pushing it, you're not trying hard enough, or you're not being aggressive enough, or not innovative enough… You want it to hurt a little bit because that means you're pushing it, but you don't want to hurt so much that you killed it all.*
>
> *Our team, we try to stay as collaborative as possible…we all have to collaborate together to make sure that we're all putting components and pieces together of a cohesive pie instead of making stuff in a silo and then hoping it all comes together at the end, because that never works.*
>
> *– Art Director, Gaming*

5.2.1.2 Artistic MacGyvers

Our informants felt strongly that software engineers should not ‘steam roll’ artists in making decisions for the team. Informants felt that collaboration between artists and software engineers should be egalitarian, with each side bringing different—equally valued—perspectives and expertise. Informants described great software engineers as working *together* with artists to produce their games; conversely, informants disliked software engineers that were dismissive of artists and dominated the decision-making process. Several informants described deleterious interactions where software engineers viewed artistic aspects of the game as merely ‘fit and finish’—to be bolted on at the very end—rather than working with artists from the very beginning to shape the direction of the project:

> *Yeah, so power-wise it's lower, "Okay, the artists just help us. Just make it pretty. Here's the product, make it pretty." This is not a good situation because, I mentioned a couple of times, the final product, people care about those. So if engineers understand those, yeah, definitely there's no such power, like the battle and egos. Those kind of things create disability [sic].*
>
> *– Cinematic Animato, Gaming*

To facilitate collaborations, all of our artist informants wanted software engineers to have some understanding of the art domain, most importantly the language and the mindset. The sentiment was that in order to work successfully with artists, software engineers needed to

develop some understanding of art. Our informants saw many instance of misunderstandings resulting from software engineers lacking an understanding of artists (and vice versa). Artists sometimes had difficulties understanding problems in the same way that software engineers understood the problems. For example, one informant described his initial frustration with time-consuming integration process imposed by software engineers, since he (the artist) cared only about seeing the (final) visual outputs. Only after the software engineers explained the process and the artist experienced the problems (e.g. breaking changes) that the process aimed to avoid, did the artist start to appreciate the engineering processes. Software engineers also sometimes did not comprehend what artists wanted. For example, an artist described having to iterate with a software engineer on the lighting and reflections for an art asset because the software engineer did not grasp the technical-artistic need for the object:

> *But sometimes there are engineers…who don't really have an eye for what looks right. Does it look right to you? Or doesn't it look right to you? And they will just plug whatever the correct or whatever they think it the right parameters and things… "That looks blown out, the bloom is way too high on that, the lighting is terrible on that."*
>
> *-Art Director, Gaming*

Furthermore, since artists and the software engineers often had different concerns, each can be myopic about the implications of their actions. Something that might seem simple or trivial from an artistic standpoint (or vice versa) could result in significant implications for the project. Therefore, out informants felt that great software engineers were able to impart understanding of the relevance to artists:

> *…being aware that not everybody has the same concerns as you, not everybody works the same as you. If you're a coder you have different things that make you scared at night versus the artist, versus the producer. Not that you have to know them all, but you at least have to know of them so that you can have a conversation with them so that when a coder is freaking out, "What do you mean change this, this and this?" And the artist is like, "Dude, it's just this one little thing." "You don't understand you completely broke the build because you did this." At least have knowledge that, "Okay, sometimes I shouldn't push that button," or whatever. Just being aware of that.*
>
> *-Art Director, Gaming*

In discussing successful collaborations, our informants described great software engineers as having an in-depth technical knowledge, which commonly helped artists in three ways. First, great software engineers used their knowledge to help scope the work (e.g. making

trade-offs) in order to keep the project within bounds of technical feasibility and schedule. Scheduling was critical for games since development was often time-bound (e.g. in time for the holiday season), with significant financial ramifications for missing dates. Our informants felt that artists would commonly suggest "pure fantasy" and needed software engineers to be the "voice of reality". Second, great software engineers offered alternatives or novel artistic possibilities based on their understanding of advances in computing, even *predicting* future capabilities. Great software engineers knew the abilities of the existing technology and anticipated the capabilities of new technologies; they would suggest technical changes to achieve a better look and feel than what was originally envisioned by the artist. Third, great software engineers worked with artists on creative solutions within the constraints of the system during development. Our informants appreciated engineers who were willing to work with them on workarounds and trade-offs to approximate their artistic goals amidst constraints:

> *[Software engineers] who, like the MacGyver kind of attitude where, "Hey, given these constraints, here's what we can make." ... So the ability to be able to say, "Well, what is it gonna take to get us there in the timeframe that we need?" And so the best software engineers that we've worked with from an art perspective are the ones who can think quickly on their feet and improvise to come up with creative solutions to help meet the needs.*
>
> *-Technical Art Director, Gaming*

Collectively, this indicated that artists envisioned a great software engineers as a 'MacGyver', who was a fictional American TV character famous for being resourceful and possessing expansive knowledge.

Our informants believed that great software engineers also needed to have certain mentalities in order to be successful in the gaming industry. Foremost, informants felt that people working in the gaming industry understood that, in order to produce a successful product, they needed to 'push the envelope'. Therefore, software engineers could *not* be risk-adverse. Second, software engineers needed to be open-minded and be willing to adapt. Our informants stated that what is 'correct' and 'best' may not be known ahead of time, and making and prototyping was often required to understand the optimal solution, similar to the mentality behind knowing by doing (Schank et al., 1999). Therefore, what the team set out to do may change once an initial prototype is produced. Software engineers should adapt to "deliver what's actually useful and

maybe not what was on paper". Finally, our informants felt that software engineers needed to be hardworking. In addition to myriad challenges throughout development, there was likely hard work at the end to push the product across the finish line. This extra work was commonly necessary because the team was attempting to 'push the envelope':

> *"Aim for the stars and hit the roof." And what is in implying is that shoot for 200% knowing that you're going to attrition down to 100%. What you don't want to do is aim for 100 and then naturally attrition down to 50, because then it's just like, there's no point. There's no point in making that… [Developers] are unsung heroes at the end to like, "Okay, we're going to make this pile of craziness actually fit and ship and run."*
>
> *-Art Director, Gaming*

5.2.1.3 Discussion

Our expert artists felt that successful games required 'look and feel' in addition to game play; therefore, only through addressing needs of both artists and software engineers can the team produce a successful product. Software engineers needed to acknowledge and value the contributions of artists. This is most closely related to the concept of creating shared success in studies of software engineers, as well as the attribute of well-mannered (or not being 'an asshole'). However, most informants felt that current Microsoft teams did not have this problem, suggesting that this might be a carry-over sentiment from previous companies or previous times (i.e. prior to the advances in digital art).

Understanding of the art domain is closely related to being knowledgeable about people and organizations from studies of software engineers; however, the focus of our informants was on *mindset* and *language*. Artists had their own 'community of practice', with their own shared understandings and vocabulary; while much of the existing research on 'communities of practice' (E. C. Wenger & Snyder, 2000) has focused on cultivating communities within organizations, little attention has been paid to different communities *intersecting* and *colliding* in project teams. Our findings indicate experts from *different* communities of practice may have special needs when working together in close synchronicity. Our informants felt that it was essential for software engineers to be able to communicate effectively. All informants valued some aspects of the attributes is a good listener, integrates understandings, and creates shared understanding, which required software engineers to understand the mindset and language of

artists. The underlying sentiment appeared to be that games development involved many unknowns and uncertainties, and having constant communications was key for success.

The mental aspects discussed by our informants closely matched the attributes of open minded, willing to go into the unknown, adaptable, and hardworking previously identified by software engineers. However, needing this combination of attributes to work in game development might be unreasonable. In our survey study, one software engineer specifically discussed his disdain for (purposely) overloaded schedules of game projects as the reason for his low rating for the hardworking attribute (Section 4.2.1.2).

5.2.2 *Content Developers*

Most of the content developers we interviewed were technical writers. Three of the five content developers produced written content for Microsoft products, ranging from dialogs boxes in Windows, to MSDN articles, to technical manuals for Microsoft cloud computing solutions. One of the remaining two managed the content publishing *process* and the other was in a 'supply-chain' role, selecting TV, music, and movie content to display in Microsoft online stores. These last two content developers were excluded from our analysis; their inclusion was likely an address book title error. In the subsequent sections, we will focus on the perspectives of our three expert technical writers.

One of our informants was a writer by profession and wrote for magazines and newspapers prior to Microsoft; the other two informants were trained and worked as software engineers at Microsoft. Our informants all saw the content developer role as a bridge between engineering intent and customer needs. Our informants felt that software engineers and technical writers needed to work together to articulate the value of the software to customers. Furthermore, our informants felt that customers often did not know how to use the (often very complicated) software products, even the technically savvy customers like IT professionals. Therefore, technical writers produced necessary instructions and explanations that enabled customers to navigate and problems solve on their own. Out informants felt that, no matter how powerful or feature-rich the software product was, customers will abandon the product if they do not know how to use it. Finally, our informants also felt that they helped to voice needs of customers,

getting software engineers to clarify features that received many ‘how to’ searches on MSDN or CSS (Customer Service and Support) calls:

> *Customers don't buy the product for content. But they can end up really disliking the product because of content… Our product should be built in a way that customers don't need content at all, right? But we're not there yet. And so when they do need content, it needs to work for them. It needs to solve their problems. And if it doesn't, they're going to blame it on the product.*
>
> *-Senior Content Developer, Enterprise*

In addition to software engineers, writers often worked with program managers and CSS. Program managers were responsible for the overall vision of the project (program managers are discussed in detail in Section 5.3.9). Our informants commented that they commonly talked with program managers instead of software engineers to understand the intent of features, especially early in the software development process. Once the software product was released, our informants examined common topics of customer support calls in addition to automated data collected from online support sites (e.g. MSDN). Technical writers then used these data to work with software engineers to provide content—MSDN articles, support materials, etc.—to help customers properly use the software features.

5.2.2.1 Explaining the software product to customers

Our expert content developers worked with software engineers to produce a wide variety of content to help customers understand the software product. At the very basic level, technical writers reviewed and edited each display string—messages displayed by the software to the customer—to ensure that it both conveyed the intent of the software engineer and could be understood by customers. Technical writers also worked with software engineers to produce basic ‘how to’ information, typically shipped with the product or online. Our informants felt that this was especially important for consumer electronics and novel software products. Customers of consumer electronics (e.g. Windows Phone and Xbox) might have limited understanding of technology; therefore, our informants felt that describing features (and feature interactions) in a manner comprehensible for a lay-person was important for a successful software product. For novel software products (e.g. HoloLens), our informants felt that descriptions and instructions

were especially challenging because they commonly involve new features and interactions, sometimes requiring novel vocabulary and metaphors.

After the software product releases, technical writers continued to work with software engineers to tackle emergent issues. By monitoring feedback channels (e.g. MSDN, CSS), our informants worked with software engineers to select salient issues and produce information for various help channels (e.g. Knowledge Base articles, MSDN articles, etc.).

The most difficult content for our expert writers were technical instructions (e.g. setup guides) that required customers to have understanding of the complex system (e.g. cloud computing infrastructure for enterprise software products). The entire system may involve many sub-systems and many configurations that all need to be specifically configured to achieve the desired results. Our informants felt that providing a working understanding of the whole system and then explaining necessary actions within that complex setting can be highly challenging, frequently requiring significant communications with software engineers:

> *…the best type of interactions that you have is to get to that level that is actually useful and a lot of software developers don't know that level. So you have to then go in there and say, okay, what about this particular part and figure that out… you may have to corral or not corral them*
>
> *-Senior Content Developer, Enterprise*

5.2.2.2 Mindful explainers

While generally acknowledging that written content was an ancillary (but necessary) part of software products, our informants that felt that great software engineers treated technical writers (and their writing tasks) with respect. Great software engineers did not ignore or put-off requests from technical writers; they responded to technical writers' inquiries in a timely manner, meeting the timelines necessary to produce written materials (e.g. editing and legal reviews). Our informants found this to be especially important for software engineers working on products with fast shipping cycles, like online services. Our informants felt that the shipping cycles were so fast that writing the necessary supporting materials (e.g. setup instructions) needed to start at the same time as the coding; delays often caused scheduling problems. Great software engineers were mindful of timing and promptly provided the necessary information:

It's continuous because there are things coming out. [We] ship something every day. Literally every day, something comes out… you have to go in there and understand how something is going to ship, have an idea of that sort of thing, and then you put up something quickly…

-Senior Content Developer, Enterprise

Though none of the informants discussed this in the open-ended portion of the interviews, when shown the full list of attributes, all three agreed that technical correctness and coding competency was critical. Our informants assumed the attribute was a given. The sentiment was straightforward; if the feature was broken (i.e. not coded correctly), then no amount of words was going to make it better:

…if the code is broken, it doesn't matter what word I put.

-Senior Content Developer, Enterprise

Beyond making a good software product, our informants felt that great software engineers recognized, acknowledge, and respected that customers were unique. In most cases, most customers are less technically savvy with low computing self-efficacy. Customers (especially of consumer electronics) sometimes had trouble with seemingly simple tasks and features. Great software engineers were willing to work with the technical writer to ensure that their features could be understood by customers, even if the explanations seemed trifling:

…my job is literally to translate this stuff so that normal humans can understand and use the product. They're not developers either; they're less technical than I am by a long shot… a lot of people don't know what their browser is. There's a lot of really low technology people who are really uncomfortable with tech. So some devs get that and they're happy to have to help.

-Senior Content Developer, Content publishing

In other cases, the deployment context of the customer may be vastly different from the development context; therefore, features that worked within the development context would not work in customer environments. Our informants felt that great software engineers understood the critical pieces (i.e. specific components or activities that must be configured in a specific manner for the feature to work) and worked with writers to effectively communicate the important instruction to customers:

> *The best [software engineer] I encountered understood the pieces that were going to trip people up and actually proactively notified me of those. So while he was going through code he went, "Oh, yeah, somebody is going to stumble on this. Oh, yeah, somebody is going to stumble on this." … Understanding how their piece fits in with everybody else's pieces and then the scenario in which it will get used.*
>
> *-Senior Content Developer, Enterprise*

Furthermore, our informants felt that great software engineers needed to be open-minded to feedback and data from customers. They undertook constructive actions to understand why customers were having problems and to address the underlying confusion.

5.2.2.3 Discussion

Though almost taken for granted, the most important attribute of great software engineers from technical writers' perspectives appeared to be technical competency. Central to technical competence was the attribute of paying attention to coding details. Our informants felt that error-free code was *essential*, since no amount of documentation and explanation can compensate for a broken feature. Technical competency, as viewed by writers, was also related to several decision-making attributes. The sentiment was that great software engineers leveraged their technical competency to *avoid* problematic decisions (e.g. breaking existing workflows) as well as understood the implications and potential pitfalls. This allowed great software engineers to work with technical writers to create appropriate explanations and guides for customers. The growing their ability to make good decisions and seeing the forest and the trees attributes were closely related to this sentiment.

Informants also felt that great software engineers understood (or at least acknowledged) that their customers were not like them. Thus, they were open to feedback (e.g. from CSS and writers) about customer pain points, and to create content for seemingly obvious features (e.g. how to make a phone call). These were closely related to the attributes of knowledgeable about customer and business and open-minded. Our informants felt that these attributes were essential to *willingness* of software engineers to collaborate with technical writers.

Finally, all of the things that the technical writers reported to 'dislike' about working with subject matter experts (e.g. software engineers) in Mehlenbacher's survey of technical writers were reported in our study (Mehlenbacher, 2000). More than 15 years after Mehlenbacher's

study, technical writers in our study were *still* discussing problems with ‘time and accessibility’, ‘respect’, and ‘communication skills’ in collaborating with software engineers.

5.2.3 *Data Scientists*

Data scientists existed in many engineering teams across the Microsoft; however, they did disparate tasks. There was no congregation within teams or similarity of functions, as with many other roles (e.g. artists or content developers). The likely reason for this lack of uniformity was that ‘data’ pervades software engineering; ‘data’ can be used for many purposes depending on context (e.g. the software feature itself, the logs for analyzing usage, or the *target* of software features). Therefore, there was no simple grouping or explanation of the ‘data scientist’ role within Microsoft.

Furthermore, the role of ‘data scientist’ was relatively new at Microsoft. All of our expert data scientists had 10+ years of experience, but most did not start their careers at Microsoft as data scientists: most transitioned from software engineers or testers at Microsoft. Only one of the four expert data scientists we interviewed was hired by Microsoft as a data scientist.

Due to their disparate functions within teams, we will discuss all data scientists separately—their context, their engagement with software engineers, and their perspectives on great software engineers—instead of examining them together. We will distinguish the data scientists by their function within their respective software engineering teams.

5.2.3.1 Data scientist who engineered software

Some data scientists at Microsoft were essentially software engineers. This was the case for one of our informants who managed a team working on the Bing search page ranker. Our informant had both software engineers and data scientists reporting to him; all of his direct reports performed similar tasks.

Our informant explained that the organization converted him and some of his team to be data scientists because their jobs involved extensive experimentation. His team experimented with improvements to the ranking algorithm (e.g. for speed, for relevancy) and shipped successful improvements directly in ranking algorithms. All members of the informant’s team

performed the same set of tasks—formulating possible improvements, developing the software, and experimenting—because he felt that this arrangement expedited development and reduced issues that were 'lost in translation', where software engineers do not fully comprehend requirements defined by data scientists:

> *I'm used to our model, where data scientists also are engineers themselves. I think that works better… There's no handoff. Right? There's no interpretation… I think if you have scientists who can actually implement code and ship it, that's useful.*
>
> *-Principal Applied Sciences Manager, Web Applications*

In discussing engagement with software engineers, one informant referred to working with the platform team on infrastructural improvements. In those interactions, our informant's perspectives were very similar to those of other software engineers working with partner software engineering teams. He wanted software engineers to verify that they understood the requirements and iterated with his team to ensure that the correct software features were being delivered. Our informant discussed that some requirements may not be feasible and unforeseen issues can arise; therefore, he wanted software engineers to frequently communicate problems and to work with his team on appropriate solutions. This approach—constant communication of status, collective problem solving, and iterative delivery—is essentially the thinking behind both Scrum (Rising & Janoff, 2000) and Spiral (B. W. Boehm, 1988) software development methods.

Our informant also felt that software engineers obviously needed to be open-minded as well as pay attention to coding details. Our informant did not discuss these attributes during the open-ended section of our interview, but when seeing the list of attributes from our previous studies, commented that these attributes were obviously important:

> *I think pays attention to coding details certainly makes sense to be the most important one. It's like saying a plumber pays attention to what he's building when he's doing . . . Anybody who's in the business pays attention to one's details. It makes a lot of sense.*
>
> *-Principal Applied Sciences Manager, Web Applications*

Our informant singled out one attribute—data-driven— as very important from his perspective as a data scientist but which he felt was not given enough attention by software engineers (i.e. not ranked high enough in our survey). The desire for software engineers to be data-driven would a common theme among the data scientists that we interviewed. Our

informant felt that data driven was important because intuition-driven decisions can often be wrong; even collecting some basic data can help avoid costly mistakes:

> *I think data-driven is very low on the list, which surprises me… So I guess there's an opinion that measuring the software outcomes is not important, but I think that's extremely important. I think a lot of work you do needs to be data-driven. You can't just say, "Well, I have a feeling this will work."*
>
> *-Principal Applied Sciences Manager, Web Applications*

5.2.3.2 Data scientist who prototyped data features

Some data scientists at Microsoft prototyped data features that software engineers then implemented. One of our informants worked on anomaly detection for Microsoft online services. Our informant had degrees in mathematics/statistics, with work experience in finance prior to joining Microsoft.

The structure of his team was very similar to that of our first informant (data scientist and software engineer working together); he sat in the same hallway as the software engineers that he worked with. The difference was that the data scientists on his team did not write the production code; the data scientist worked with the software engineers to fully actualize the features. Our informant focused on this collaborative effort to fully implement the features that he prototyped in his discussions of great software engineers:

> *My neighbor is a software engineer and my neighbor's neighbor is a software engineer…We do models then we kind of close the gap between business and engineers. We develop strategies and then we figure out how we want to do [them]. Then software engineers, they really help realize our wishes, so we work really close.*
>
> *-Principal Data Scientist, Web Applications*

Our informant felt that, in his domain, great software engineers need to be very detail-oriented with a full understanding of behaviors of the software system. Since his features dealt with large number of transactions involving money, even minor issues affecting a small percentage of transactions could be costly for Microsoft. Therefore, great software engineers needed to fully understand the risks and consequences of their choices; our informant felt that, when problems can result in lost money for clients, an explanation of ‘I don’t know’ was not acceptable.

Our informant further described great software engineers in his domain as flexible and fast. Great software engineers understood that issues involving money needed to be fixed immediately, often outside of regularly planned development cycles. Therefore, our informant appreciated software engineers who were able to quickly fix (or at least temporarily patch) issues and were willing to adjust their development plans to accommodate unexpected interruptions.

> *When you are working on business, you actually impact the customers in real time. You cannot ask the customer, "Okay. We know that there's a bug. Wait for three days, we're going to fix the bug." It's not going to work… Sometimes it can be short term solution, but need to pay immediate attention.*
>
> *…They need to be very flexible. I know we have some release cycles and we have to do some code review and we want to make sure our work also has good quality. By it does not means we can slow down. It doesn't mean we have to follow step by step, without changing it a bit… you do have to get a little creative sometimes*
>
> *-Principal Data Scientist, Web Applications*

5.2.3.3 Data scientists who consulted on usage of the software product

Some data scientists at Microsoft produced software that reported on usage of the software product. Two of our informants built data processing and monitoring systems that took usage logs to report on the status of the software product. These were 'shadow' software systems; their value was in providing information about the actual software product and would not exist without the original. Nonetheless, our informants believed that having data on the state of the software system was essential for an organization to improve. These data would allow the organization to track progress, assess outcomes of investments, and identify new areas for improvement.

In this monitoring and improvement process, our informants felt that data scientists acted as consultants to software engineers of the software product. Data scientists worked with software engineers to clarify vague concepts (e.g. success and failure) and to instantiate them with concrete metrics. Our informants then created automated systems that enabled software engineers to track those key metrics and to assess effects of changes. Finally, our informants worked with software engineers to analyze the data and to develop actions to improve the software product:

…it's actually more that the data scientist is more on the engineer side to help improve the system…

.. analyzing the data, provide a daily scorecard. I provide a metrics that the developer can come and see whether their changes improved, with what they have done actually had made the system better. But at the same time data is used to improve our system automatically, programmatically, interacting [sic] with developer.

-Principal Data Science Manager, Web Applications

Our informants felt that great software engineers were open-minded and iterative, willing to listen to the data and make quick adjustments. Our informants felt that in order to make progress, software engineers needed to be interested in knowing and understanding more about their software product (e.g. how many customer did it have, who were these customers, what features were they using, etc.). The great software engineers were open to experimenting with all aspects of their software product and to empirically assessing the benefits and drawbacks of those changes. Our informants felt that great software engineers based their decisions on actual data rather than intuition. Above all, our informants felt that great software engineers, perhaps due to the fast-paced nature of online software, made changes continuously and quickly:

"I have idea [sic]. Don't see me for three months. In three months I will build a cool thing." No. I want to see it the next day, even better, tomorrow, a little bit, and more better. Can you get feedback about what you just built, just a little bit? Because every time we build something, every time we have feedback, we can say, "Tomorrow let's change. Let's change it." Now I am looking for this kind of engineer…

- Senior Data Scientist, Applications

Finally, our informants also wanted software engineers to be intimately familiar with their software products. Our informants felt their software product were complex; in addition to complexities within the software product itself, the engineering system (e.g. development branches with anomalous builds) or client-side software (e.g. browser refresh behavior and plug-ins) can also have idiosyncrasies that corrupt the data. Therefore, our informants wanted software engineers to, as much as possible, make error-free changes, and to proactively notify data scientists when changes may affect their monitoring and reporting data:

Data, it's very hard to be accurate. Data is very hard to be correct. I get garbage data almost all the time…

However, I work with some developers that are just incredible. I get mail from them. "Hey, I'm changing this today because of this. I realize the data I feed to you can be better. I make these changes." That's the best experience I've ever had.

…And they make my life much better. And the worse thing is, sometimes I don't even know the data is wrong, and I publish the data. I make a big business decision based on the data, and it can hurt. It can be millions of dollars because the data is wrong. So, yes, pay attention to detail!

-Principal Data Science Manager, Web Applications

5.2.3.4 Discussion

The fact that our expert data scientists wanted software engineers to be open-minded and data-driven was no surprise; those are the tenets of data science. The concept of experimentation was also central in our informants' discussions of great software engineers. Close to the mental attributes of continuously improving and willingness to go into the unknown, our informants' sentiment on experimentation was akin to a philosophy of software engineering. Rather than a 'build to last' mentality, our informants felt that great software engineers had a 'fail quickly' mentality—getting to the best answer quickly by iterating through variations. This may reflect emerging trends within the software engineering domain to better leverage data to construct software products (Economist, 2010).

Our informants wanted technical excellence mostly to ensure that their own features were correct and did not break. Their underlying sentiment was that of mutual dependence, likely related to the creating shared success attribute discussed in interviews with software engineers. Another contributing factor was likely the 'garbage in garbage out' problem. Data validation and data cleansing are commonly the most expensive and time-consuming parts of data analyses efforts (D. Fisher et al., 2012). Therefore, our expert data scientists' opinions may reflect the major pain points that they wanted software engineers' help to ameliorate.

5.2.4 *Design Researchers*

Design researchers at Microsoft are also called UX researchers, user researchers, or usability engineers. Throughout Microsoft product divisions, design researchers conduct qualitative research on customers. All of the expert design researchers we interviewed had advanced degrees

in psychology or sociology. Our informants felt that design researchers provided engineering teams with knowledge on the holistic experience of users with the software product.

Even though our informants did a variety of tasks, all had performed usability testing at some point in their Microsoft careers. Usability testing at Microsoft entailed bringing the intended customers in, letting them use the software feature or product, observing their usage, and asking them questions about their experiences. Conducting his qualitative research appeared to be the central function of design researchers at Microsoft.

In addition to usability testing, our informants performed a diverse set of user-centric tasks. One informant examined communities around user-generated content (UGC) in online games to understand user needs and to develop guidelines that would help the development of other Microsoft games. Another informant organized outreach initiatives to cultivate fans and to generate interest in Bing. Yet another informant oversaw consistency of user experience across Office applications when the application migrated to iOS and Android platforms. The common links between these tasks performed by our expert design researchers were their qualitative nature and their focus on users.

Microsoft, like many technology companies, is investing in 'big data'—the collection, analysis, and leveraging of customer telemetry data. Nonetheless, our expert design researchers felt that qualitative research will always be needed. Contrasting with the quantitative data collected by 'data scientists who consulted on usage of the software product' (discussed in Section 5.2.3.3), our informants felt that those behavior data— 'big data'—cannot explain intent. Data collected through instrumentation cannot inform software engineers *why* users are (or are not) performing certain actions. Our informants felt that design researchers provided that qualitative understanding to software engineering teams:

> *We do a lot of instrumentation where we can see what someone is doing, so it's behavioral data, that's through instrumentation…. It doesn't explain why. What we bring to the table is the why part of it or what is the intent… It's like "Aha! That's why someone wants it."*
>
> *-Principle Design Researcher, Web Applications*

5.2.4.1 Ensuring that users can actually use it

The most common interaction between design researchers and software engineers was assessing near-completed software features using usability studies. Many software engineering teams worked with design researchers to ensure that their end-users can use their software products as intended. Our informants felt that software engineering teams, especially the software engineers, needed to know how the intended user used their features; contextual factors (e.g. established workflows) and physical limitations like hand size that may affect usage. This knowledge was especially important for novel features or product, where typical usage patterns were unknown. The software engineering team might have a desired usage pattern in mind, but customers might not use the software product as intended. Our expert design researchers provided that information and often brought software engineers along to see how customers used their software:

> *But also kind of late in the cycle, once we've got to a stable alpha or a beta--we often, for a lot of products, you really need to see it in the user's own context… "Are people really going to use this new world? We need to see if we need apps to be available. We need people to be living with it at home for us to really understand how they're going to integrate that into their real world right now"…*
>
> *Any time we go out in the field or in the lab, we invite team members to come with us and that's when dev would have the opportunity to join in and connect with customers.*
>
> *-Senior Design Research Manager, Applications*

Our expert design researchers also conducted user research prior to the initiation of software project to understand user needs, and then worked with software engineers to develop prototypes to assess the viability of features. Our informant discussed going out to user environments to understand the users' context, their existing solutions, their needs, and any 'blockers' they were encountering. Information provided by this process helped software engineering teams make decisions about what software features to include in their software products. Based on this understanding, design researchers sometimes worked with engineering teams to iteratively build and test prototypes via usability testing. Sometimes, these prototypes were cardboard or HTML mockups; other times, these were interactive prototypes built by software engineers, which allowed teams to examine user interactions. This process helped

software engineering teams understand user reactions to various design options and allowed the teams to fine-tune their designs:

> *So that meant going out and learning about the current state of things, what people were doing in their working environments, what their needs were, what blockers existed to accomplish what they wanted to accomplish now. Bringing that back to the developers and having them start to prototype solutions to these problems then working with them to sort of hone those prototypes through user testing… help alleviate the decision-making tax, or your cost on decision about things they see at user end.*
>
> *-Senior User Researcher, Gaming*

Our informants stated that they generally work with program managers—PMs—(discussed more in detail in Section 5.3.9) instead of the software engineers. PMs were typically responsible for software features overall and interfaced with the other functional roles. Our design researchers indicated that they commonly provide information to PMs and worked with PMs on usability testing. Needing to work through an 'intermediary' would prove to be problematic for many roles, as the PM was sometimes a barrier to getting actual technical information from/to software engineers:

> *…we tend to interface more so with the [program manager]… owner of a feature or a product, and he or she then interfaces with different functional roles to deliver that feature ultimately. They are responsible for spec-ing it, and they are the people then who tend to become the project drivers for getting it out the door.*
>
> *-Principal Design Researcher, Web Applications*

5.2.4.2 Respectful collaborators

In discussing great software engineers, our informants focused on three detrimental attributes that great software engineers should avoid. The first was that great software engineers should strive to be data-driven and open-minded rather than not believing data. Our informants discussed software engineers feeling that users in the usability studies were not intelligent enough to properly use the feature as they had designed it. These software engineers then refused to adjust their software features to address the usability problems, feeling that 'dumbing down the experience' would compromise their engineering artistic integrity. Our informants assessed these poor software engineers as having 'self-referential' problems; they made decisions by referring to their own experiences and experiences of 'folks down the hall'. Informants felt that

great software engineers understood that their users may not be like them or their colleagues. Great software engineers accepted that they may not know what is best for their customers. Great software engineers were open to learning from the usability studies to improve their software features:

> *I think [this great software engineer] definitely took the approach that he didn't know best, that it really is our customers that we need to be understanding what is working for them, what isn't working, what their needs are and that they're right. Yeah, we might be smarter in different ways and understand, "Oh, if you only did it this way, you could be getting so much more productivity out of what you're doing," but that's not how the customer thinks about it. And it doesn't fit in with their approach or their life… He listens. He's open to input. He always was soliciting input from a variety of functional teams.*
>
> *-Senior Design Research Manager, Applications*

For this first detrimental attribute, our informants felt that an effective solution was to bring software engineers to usability testing sessions. Our informants discussed that software engineers got it once they saw intelligent professionals struggling with their software features. Seeing usability studies firsthand enabled software engineers to understand that usability study results come from smart people and the problems they encountered were due to the software features. This commonly resulted in gaining trust for the information provided by design researchers and facilitating future engagements:

> *Usually if you get a developer, even a mildly conscientious developer, even someone who has any pride in their work at all, into a lab on the other side of the glass watching users, you can usually break through to them… I was, in their words, testing secretaries. I wasn't testing smart people. Seeing a lawyer struggle with it, seeing someone who's paid more per hour than they were by quite a bit, not able to do it, get really frustrated, clicked with some folks.*
>
> *…Once someone comes on board and they want to do the right thing for users then usually it's going to be a good relationship.*
>
> *-Senior User Researcher, Gaming*

The second detrimental attribute discussed by our informants was that software engineers thought that qualitative research could be done by anyone. Some software engineers did not respect the design research role; informants discussed software engineers reading various articles or materials and then believing that they were qualified to conduct and interpret qualitative

research. This caused problems in collaborations because these software engineers would then question results and suggestions of design researchers, creating adversarial situations. Our informants firmly believed that, without extensive knowledge in qualitative research and a foundation in social science, interpretations would not be correct. They expressed frustration at the lack of respect:

> *My background is psychology, and behavioral research science is something they can totally just intuit… they can figure "I can learn languages" so they can learn what we do. So they'll come to a meetings and say, "I was reading yesterday about this principle and I think you're doing this the wrong way because XYZ. Sometimes they're applying the information right, sometimes they're applying it wrong. They're almost always applying it in too narrow a scope to understand the full context.*
>
> *Those folks are tough to deal with because they will devote a ridiculous amount of time actually trying to build a case against anything you're doing and ultimately at some point you have to deal a blow to them about that to deal with it, and it becomes an adversarial relationship.*
>
> *You wouldn't want me to go online and start trying to build code and suggest you inject that code into your code base… In the same way you might able to give me some information you've found someplace but you're not going to be able to put it in the full context of my expertise and you're not going to be able to put in the full context of understanding human behavior and psychology.*
>
> *-Senior User Researcher, Gaming*

Finally, our informants felt that some software engineers simply did not care. These software engineers were content 'checking off boxes' for their features; they did not take usability (usually an unspecified aspect of software features) into considerations and would not fix usability issues. Our informants felt that software engineers should consult design researchers on how users would approach specific scenarios to ensure that their features were usable; these discussions would enable software engineers to make better decisions about their software features. Furthermore, great software engineers did not neglect usability issues; they took the time to make refinements and corrections based on findings of usability studies:

…this sounds really glib but it's true. I've worked with developers who are lazier... they're checking off boxes, and they won't want to go any farther outside of that. The little bit of extra effort to make something work right…

They're smart people, these are smart folks, but they're not going to take the time to interpret through that intelligence and say, "Oh, what they probably meant here was, or what is likely here is."

-Senior User Researcher, Applications

5.2.4.3 Discussion

Above all else, our expert design researchers want software engineers to respect and appreciate the contributions of design researchers. This discrepancy may be due to the differences between the quantitative world of software engineers and the qualitative world of design researchers; software engineers may not understand the value of qualitative data. The notions of respect and appreciation of others are partly related to the knowledgeable about people and organizations and open-minded attributes discussed by software engineers (Section 3.3.2.5 and 3.3.1.2); furthermore, the manner in which our informants discussed poor software engineers suggests being well-manned (i.e. not being an 'asshole') may also relevant.

Problems with 'self-referential' decision-making are closely related to the knowledgeable of customer and business attribute. Great software engineers needed to base their decisions on knowledge about the intended user and not themselves. Avoiding 'designing-for-oneself' is the underlying concept of 'apprenticing with the customer' approach to software design (Beyer & Holtzblatt, 1995), and is one of motivating factors for Cooper in *Inmates Running the Asylum* (Cooper, 1999).

Interestingly, our informants rarely discussed problems with communication, unlike experts in other roles whom we interviewed. A likely explanation is that the well-defined nature of engagement—usability testing—simplified communication. The roles and responsibilities of designer researchers and software engineers were well defined, likely reducing the complexity of communications. Most of the problems expressed by our informants were around software engineers not believing the information.

5.2.5 *Designers*

Our informants characterized the purpose of designers at Microsoft as ensuring enjoyable user interactions with the software product. Typically, this involved two aspects: visual design and user interaction design. Most of our expert designers had backgrounds in art and graphic design, many stating that migrating to interaction design with the rise of the software industry was the logical career move. Designers are pervasive throughout Microsoft with higher concentrations in teams with user-facing software features (e.g. Bing).

Our informants felt that there was a general split between visual designers and interaction designers. Interaction designers focused on user interfaces, ensuring that users can easily use interfaces and can understand the information presented in those interfaces. As such, in addition to usability, our informants felt that interaction design involved 'information architecture': showing users the right amount of information, at the right time, and at the right place, to enable the users to accomplish their tasks without overwhelming them:

> *[Interaction designers] stick with the information architecture. They stick with the user flow and the wireframes.*
>
> *-User Experience Visual Designer, Gaming*

Our informants stated that interaction designers typically worked with 'wireframes', which are sketches that specify location, content, interactions, and workflows of user interfaces. Once the interactions and the user interfaces were finalized, visual designers produced the specific visual assets (e.g. images, logos, and CSS stylesheets) needed by the designs.

Visual designers made visual elements for the software product, including icons, logos, background, layouts, and even marketing materials; furthermore, they ensured that the software product overall was atheistically pleasing. While sharing some of the same tasks (e.g. creating visual assets) as artists (discussed in Section 5.3.1), in addition to visual aesthetics (e.g. color pallets, typography, style considerations, etc.) visual designers also focused on the usability of the artistic assets (e.g. user interfaces atop the images). For example, depending on whether the image was going to be rendered on a PC or a phone, visual designers might need to change the size and composition of the images. Visual designers took those factors into consideration in

designing the visuals, made different versions of the visual depending on the intended user contexts:

> *…what does this look like on a mobile device? Well, obviously you've only got that much room for a mobile device and you can see that the page is not set up here to scale… Does this work on tablet surfaces, because now, a tablet surface might be like that big. But there's also higher resolutions of tablets now, so what happens?*
>
> *-Senior UX Designer, Web Applications*

Our informants stated that designers transitioned fluidly between visual design and interaction design; most of the informants had filled both positions in their careers at Microsoft. Sometimes, especially in small teams, designers provided both visual and interaction designs, contributing whatever the team needed in the design space:

> *Something that I've found in this role particularly, it's very much a jack of all trades. So the ability to be very flexible, because the team size can oftentimes fluctuate… I end up doing a lot of different things... online advertising and design, even on XBox's splash page… in addition to the entire website design… not just coming up with the flow or the user experience that someone goes through when they're on the website, but then also all the images that go in there.*
>
> *-User Experience Visual Designer, Gaming*

Designers at Microsoft commonly had the same managers as design researchers and reported up through the same management chain; many of our informants reported working closely with design researchers. Both designers and design researchers focused on ensuring that users were able to use the software product. Design researchers commonly conducted usability studies to assess user reactions to the designs produced by the designers.

5.2.5.1 Crafting enjoyable interactions

At Microsoft, designers worked alongside software engineers to produce software features. The prevailing sentiment among our informants was that software engineers were responsible for what happened underneath the covers, getting the software feature to “work just right”, including reliability, scalability, and compatibility (e.g. across browsers). Designers were responsible for how users interacted with the software feature, conducting usability testing with design researchers and iterating on the design to ensure users are able to use the software features as intended:

> *Good developers will want to collaborate with designers because developers are all about what happens behind, in the code base, to make things render on the page. So they spend a lot of their life writing code… So they're not thinking about how users interact with the page so much, and that's the job of the designer. The designer is thinking through these issues, the designer is the one that's in usability lab, testing the prototype, working with user researchers to try to figure out why a user isn't interacting with the design so much.*
>
> *-Senior UX Designer, Web Applications*

Our informants discussed engaging with software engineers each step along the software development process. At the beginning of the software development cycle, designers worked with software engineers to create a shared understanding of the problem that the software feature is intended to solve. This mind-melding process was often aided by wireframes to communicate user interactions and results. Designers and software engineers iterated to arrive on the intended design, with designers proposing the ideal user interactions and software engineers providing input into the technical feasibility. While software engineers were developing code to realize the designs, the visual designers worked to produce the visual assets that developers need to 'plug-in' to the final software feature. In addition to periodic sync-ups to ensure that the design was being realized as intended, designers and software engineers would often have spontaneous interactions to clarify understandings about the design. At any time during the development process, designers might also take the current design (e.g. wireframes or prototypes) and work with design researchers to better understand design decisions and options. Based on the outcomes of those usability tests, the designer might propose adjustments to the design. Towards the end of the software development cycle, visual designers would provide the final artistic assets, and then the entire team might take a final pass through the product holistically to make final adjustments:

> *And then at the very end of the cycle, right before you ship, everybody comes back and looks, does a review of where it ended up. At that point in time, the visual designer is able to make small, little visual changes just to make sure everything renders right, and then you release it into the world.*
>
> *-Senior UX Designer, Web Applications*

Informants felt that the primary challenge facing software engineers and designers was reducing complexity of software features. Our informants felt that many software features involved large amounts of technology that would overwhelm and frustrate the typical user;

therefore, software engineers and designers needed to work together to design and implement software features that are easy and enjoyable for users to use:

> *To make intuitive user experiences requires obfuscating huge amounts of complexity from the user. And that is almost always on the backend. The frontend's gonna have one button on it… it's removed eight. Those eight buttons were all really important, but we can do it with one. How do we do it with one? That complexity is really critical, and that's hard.*
>
> *-Senior Design Lead, Applications*

5.2.5.2 Deferential creators of shared understanding

The dominant sentiment among our informants was that great software engineers left design decisions to designers. Great software engineers respected the design discipline, and did not think that they could do the designers' job. Our informants felt that great software engineers understood that designers had specialty knowledge and experience that enabled them to produce designs that provided users with enjoyable experiences. Therefore, great software engineers left design decisions to designers and focused on realizing those designs through technology. Our informants felt that designers would not tell software engineers what code to write, so software engineers should not tell designers how to design:

> *An important attribute is when developers also respect the expertise of designers, understanding that there's a time for feedback… but for them to also defer to designers when it comes to the design and the user experience.*
>
> *Because just as a user experience designer will not tell a developer how to do their job, so too should a developer be very respectful of the designer's position and their years of expertise in the field.*
>
> *-User Experience Visual Designer, Gaming*

Many informants discussed bad experiences with software engineers that lacked respect for designers. Some software engineers, when encountering problems with design, would produce fixes without consulting designers. Our informants stated that the design would usually be suboptimal; the software engineers had neither the design knowledge nor the (hundreds and hundreds of hours of) experience observing actual users interacting with interfaces. Software engineers' solutions commonly did not fully consider all aspects of the users' interactions and needed to be reworked. Also, some software engineers felt that designers only made things 'look pretty', and did not include designers in the designing of software features. Those software

engineers would come to designers towards the end of the development process and tell them to "put some UI on it"; this commonly results in software features that were unusable and required substantial redesign. Our informants felt that great software engineers respected the design discipline and engaged/consulted early and often with designers on the right designs:

> *A lot of times what happens is, we will get a team that has done a lot of development work and they will say, "Hey UX guys, can you put some UX onto this app or feature?"… that makes it really challenging for us because a lot of times the functionality is really awesome, but a user might not understand it or understand how to use it… "Well you guys need to rearrange this whole thing."*
>
> *-Senior UX Designer, Web Applications*

Our informants also felt that great software engineers did not rush into coding, rather they took the time to fully understand the problem and worked with designers on the best design trade-offs. Our informants felt that great software engineers thought through problems and were open-minded when considering input from designers. Great software engineers did not rush into coding before understanding user goals and the nuances of their needs (e.g. common cases and edge cases). They worked with designers and design researchers to arrive at designs that best accounted for all scenarios, instead of blindly rushing to code solutions. The consequences of rushing ahead without sufficient design consideration were poor or incomplete designs that were unchangeable (or too costly to change) and that resulted in suboptimal experiences for users.

> *What happens a lot of the time is…we will be presented with a project, there is a timeline suggested, and we end up getting dev involved in building code too early. So what design is doing is, we are doing design and research that may actually go against what is being built and it becomes this awkward…we are saying you actually need to change it based on our user research, but they have already invested time into it so they don't want to change it or it's already too far along… the end product doesn't meet the users goals or they are not able to use it as easily as they should be if we had done that upfront research.*
>
> *-Senior UX Designer, Web Applications*

Our informants also appreciated great software engineers that were willing to try new designs and go through multiple iterations to perfect design ideas. Our expert designers all wanted to work with software engineers who were open to trying new or different designs: ideas that had not been tried before or might be not have been technically feasible previously. Since software engineers were usually not obligated to try new designs, our informants valued software

engineers that were willing to "try something new", leveraging their knowledge of the latest advances in technology or of workarounds to overcome technical challenges:

> *Developers know what works and what is technically feasible and it's easy for them to just say no we are not going to do that. It's great if they say well, that will be technically difficult but we can look at it and see and look a little further into it… "Yeah, we are excited to do something cool so we want to help you make this cool thing." That's the kind of developers I like to work with.*
>
> *-Senior UX Designer, Web Applications*

Our informants also felt that designing (especially for new designs) was a process; designers often needed to create a design, assess user reactions, and then refine their design. Therefore, our informants appreciated software engineers who were willing to work with them through multiple versions to arrive at the best experience for users. This typically involved many iterations of minor adjustments or fixing interactions at edge cases. Our informants felt that great software engineers were willing to "sweat the details and try to polish" for a better software product.

Finally, our informants felt that great software engineers worked to clarify understanding. Great software engineers paid attention to details and recognized missing elements or inconsistencies in designs. They would then immediately raise questions to get clarity on these inconsistencies. Our informants felt that designers and software engineers may have different interpretations of the problem; therefore, to avoid divergent efforts that solved different problems, great software engineers would get clarity—often using face-to-face meetings—to ensure that everyone operated toward the same goal. Our informants appreciated software engineers who did not proceed based on assumptions, but rather they made consistent and ongoing efforts to reach shared understanding with designers:

> *…there were a couple of developers that really understood how to pull information out of you. So if they didn't understand something, they would drill in deeper and get more clarity to the point where there wasn't any ambiguity so you both knew exactly what was expected for the best outcome.*
>
> *And that's probably one of the most important aspects of the process, is just being able to come together and sit down and talk through things. And the more you can sit down and get clarity upfront, the more successful the outcome is going to be at the end.*
>
> *-Senior UX Designer, Web Applications*

5.2.5.3 Discussion

Designers had strong feelings about software engineers respecting their expertise and not trying to 'do their job'. The underlying cause might be that some roles (e.g. artists, electrical engineers, and mechanical engineers) performed functions that software engineers readily recognized were outside of their scope of expertise; some roles were not deemed essential (e.g. content developers and service engineers). Design of software features appears to be an important task that some software engineers believed they could do themselves; however, many of our informants believed that software engineers could not do it effectively.

Deferring to experts is related to the sentiments behind self-reflecting and asks for help; both involve recognizing one's own limitations and seeking help from others. However, letting designers make the design decisions had the added elements of willing to trust others since the design decisions often directly affected the software feature's success. The 'lack of respect' from software engineers may not be malicious but rather reflect the desire to control the situation.

'Not rushing ahead' is related to the attributes of systematic and grows their ability to make good decisions. Systematic software engineers did not rush into actions, rather they took the time to be circumspect about how they should proceed. This matched our informants' sentiments that software engineers needed complete understanding of the problem space, including compromise within the design, in order to make the best engineering decisions. Our informants also hinted that various engineering decisions were not 'robust'; once made, they precluded various design options. Software engineers rushing ahead with actions that 'locked' the team into poor designs was one central problems discussed in *Inmates Running the Asylum* (Cooper, 1999). Therefore, obtaining more information (especially information from designers) helped great software engineers reduce the likelihood of mistakes.

Finally, our expert designers emphasized establishing and maintaining shared understanding, which might have been due to the highly coupled and time-sensitive nature of the tasks performed by designers and software engineers. Our designers expressed sentiments that included all aspects of being a good communicator found in our prior studies of software engineers—is a good listener, integrates understanding of others, and creates shared

understanding with others—along with maintaining shared understanding. Other roles with shared timelines and interdependencies (e.g. electrical engineers and mechanical engineers, discussed in the next two sections) shared this opinion of needing to be in 'lock step' during development.

5.2.6 *Electrical Engineers*

Our expert electrical engineers described their role at Microsoft as "making physical things with electrons flowing through them" which were typically circuitry inside Microsoft consumer electronics (e.g. Xbox and Surface). Our informants indicated that some of their fellow electrical engineers also had 'hardware engineer' titles; however, our informants drew a distinction between electrical engineers and 'mechanical engineers' (discussed in the next section), who built physical components that housed the electronics.

Our informants specialized in a variety of tasks involving hardware. Some architected circuitry: selecting, positioning, and diagraming chips, processors, ports, power supply, wiring, etc. on circuit boards. Some informants prototyped and debugged circuits, working-out the kinks in designs prior to full-scale manufacturing. Some worked on specialized chips that performed special electronic processing. All of our expert electrical engineers had degrees in electrical engineering.

Many of our informants reported working on 'matrixed' teams. To produce a product, electrical engineers worked on 'program teams' that included program managers, mechanical engineers, electrical engineers, and software engineers; these were the core disciplines that collectively *engineered* the product. In addition to their program team, electrical engineers also reported up through their own specialties (i.e. leads and managers of electrical engineers were electrical engineers). The sentiment among our informants was that their fellow electrical engineers were best able to evaluate and help their work. Our informants reported having their designs reviewed by peers who worked on other program teams but understood electrical engineering:

> *Now, think of a matrix structure. You have a vertical structure, which is your specialties… But each of those electrical engineers will be assigned to a different program team. And if you look horizontally across that matrix, that would be a program*

team. You have these vertical columns where you have job structure, job features like firmware, like electrical, like mechanical. Then horizontally across, there would be a program and the program forms for the duration of shipping that product. You'll be assigned to a program, you'll collaborate amongst not only the different departmental peers to ship this product but when you need help, you go into your vertical column of your tree and you start asking your peers for their help for what they might have done in the past… When you have design reviews, you drag in all the people from your department and other departments and they all look at your design and critique it and give suggestions to make it better.

-Senior Electrical Engineer, Devices

Our informants stated that the software engineers they interacted with were predominantly 'embedded' software engineers (also referred to as 'firmware' engineers). Though embedded software engineers tended to be on the same team and had the same titles as other software engineers, our informants differentiated embedded software engineers as those that developed code residing on the hardware, whereas other Windows software engineers developed code residing on the operating system (including drivers and applications). These embedded software engineers programmed the chips to process the electric signals and translated operating system commands to electrical outputs. Our informants reported working with embedded software engineers; several of the informants exclusively dealt with embedded software engineers and had no interaction with Windows software engineers. Our informants did not consider embedded software engineers to be electrical engineers because embedded software engineers did not design the circuitry; they were given the circuitry and were responsible for writing code that 'enabled it'. Nonetheless, our informants felt that embedded software engineers needed extensive knowledge of electronics and hardware, far beyond the knowledge needed by Windows software engineers:

It could be something like sampling registers, checking for button presses, reading the data from an optical engine and then doing something with it… That's the embedded firmware software aspect of it. There's a tight coupling of system level design architecture between the hardware folks, which is myself, and the embedded firmware folks.

-Senior Electronic Engineer, Devices

5.2.6.1 Working as a collective to engineer the consumer electronics

Our expert electrical engineers worked on program teams with software engineers (and other experts, such as program managers and mechanical engineers) to produce consumer electronics.

Some of the products that our informants worked on included the Xbox (encompassing the Kinect), HoloLens, the Surface (the tablet/laptop, not the table computing device), Microsoft Keyboard/Mice, and Nokia phones.

The development process typically started with the program team assembling to examine the high-level requirements for the product. At this point in the project, the people involved are typically very experienced engineers. This group worked together to define the scope of the program, choosing the functions and features to deliver as well as the choices in hardware and software to deliver them. Many of these choices were tightly coupled. For example, one informant detailed how the electrical engineer's choice of the CPU (ARM, proprietary, Atom, or Intel) had implications for the software engineer in terms of memory capacity and memory speed, affecting the overall functional capabilities of the product. Our informants indicated that the engineers making these decisions were usually senior engineers (e.g. technical leads) because they were making difficult decisions with many considerations, amidst great uncertainty, and sometimes years in advance of product development:

> *A product kicks off and marketing folks come up with "We need a product X to do this particular feature." You start out at the system level. From that you get your design team involved to try and architect. That would be your mechanical engineers, your electrical engineers, your firmware engineers, and sometimes your software engineers will come together and figure out what pieces you're required to achieve that marketing goal of product X.*
>
> *…the senior people on the team that have experience in doing that, they take a system level role at the very beginning when you have a marketing concept for a particular product. They start architecting the system architecture for it. And then that iterates multiple times and when it gets to a point that it seems feasible, then you start getting more team members involved to break it down into actually more tactical, executable blocks.*
>
> *-Senior Electrical Engineer, Devices*

Due to cost and physical constraints, many of the program team's decisions were difficult compromises and trade-offs. Our informants felt that the program team needed to thoroughly think through the choices to ensure that they considered and addressed as many of the potential problems as possible ahead of time. This typically involved thinking through the necessary parts and implications associated with each choice; the choices commonly affected multiple disciplines. One informant used a scroll wheel on a mouse an example. Mechanical engineers

would choose the physical component based on physical dimensions and functional requirements. Electrical engineers would then take into consideration how the data comes off the component (e.g. optically or electronically) to decide how to read and transport the data to the microcontrollers. The embedded software engineers would then decide how to communicate the data to the PC, including considerations for efficient algorithms and sampling intervals. Whenever the program team encountered an issue, the team would have to figure out where an adjustment—mechanical, electrical, or firmware—should be made to best address the issue.

After the program team set the plans, the individual disciplines independently fleshed out and produced their own parts; nonetheless, our informants indicated that the program team continued to work closely throughout the development process. In addition to periodic sync ups, the program team would usually come together during major milestones (e.g. prototype complete) to verify that the product was functioning as expected and that the project was progressing on-schedule. One informant described software engineers as 'concurrently' developing code to exercise the hardware that he was designing, based on design documents and reference platforms, so that the prototype could be tested on-time:

> *And it's not I close my door to design circuitry, once I have my circuit design on the paper. I assume it will work. Then I throw over to software guys to write the code… before I get my hardware, software guy is already working…by reading my schematic, by looking at my layout… And they also use off the shelf reference platform…*
>
> *When [the] factory builds the first prototype right off the production line, software guys will try to load its code into my prototype. Once it's loaded the hardware guys will take it over. Say okay let me fire it up. Let me start to verify of all the function for each module step by step. Software guys sitting next to him. Every time he looks at how they probe the signal, they look at how they verify the functionality, and provide real time feedback because if this doesn't work the team has to work together to see what causes failing… So you can see that integrational interaction is real close. This is the only way to make it work.*
>
> *-Director, Devices*

As alluded to in the quote above, our informants stated that they commonly worked with software engineers to overcome unforeseen issues during development. Our informants felt that the best products were almost always trade-offs; many choices were physically impractical or commercially infeasible. Depending on the desired functionality, a cheaper chip may be perfectly sufficient and better than a more powerful choice; however, these compromises sometimes lead

to unforeseen problems. One informant discussed an unforeseen issue in an audio controller that restricted the device from muting completely, which required software engineers to address the problem in the codec firmware. Our informants felt that to hit market ‘price points’ they often had to pick hardware that was less than ideal, and then worked with software engineers to fix problems caused by the selection and to deliver the desired functionality:

> *I think the biggest burden then drops onto the software guys or firmware guys…where they become more and more responsible to keep the cost down. So it's anything you can't do in hardware like, "Oh, can you throw that in software?" They’ll be trying to figure out now that they had their code written in a nice format…and now they're hacking in new pieces to fulfill problems we have.*
>
> *-Electrical Engineer, Devices*

5.2.6.2 Hardware-speaking system thinkers

All of our expert electrical engineers felt that great software engineers should be able to ‘speak hardware’ and have good understanding of both software and hardware domains. Great software engineers that our informants worked with had knowledge of the vocabulary and nomenclature of electrical engineering, which enabled them to effectively communicate their needs and requirements, as well as understand explanations and feedback from electrical engineers. These great software engineers also had a working understanding of the hardware domain, which allowed them to understand the limitations and the capabilities of available hardware, and work within the expectations and processes of electrical engineers. Our informants felt that in order for software engineers and electrical engineers to work together effectively, they needed to be able to have deep technical discussions; consequently, software engineers (and electrical engineers) needed to understand each other’s language and have shared understanding of both software and hardware contexts:

> *The really great ones have a really good understanding of both, the software and the hardware. Like I said, figuring out the limitations of what the hardware can do and what it can't do and asking the right questions and phrasing it right to get it either in software lingo or hardware lingo. You know, the people that can do that are fairly rare…there's different terminologies and different expectations.*
>
> *-Senior Architect, Devices*

Our informants discussed both positive and negative situations in which the software engineer's knowledge of the 'lingo' and understanding of both software and hardware domains were important. The most common complaint among our informants was the lack of understanding of the physical limitations of hardware (e.g. power). Our informants mentioned frustrating situations when software engineers asked for functions that were 'ridiculous':

> *Sometimes people can get stuck on something that physically isn't going to work… Well, there is a defined of what bandwidth you get, here is, this is physically what it's going to take… and just physically not possible… Don't wait on your plans without looking at the reality of your situation.*
>
> *-Senior Architect, Devices*

Conversely, our informants also discussed software engineers having mistaken knowledge about the limitations and capabilities of hardware. Because the software engineers were unaware, they did not ask for available capabilities and functionalities, leading to inferior products. Our informants felt that software engineers should have meaningful exchanges with electrical engineers to fill knowledge gaps to decide on the best choice of components and functions. The sentiment was that software engineers needed to be knowledgeable and needed to continually update their knowledge; great software engineers knew the limits of hardware, and would constantly update their understanding.

Since the products were *composites* of hardware and software, the problems that the program team encountered could commonly be solved by either software or hardware, but with different compromises. Recognizing the options and understanding the costs and benefits of *all* options enabled great software engineers to select optimal choices. Informants felt that the lack of holistic understanding often resulted in inferior products despite similar hardware (e.g. worse battery life, performance, etc.). Our informants felt that great software engineers worked with electrical engineers to efficiently solve issues:

> *Sometimes what's very hard to do in software, for instance, is very easy to do hardware. For instance, one of the things we worked on, call it some accelerator block that did this and it took many milliseconds to do in software. And we're like, "Well, we can just do this one little thing and that's really what you want? Okay, that's easy to do."*
>
> *-Senior Architect, Devices*

A direct corollary, our informants appreciated great software engineers that helped out when issues could be better resolved within software. As discussed in the previous section, our informants admitted that they frequently had to design hardware "that was not purely the best from an electrical perspective" because of required compromises (usually cost). They mentioned working with great software engineers to overcome limitations of the hardware to deliver on the system-level requirements. Our informants felt that great software engineers recognized the holistic nature of their product and worked with electrical engineers to make the best collective system. Great software engineers were not myopically focused on their software parts, but rather, worked with the program team to deliver the best system under constraints:

> *There's no perfect hardware. There's no perfect software. How do you make your software to make my hardware perfect?... So it's help each other, working together.*
>
> *-Director, Devices*

Another common complaint among our expert electrical engineers was the lack of knowledge about scheduling and development constraints of electrical engineering. Our informants stressed that the hardware domain operated with very different timelines, and therefore, software engineers who did not understand the differences were difficult to work with. Our informants stated that hardware is significantly less malleable than software. Once the electronics were 'burned', changes and fixes could take months, if they were possible at all. One informant explained that designs can take months to produce, fabrication (which sometimes took place overseas) can take weeks, and testing can be another several weeks; a typical hardware cycle may take more than three months. Furthermore, changes and fixes were also expensive due to the materials and electronic components involved (many of which were prototypes). On the other hand, our informants felt that software changes and fixes were fast and cheap—recompilations were essentially free and could be done in less than an hour. Our informants felt that software engineers that had similar expectations about hardware were often not successful, because the hardware modifications they wanted did not fit within project timelines. This often resulted in incomplete or poor quality products that would require future product iterations to fix:

Since [the software engineers'] programs are very malleable, they can make changes up to the last second, right? … And it's hard to get across, "No this is fixed. Once it's burned it's not going to change."

-Senior Architect, Devices

Related to the scheduling issue discussed in the previous paragraph, our informants also discussed needing to be able to trust the information provided by software engineers, especially scheduling estimates. Due to the highly interdependent nature of their disciplines, electrical engineers needed software engineers to provide accurate estimates of their deliverables. Software delays can delay testing and have rippling effects on future deliverables. Our informants felt that they should be able to trust great software engineers to deliver their parts on schedule:

So we need very specific dates from them early on in order to provide a schedule so that we can meet our dates… when the chip needs to complete… we're trying to meet a product cycle.

So we don't even know if we're gonna do the chip yet and it's usually based on when we get this [software] stuff. So if we don't have any of that with these dates and deliverables that they guarantee then it could mess up the whole course of the chip.

-Hardware Engineer, Devices

Finally, our informants had interesting perspectives about the code produced by great software engineers. Our informants felt that coding problems close to the hardware can be very difficult to isolate and debug, often causing significant delays in development. Therefore they wanted great software engineers to produce error-free code. However, they also wanted great software engineers to know when to break the rules to deliver a better product. Our informants discussed scenarios in which doing what is 'correct' resulted in efficiencies, and stated that they wanted software engineers to make appropriate compromises. One informant discussed diagnosing differences in battery life differences between two products to find that the firmware engineer on the better product was delaying writes to the disk—not correct, strictly speaking—so that the writes can be synchronized and be more efficiently performed:

Those ones who's willing to break the rules. Not because they were told that's not the right way to do it… You have to be able to think outside of what you're educated.

…the app want to access the SSD send the request over OS say okay pass over to the driver, driver say oh yeah you get it, okay. Another app send request over so that your SSD is busy all the time. The downside is like you consume more power because your system is never in the low power state.

-Senior Electrical Engineer, Devices

5.2.6.3 Discussion

By all our accounts, electrical engineering was very different from software engineering. Electrical engineers worked on electrical components that were significantly harder to change than software, their development cycles were much longer, and they had a different language—typically the names of components they worked on and the terminology associated with working with those components. For software engineers to work effectively with software engineers, our informants felt that software engineers needed to have a working understanding of electrical engineering. This sentiment resembles the attributes of knowledgeable about people and organizations, knowledgeable about the technical domain, and creates shared understanding; however, it goes beyond the sentiment expressed in interviews with software engineers. Knowledge and understanding in the context of working with electrical engineers involved knowing and *technically* understanding *electrical engineering*, an entirely different technical field.

Knowledge and understanding of electrical engineering underlies most of the other attributes of great software engineers discussed by our informants. Being able to have meaningful exchanges with electrical engineers, to decide on appropriate trade-offs for the product, to have reasonable schedules, and to collectively solve system-level problems all required software engineers to understand electrical engineering. The need to have deep understanding of another technical field was more pronounced for electrical engineers than for any other group of experts that we interviewed.

5.2.7 *Mechanical Engineers*

The two mechanical engineers that we interviewed were both in the Xbox division; one worked on Xbox controllers and the other worked on the Kinect. Initially we included ‘hardware engineers’ as ‘mechanical engineers’; however, after completing interviews, we found that ‘hardware engineers’ were ‘electrical engineers’. We included ‘hardware engineers’ with ‘electrical engineers’, leaving us with only two mechanical engineers. The low number of mechanical engineers was not an issue, as mechanical engineering was a well-defined discipline at Microsoft. Mechanical engineers worked on ‘anything that has a physical embodiment that ends up in the customer's hands’. As with electrical engineers, mechanical engineers were concentrated in divisions that made consumer electronics (e.g. Xbox).

The two expert mechanical engineers that we interviewed both had mechanical engineering degrees and had prior work experience at other electronics manufacturing companies prior to Microsoft—HP and Xerox. At Microsoft our expert mechanical engineers designed the parts, the plastics, the metals, and the stuff that goes around the PCBs (printed circuit boards), and even how big the PCB is.

5.2.7.1 Translating the physical to the digital

Our expert mechanical engineers reported working mostly with ‘firmware’ engineers; their infrequent interactions with platform-level software developers usually occurred through program manager (PMs, which are discussed in Section 5.3.9) intermediaries. We focused on engagements with firmware software engineers in our discussions. As with electrical engineers, our informants reported working with software engineers in program teams to produce consumer electronics.

Our informants discussed collaborating with software engineers at three time points. First, before the product is conceptualized, mechanical engineers worked with software engineers to prototype and to experiment with the latest advances in technology. Since mechanical changes often required corresponding firmware changes, one informant reported working closely with his software engineer counterpart to rapidly try new mechanical changes. Second, our informants discussed coming together with all the disciplines (e.g. electrical

engineers, PMs, legal, industrial design, etc.) at project initiation to scope and schedule the engineering effort. As described for electrical engineers in Section 5.2.6.1, this process usually involved collectively making trade-offs for the product. Finally, our informants discussed working with software engineers towards the end of the project to resolve last-minute issues, which, after the physical product was finalized, could be address in firmware:

> *One thing that happens often at the end of a project when mechanical design is pretty much firm, electrical design is set, but you find a bug that has to be solved and you have no time, it's often on the firmware team's shoulders to process those signals or do something in a certain way, add a new algorithm maybe, that will clean something up or resolve an issue in a very short amount of time.*
>
> *-Senior Mechanical Engineer, Devices*

5.2.7.2 Informed action-takers

The sentiment among our informants was that great software engineers were doers: just execute and make it work. One informant discussed a great software engineer that was able to quickly turnaround a firmware change that enabled him to test a fix, bypassing some typically standard processes to produce a revised version of firmware. Another informant discussed a great software engineers that came up with a firmware workaround for a hardware problem to enable progress while the hardware fix was going to take a month or two. Furthermore, our informants felt that some problems can often be overanalyzed and some outcomes are impossible to forecast; therefore, to make progress, great software engineers were willing to take forays into the unknown:

> *To do innovative products you need people willing to take the risk. And make the calculated judgment, get the data they can and make a decision… they may not know all the details of how do we execute it before the decision is made, but they're willing to do it anyway.*
>
> *-Senior Mechanical Engineer, Devices*

Our informants felt that though great software engineers were willing to just ‘do it’, they made sure that their decisions were *informed*. Our informants discussed several aspects of being informed. Foremost, our informants felt that great software engineers knew their own domain

and did their jobs well because they understood that mechanical components were useless without firmware:

> *I don't know how clean his code has to be to fit in the memory space that we have on the [system on chip] or I don't know how he has to optimize it for latency. All of that is obscure to me, or opaque to me. But I know the better he is at writing code for small spaces or low latency, the better he understands his tools, the quicker I can get my deliverable into my role.*
>
> *-Senior Mechanical Engineer, Devices*

Second, our mechanical engineers wanted software engineers to understand mechanical engineering, specifically, the language, the constraints, and the timelines. For example, one informant discussed wanting software engineers to know that the mechanical hardware development cycle was much, much longer than that of software or firmware. Our informants felt that having some understanding of mechanical engineering facilitated and expedited communications; software engineers and mechanical engineers could quickly understand each other and avoided unrealistic expectations. Great software engineers intuitively knew the limitations and capabilities of the system for making and communicating decisions. Finally, our informants felt that great software engineers wanted to be informed by asking questions, soliciting feedback, and being open to new information that might change their thinking.

> *Open, honest, and trusting conversation. I used that word "relationship" earlier… having people who intend to communicate openly, honestly, collaboratively, that's what that interaction should be like. That's the most productive, the most innovative way that can go.*
>
> *-Senior Mechanical Engineer, Devices*

5.2.7.3 Discussion

Overall, sentiments of our expert mechanical engineers matched those of our expert electrical engineers. This was expected as mechanical engineers and electrical engineers engaged with software engineers in the same context, working together on consumer electronics. Furthermore, mechanical engineering, like electrical engineering, is an entirely different engineering field with its own domain-specific considerations and constraints. Therefore, many of the same engagement issues like understanding each other’s language, understanding constraints, and effectively communicating are shared concerns between electrical and mechanical engineers.

As with electrical engineers, it is questionable whether expecting software engineers to have in-depth *technical* understanding of another engineering field is reasonable; however, some luminaries feel that this may be expected of engineers. David Parnas said in his opinion piece on software engineering programs: “Licensed professional engineers often take responsibility for some complete product, which means that they require extensive knowledge outside of their engineering specialty. A mechanical engineer might have to do some electrical power design, or an electrical engineer might have to look at the mechanical aspects of a motor or servo-mechanism” (Parnas, 1998).

5.2.8 *Product Planners*

One informant aptly described the function of product planners as providing engineering teams with understanding about the ‘five Cs’: *customers*—who they are, what they want, and why they want it, *company*—core strengths of the organization and the executives’ vision of the future, *competitors*—not just direct competitors but also substitutes, *collaborators*—partners that can help, and *context*—relevant trends in the market, in society, and in the technology domain. Informants felt that product planners do the leg work to bring that business information to software engineers—typically the “engineering leadership team”—enabling them to make decisions. Product planners were typically in ‘an advisory role or a consultant role’ to software engineering organizations:

> *Are we gonna go invest in this or in that? Are we gonna make this a higher priority or a lower priority? ... in many respects you're sort of an advisory role or a consultant role to the leadership team who can do the leg work to kinda bring the data to bear and help free things up for the decision-making, right?*
>
> *-Principal Product Planner Manager, Enterprise*

Our expert product planners came from diverse educational backgrounds and worked in a variety of areas. One was trained as a bioengineer, started working at Microsoft developing medical imaging software, and had been a program manager; one had graduate degrees in statistics and political science; another one came to Microsoft directly from business school and had been in marketing. Our informants worked in a variety of areas, ranging from software products (e.g. SQL, Phone, Devices) to specific feature areas (e.g. Education).

Our informants said that product planners typically did two types of work: overall market intelligence research or specific research for a feature. Our informants described overall market intelligence as a strategic role, focusing on the overall direction of the market and the technology area rather than individual features. One informant reported using primary and secondary research to provide revenue opportunities and projections for various markets:

> *…through a significant amount of market analysis, looking at syndicated research reports, things that are talking about the market space in general, where the opportunities are, doing market opportunity analysis and market sizing exercises to say, "This is where we think the volume or the revenue play is going to be,"*
>
> *-Principal Product Planning Manager, Devices*

Our informants also discussed conducting targeted market research to guide feature development. One informant gave an example of doing research to support a 'classroom orchestration software'. The informant described researching specific needs of teachers, like 'turn off these devices and have all eyes on the teacher' and 'temporarily block applications or websites'. The informant also described understanding value propositions of various features to teachers: which features are 'table stakes' and which ones are 'differentiators'. In addition, the informant recounted providing information on competitors and their planned features, as well as helping to make difficult trade-offs—keeping the features that will attract customers to the product—toward the end of release cycle. While some of our informants focused on one discipline, others reported doing both kind of market research for their organization:

> *Think of my role as 50/50. Fifty percent is focusing forward looking. So I'm sitting there saying "These are trends that I'm seeing based on behaviors that are happening today in the market and things that you ought to be thinking about…"*
>
> *Fifty percent of my time is with engineers that are currently building products today, who are sitting there saying I'm wrestling with this feature… I only have enough resources to build one of those things. What's the thing I should build? And is anybody else building it? Does it care? Is it a table stakes, will it really matter?*
>
> *-Principal Product Planning Manager, Applications*

With their focus on understanding customers, product planners share similarities with designers (discussed in Section 5.3.5) and design researchers (discussed in Section 5.3.4); however, the focus on business questions is unique to product planners. Product planners not only looked at

the needs of the customer, but also examined the ‘market’ around that need, including alternatives offered by competitors and revenue potential.

5.2.8.1 Deciding on engineering initiatives for next release

Product planners typically advised engineering leadership teams, designed to help those that decided the overall direction of the entire software engineering *organization*. As such, product planners typically work with very experienced software engineers, those entrusted with making decisions that affect the actions of other engineers.

Our informants discussed interacting with engineering leadership teams in various ways; all were variants of providing information about the *business* implications of their decisions. Informants stated that engineering leadership teams usually understood the limits of technical feasibility— ‘what we think we actually can do’. The product planners then helped to provide the business context around those engineering choices— ‘what do we think we can achieve from a business standpoint to make money’. Sometimes, the business context provided by product planners had implications beyond feature choices; one informant discussed strategic decisions affecting the structure and composition of the engineering organization:

> *…Not only how we build the products but how we're going to set up our structure and strategy. How many devs we’re going to put on those new tools and instead of having enterprise devs who modify something… we need to have dedicated devs that are doing something very differently.*
>
> *-Principal Product Planning Manager, Applications*

Beyond strategic decisions, our informants also discussed working with software engineers on tactical decisions during development. Our informants felt that once the engineering leadership team made the strategic decisions, frontline software engineers often still needed to make tactical trade-offs—usually selecting between features—that required understanding of business implications. Software engineers enlisted the help of product planners to understand the business implications of the trade-offs aiming to meet business goals while keeping on schedule:

We decided to invest in here, but then when you go and do it, there's choices that still need to be made. And so there's that interaction with them about well, do we do it this way or do we do it that way? Can we trade this off for that, or not?

And particularly if schedule comes into it, it's like okay well, this doesn't fit. What can we give up to do that? And so, you know, we're trying to keep the overall strategy and business model whole while still working with the practicalities of what's technically feasible…

a lot of times those decisions get made with some high level technical feasibility but of course once you get down into the details of it, other things start to come up, right, that weren't really considered at the top level…

-Principal Product Planner Manager, Enterprise

Several informants also discussed taking software engineers on "field trips" to talk to customers. Our product planning experts felt that, in addition to secondhand understanding provided by product planners, software engineers also needed firsthand knowledge of customers in their element to "really get a sense of what's going on". This helped to protect against having understanding only based on the software engineer's surroundings—'the 98052 problem', zip code of Microsoft's main campus in Redmond, WA.

One informant described taking software engineers to the CES show in Las Vegas to see the real products and to talk to the developers to understand their thinking:

I would love to take a couple of developers on a field trip to CES… most of them are stuck here at Redmond all the time… why they're building it, how it came to be, what tools are they using, what frustrations they have, why are they choosing open source, where did they learn about it, how did education in their university impact their decision to do things differently in a way.

…I think is super valuable for developers to have firsthand experience… They're getting it secondhand from us, but I'd like to see more of the firsthand integration taking place. That would be good.

-Principal Product Planning Manager, Devices

5.2.8.2 Impactful decision makers

With nearly all of the engagement between product planners and software engineers coming through the course of decision-making, nearly all of the attributes of great software engineers discussed by our expert product planners concerned effective decision-making. Most of our

informants acknowledged that technical skills were ‘obviously’ important, since buggy code causes ‘all kinds of problems’ for other people on the team; however, our informants felt that great software engineers went beyond being excellent coders. Great software engineers understood the business context and reasoning around the code they were developing, which enables them to make better holistic choices for the business success of their products:

> *They understand the full picture…they have an understanding of why the product is being created, for whom, what kind of problem it's solving and how it does differentiate against the competition…*
>
> *So, most senior people have the larger view of the big picture. And the big picture is how does the product fit in with the business impact? Why are we doing this specific product? How does a specific feature make that product more viable for the end user or more competitive against our competitors or more realistic?*
>
> *-Principal Product Planner, Devices*

Our informants felt that fully understanding the context was central to making effective decisions. By context, our informants generally meant *qualitative* understanding of customers, including their habits and their motivations behind behaviors. Our informants felt that understanding customer pain points was essential to creating software products that customers would buy. Our informants further believed that this knowledge needed to be thorough, not simply anecdotal. Great software engineers had a nuanced understanding of the market, and where variation existed, they understood the differences. The overall sentiment was that unless great software engineers understood the context and the reasoning behind their software product, their software product might be technically excellent but not commercially successful:

> *So it really doesn't matter if you're an amazing engineer and you can write stuff and you're open minded if you have no idea what the market is telling you and you built a crap product.*
>
> *-Principal Product Planning Manager, Applications*

Though our informants acknowledged that software engineers were often—correctly—focused on ‘building the damn product we need them to go build’, our informants felt that understanding the ‘why’ behind the engineering plans enabled great software to build better software products. Our informants discussed great software engineers as using their technical domain knowledge and their understanding of the ‘why’ to suggest enhancements—beyond what

product planners had envisioned—that increased the product's value proposition to customers; also, our product planners appreciated great software engineers using that knowledge to suggest courses of action that were robust to future technological developments:

> *Great developers are the ones who add two extra layers to their thinking. One is the full understanding of the business value of the work that they're actually doing. And the intent of the business oftentimes helps them understand why it's so important to do things, which leads to the second thing, which is really creativity. Once you've got an understanding of why we're trying to accomplish something, their ability to get creative around solutions*
>
> *-Principal Product Planning Manager, Devices*

On the flip side, our informants felt that great software engineers should be open-minded. Numerous informants discussed frustrating situations when software engineers refused to take input from product planners. One informant described a bad experience: a software engineer insisted on his approach 'come hell or high water'; when other members of the team asked for clarifications, the software engineer resisted, telling them, "That's not your problem. It's mine. Let me go get it done." Our informants felt that even though software engineers wrote the code and decided 'what goes in and out', they should not exploit their position of power; they should be open to letting others influence their decisions. Informants felt that with so many "different paths to solving the problem" software engineers should not be rigid in their thinking. Not taking input from other experts and not considering alternatives may lead to an inferior product and may burn a bridge with colleagues that prevents future collaborations:

> *…they understand one way to do things, and that's the way that they want to approach it, and they're going to go off and go do it that way come hell or high water. And even when people ask for more clarifications so they can understand, like, "Why are we doing it this way?" there are developers who just resist that and say, "That's not your problem. It's mine. Let me go get it done."… they look at themselves as being the end game and recognize that we have nothing if we don't have people who can write code. Therefore, because they're the ones who write the code, they get to just make the decisions of what goes in and out. And those are the ones who, I think, are really challenging.*
>
> *-Principal Product Planning Manager, Devices*

Our informants further explained that being open-minded allowed for diverse perspectives and ideas, leading to a better product: "create something that is better as a whole, with all of our ideas." Great software engineers worked with others to combine and integrate

ideas to arrive at better solutions. Seeking a heterogeneous set of perspectives and taking feedback from other experts helped great software engineers to avoid ‘blind spots’ or problems that they could not have or had not considered themselves. Our informants felt that being open-minded also avoided confirmation bias. Software engineers commonly depended on their own experiences, which caused them to be selective about data that they paid attention to, selecting those that fit their perspective. One informant described a situation where a software engineer had experiences with their own children and treated their observations as fact. The informant felt that while personal experience was valuable, a great software engineer understood that his personal experience was a single data point and was willing to supplement his understanding with differing data from other people and other sources:

> *I think one of the things that all of us, myself, suffer from is something that I call confirmation bias. Meaning you start to see the environment that you're in and you start to look at things from your lens… well my child is in this school and we experience this, therefore that's the answer. When developers have observation or opinion and they treat it as fact, bad things happen.*
>
> *And so [a better situations is] when you have devs that are willing to engage to say it's a piece of information that I need to make a decision…and bring in their own information too, but looking at that holistically is where I've seen developers make the best trade-offs.*
>
> *-Principal Product Planning Manager, Applications*

Another attribute that our informants felt was common among great software engineers was the ability to effectively communicate complicated technology concepts. Our informants felt that the ability to “take a really complex concept” and explain it to others, such as product planners, enabled the entire team to better understand the situation and to arrive at better and more cohesive decisions. Our informants felt that software engineers had the best understanding of the technical reasons for taking various courses of action (e.g. advances in technology will require costly rework in the next release); therefore, sharing the logic behind the decision was critical, improving the understanding of others and facilitating goodwill among the team. One informant described this sharing as the inverse of data gathering activity of product planners; great software engineers needed to disseminate their knowledge so other team members could use the information to better plan their activities and to better negotiate with others:

> *It's almost the reverse if you're an engineer … that V-team is made up of a bunch of people who are looking for the most efficient way to go get something done, and they may*

not all understand all the technical details of what actually has to happen. But if you can communicate that clearly to them, there's going to a much higher probability that they're going to listen. They're just going to go, "Oh, yeah, okay. Now I understand why we need to do this. Now I can go help defend to my management team why this project is a three-week project instead of a one-week project," right? ...And [other team members] have to be able to share that out to their management, to leaders, to whoever, and convince them of that as well.

-Principal Product Planning Manager, Devices

5.2.8.3 Discussion

In our interview study of software engineers, we concluded that effective decision-making was critical for software engineers, especially those in leadership positions (see our discussion in Section 3.4). Sentiment from the expert product planners who mostly worked with software engineers on 'engineering leadership teams', reinforced our previous findings. Attributes of great software engineers discussed by our informants spanned nearly all of the attributes associated with making effective decisions in our previous study—knowledgeable about their technical domain, knowledgeable about customer and business, grows their ability to make good decisions, updates their decision-making knowledge, mentally capable of handling complexity, and sees the forest and the trees. The attributes not receiving much discussion by our expert product planners—knowledgeable about tools and building materials, knowledgeable about software engineering processes, and knowledge about people and the organization—were not surprising as those attribute concerned everyday technical execution of software engineering projects, which our informants rarely observed.

Interestingly, most of the attributes not directly related to effective personal decision-making, were related to enabling effective team decision-making. For example, our informants highlighted that great software engineers integrated and built upon ideas of others as well as disseminated technical knowledge; both behaviors are aimed at better team decisions. While these behaviors were ostensibly for creating shared success, the sentiment of our expert product planners was that these behaviors helped to improve others' decision-making.

5.2.9 *Program Managers*

Of all roles that we interviewed, the largest group, by far, was program managers (PMs). As described previously (Section 5.1), we studied 3,411 senior level PMs, the most populous group

among the expert non-software-engineers; the next most was ‘service engineer’ at only 506. Nearly every software engineering team at Microsoft had a PM.

Most program managers at Microsoft were ‘feature’ PMs, working closely with software engineers to produce the software product; we focused on these ‘feature’ PMs in our analysis. However, some experts, performing specialized non-engineering tasks, also had ‘program manager’ titles. One of the PMs we interviewed was responsible for working with governmental security agencies world-wide to ensure trustworthiness of the Windows platform. The PM discussed working with governments to provide assurances that Microsoft software had high integrity and security: “there’s no backdoor, there’s no non-declared functionality.” We also interviewed ‘process’ program managers; some managed relationships with OEM partners (e.g. Dell and HP), exchanging technical data that facilitates development and maintenance of Microsoft software running on the OEM’s hardware.

Among expert non-software-engineers, feature PMs worked the closest with software engineers in the development of software products, both for pure software products and hardware products with software components (e.g. consumer electronics discussed in Section 5.3.6 and Section 5.3.7). With the exception of data scientists (discussed in Section 5.3.3), the feature PMs we interviewed had offices in close proximity to the software engineers with whom they worked. In contrast, many other expert non-software-engineers (e.g. product planners) sat with experts of the same role, located in buildings away from the software engineers that they worked with.

The distinguishing characteristic of the expert PMs that we interviewed was technical knowledge about their software product (relative to all roles except expert data scientists who engineered software); almost all of our informants had deep technical knowledge. Most of our informants had degrees in computer science or their specialty area; many had prior software development experience. One informant stated that being a PM at Microsoft was not merely being a ‘schedule jockey’; effective PMs needed to understand the underlying technology in order to manager programs effectively:

> *…a weak PM, it's a PM who doesn't have some technical skills. They may be awesome communicators but they cannot really understand some architecture, some technical architectures, some backend features, read code, maybe sometimes do some scripting for analysis. That's not the developer's job to help to unblock the PM with every single question.*
>
> *-Senior Program Manager Lead, Applications*

The role of program management at Microsoft is seemingly simplistic yet complex in practice. At a high level, PMs were responsible for ensuring the success of the project; however, the specific tasks performed by PMs varied greatly between teams.

5.2.9.1 Setting and executing the vision

As explained in the book *Showstopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft* (Zachary, 1994), program management at Microsoft originated to allow software engineers to focus on coding. Consequently, many non-coding software engineering tasks in the Microsoft engineering teams were performed by PMs. Our informants stated that PMs owned "the end-to-end experience for a scenario, a feature set, a product... defining what that looks like and actually driving it with the development and quality teams in order to deliver." At a broad level, program management at Microsoft was a combination of requirements definition and prioritization (tasks commonly performed by software engineering leads and managers at other organizations), scheduling, budgeting, and tracking (tasks typically performed by project managers at other organization), as well as communicating and coordinating with other teams and experts. One informant compared being a PM to being to a CEO:

> *Sometimes I'm thinking of myself as CEO because I need to really oversee all the disciplines, including marketing there as well. So I also have close connection with the engineer here but… It's really to say there's one team. There's no one directly reporting to me, working with me as a crew…*
>
> *-Senior Program Manager, Devices*

Though the exact set of tasks performed by PMs varied greatly, our informants discussed several common themes. In the beginning of the software development cycle—referred to as the 'conception' phase of the project—our informants discussed setting the vision for the project, establishing what the software engineering effort should accomplish. Our informants discussed

looking at the market and customer pain points to identify gaps, and then proposing features to close that gap. Some of the specific tasks mentioned included triaging user-reported bugs, conducting user and market research, gathering input from other experts, composing the business case, and drafting solutions. Many informants discussed producing a "functional spec" that described the business case, the customer scenarios, the proposed solution, and the requirements for the solution. However, our informants stated that the functional specifications did not include technology choices; rather, it described what the new experience should be, leaving the 'how' to the software engineers:

> *So on that final solution, identify the problem, quantify it, write a functional spec of what the user wants to see, and propose a solution to the dev team. The solutions should not be a technical solution... It's more like, this is how we think it should be. And the devs usually own the implementation side. They own the actual solution for the problem we are proposing.*
>
> *-Senior Program Manager Lead, Web Applications*

Our informants discussed working closely with software engineers on the functional spec and the subsequent technical decisions. For specifying functional requirements, our informants discussed consulting with software engineers to ensure that the proposed requirements were feasible, since software engineers would ultimately have to implement software features to meet the requirements. One informant discussed giving "a really early heads up" to the software engineers so that they can raise any concerns to forestall future problems. For technical choices, our informants discussed ensuring that all the right questions are asked, enabling software features to meet requirements in the most optimal manner. For example, one informant discussed a choice between using of an existing technology and building one of their own; the informant asked questions about consistency of user experiences across products, the cost savings of leveraging an existing solution, and risks associated with taking on a dependency. The informant stated that though software engineers may have preferences for writing their own code, PMs helped to ensure that "all of the right questions are being asked and that the customer experience is being thought through" when looking at different technical choices.

The second theme was creating schedules and managing timelines. Our informants coordinated with their software engineers and, more importantly, partner software engineering teams on timelines to deliver the requirements on time. These coordination meetings would

commonly involve senior software engineers and PMs from all of the teams. The key was to ensure that all of the partner teams agreed to the plan (with a schedule, deliverables, and milestones):

> *…you can pull together your hardware team that consists of electrical engineers, mechanical engineers... an audio engineer…optical engineers… industrial design engineers… We work with partner organizations over in [Windows] then that do the software development, that do the shell or the drivers, the user, the UI.*
>
> *-Senior HW Program Manager, Devices*

The PMs—frequently working with PMs and senior software engineers in partner teams—subsequently tracked and monitored progress along the schedule. This commonly meant periodic 'status update' emails and sync-up meeting to keep the stakeholders informed, as well as to discuss issues or problems that come up. Informants discussed "driving dependencies with other teams" to ensure that the assets developed by partner teams (e.g. hardware engineering) met expectations and to 'unblock development on both sides'. In addition to daily activities, our informants also discussed doing various milestone tasks, such as preparing demos and presentations for executive reviews, setting business metrics for measuring success, filing patents, and ensuring compliance with various policies (e.g. accessibility and internationalization):

> *Everything… helping to define the metrics around the product, making sure that it's set to ship from an international perspective, getting it through compliance and actually shipping it out the door.*
>
> *-Principal PM, Applications*

The final theme was facilitating communication between teams, commonly acting as the "translator" for software engineers. Among the many functions of PMs, this function—acting as the intermediary between software engineers and others—was the most controversial. As alluded to in the previous paragraphs, much of the communications between teams are mediated—sometimes conducted completely—by PMs. This aligned with the purpose of PMs freeing up software engineers to focus on coding. PMs helped software engineers by translating user needs into the 'lingo' of the software engineer so that they understood the requirements:

> *…[my developers] are both PhD. So my key role is how to make sure I can communicate and manage a project in a way talking all the researchers [sic]. At the same time I can talk with the product team to make sure they understand, "this is what it means for user." -- transfer of the different language, research language into product language. And the second thing is most kind of PhD researcher not really having a scheduled timelines [sic].*
>
> *-Senior Program Manager, Devices*

PMs were the primary points of contact between many expert non-software-engineers and the software engineering teams. For some, this bridge was beneficial because PMs helped facilitate conversations with software engineers, providing clarity about the questions being asked. However, for others, PMs were considered a hindrance. The PMs blocked access to software engineers—the people who had actual technical answers—for the purported reason of keeping software engineers focused on coding.

5.2.9.2 Vocal prognosticators

With their focus on facilitating successful completion of software engineering projects, our expert PMs emphasized attributes of software engineers that helped to avoid problematic plans and deviations. The general sentiment among our informants was that software engineers often had critical insights, during plan formulation as well as during execution, and therefore, needed to 'speak up' in order for the project to be successful:

> *Be blunt and honest with me. Tell me how it is, why it is… I want to know it upfront. If it's sugar-coated, you can't address it in as timely manner as probably as needed or in a direct manner as probably as needed… That can later come back and cause more problems than good.*
>
> *- Senior HW Program Manager, Devices*

Our informants felt that great software engineers commonly had the best and most complete technical knowledge about how to implement the desired functionalities; therefore, PMs appreciated great software engineers that spoke up during the planning phases to help avoid bad choices. Our informants expressed the sentiment that, too often, software engineers get recognized for 'fighting fires'; however, PMs preferred to work with great software engineers "who prevent the fires before they even start". Our informants felt that great software engineers foresaw challenges, asked the right questions, and helped PMs plan the project to avoid

problems. Our informants wanted great software engineers to articulate ‘what it’s going to take’ to implement desired features, helping PMs who commonly only had ‘black box’ knowledge. Our informants felt that by providing the implementation details, even at a high level, great software engineers helped PMs to set more realistic schedules. Great software engineers also asked key questions to ensure that the important decisions were well thought through. Our PM informants readily admitted that they often did not have the most in-depth technical knowledge and needed great software engineers to help ask questions and suggest better solutions:

> *Part of [the software engineer’s] job is to make sure that you understand the technical hurdles and realistically what it's going to take to deliver so that you're not over promising, so that you do understand whether or not you can deliver what you're promising to the customer… if there's a challenge that's going to make it difficult to deliver that maybe it's not the right feature or the right time or the right implementation.*
>
> *-Principal PM, Applications*

Our informants also felt that great software engineers knew about changes in the technology domain, underlying technologies, and supporting components (e.g. a rewrite of an existing component). Therefore, providing expertise helped PMs avoid risky or problematic technology choices during planning. Our informants depended on great software engineers to provide the full information (e.g. latent dependencies and points of contact) to allow PMs to fully scope and manage the software engineering effort. For example, one informant discussed great software engineers knowing dependencies ‘from the code level’ that PMs may not realize, which may include ancient code ‘touching ten years ago’. An overall sentiment among our informants was that software engineers needed to question choices for the good of the project and to be ready to communicate why certain choices were optimal. Our informants felt that questioning decisions—especially working across disciplines, such as electrical engineers and mechanical engineers—led to better products by avoiding choices that lazily followed previous decisions. Furthermore, our informants discussed PMs had to justify choices and answer difficult questions about the team’s decisions (often with future of the project in the balance) when presenting plans and updates to management and executives (e.g. during milestone reviews). Therefore, our informants felt that hearing challenges and explanations internally first—gaining a better understanding of the ‘why’—helped them to better represent the team:

And, if we don't challenge amongst ourselves, amongst the functional teams, when we go up to our management reviews, they're vicious, they're brutal. Our [General Manager] and our VPs, it's their job not to hold back. They need to challenge us and question us. And if we don't have appropriate answers, then we failed….we need to be able to all understand and come together, here's what we're doing and why we're doing it…

-Senior HW Program Manager, Devices

In addition to the planning phase, our informants also felt that software engineers needed to be vocal during the execution phase. PMs wanted software engineers to speak up when something could jeopardize the schedule. Our informants discussed numerous disasters where software engineers did not mention that they were 'blocked', leading to bad surprises that delayed project timelines. Our informants explained that an important function of PMs was 'unblocking' software engineers; however, PMs could only do so if software engineers communicated issues and problems. Great software engineers knew when they were on track and when they were blocked; they readily communicated issues and enlisted help to address issues out of their control (e.g. waiting on actions from another team). One informant discussed a frustrating experience with a software engineer who would constantly only reply with "yes, we're making progress" without any details for over a week, causing the team concern about meeting the schedule and consider cutting an important feature. Our informants felt that great software engineers were transparent about their progress and any issues that they are encountering; this effective communication allowed stakeholders (e.g. their manager and PMs) to properly assess the situation and to prepare backup plans when needed:

[Great software engineers] were able to articulate the problems that they are seeing in the system, and follow up on them…And we have methods and we have ways to do it. We have a daily scrum. You should just go surface these things there, just don't sleep on them…there is some level of transparency between the devs… they minimize risks and they surface risks and the PM or the dev manager can have a backup plan. The more you identify these problems early in the process, the better. If you just keep them as surprising issues at the end, nobody is able to handle them. When the plane is landing, you cannot just go say, "Oh, the engine is not working now. Oh, I knew about it a week ago."

-Senior Program Manager Lead, Web Applications

Aside from proactively communicating important information, our informants also felt that great software engineers were willing to adapt and to continuously improve. One informant discussed a good experience with a software engineer that was willing to learn how to program iOS apps to help the team. Even though the software engineer was not familiar with iOS

programming (being a Windows programmer), the software engineer was willing to “try to learn and just help keep this going.” Our informants felt that great software engineers saw changing needs as opportunities to acquire new skills or to try new things, and admired the attitude and aptitude of great software engineers to continuously learn and improve. Our informants had low regard for software engineers that refused to stray from their comfort zones and felt that software engineers that were intransient quickly ceased to be useful to the team:

> *They have the attitude to learn more. They have some hunger to learn more and help. And also, they have the ability to learn… People shy away from some machine learning problems and some people just actually say, "You know what? I am on this team and that's an opportunity for me to learn machine learning." And some people say, "You know what? I am not a machine learning guy. And I don't even have interest to learn machine learning," …If the team is about machine learning, you can either become flexible and learn this new technology or get out of the team.*
>
> *- Senior Program Manager Lead, Web Applications*

Our informants also felt that the desire and willingness to improve helped the software products. Great software engineers were constantly seeking to improve customer experiences and to leverage better technologies. Our informants felt that great software engineers were not locked into ‘doing it their way’ and were willing to make the necessary changes to evolve the software product. Great software engineers were passionate and excited to improve upon the status quo:

> *I see passionate [sic] here…I go to talk a dev about an idea. I really want to see them jumping up and down with me about the idea. … "Just think with me about it." The desire to turn ideas into reality is kind of the entrepreneurial thinking.*
>
> *- Senior Program Manager Lead, Web Applications*

Though none of our informants mentioned technical excellence in the open-ended portion of interviews, nearly all pointed to the importance of technical excellence when presented with the entire set of attributes from previous studies. The sentiment appeared to be that having solid technical skills was a ‘baseline’. Most of our informants felt that all the software engineers they worked with were technically competent, ‘otherwise they cannot come to Microsoft’. At the end of the day, software engineers were the ones touching the code; therefore, software engineers needed to be able to write solid code in order to successfully deliver the software product:

It's very important especially in this company, software engineer is really touching the code [sic], touching the product closely… if I'm just touching this block because the whole thing is broken.

-Senior Program Manager, Applications

5.2.9.3 Discussion

The important attributes discussed by our expert PMs involved many aspects of 'effective communicators' discussed by software engineers: is a good listener, integrates understanding of others, and creates shared understandings with other. In addition, the sentiment among our informants was that great software engineers were proactive with their communications. Our informants wanted software engineers to be forthcoming with their information, not simply communicating when elicited. This may be due to fact that PMs, though technically astute, often did not have in-depth or full technical understanding of their software products; hence, they often were not even aware of various options and possible problems. Therefore, in making planning decisions and managing risks for the team they needed input from software engineers.

The literature on project managers in software engineering teams commonly discussed risk management for the team (Barry W. Boehm, 1991) (Ropponen & Lyytinen, 2000). Our findings indicate that an important aspect of risk management—not receiving much attention in the literature—may be having great software engineers that proactively provide needed information. They would enable project/program managers to take better and faster remedial actions.

In other attributes discussed by our informants, proactive action was also evident. The desired attribute of software engineers informing the team when they were 'blocked' has elements of managing expectations and asks for help discussed by software engineers; in addition, it reflected PMs desire for software engineers to initiate communication. The need for software engineers to continuously seek improvement often emerged and echoed the continuously improving and desires to turn ideas into reality attributes; both reflecting the sentiment that software engineers should be self-motivated to take actions.

5.2.10 *Service Engineers*

The role of service engineering at Microsoft is undergoing change; we discerned three kinds of 'service engineer' among our four informants. One informant was a 'network architect', managing a team that built network 'topologies' that ran Microsoft services (e.g. Azure and Office 365). Our informant stated that since Microsoft did not have a network architect role, rather, he (and others like him) was given the title of 'service engineer'. Two informants ran IT operations, working where software engineers in a DevOps model—an software engineering practice that emphasized collaborations and communications between software engineers and IT profession to automate and expedite the process of software delivery and infrastructure changes (Roche, 2013). The final informant developed services for internal teams.

In our examination of service engineers, we will discuss the perspectives of the three kinds of service engineers that we interviewed separately.

5.2.10.1 Network architect

One informant managed of a team of services engineers that "build the networking infrastructure inside the data centers for Microsoft's online services"; he stated that in other companies he would be called a network architect. Our informant stated that his team developed the network designs on top of which Microsoft services team developed their product, including Office365, Azure, and Skype. Our informant described working with software engineers to design the software and then, acting as the customer, deploying it see if it worked as expected in practice:

> *We also do a lot of testing and piloting with them, so early in beta sort of times, they'll give us builds of the software, we'll put in our labs here, we'll test it, make sure it works the way we think it's supposed to and give them feedback on that, too…They'll give it to us and we'll test it and give them feedback on it, just like what a customer would do during a beta trial.*
>
> *-Principal Service Engineering Manager, Enterprise*

Our informant discussed service engineers and software engineers at Microsoft having an uneasy relationship. He lamented that many Microsoft software engineers did not view networking as a 'discipline' and lacked respect for service engineers. Our informant felt that this was misguided because networking required a specialized set of skills: "if you don't have

experience building a network that will support 500,000 servers, you will do it wrong." In addition to interpersonal issues, our informant also discussed industry trends whereby software engineers threatened the future of the service engineering role. He stated that software engineers were displacing 'network architects' by writing software that automated the deployment process:

> *In the Cloud space, specifically, like the three big, Google, Amazon, Microsoft, the career of a network architect or engineer is diminishing to be replaced by a software engineer because the networks are so large that they can't be done by a person anymore. They have to be written, they have to be automated in code.*
>
> *-Principal Service Engineering Manager, Enterprise*

With the tension between software engineers and service engineers as backdrop, our informant discussed several attributes of great software engineers. Our informant felt that great software engineers—those developing software services—understood that the network had great impact on the quality of their service; therefore, they had working knowledge of their infrastructure and worked closely with service engineers to ensure that their services work as expected. Our informant described an example where the network topology of an existing service for one team made adoption by another team difficult; he had to implement a temporary work-around and work with the first team on a permanent solution to "line up the physical infrastructure and logical infrastructure".

Our informant also felt that, in order to work effectively with service engineers, software engineers needed to be willing to take feedback. Our informant felt that great software engineers did not simply "hand it to us and walk away"; they included service engineers early in the design process, earnestly listened and discussed problems they raised, and then worked with service engineers to improve the product after deployment. However, our informant felt that many software engineers were not open to ideas for improvement, especially from service engineers:

> *being able to accept feedback from folks that...our experience as an infrastructure team is often the software engineers sort of think that because it's software it's a higher tier and more important, and don't often either take criticism well or advice like, "If you wrote your software this way, it would work much better on this infrastructure."*
>
> *-Principal Service Engineering Manager, Enterprise*

5.2.10.2 IT pro

Two of our informants were in IT Operations, which deployed, tested, configured, and monitored the software services. One informant discussed deploying services to data centers, and then setting up the underlying failover and backup settings. The other informant discussed running tests and monitoring services running on "BigIron routers, which are these hulking beasts that carry terabits of traffic" to ensure that the service is working as expected on production firmware under actual load.

In describing engagement with software engineers, both informants used the 'DevOps model' to describe the relationship. Our expert service engineers worked with software engineers to quickly and frequently deploy iterations of the software services to production. In addition, when failures occurred, service engineers worked with software engineers to isolate, debug, and resolve the issues.

As with the 'network architect', our two IT pros reported incidences of condescension from software engineers and highlighted 'mutual respect' as an attribute of great software engineers. Our informants appreciated software engineers who were willing to take input and feedback on improvements from service engineers based on their experiences from daily operations. One informant expressed the desire to be able to go to developers with problems that he was seeing and have a frank back and forth discussion about the issues. Our informant felt that, regardless of the question, the software engineer should not be dismissive and think, "Oh, that guy's an idiot." Our informants felt that there will be instances when the engineer—software or service—does not understand the situation (e.g. a feature requirement or how a component works); therefore, great engineers needed to be willing to ask questions as well as provide explanations. However, both our informants felt that the software-engineering-centric culture at Microsoft often put service engineers at a disadvantage in engagements.

Our informants also discussed understanding the bigger picture as an attribute of great software engineers. Our informants felt that great software engineers understood that software services encompassed the software, the infrastructure, and the daily operations of the software service; therefore, software development did not end when the code is done. After the software

has been ‘released to the web’ and customers start using it, software engineers should work with operations to find bugs that escape during development. Furthermore, great software engineer thought about the holistic impact of their software features, not simply the coding aspects. One informant described a bad situation when a software engineer only thought about completing his feature but ignoring the security implications when put into production, viewing it as a concern to be ‘thrown over the fence’ at service engineers. Our informants felt that great software engineers took concerns of entire services into consideration:

> *The [software engineering] guys that we interface with, that I think do a fantastic job of being able to communicate the requirements is they understand not only the code, but they understand, at least high level, what the network side of it needs.*
>
> *-Senior Service Engineering Manager, Enterprise*

5.2.10.3 Internal services developer

One of our informants was a solution architect designing SharePoint solutions producing custom SharePoint services for internal teams, effectively an IT developer. The informant described the confusion with her title as a result of multiple team mergers and title transitions leaving her with a job title that did not reflect the actual tasks that she performed. She described her projects as designing ‘service fabrics’ (e.g. provisioning and configures SharePoint VMs) to bring together multiple sources of customer usage data for Microsoft products.

Our informant discussed collaborating with software engineers—feature owners of the target software products—to produce her data processing solutions. She learned about the data in production by the software engineer’s features and gathered processing requirements, including about how the data should be connected with other data ‘downstream’ or ‘upstream’.

Essentially a software engineer whose customers are other software engineers, our service developer expressed admiration for ‘innovative solutions’, designs that met all of the challenging requirements and contextual considerations:

> *…smart solution that meets your business goals, requirements, everything that minimal engineering, and with all around your service fabric…*
>
> *-Senior Service Engineer, Applications*

Our informant further felt that great software engineers had a great depth of knowledge, and were effective in conveying that knowledge to others. She discussed working with great software engineers that knew, in great detail, what data were collected and what information (e.g. user behaviors) those data captured, as well as how that data fit with within the business objectives and connected other data. In addition, our informants felt that great software engineers were able to clearly and succinctly explain that understanding to another software engineer so that another software engineer—our informant in this case—could then comprehend the scenarios and data processing requirements (i.e. what the software engineer wanted the service to capture):

> *[A great software engineer] who is also able to speak out or being very crisp. It needs to be precise, concise and then able to convey what you want to convey in very small words [sic], not too much of stories, not derailing from the requirements, or not carried away by a lot of other stuff… Business goals is going to be the main requirement. You don't get carried away by the supporting things or lose your track… it's very essential to go in-depth as well as on breadth.*
>
> *-Senior Service Engineer, Applications*

5.2.10.4 Discussion

Aside from the one service engineer who was effectively a software engineer, the rest of our service engineers all wanted software engineers to appreciate the expertise of service engineers—their knowledge of networking and operations aspects of software services. This sentiment, while related to the concept of knowledgeable about people and organizations discussed in interviews of software engineers, was closer to the concept of respecting other experts.

Interestingly, our informants did not emphasize the need to be well-mannered, which is how software engineers typically discussed condescending behaviors in other software engineers. Rather, our expert service engineers emphasized attributes related to being open-minded, as well as being able to see the big picture, sees the forest and the trees. The underlying understanding appeared to be that service engineers recognized that software engineers were central and critical to Microsoft's software services business (i.e. service engineers were, in fact, second class citizens); nonetheless, our experts wanted the software engineers to recognize that successful software services required service engineers.

Many of our informants described their relationship with software engineers as DevOps; Roche, in his description of the DevOps model (Roche, 2013), characterized it as an evolution of quality assurance. While the migration to combined engineering was largely discontinued with the traditional tester role at Microsoft (Locke, 2014), our findings indicate that the service engineer may be the new tester for a services-centric Microsoft. With the responsibility to ensure that software services ran with quality after release, many of our informants reported the same issues (e.g. throwing software 'over the wall' and lack of respect) reported in historical accounts of testers at Microsoft (Zachary, 1994).

5.3 DISCUSSION

In this study, we sought to understand the perspectives of expert non-software-engineers on the software engineering expertise. To our knowledge, this is the first time this topic has been systematically investigated. Overall, our expert non-software-engineers recognized that software engineers performed a critical engineering task—writing code—without which their product would not exist. Consequently, great software engineers were foremost expected to be great at their own jobs; without quality software, other considerations were often moot. Second, great software engineers were expected to speak up about potential problems and progress, since they usually had the best knowledge about the technical details and their implications. This included scoping and innovating during planning, as well as updating timelines and expectations during development; great software engineers ensured that all team members had the information they needed to make their decisions. Third, our informants generally felt that great software engineers recognized that they were not experts on all aspects of the product; great software engineers listened to and leveraged the knowledge of other experts. Great software engineers understood and appreciated the contributions expert non-software-engineers.

For practitioners and educators, this knowledge may help to train and educate software engineers, particularly those working (or will be working) in interdisciplinary teams. At minimum, for practicing software engineers, the myriad (negative) stories and examples behind these findings may spur some introspection about how they are engaging (and should be engaging) with expert non-software-engineers. For researchers, the findings may be a starting point for many additional research efforts. For example, numerous studies can examine each of

the important attributes rated by software engineers that were not found to be important by expert non-software-engineers.

Aside from the general insight above, we also found two interesting overarching insights among our findings. In the following three sections, we will detail these two observations and their implications for research, and then conclude with a discussion of the threats to validity.

5.3.1 *Conditions for Equality*

Many of our expert non-software-engineers felt that software engineers did not view them as equals; however, this feeling was not universal and other experts did not have similar sentiments. Understanding the conditions that lead to equality—real or perceived—between software engineers and non-software-engineers may be worthwhile future work. As many expert non-software-engineers perform important functions in software engineering teams, eliminating the kinds of issues described by our informants may be essential to the long-term success of teams and organizations.

Informants in numerous roles expressed feelings of perceived inequality, including Content Developers (Section 5.2.2), Design Researchers (Section 5.2.4), Designers (Section 5.2.5), Product Planners (Section 5.2.8), and Service Engineers (Section 5.2.10). Commonly, these experts commonly discussed software engineers developing and shipping the software without involving them. One content developer informant expressed frustration with software engineers being non-responsive or late with documentation requests; several design researchers stated that software engineers sometimes made decisions based on their own experience rather than conducting usability testing. A designer informant discussed bad experiences with software engineers making UX decisions without seeking their guidance; a product planner informant discussed software engineers ignoring their suggestions. One service engineer felt that software engineers sometimes threw software ‘over the wall’ at them. All of these informants wanted software engineers to appreciate the contributions of their role to the overall success of the software product and to recognize the specialized skill necessary to do their functions well.

Informants in other roles expressed feeling of equality in their collaborations, including Artists (Section 5.2.1), Electrical Engineers (Section 5.2.6), and Mechanical Engineers (Section

5.2.7). Historical perspectives provided by artist informants were especially interesting. Several artist informants discussed ‘steamrolling’ and inequitable decision making by software engineers in the past. However, they felt that their current Microsoft teams did not have those issues. Informants hinted that underlying reasons may include maturation of the gaming industry making art/atheistic essential to success, advances in game development technology (e.g. game engines) enabling artists to be more self-sufficient, or the culture at Microsoft. More understanding of the conditions—both what and why—may help software engineering teams perform optimally, especially teams with non-software-engineers that our study indicate as possibly treated inequitably by software engineers.

5.3.2 *Challenging Engineering Processes*

For many of our non-software-engineer informants, problems during collaborations (and consequently their desired attributes of great software engineers) might have been direct results of the engineering processes of their products. However, as the engineering approaches were highly constrained by their software products, we did not see obvious adjustments, making in-depth investigation of mitigations potential future work.

Informants that worked on games (e.g. Artist and Designers) indicated that their teams often needed to ‘push the envelope’, necessitating substantial hard work, especially close to shipping dates. As explained by our expert Artists, the video games industry is hyper-competitive, and game studios must constantly be ‘aiming for the stars’ to remain competitive in the marketplace. This context likely contributed to Artists desiring software engineers who can ‘hack’ something together and who are willing to do extra work at the end of the schedule. However, this engineering process is likely detrimental to code quality (Nagappan & Ball, 2005) and software engineers working on games disliked being expected to do extra work (see discussions of the hard working attribute in Section 4.2.1.2). Easy solutions may not exist; given the competitive nature of the gaming industry, any additional resources would probably be put towards more features, instead of reducing the strain on software engineers and other experts.

Consumer electronics is another area where informants hinted that the engineering process led to problems. In this case, the physical nature of consumer electronics necessitated essentially a ‘waterfall’ process. ‘Physical’ electronics required the program team to make many

important engineering decisions upfront, since even minor hardware changes or fixes took months (if possible at all). This development approach puts considerable strain on the various experts involved, including Software Engineers, Electrical Engineers, and Mechanical Engineers. Not only do they have to be experts of their own technical domains, they also needed considerable technical knowledge of the *other* technical domains. They needed to think through implications of their decisions on others as well as comprehend and communicate with the other experts; many expert Electrical Engineers and Mechanical Engineers wanted software engineers to understand their technical domains. However, asking software engineers to have expertise in multiple separate engineering disciplines may not be realistic.

Today, the typical solutions to the problems above are to find *great* software engineers with all the desired attributes; however, finding these 'unicorns' is not a sustainable practice, especially with the increasing demand for software engineers. Therefore, it may be incumbent on researchers to find workarounds or alternative engineering approaches that address these structural problems.

5.3.3 *Threats to Validity*

As with any empirical study, there are various threats to validity. The main threat to *construct validity* is our informants' understanding of the attributes from our previous studies of software engineers. Since our informants were not software engineers and were commonly from very different communities of practice, they might not have interpreted the attributes and descriptions in the same manner as software engineers. Furthermore, due to time constraints we were not able to describe each attribute in detail in our interviews. This threat is mitigated by our selection of experts (typically those with 5+ years of experience), as our all our expert non-software-engineers had worked with software engineers and were likely familiar with the attributes described. In addition, we selected our informants from the same organization as our software engineers, increasing the likelihood that our informants were familiar with the terminology and sentiments described. Most importantly, we did not attempt to pigeonhole the attributes discussed by our expert non-software-engineers into attributes discussed by expert software engineers. We examined the data for each role separately, retaining the wording and sentiments of the expert non-software-engineers.

The key threat to internal validity is from the interpretation of the data; other researchers may interpret the data differently. Nonetheless, I feel that I may be uniquely qualified to accurately interpret the data since I work at Microsoft (having an understanding of the organization context), am familiar with software engineering at Microsoft (having shipped a feature in Windows Vista SP1), and am knowledgeable about the area (having conducted two prior research projects on this topic).

We note three threats to the external validity of our study. First, findings from our sample of expert non-software-engineers might not extend to all expert-non-software-engineers at Microsoft. Though we interviewed multiple experts for each role, the proportion of experts we interviewed in each role was small; differing opinions may exist. Second, our findings may not extend to expert non-software-engineers outside of Microsoft. For example, the Program Manager role (discussed in Section 5.2.9) is likely unique to Microsoft, and findings may not extend to project managers at other organizations, e.g. ‘project managers’. Nonetheless, many of our informants had work experience at other organizations, increasing the external validity of our study, notably Artists (discussed in Section 5.2.1), Designers (discussed in Section 5.2.5), Electrical Engineers (discussed in Section 5.2.6), and Mechanical Engineers (discussed in Section 5.2.7). Furthermore, we believe that our methods and findings are appropriate given the dearth of knowledge in this area and the exploratory nature of our study.

Chapter 6. WHAT MAKES A GREAT SOFTWARE ENGINEER

In this thesis, we have interviewed expert software engineers, surveyed many more, and talked to expert non-software-engineers to gain a holistic, contextual, and real-world understanding of software engineering expertise. We learned about various attributes considered to be a great software engineer, how expert software engineers rated those attributes, and the perspectives of expert non-software-engineers. In this chapter, we will return to our original question: what makes a great software engineer?

In subsequent sections, we will synthesize our findings and discuss the salient aspects of being a great software engineer. We will discuss each aspect: how it manifested in our studies and how it relates to findings in related work.

6.1 BE A COMPETENT CODER

Our results indicated that the most important aspect of being a great software engineering is being a competent coder. While previous studies about software engineering expertise tout various 'soft skills' (see Section 2.6), the experts in our study—software engineers and non-software-engineers alike—the ability to write good code was essential.

The understanding is straightforward: without code, there is no software; therefore, great software engineers need to be able to write good code. Producing software is the basis for ACM's definition of a software engineer: 'people who write software to be used in earnest by others' (Shackelford et al., 2006). In addition, our informants felt that the code needed to be a sufficient quality. Expert non-software-engineers stated that even if everything else about the software product is great—great art, excellent design, wonderful documentation, etc.—it will not be successful if the software is full of bugs. Most of our expert non-software-engineers knew little about software development and depended on the software engineer to write the code; therefore, they expected and needed the software engineer to do the own job right. Software engineers agreed; paying attention to coding details was the top rated attribute in our survey. Our

expert software engineers stated that they did not respect software engineers who could not get the basics right (i.e. "wrote shoddy code").

Being a competent coder was necessary to be a great software engineer, but not solely sufficient. While our informants felt that a software engineer cannot be great without this attribute, they also felt that simply having this attribute does not make a great software engineer. For many informants this attribute was a 'baseline', and felt that most software engineers at Microsoft were competent.

The threshold for competence also appeared to be low. Software engineers' rankings of attributes showed that although paying attention to coding details (entailing error handling, memory consumption, performance, security, and style) was the highest ranked attribute, the next software-product-related attribute (the fits together with other pieces around it attribute) was ranked 10th. There appeared to be a low threshold that software engineers needed to achieve, beyond which other attributes (like the ones we discuss in subsequent subsections) become more important.

Nevertheless, coding competence received near universal acknowledgement by our experts. This finding is a vindication for the ACM's Computing Curriculum for Software Engineering (Shackelford et al., 2006), which focused largely on technical coding skills. These findings also align with various studies that observed everyday activities for software engineers ((Ko et al., 2007), (Latoza et al., 2006), (Singer et al., 1997), (Perry et al., 1994)); though software engineers spend time doing other activities, much of their time is still spend coding. In addition, our results largely justified research efforts aimed at understanding and closing the gap between novice and expert coders (see Section 2.4). Even though focusing on coding may be myopic, given that it is a necessary skill for software engineers (as our findings confirm), ensuring that novices are competent coders first is likely a good starting point. Conversely, our findings suggest that not considering technical skills is a major limitation of several research efforts that solely focus on 'soft skills' of software engineers (Kelley, 1999b) (Ahmed et al., 2012). The lack of consistent findings between various human factors and engineering outcomes, as discussed in Cruz et al. (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.

6.2 MAXIMIZE CURRENT VALUE OF YOUR WORK

The economic concept of 'risk and expected returns' (Ventures, 2000) explains numerous seemingly contradictory attributes and sentiments in our study (discussed below). When applying this economic lens, including consideration of probabilistic future value (possibly negative) and the time available for actions, a coherent theme emerges. Great software engineers, taking into consideration the context of their software product, maximized the value of their actions—adjusted for probable future values and costs.

The first area of (apparent) contradiction was that many software engineers discussed great software engineers designing their software with the future in mind, e.g. long-termed (discussed in Section 3.3.4.6) and anticipates needs (discussed in Section 3.3.4.8). To them, great software engineers took time and effort to ensure that their software was resilient to possible future changes. However, also discussed in those sections, many other software engineers disagreed. The dissenters felt that predicting the future was futile; they felt that experimentation, faster iterations, and a willingness to make changes were better. In their view, long-termed and anticipates needs were detrimental attributes since they wasted effort and resources on a future that may not occur.

These seeming conflicting opinions reconciled when viewed with an economic lens. The software may incur future costs to service and repair (i.e. incur 'engineering debt'). Therefore, the current value of software engineering work needs to take into consideration probable future costs of repair and maintenance. For software with long lifespans and high repair costs, software engineers should think ahead (i.e. be long-termed and anticipating needs). However, in other situations the software may have a short lifespan or have low repair costs (e.g. updating online services compared to patching boxed software); in those situations, great software may rightly defer future costs (i.e. 'build for now').

The second area of contradiction was the expectation that great software engineers should take the time to thoroughly think through the problem. The systematic attribute (Section 3.3.1.13) entailed not jumping to conclusions and not acting too quickly; the elegant attribute (Section 3.3.4.5) involved thinking deeply to coming up with simple solutions to difficult

problems; the fits together with other pieces around it attribute (Section 3.3.4.2) entailed accounting for the relationships with surrounding components. However, many software engineers also wanted great software engineers who would just go ahead and 'do it'. The willingness to go into the unknown attribute (Section 3.3.1.10) was about the willingness to take action with incomplete information, and the executes with no analysis paralysis attribute (Section 3.3.1.3) was explicitly about the need to *stop thinking* and start doing. The same incongruity was also present among our expert non-software-engineers. Nearly all of them expected great software engineers to produce quality software, as discussed in the previous section; however, on numerous occasions and across many roles, expert non-software-engineers also wanted software engineers to 'hack' a solution together (for example, see Section 5.2.1.2 for Artists) and to bypass established processes to get them something quickly to 'unblock' their tasks (for example, see Section 5.2.7.2 for Mechanical engineers).

From an economic perspective, software has value (i.e. makes money) only after deployment; however, for some products there is a timing element that greatly affects these future benefits. Products like games and consumer electronics have market conditions that incur significant revenue penalties for missing certain deadlines (e.g. the holiday season). Through this lens, contradicting opinions about speed of actions makes economic sense. High-quality software saves on future repair and maintenance costs; however, those savings must be weighed against possible forfeiting of revenue. Therefore, while having high-quality software is generally good, there may be situations, especially close to 'ship dates', where producing a 'hack' makes more economic sense than having a complete solution that takes more time.

The importance of risks and expected returns may be context specific, as Microsoft is a for-profit organization. Nonetheless, related concepts are often discussed in research literature on bug triaging for open source software projects. (Anvik et al., 2006) (Ko & Chilana, 2011); open-source software engineers take possible future issues (e.g. 'regression' or 'reopen') into consideration when deciding whether/how to fix a problem. Interestingly the education literature is largely silent on this issue. For example, ACM's curriculum (Shackelford et al., 2006) prescribes a set of skills but has little information about when or even whether to use those skills. Things like 'software architecture' and 'software verification' are great in theory; however, our

findings indicate that, in the economics of real-world software engineering, the best solution may sometimes be ‘quick and dirty’.

6.3 PRACTICE INFORMED DECISION MAKING

As we discussed in Section 3.4.1, software 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. However, rather than outcomes (which were often confounded by future uncertainties and outside factors), we found the process of acquiring needed information to make good decisions to be the most important aspect of effective decision-making. Great software engineers differentiated themselves by going through the right processes to make informed decisions.

In discussing this theme, we use the framework of ‘rational decision-making’ described in Simon’s 1955 paper (Simon, 1955). We believe this framework captures decision-making in the software engineering context better than the intuition-driven ‘naturalistic decision-making’ advanced described by Zsambok and Klein (Zsambok & Klein, 1996). In most situations software engineers identified the decision to be made, systematically identified the alternatives, thought through potential outcomes, estimated the likelihood of those outcomes, approximated the value of those outcomes, and then decided among those courses of action (if any).

Many attributes of great software engineers concerned their effectiveness in making decisions (see all of the attributes in Section 3.3.2); we found those associated with the ‘information gathering’ activities described in Simon’s paper (Simon, 1955) to be the most important. Software engineers often did not have the information they needed to make their decisions; great software 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 (discussed in Section 3.3.1.13) described actually undertaking the ‘information gathering’ activity, the asks for help attribute (Section 3.3.3.11) concerned seeking out those with the best information, and the open-minded (Section 3.3.1.2) and data-driven (Section 3.3.1.17) attributes both describe great software engineers willingness to let new information influence their decisions.

Conversely, many negative attributes of bad software engineers discussed by our experts were symptoms of not gathering or not using the right information to make decisions. In discussing the data-driven attribute (Section 3.3.1.17), informants lamented that some software engineers had confirmation bias, selecting only the information that confirmed their initial understanding. The same problem was reported by various expert non-software-engineers, like Product Planners in Section 5.2.8.2. In addition to confirmation bias, numerous expert non-software-engineers also described self-referential problems. Design Researcher (Section 5.2.4.2), Designers (Section 5.2.4.2), and Product Planners (Section 5.2.8.2) all described software engineers as overly reliant on their own experience and not letting other data supplement or change their understanding.

The process of decision-making has received little direct attention in the software engineering literature. It is not mentioned in the ACM curriculum, and, as we discussed in Section 3.4.1, we are not aware of any direct research on the topic within the context of software engineering. Nonetheless, aspects of decision-making, good and bad, are sprinkled throughout the software engineering literature. Bug triaging, examined by many researcher (Anvik et al., 2006) (Jeong et al., 2009)(Podgurski et al., 2003)(Runeson et al., 2007)(Bertram et al., 2010) is effectively a decision-making process. The work by Gobeli et al. (Gobeli et al., 1998) examining effective (and not effective) conflict resolution approaches within software engineering teams touches on making decisions. Consulting with team members to decide how best to implement a feature or to fix a bug is mentioned in various studies that examine everyday activities of software engineers (Ko et al., 2007)(Latoza et al., 2006). Perhaps now is the time for software engineering educators and researchers to pay attention to decision-making within their education and research efforts.

6.4 Enable Others to Make Decisions Efficiently

Shrouded in polite descriptions like creates shared understanding with others (Section 3.3.3.3) and creates shared success for everyone (Section 3.3.3.15), a major theme in our interviews—software engineers and non-software-engineers alike—was *don't make my job any harder*. Great software engineers made others' jobs easier by helping to them make their decisions more efficiently (or, at minimum, they did not make them worse).

We noticed this theme surface as we discussed software engineers having—though more commonly, not having—various attributes. This sentiment was most apparent in the honest attribute; in almost every instance where honest was discussed (Sections 3.3.3.4, 4.2.1.1, and 5.2.9), informants described negative situations when software engineers lacked honesty. One informant could not act upon the feedback from a software engineer because he would "misrepresent something or make them look better." An informant described poor software engineers that "would lie to me about [their component's] availability and maturity in order to get me to be a user and justify their own existence to management." A program manager complained that "if it's sugarcoated, you can't address it in a timely manner as probably is needed or in a direct manner as probably is needed". Poor software engineers did not provide information (or worse, provided misinformation) that caused our informants grief.

The theme of not causing problems for others was also evident in the discussions of many other attributes. The manages expectations attribute (Section 3.3.3.5) contained discussions about software engineers derailing a project by not speaking up about potential delays. The self-reflecting attribute (Section 3.3.1.5) entailed software engineers proactively changing plans when they realized current plans were untenable; the same sentiment underlies the asks for help attribute (Section 3.3.3.11). In addition, for many informants, the creates shared understanding attribute (Section 3.3.3.3) was about great software engineers helping them understand the reasoning—commonly, pitfalls and potential problems—behind various options so they can make appropriate selections or explain decisions to management. For these attribute, informants discussed software engineers without the attributes preventing other software engineers from taking corrective actions to avoid bad outcomes for the team and ultimately making their jobs *harder*.

There is little direct mention of "*don't make my job any harder"* in the research literature, even though there are hints in various qualitative studies of software engineering efforts. For example, Ko et al. (Ko et al., 2007) found 'maintaining awareness' to an important concern for software engineers, and Latoza et al. (Latoza et al., 2006) found 'team code ownership and the moat' (which facilitated understanding within the team and limited outside perturbations) to be a common theme. This latent sentiment may be especially difficult to detect using research methods that do not dig deeper into the reasoning behind stated opinions; we found this theme in

the discussions of the implications of attributes. Various research methods like surveys (Lethbridge, 1998), meta-analysis (Radermacher & Walia, 2013), and secondary analysis (Ahmed et al., 2012) may not be able to detect this sentiment. Our findings suggest that a complex phenomenon like software engineering needs to be studied using qualitative studies that provide in-depth understanding, in addition to quantitative methods.

6.5 CONTINUOUSLY LEARN

Aptly and succinctly summarized in our section on the continuously improving attribute (Section 3.3.1.1), "engineers do not start their careers being great; young software engineers needed to learn and improve to become great… the software field was rapidly changing and evolving, unless engineers kept learning, they would not become and would not continue to be great."

In addition to continuous improving, which is a direct derivative of the continuously learning concept, numerous other attributes were also related. Many informants discussed the curious attribute—wanting to know how things work—(Section 3.3.1.7) as a motiving factor behind learning. Both grows their ability to make good decisions (Section 3.3.2.7) and updates their decision-making knowledge (Section 3.3.2.8) were about learning and continuously re-learning how to make the best decisions. Asks for help (Section 3.3.3.11) and integrates understanding of others (Section 3.3.3.2) both involved effectively learning from others. Finally, the concept of being open-minded, both as the attribute described by software engineer (Section 3.3.1.2) and as the sentiment in interviews with expert non-software-engineers (Section 5.2.4.2, Section 5.2.8.2), derived from situations where software engineers had to learn and utilize new information.

As made evident by attributes and sentiments throughout our research, a great software engineer is not a one-time designation; it is an ongoing progress. This aligns with sentiments in relate work. McConnell in *Code Complete* (McConnell, 2004) stated that curiosity is an important personal characteristic for software engineers because it promotes "keeping up with changes and seeking ways of doing their job better." The Vice President of People Operations at Google stated, "significant learning and growth occur after college and that many skills to succeed in industry are not the same ones you need to succeed in school" (Bryant, 2013). Codes

of ethics from other fields, e.g. medicine (AMA, 2001) and traditional engineering (NSPE, 2007), suggest that continuously learning is a requirement that is shared across all learned professions.

6.6 SUMMARY

In summary, the five aspects of software engineering expertise we found in this dissertation were:

- Be a competent coder
- Maximize current value of your work
- Practice informed decision-making
- Enable others to make decisions efficiently
- Continuously learn

Overall, software engineering expertise holistically encompassed internal personality traits, ability to engage with others, technical capabilities, and decision-making skills (an area not emphasized in previous studies).

Within software engineering research, the one area that covers as broad a set of concerns is software development processes/methodologies (Section 2.3); however, we note a salient difference. While software development processes/methodologies commonly prescribed some of the same attributes as our findings, their focus was on software engineering teams and many of their attributes may not be important for individual software engineering expertise. For example, the Capability Maturity Model (Herbsleb et al., 1997), in Basic Level 2 (‘repeatable’), prescribes ‘software project planning’ and ‘software project tracking and oversight’. In our study at Microsoft, these activities were commonly performed, with or by the expert non-software-engineers (e.g. product planners, Section 5.2.8, and program managers, Section 5.2.9). Therefore, it is likely that some areas of concern discussed in research on software development processes/methodologies are not essential to *individual* software engineering expertise. An

interesting area of future research may be to discern which software engineering activities could be (or should be) off-loaded to expert non-software-engineers to promote *organizational* success, freeing software engineers to focus on the critical task of producing good code.

Chapter 7. SOFTWARE ENGINEERING EXPERTISE WITHIN THE CONTEXT OF HUMAN EXPERTISE

Software engineering expertise is a part of human expertise, a significantly broader research area. Within the framework of human expertise research, this dissertation can be viewed as an 'ecologically-valid study' of 'an ill-defined problem', as defined by Ericsson and Smith in *Toward a General Theory of Expertise* (Ericsson & Smith, 1991). Rather than examining actions and behaviors within constrained or synthetic 'lab' situations, we have attempted to understand a phenomenon as it occurs in real and complex settings. In this chapter, we will relate the findings in this dissertation to research and knowledge in human expertise, discussing insights and implications.

Before discussing our findings, we note that various studies in human expertise have examined aspects of software engineering. Other than software engineering focused studies comparing novices and experts (Section 2.1) several research studies have examined programming from a human expertise perspective. Soloway, Adelson, and Ehrlich summarized their studies examining the cognitive underpinnings of program comprehension; the authors proposed two constructs that help advanced programmers to comprehend programs quickly: program plans (program fragments that present typical sequences) and rules of programming discourse (conventions in programming) (Soloway, Adelson, & Ehrlich, 1988). The authors tested their theory by creating two programs that adhered (and did not adhere) to plans/rules and analyzed novices and experts by using fill-in-the-blank questions. The authors found that experts performed better than novices in adhering programs but performed at a level similar to novices in non-adhering programs. Adelson and Soloway examined expertise in software design by studying three expert software designers; the authors observed that the experts used mental models that began as abstract but became progressively more concrete, used balanced development in which components were iteratively designed to the same level of detail, kept notes about what needed to be done later, and mentally 'executed' their designs (Adelson & Soloway, 1988). Sonnentag, Niessen, and Volmer surveyed studies involving software design

expertise; they identified five areas of concern (requirement analysis and design, program comprehension and programming, testing and bugging, knowledge, and communication and cooperation) and two ways of defining experts (more experience and higher performance) (Sonnentag, Niessen, & Volmer, 1991). The authors discussed the distinguishing characteristics of experts in each of the five areas. Though prior work contains interesting insights about understanding and developing code, real-world software engineering goes beyond programming (in isolation). In the subsequent sections, we will examine insights from our study of the broader and more complex phenomenon of real-world software engineering.

A thorough discussion of the voluminous literature in human expertise is beyond the scope of this dissertation. In our discussions, we will describe and reference relevant theories and studies where appropriate. Readers interested in a deeper and broader understanding of human expertise are encouraged to read comprehensive texts on the topic: *The Cambridge Handbook of Expertise and Expert Performance* (Ericsson, Charness, Feltovich, & Hoffman, 2006), and *The Nature of Expertise* (Chi, Glaser, & Farr, 2014).

7.1 Actions Amid Chaos

Real-world software engineering is significantly more challenging (and complex) than constrained scenarios studied by prior work on human expertise. Compared to the programming and program comprehension tasks examined by prior work, software engineering (as we observed at Microsoft) involved many activities outside of coding (e.g. consulting experts), spanned significantly longer periods of time (several weeks to several years), and had more complex technically-contextual considerations beyond correctness (e.g. long-term viability of changes, Section 3.3.4.6, and structuring components to be updated/changed efficiently, Section 3.3.4.4). More importantly, real-world software engineering is replete with unexpected disruptions. These disruptions often involved important changes in underpinnings of project; teams changed priorities/objectives, underlying technology evolved constantly, and time/resource shortfalls occurred frequently (e.g. partner teams missing deadlines).

Writing good code, while critically important, is not sufficient. ‘Being a competent coder’ is one of the most important aspects of software engineering expertise (Section 6.1);

however, most informants considered this attribute as a ‘baseline’. Once the requirements were well-understood and documented (and remained unchanged), informants felt that most software engineers—at least those working at Microsoft—could competently produce the needed software. The challenge, it appears, is what to do when something unexpected happens.

Prior research found that unknown situations and perturbations in well-understood patterns was where human expertise typically breaks down. Chase and Simon found that by placing chess pieces in unfamiliar arrangements (e.g. impossible position), the ability of experts to recall board positions regressed to the abilities of novices (Chase & Simon, 1973). Soloway et al. found the same regression in novice and advanced programmers when programming rules/plans were broken (Soloway et al., 1988). Johnson found that when “uncontrolled intervening events occurs between the choice and the outcome”, decisions of experts were not consistently better than those of novices (Johnson, 1988). The advantages of expertise appear to disappear when confronted with the unexpected.

Yet, in our studies of great software engineers, we found that some attributes of their expertise lie precisely in their ability to effectively deal with the unexpected. Being adaptable to new settings (Section 3.3.1.14) and willing to go into the unknown (Section 3.3.1.10) involved the mentality of expert software engineers facing the unknown and the unexpected. Expert non-software-engineers discussed successful collaborations where great software engineers adroitly reacted to unexpected problems, often by-passing or short-cutting processes (e.g. fixing an unexpected bug in the payment processing system, Section 5.2.3.2, and quickly producing firmware to unblock prototyping, Section 5.2.7.2). Great software engineers were also unafraid of trying new technology and going into the unknown (e.g. trying new gaming technologies discuss by artists, Section 5.2.1.2, and programming iOS apps discussed by a program manager, Section 5.2.9.2). We found that great software engineers handled disruptions (which were *very* common) gracefully.

Our findings suggest that a distinguishing trait of software engineering expertise may be how expert software engineers deal with unexpected situations, where traditional human expertise would normally breakdown. More investigation of dealing with unexpected disruptions

may be an interesting area of future research for software engineering expertise as well as human expertise.

7.2 Decision-making but With Possibly Incorrect or Incomplete Information

Nearly all research into human expertise involves, to some degree, the ability of experts to process information. Notably, in studying expert and novice chess players, Newell and Simon found that experts were able to quickly translate ‘patterns’ into ‘chunks’ for efficient mental processing (Newell & Simon, 1972). Subsequent research would find that efficient pattern recognition (selective intake of information) and reductions in the ensuing search for optimal actions are hallmarks of expertise (Chi et al., 2014). Yet, our findings suggest that expert software engineers are often confronted with incorrect or incomplete information, often unbeknownst to the engineer. What happens when the information expert software engineers depend upon is not dependable?

Informants indicated that information others provided was sometimes incorrect. Sometimes this was accidental; people often only had partial knowledge of the situation (see the discussion of the integrates understanding of others attribute in Section 3.3.3.2.). Other times people deliberately provided bad information; informants discussed deleterious situations where software engineers would misrepresent the situation to their own benefit (see discussion of the honest attribute in Section 3.3.3.4) or selectively used data that suited their purposes (see discussion of the data-driven attribute in Section 3.3.1.17).

Our informants also discussed important information being *missing*—some bad software engineers simply did not communicate *anything*. Missing needed information is the sentiment behind the manages expectations attribute (Section 3.3.3.5). Needing undeclared knowledge was also behind 6.0% of software engineers rating the trades favors attribute as ‘detrimental’; follow-up interviews found that software engineers did not like needing undeclared processes (i.e. information they did not have) to achieve their goals. Numerous expert non-software-engineers also criticized lack of communications from software engineers. Product planners discussed wanting software engineers to provide technical understanding so that rest of the team can make

informed decisions (Section 5.2.8.2); program managers discussed needing progress updates from software engineers so they can avoid deviations from timelines and plan for mitigations (Section 5.2.9.2).

By all accounts, expert software engineers (and even expert non-software-engineers) did not have good solutions or strategies for dealing with bad or missing information. Many resigned to criticizing bad behaviors and calling on software engineers to improve their abilities to provide needed information (see discussion of 'enabling others to make their decisions efficiently' in Section 6.4). Some informants declared that they would simply leave teams with significant problems (honest, Section 3.3.3.4). Dealing with bad or missing information may be an area where researchers can devise processes and procedures to help software engineers and their teams. For example, the practice of 'daily stand up' in the Scrum development process (Rising & Janoff, 2000) helps software engineers develop the habit of providing updates of their progress.

7.3 TEACHERS: A REQUISITE FOR DELIBERATE PRACTICE

Ericsson et al. (Ericsson et al., 1993) found that expertise required deliberate practice:

> *…to improve performance it is necessary to seek out practice activities that allow individuals to work on improving specific aspects, with the help of a teacher and in a protected environment, with opportunities for reflection, exploration of alternatives, and problem solving, as well as repetition with informative feedback.*

Among the many requisites discussed above, the most interesting were 'help of a teacher' and 'informative feedback'. Many of our expert software engineers discussed great software engineers who mentored and helped them. As teachers, the expert software engineers provided important feedback and guidance that helped our informants improve and become great software engineers themselves. Many aspects of 'positively influencing others' are related to great software engineers effectively growing other great software engineers (e.g. mentoring, Section 3.3.3.8, and creates a safe haven, Section 3.3.3.10). This aligned with the perspectives of research in human expertise; having teachers and receiving feedback were requisites for becoming an expert.

Human expertise literature finds training of other (future) experts to be an important element of expertise for many fields. Amirault and Branson discusses ‘masters’ training ‘apprentices’ as one of the key aspects of expertise in craftsman guilds (Amirault & Branson, 2006); the same approach persists in universities today, underlying the doctorial process (both how a candidate attains a doctorate and who is qualified to train those candidates). Software engineering, based on interviews, appeared to be another field where having ‘teachers’ is essential.

Yet, both mentors and creates a safe haven ranked low in our survey of expert software engineers, ranking 48 and 49 (respectively) out of 54 attributes. The apparent disconnect between the importance of teachers and the low importance ratings for its associated attributes may be an area of future research. It may well be that the attributes connected with being a good teacher—providing feedback and creating a safe environment for growth—are not attributes of *individual* expertise, but are critical for the growth of expertise in the software engineering *field*.

7.4 SUMMARY

Software engineering, as we observed, is a complex phenomenon. The expected expertise goes beyond ‘practiced skills’, like typing, memory, and calculations (Chi et al., 2014). Software engineering involves actions, but does not include ‘motor skills’ expertise, like music, sports, or dance (Ericsson et al., 2006). It is not a single isolated activity—software design—and constitutes many other activities involving many other agents.

Even though our ‘ecologically-valid study’ examination of software engineering expertise did not aim to contribute to theories of human expertise, our findings (when related to existing knowledge in human expertise) yielded several interesting insights. First, human expertise literature indicates that expertise commonly breaks down in unexpected situations; yet, many aspects of software engineering expertise involved effectively handling the unexpected. Software engineering expertise may specifically involve ‘gracefully’ dealing with situations where traditional expertise breaks down. Second, human expertise (software engineering expertise included) is dependent on having good information; yet, we find that software engineers often have to deal with bad or missing information. Experts in our studies did not

appear to have effective strategies for dealing with these informational problems. Finally, human expertise literature indicates that having teachers who can provide informative feedback is a requisite for acquiring expertise. Yet, even though many of our expert software engineers discussed mentors helping them gain expertise, attributes associated with growing and developing others received low importance ratings in our surveys. There may be a disconnect between needing those attributes to be an expert and needing those attributes to have experts in the software engineering field. We feel that investigating these questions may lead to interesting findings that may advance our understanding of software engineering expertise, and possibly of human expertise.

Chapter 8. CONCLUSION AND FUTURE WORK

The goal of this dissertation was to gain a holistic, contextual, and real-world understanding of software engineering expertise. The dissertation described three studies triangulating on this understanding using different methods and from different perspectives.

We started our research arch by interviewing experienced software engineers to understand attributes of software engineering expertise. Not only did we extract a holistic set of attributes of software engineering expertise from interviews with 59 experienced software engineers (over 60 hours of interviews and 388,000 words of transcripts), we also elicited understanding about why each attribute was important in real-world engineering of software.

In the next study, we built on the qualitative understandings from our interview study with a mixed-methods study examining the relative importance of the attributes and relationships with contextual factors as well as exploring why the attributes rated highly (and lowly) and why various contextual factors affected the rankings. In one of the largest studies of real-world software engineers that we are aware of, we received survey responses from 1,926 experienced software engineers. In addition to quantitative data, we also gained qualitative understanding about the most importance and the least important attributes from follow-up email interviews with 77 respondents.

Finally, we complemented our understanding by interviewing expert non-software-engineers that collaborated with software engineers. We interviewed 46 expert non-software-engineers in 10 different roles: Artists, Content Developers, Data Scientists, Designers, Design Researchers, Electrical Engineers, Mechanical Engineers, Product Planners, Program Managers, and Service Engineers. We gained different and diverse (yet in many ways similar) perspectives on software engineering expertise in practice, enriching our understanding of what makes a great software engineer.

We discussed key findings, insights, and limitations of each study; furthermore, we synthesized the findings from all three studies to deduce what we know about software

engineering expertise. Finally, we compared our findings about software engineering expertise with expertise in other professions to identify key differences and similarities.

In the subsequent sections in this chapter, we will first discuss an idea for future research that can build on findings in this dissertation (and address some of its limitations). Then, we will restate the major contributions of this dissertation. We will then close with some final remarks.

8.1 FUTURE DIRECTION

Software is ubiquitous today, understanding of what makes a great software engineer—the person that produces the software—is increasingly important. Our dissertation has provided foundational knowledge about the attributes of software engineering expertise, including definitions, explanations, and ratings. However, our understanding comes from a single organization, albeit a diverse and important organization, Microsoft is very different from other software producing organizations in several ways. Future studies may wish to replicate and expand the work detailed in this dissertation at other organizations to expand our understanding.

Foremost, Microsoft is a *for-profit* organization. However, some software producing organizations do not aim to make money and do not pay their software engineers; the most common and the most important is open-source software projects. Open source software projects generally do not pay their software engineers and do not charge money for acquiring their software (though some are supported by for-profit organizations, e.g. Eclipse backed by IBM). Since many aspects of software engineering expertise in our findings being related to economic considerations (e.g. ‘maximize current value of our work’, Section 6.2), future studies may want to examine software engineering expertise where the organizational objective is not to the economic goal of making money.

Second, Microsoft is a software-centric organization where software engineers are held in the highest esteem (Section 5.3.1). However, in many other organizations, software engineers are ancillary roles, supporting other parts of the organization. Several expert non-software-engineers felt that some of the software engineers’ perceptions (and the actions they engender) were due to software engineers having too much power at Microsoft. Software engineers in non-software-centric organizations may have different opinions about the importance of various attributes of

software engineering expertise. Some software engineers mentioned in the interviews that the difference in treatment at Microsoft versus other non-software-centric organizations may lead to differences:

> *I will not work at a company where what I do is not what the company does… Financial firms is a good example too because while certain benefits are attractive, you are not the reason they exist. You have to suffer to make the people whose existence is crucial happen.*
>
> *-SDE2, Windows*

With important software engineering taking place in non-software-centric organizations (e.g. financial organizations as mentioned in the quotation above), understanding differences in perspectives—not simply *what* attributes are viewed different, but also *why*—may be valuable future work.

Third, future studies may want to explore difference (and similarities) at similarly successful software engineering organizations like Google, Apple, Amazon, or Facebook. Like Microsoft, many of these organizations have diverse product offerings and development contexts (e.g., Apple has consumer electronics and cloud-based services). Therefore, not only would replicating our studies at these organizations enhance confidence in the findings in this dissertation, the studies may further explore product-related effects on software engineering expertise (e.g. consumer electronics and software services-related issues, Section 5.3.2).

Finally, investigating the *negative* aspects of bad software engineers may be important future work. This dissertation has focused on *positive* attributes of great software engineers at Microsoft; yet, we have found that informants often discussed avoiding bad mindsets and actions that would preclude a software engineer from being considered great. There will be methodological and ethical challenges with studying the darker side of software engineering; nonetheless, evidenced by hidden feeling in qualitative interviews (e.g. the 'don't make my job any harder' sentiment discussed in Section 6.4), the findings may be important. Software engineering expertise likely involve not only the acquisition and practice of positive attributes, but also the recognition and avoidance of negative ones.

Regardless of the direction, future studies should utilize both qualitative and quantitative methods. As discussed in Section 6.4, software engineering is a complex phenomenon, necessitating qualitative methods to deeply understand the reasoning and meaning behind statements and quantitative data to elucidate important insights and findings. Future studies should begin with qualitative interviews to gain a contextual understanding of perceptions of software engineering expertise in these organizations as well as to check understanding of the attributes identified in this dissertations—both similarities and differences. After adjustments and modifications to fit the organizational contexts, studies should follow with quantitative surveys at each organization to understand perceptions at scale. Finally, the studies should follow up with additional qualitative interviews to understand differences (and similarities) with findings in this dissertation. Through both qualitative and quantitative methods, future studies can further our understanding of software engineering expertise.

8.2 SUMMARY OF CONTRIBUTIONS

In this thesis, we have contributed a holistic, contextual, and real-world understanding of software engineering expertise. Specifically, we have provided the following eight contributions to our knowledge:

- A list of attributes of software engineering expertise from expert software engineers
- Contextual understanding of why expert software engineers think the attributes are important for the engineering of software
- A model that relates the attributes together
- An importance ranking of the attributes by expert software engineers
- Understanding of the reasoning behind the importance rankings
- Understanding of the relationship between contextual factors and the importance rankings
- Attributes of software engineering expertise that expert non-software-engineers think are important

- Contextual understanding of why expert non-software-engineers think the attributes are important

8.3 Implications for Researchers, Educators, and Practitioners

The knowledge in this dissertation may have wide-ranging implications for software engineering research, practice, and training.

8.3.1 *Researchers*

Our findings may have several implications for researchers. Foremost, to better understand and leverage attributes of software engineering expertise examined in this dissertation, we need measurements that operationalize the attributes. These will be essential in enabling rigorous science to better understand how the attributes vary and their effects on teams and outcomes. Such measurements may also form a critical foundation for managers to identify and cultivate talent, for novices to improve, and for educators to assess learning outcomes.

Second, our findings pinpoint several pain points that software engineering methodology researchers may want to address. We discussed several challenging software engineering processes in Section 5.3.2, caused by nature of the software products being developed. We discussed problems that expert software engineers (and expert non-software-engineers) have with bad information in Section 7.2. Better software engineering methodologies that address these issues may help software engineering teams.

Third, researchers may also want to look deeper into cultural variations and the impact on effective software engineering. Our findings indicate that cultural differences impact perceptions about software engineering expertise, as discussed in Chapter 4. Since many software development organizations are multinational, researchers may want to help practitioners understand the conditions in which software development organizations should (or should not) adapt to local cultural norms (versus instituting organizational standards) in their distributed software engineering efforts.

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 tradeoffs or see the forest and the trees when making decisions. Tools research may also explore facilitating and training engineers, especially novices, in the attributes of software engineering expertise.

8.3.2 *New Software Engineers*

Our findings have possible implications for new software engineers. Foremost, new software engineers should prioritize joining teams/organizations with good mentors and teachers. Software engineering expertise likely requires having mentors and teachers to provide guidance and feedback, as is the case with other kinds of human expertise, discussed in Section 7.3. However, our findings suggest that serving as teachers/mentors may not be a priority for many software engineers, and thus may be neglected in some software engineering teams. Young software engineers should avoid those organizations, and instead seek organizations that will provide the guidance and environment they need to become great software engineers.

For new software engineers who are unsure of how to become great (beyond being a good coder), our findings enumerate a prioritized set of attributes that they may aspire to achieve. Improvements may come from training, projects at work, mentoring, or self-adjustments (e.g. for personality traits). This may also yield interesting insights on whether various attributes (especially personality attributes) are trainable or innate.

Finally, our findings may also help new software engineers better present themselves to employers. Since our findings indicate that expert software engineers and managers value these attributes, novice software engineers may consider demonstrating to employers that they have or can develop these attributes. This also extends to highlighting the qualities when authoring their resumes or presenting themselves in interviews.

8.3.3 *Leaders of Software Engineers*

Our findings have possible implications for leaders of software engineers. Foremost, our findings conclude that many attributes that are important for engineers in senior and leadership positions,

such as mentoring, raising challenges, and walking the walk. Therefore, software engineers in leadership positions (or those working to become leaders), may seek to acquire or improve in those areas.

Beyond improving themselves, our findings may help managers make more effective hiring decisions. Managers may better identify candidates that fit the culture and context of the team. They may also be better equipped to avoid engineers without various important attributes, such as not aligned (off doing their own projects), not well-mannered (being an 'asshole', as many engineers described it), or not asking for help.

Our findings also suggest that current hiring practices—typically, one-day interviews—could be improved. Some important attributes of software engineering expertise (Section 4.3.1), such as the 'desire, ability, and capacity to learn', may more longer time to assessed in that short time. Approaches like Microsoft's successful internship program may be better alternatives; over several months and based on real-world projects (albeit scoped and non-critical), teams can better assess applicants' abilities, behaviors, and growth potential.

Finally, our findings strongly suggest that managers of software engineers should cultivate the attributes within their teams. Managers may consider using the findings to build a culture that is conducive to attracting, producing, and retaining great engineers.

8.3.4 *Educators*

Lastly, our findings 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 Computing Curricula (Shackelford et al., 2006), we found this attribute to be a key part of software engineering expertise (Section 6.3). A course specifically about decision-making (e.g. discussing Simon's model of rational choice (Simon, 1955), Klein's naturalistic decision-making approach (Zsambok & Klein, 1996), or case studies of software engineering decisions) may be valuable to students.

Another important area that may need more attention from educators is collaborations with non-software-engineers. The fact that expert non-software-engineers continue to be plagued

by decade-old problems in their collaborations with software engineers is vexing. As software engineering is a *socio*technical undertaking, educators may need to improve the ability of software engineers to collaborate not just with other software engineers, but also expert non-software-engineers who are essential to the success of real-world software engineering efforts.

Software engineering educators may need to reexamine their teaching methods. Most attributes of software engineering expertise involve *how* rather than *what*, whereas most instructions in software engineering focus on teaching skills and knowledge (the *what*), such as prior work on tools for automated testing and analysis. Educators may consider improving how software engineering goals are attained. For example, existing project-based courses can use attributes presented in this paper to help student evaluate each other's behavior, as well as grading non-functional attributes of the code, such as elegance, anticipates needs, and creative. Educators may also consider providing students with knowledge about *when* to use various skills. Our results indicate that various conditions exist in which eschewing best practices makes the most economic sense (Section 6.2). Educators may want to provide their students this knowledge to enable them to be effective under real-world conditions.

Finally, educators may consider explicitly discussing what students will not learn in school, allowing them to be aware of potential knowledge gaps and empower them to seek out opportunities outside of the academic setting (e.g. internships or open-source projects). For example, attributes like self-reliant may not be reasonable to teach in an academic setting and might be better learned through mentorships/internships; nevertheless, educators should consider informing students that it is a critical component of software engineering expertise.

8.4 Final Remarks

This dissertation has demonstrated the following thesis:

Experts involved in the creation of software view software engineering expertise as holistically encompassing internal personality attributes, attributes regarding engagement with others, in addition to technical capabilities in designing and writing code. Furthermore, the ability to make good decisions (e.g. choosing what software to write and how to write), which has not yet been articulated by previous research studies, is also critically important. The key aspects of

being a great software engineer are: writing good code, adjusting behaviors to account for future values and costs, practicing informed decision-making, avoiding making others' jobs harder, and learning continuously.

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 'SD Commit'.

BIBLIOGRAPHY

Adelson, B., & Soloway, E. (1988). A model of software design. In *Nature of expertise* (pp. 185–208).

Ahmed, F., Capretz, L. F., & Campbell, P. (2012). Evaluating the demand for soft skills in software development. *IT Professional*, *14*(1), 44–49.

AMA. (2001). American Medial Association Principles of Medical Ethics. Retrieved January 1, 2016, from http://www.ama-assn.org/ama/pub/physician-resources/medical-ethics/code-medical-ethics/principles-medical-ethics.page?

Amirault, R. J., & Branson, R. K. (2006). Educators and expertise: A brief history of theories and models. In *The Cambridge handbook of expertise and expert performance* (pp. 69–86).

Anvik, J., Hiew, L., & Murphy, G. C. (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).

Beck, K., Beedle, M., Bennekum, A. van, Cockburn, A., Cunningham, W., Fowler, M., … Thomas, D. (2001). Manifesto for Agile Software Development. Retrieved December 3, 2016, from http://www.agilemanifesto.org/

Begel, A., & Simon, B. (2008). Novice software developers, all over again. In *Proceedings of the Fourth International Computing Education Research Workshop* (Vol. 1, pp. 3–14).

Begel, A., & Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 12–23).

Bellovin, S. (2013). Why healthcare.gov has so many problems. Retrieved January 1, 2015, from http://edition.cnn.com/2013/10/14/opinion/bellovin-obamacare-glitches/

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).

Beyer, H. R., & Holtzblatt, K. (1995). Apprenticing with the customer. *Communications of the ACM*, *38*(5), 45–52.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, *21*(5), 61–72.

Boehm, B. W. (1991). Software Risk Management: Principles and Practices. *IEEE Software*, *8*(1), 32–41.

Borchers, G. (2003). The software engineering impacts of cultural factors on multi-cultural software development teams. In *Proceedings of the 25th International Conference on Software Engineering* (pp. 540–545).

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, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional.

Bryant, A. (2013). In head-hunting, big data may not be such a big deal. Retrieved March 10, 2015, from http://www.nytimes.com/2013/06/20/business/in-head-hunting-big-data-may-not-be-such-a-big-deal.html

Bureau of Labor Statistics, U. S. D. of L. (2015). Software developers. Retrieved January 1, 2015, from http://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

Burrus, D. (2013). The internet of things is far bigger than anyone realizes. Retrieved January 1, 2015, from http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/

Capretz, L. F. (2003). Personality types in software engineering. *International Journal of Human Computer Studies*, *58*, 207–214.

Carver, J. C., Nagappan, N., & Page, A. (2008). The impact of educational background on the effectiveness of requirements inspections: an empirical study. *IEEE Transactions on Software Engineering*, *34*(6), 800–812.

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, *4*(1), 55–81.

Chi, M. T. H., Glaser, R., & Farr, M. J. (2014). *The nature of expertise*. Psychology Press.

Clark, H., & Brennan, S. (1991). *Perspectives on Socially Shared Cognition*. American Psychological Association.

Cooper, A. (1999). *The inmates are running the asylum:[Why high-tech products drive us crazy and how to restore the sanity]*. Sams.

Cruz, S., da Silva, F. Q. B., & Capretz, L. F. (2015). Forty years of research on personality in software engineering: A mapping study. *Computers in Human Behavior*, *46*, 94–113.

Czerwinski, M., Horvitz, E., & Wilhite, S. (2004). A diary study of task switching and interruptions. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, *6*(1), 175–182.

Dabbish, L., Mark, G., & Gonzalez, V. M. (2011). Why do I keep interrupting myself ?: environment, habit and self-interruption. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 3127–3130).

Economist. (2010, February). Data, data everywhere. *Economist*.

Edmondson, A. (1999). Psychological safety and learning behavior in work teams. *Administrative Science Quarterly*, *44*(2), 350–383.

Ericsson, K. A., Charness, N., Feltovich, P. L., & Hoffman, R. T. (2006). *The Cambridge handbook of expertise and expert performance*. Cambridge University Press.

Ericsson, K. A., Krampe, R. T., & Tesch-romer, C. (1993). The Role of Deliberate Practice in the Acquisition of Expert Performance. *Psychological Review*, *100*(3), 363–406.

Ericsson, K. A., & Smith, J. (1991). *Towards a general theory of expertise: prospects and limits*.

Cambridge University Press.

Fisher, A., & Margolis, J. (2002). Unlocking the Clubhouse: the Carnegie Mellon Experience. *ACM SIGCSE Bulletin*, *34*(2), 79–83.

Fisher, D., DeLine, R., Czerwinski, M., & Drucker, S. (2012). Interactions with big data analytics. *Interactions*, *19*(3), 50.

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

Gobeli, D. H., Koenig, H. F., & Bechinger, I. (1998). Managing Conflict in Software Development Teams: A Multilevel Analysis. *Journal of Product Innovation Management*, *15*, 423–435.

Gugerty, L., & Olson, G. M. (1986). Debugging by skilled and novice programmers. *ACM SIGCHI Bulletin*, *17*(4), 171–174.

Guo, P. J., Zimmermann, T., Nagappan, N., & Murphy, B. (2011). "Not My Bug!" and Other Reasons for Software Bug Report Reassignments. In *Proceedings of the ACM Conference on Computer Supported Work*.

Herbsleb, J., Zubrow, D., Goldenson, D., Hayes, W., & Paulk, M. (1997). Software quality and the Capability Maturity Model. *Communications of the 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, D. A., & Chicken, E. (2013). *Nonparametric Statistical Methods* (3rd Editio). Wiley.

International Game Developers Association. (2008). IGDA Curriculum Framework: The Study of Games and Game Development. *IGDA Education SIG*, (February), 41. Retrieved from http://www.igda.org/wiki/images/e/ee/Igda2008cf.pdf

Iqbal, S. T., & Horvitz, E. (2007). Disruption and recovery of computing tasks: field study, analysis, and directions. *Proceedings of CHI '07*, 677–686.

Ivory, M. Y., & Hearst, M. A. (2001). The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, *33*(4), 470–516.

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). Amsterdam, Netherlands.

Johnson, E. J. (1988). Expertise and decision under uncertainty: performance and process. In *The nature of expertise* (pp. 209–228).

Joint Task Force on Computing Curricula. (2014). *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. *ACM Curricula Recommendations*.

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

Kelley, R. E. (1999b). How to be a star engineer. *IEEE Spectrum*, *36*(10), 51–58.

Kidder, T. (2000). *The Soul of a New Machine*. Back Bay Books.

Ko, A. J. (2006). Asking and Answering Questions About The Causes of Software Behaviors, 1–23.

Ko, A. J., & Chilana, P. K. (2010). How power users help and hinder open bug reporting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1665–1674).

Ko, A. J., & Chilana, P. K. (2011). Design, discussion, and dissent in open bug reports. *Proceedings of iConference '11*, 106–113.

Ko, A. J., 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).

Kohavi, R., Frasca, B., Crook, T., Henne, R., & Longbotham, R. (2009). Online experimentation at Microsoft. In *Workshop on Data Mining Case Studies and Practice*.

Krishnamurthi, S., & Felleisen, M. (1998). Toward a formal theory of extensible software. In *ACM SIGSOFT Software Engineering Notes* (pp. 88–98).

Latoza, T. D., 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).

Lethbridge, T. C. (1998). A Survey of the Relevance of Computer Science and Software Engineering Education. In *Proceedings of the Conference on Software Engineering Education and Training* (pp. 56–67).

Locke, C. (2014). Combined engineering @ Microsoft. Retrieved January 1, 2016, from http://blog.teleri.net/combined-engineering-microsoft/

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

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *2*(4), 308–320.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd Editio). Microsoft Press.

Meade, A. W., & Craig, S. B. (2012). Identifying careless responses in survey data. *Psychological Methods*, *17*(3), 437–455.

Mehlenbacher, B. (2000). Technical writer/subject-matter expert interaction: The writer's perspective, the organizational challenge. *Technical Communication*, *47*(4), 544–552.

Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, *68*(4).

Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering* (pp. 284–292).

Newell, A., & Simon, H. (1972). *Human problem solving*. Prentice-Hall.

NSPE. (2007). National Society of Professional Engineers Code of Ethics for Engineers. Retrieved January 1, 2016, from http://www.nspe.org/resources/ethics/code-ethics

Parnas, D. L. (1998). Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, *6*, 19–37.

Pendharkar, P. C., & Rodger, J. A. (2009). The relationship between software development team size and software development cost. *Communications of the ACM*, *52*(1), 141–144.

Perlow, L. A. (1999). The Time Famine : Toward a Sociology of Work Time. *Administrative Science Quarterly*, *44*(1), 57–81.

Perry, D. E., Staudenmeyer, N. a., & Votta, L. G. (1994). People, organizations, and process improvement. *IEEE Software*, *11*(July), 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).

Poile, C., Begel, A., Nagappan, N., & Layman, L. (2009). Coordination in Large-Scale Software Development : Helpful and Unhelpful Behaviors. *Microsoft Research Technical Report*.

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

Radermacher, A., & Walia, G. S. (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).

Raymond, E. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* (Revised Ed). O'Reilly Media.

Rising, L., & Janoff, N. S. (2000). The Scrum Software Development Process for Small Teams. *IEEE Software*, *17*(4), 26 – 32.

Robillard, M. P., Coelho, W., Murphy, G. C., & Society, I. C. (2004). How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, *30*(12), 889–903.

Roche, J. (2013). Adopting DevOps practices in quality assurance. *Communications of the ACM*, *56*(11), 38–43.

Ropponen, J., & Lyytinen, K. (2000). Components of software development risk: how to address them? A project manager survey. *IEEE Transactions on Software Engineering*, *26*(2), 98–112.

Rothwell, J. (2014). Short on STEM talent. Retrieved January 1, 2015, from http://www.usnews.com/opinion/articles/2014/09/15/the-stem-worker-shortage-is-real

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). Minneapolis, MN, USA.

Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programmmg performance. *Communications of the ACM*, *11*(1), 3–11.

Sandusky, R. J., & Gasser, L. (2005). Negotiation and the coordination of information and activity in distributed software problem management. In *Proceedings of International Conference on Supporting Group Work* (pp. 187–196).

Schank, R. C., Berman, T. R., & Macpherson, K. A. (1999). Learning by doing. In C. M. Reigeluth (Ed.), *Instructional-design theories and models: A new paradigm of instructional theory* (pp. 161–181). Lawrence Erlbaum Associates.

Schraw, G. (1998). Promoting general metacognitive awareness. *Instructional Science*, *26*(1-2), 113–125.

Shackelford, R., Andrew McGettrick, Robert Sloan, Topi, H., Davies, G., Kamali, R., … Lunt, B. (2006). Computing Curricula 2005: The Overview Report. *SIGCSE Bulletin*, *38*(1), 456–457.

Simon, H. (1955). A Behavioral Model of Rational Choice. *Quarterly Journal of Economics*, *69*, 99–188.

Simon, H. (1976). *Administrative Behavior* (3rd ed.). The Free Press.

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).

Soloway, E., Adelson, B., & Ehrlich, K. (1988). Knowledge and processes in the comprehension of computer programs. In *Nature of expertise* (pp. 129–152).

Sonnentag, S., Niessen, C., & Volmer, J. (1991). Expertise in software design. In *The Cambridge Handbook of Expertise and Expert Performance*.

Sowe, S., Stamelos, I., & Angelis, L. (2008). Understanding knowledge sharing activities in free/open source software projects: an empirical study. *Journal of Systems and Software*, *81*(3), 431–446.

Trifonova, A., Ahmed, S. U., & Jaccheri, L. (2009). SArt: towards innovation at the intersection of software engineering and art. *Information Systems Development*, 809–827.

Tuckman, B. W. (1965). Developmental sequence in small groups. *Psychological Bulletin*, *63*(6), 384–399.

Turley, R. T., & Bieman, J. M. (1995). Competencies of exceptional and nonexceptional software engineers. *Journal of Systems and Software*, *28*(1), 19–38.

Valett, J. D., & McGarry, F. E. (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, C. B. S. and C. U. D. K. (2000). Risk and return: expected return. Retrieved January 1, 2016, from http://ci.columbia.edu/ci/premba_test/c0332/s6/s6_3.html

Walkowski, D. (1991). Working successfully with technical experts—from their perspective. *Technical Communication*, *38*(1), 65–67.

Wenger, E. (1999). *Communities of practice: learning, meaning, and identity*. Cambridge University Press.

Wenger, E. C., & Snyder, W. M. (2000). Communities of practice: the organizational frontier. *Harvard Business Review*, *78*, 139–145.

Wikimedia Foundation. (2015). Sony pictures entertainment hack. Retrieved January 1, 2015, from https://en.wikipedia.org/wiki/Sony_Pictures_Entertainment_hack

Zachary, G. P. (1994). *Showstopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. Free Press.

Zetter, K. (2013). Target admits massive credit card breach; 40 million affected. Retrieved January 1, 2015, from http://www.wired.com/2013/12/target-hack-hits-40-million/

Zsambok, C. E., & Klein, G. (1996). *Naturalistic Decision Making (Expertise: Research and Applications Series).* Lawrence Erlbaum Associates.

APPENDIX A: SURVEY RECRUITMENT EMAIL

From: Paul Li

Sent: Wednesday, December 10, 2014 2:55 PM

To: [Expert Software Engineer at Microsoft]

Subject: Microsoft Research: What Makes A Great Developer?

Hi Adi,

We're doing a study on what makes someone a great developer. Would you be willing to fill out a quick **20 minute survey**, providing us with your expert opinion? We selected you from the company directory based on your title and experience.

The survey is anonymous and you are not obligated to participate. If you complete the survey, you will receive a report of the insights, as well as be entered into a drawing for **one of two $75 Visa gift cards.**

[Customized link to survey]

Thanks,

Paul Li

Senior Data Scientist, Microsoft; Ph.D. Candidate, Information School, University of Washington

Andrew Ko

Associate Professor, Information School, University of Washington

Andrew Begel

Senior Researcher, Microsoft

APPENDIX B: SURVEY

What Makes A Great Developer?

_____

Welcome!

What makes a great developer?

We need feedback from experienced developers, like yourself, about the importance of a set of attributes for being a great developer, based on developers you've worked with. We will be asking about attributes of developers in four groups: personal characteristics, decision making, interacting with others, and producing software.

This 20 minute survey is anonymous, you are not obligated to participate, and you can return to the survey later if you don't finish. If you complete the survey, you will receive a report of the insights, as well as be entered into a drawing for one of two $75 Visa gift cards.

Thanks
Paul Li: pal@microsoft.com: Senior Data Scientist, Microsoft; Ph.D. Candidate, Information School, University of Washington
Andrew Ko: ajko@uw.edu: Associate Professor, Information School, University of Washington
Andrew Begel: abegel@microsoft.com: Senior Researcher, Microsoft

_____

To get started, we'd like to know a bit about you

1) What is your current Microsoft job title?*

_____

2) What is your gender?*
( ) Male
( ) Female
( ) Other
( ) Decline to state

3) What is your age? (optional)

_____

4) How many years have you been a professional software developer (not including internships)?*

_____

5) How many different software development companies/organizations---including for profit companies, universities, and open source projects---have you worked for or contributed to?*

_____

6) How many years have you worked at Microsoft?*

_____

7) How many years have you been working on your current product area at Microsoft?*

_____

8) What educational degrees have you received?*
[ ] Bachelors/Associates
[ ] Masters (not Masters of Business Administration)
[ ] MBA (Masters of Business Administration
[ ] Doctorate
[ ] Other

9) What was the area of concentration of your Bachelors/Associates degree?*

_____

10) What was the area of concentration of your Masters degree?*

_____

11) What was the area of concentration of your Doctorate degree?*

_____

12) What degree did you receive?*

_____

13) Have you ever been a manager of developers (not including interns or vendors) at Microsoft (e.g. Lead or Manager)?*
( ) Yes
( ) No

14) Do you work in the United States?
( ) Yes
( ) No

15) What non-US country do you work in?*

_____

16) Have you ever worked, as a developer, in a non-US country?*
( ) Yes
( ) No

17) What non-US country did you work in the longest?*

_____

18) For how long (in years)?*
_____

19) What was the first language you learned (e.g. English, Spanish)?*
_____

20) What best characterizes the target customers/users of your software?*
( ) Our target customers/users are internal teams
( ) Our target customers/users are external people/organizations
( ) Both

21) How frequently do you release your software?*
( ) Daily
( ) Weekly
( ) Monthly
( ) Yearly
( ) Other

22) Please explain...
_____
_____
_____
_____

23) What best characterizes the software you currently produce?*
( ) The software run on customer/user devices. To change or upgrade the software, updates are shipped to customer/user devices to be installed.
( ) The software run on our servers and are accessed remotely (e.g. a service). To change or upgrade the software, changes are made on our servers; all future access are updated.
( ) Both

24) In the past year, how many developers have your worked with closely in producing your software?*
_____

_____

Personal Characteristics

In this section, we ask about 18 attributes of a developer's personality. We'll describe an experienced developer---whose primary responsibility is to develop software---with the attribute, along with a supporting quote. Please judge the importance of the attribute for being a great developer, based on developers that you've worked with in your career.

Continuously improving

A developer that is continuously improving is constantly looking to become better. This can mean improving themselves (e.g. learning new skills, learning new technologies, learning to do things better), their product (e.g. simplifying features, refactoring code), or their surroundings (e.g. automating processes)

25) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Passionate

A passionate developer is intrinsically interested in the area they are working in (i.e. they are not just in it for a pay check).

26) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Open-minded

An open-minded developer is willing to let new information change their thinking. They do not believe they know everything and will consider new information if it has merit.

> *" ... the problem is sort of in a way the inverse of sharing, which is people not being willing to take the input of others, to take what others are trying to share with them ...You've heard of NIH – not invented here. That's a huge problem." -Office developer*

27) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Systematic

A systematic developer does not rush to conclusions or jump to conclusions; they address problems in a systematic and organized manner.

28) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Data-driven

A data-driven developer measures their software and the outcomes of their decisions. They let actual data drive actions, not depending solely on intuition.

> *"If you're designing your feature, you need to put some telemetry features into there, collect customer data, and take some of that into account while you're making the next wave of decisions... Being data driven rather than instinct driven."–Server & Tools developer*

29) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Productive

A productive developer achieves the same results as others faster, or takes the same amount of time as others but produces more.

30) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Persevering

A persevering developer is not dissuaded by setbacks and failures; they keep on going, keep on trying.

> *"Ultimately, I will never give up. I will live here day and night to make sure it happens... intelligence is required but the people that continuously say, 'okay, I won't give up. I will try to find out a solution.' Those people always succeed." -Dynamics developer*

31) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Hardworking

A hardworking developer is willing to work more than 8 hr days to deliver the software product.

> *"Sometimes there's something that's just arduous. You really just need to grind through, like running a marathon. It's a long grind, hours and hours... " -Server & Tools developer*

32) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Curious

A curious developer desires to know why things happen and how things work (e.g. how the code and the conditions produce a software behavior).

> *"A curiosity. I think having [a need to know] how things work, why things work the way they work... Wanting to tear something apart, figure out how it works, and understand the why's" -Xbox developer*

33) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Willing to go into the unknown

A developer that is willing to go into the unknown is willing to step outside of their comfort zone to explore a new area (e.g. new technologies, new tools, new role, etc.), even when there might be risks or when benefits are not immediately known.

> *"People are just naturally going to gravitate towards their comfort areas and just kind of hang out there... But if you're willing to take those risks and learn about other things, and then actually apply them, they can help move you forward. But applying them might mean getting out of your comfort zone." -Windows developer*

34) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Adapts to new settings

A developer that adapts to new settings continues to be valueable to the organization even with changes in their environment, such as changes in what they work on and changes in their team.

> *"Things are going to change. What are you going to do about that? Are you going to be one of the people that is helping to change? ... everything from values to fit into the group, or the product, or the problem you're trying to solve... How are you going to take and adapt your situation to move forward, and how do you adapt to work with what you have to work with?" -Service Engineering developer*

35) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Self-reliant

A self-reliant developer gets things done independently and does not get blocked easily; they get around problems by leveraging their abilities and other resources (e.g. asking experts for help).

36) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Self-reflecting

A self-reflecting developer can recognize when things are going wrong or when their current plan is not going to work, and then self-initiate corrective actions.

> *"... a little bit of an intuition, and maybe the ability to see where you're going wrong, and then step back. So, self-reflection is important: being able to recognize, yeah, this ain't working, I better start over." -Xbox developer*

37) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Aligned with organizational goals

A developer that is aligned with organizational goals takes actions for the good of the product and the organization, not for their own self-interest. They do what is good for the organization, not just what interests them.

38) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Executes

A developer that executes does not have analysis paralysis. They know when to stop thinking and to start doing.

> *"They should not be just idealistic software designers, where you can think a lot; they should not get into analysis paralysis... write the most optimal solution for the problem on hand."-Phone developer*

39) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Craftsmanship

A developer that has craftsmanship takes pride in their work. They want their output to be a reflection of their skills and abilities.

> *"Really being able to demonstrate something that you've done, that you're really proud of it, and speak to it well. When you do your work, you take pride in the fact that it's quality work." -Xbox developer*

40) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Desires to turn ideas into reality

A developer that desires to turn ideas into reality takes pleasure in building, constructing, and creating software.

> *"You have an urge to create. You get satisfaction from creating...They feel more accomplished at the end of the day if they've actually built something... wrote some code." -Windows developer*

41) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Focused

A focused developer allocates and prioritizes their time for the most impactful work. They do not let the numerous daily distractions and tasks overwhelm them.

"In an environment like Microsoft where there's a lot of meetings and interruptions… A developer has to figure out how to get their focus and when to get their focus. ...Figure out when he can get away from the chaos of the day-to-day, [and then] he could come back and make very good use of that time."-Windows Services developer

42) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Decision Making

In this section, we ask about 9 attributes of a developer's ability to make good decisions. We'll describe an experienced developer---whose primary responsibility is to develop software---with the attribute. Please judge the importance of that attribute for being a great developer, based on developers that you've worked with in your career.

Knowledgeable about people and the organization

A developer that is knowledgeable about people and the organization is informed about the people around them: responsibilities (i.e. organizational structure), knowledge (i.e. the domain experts), and tendencies.

43) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Sees the forest and the trees

A developer that sees the forest and the trees can reason through situations and problems at multiple levels of abstraction: technical details, industry trends, company vision, and customer/business needs.

"The really great developers are the ones who find the sweet spot in between two extremes. [They] are able to understand and consider the very large picture; while at the same time, work at a very detailed level, and not get lost and bogged down in the details." -Office developer

44) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Updates their decision making knowledge

A developer that updates their decision making knowledge does not let their understanding and thinking stagnate; they update their decision making with regards to changes around them (e.g. new technologies, industry trends, organizational changes).

"...the world has changed ...Unlearning: the things that I used to do five years ago that made me successful don't matter anymore; in fact, they can get me into trouble right now... I would assess their ability to unlearn: after a while, two thirds or three quarters of what you know is still valuable, quarter to a third is the wrong thing... the trick is to figure out which is which really quickly..." –Server & Tools developer

45) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Mentally capable of handling complexity

A developer that is mentally capable of handling complexity is able to comprehend and understand complex situations, especially ones involving multiple layers of technology and many interacting/intertwining software.

"... [Being] able to solve deep architectural problems, come up with a design that spans multiple different components... Some people's brains operate faster than others... [it's] an indicator of intellectual horsepower."-Servers & Tools developer

46) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Knowledgeable about their technical domain

A developer that is knowledgeable about their technical domain is thoroughly conversant about their software product, their technology area, and their competitors.

47) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Knowledgeable about customers and business

A developer that is knowledgeable about customers and business understands the role their software product plays in the lives of their customers and the business proposition that it entails.

"...understanding your customer, find out what they've got, what they want, what they already do, what's the delta you can provide, how can you help, and then go find a simple solution to it. Because at the end of the day, we are a for profit company." -Xbox developer

48) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Knowledgeable about tools and building materials

A developer that is knowledgeable about tools and building materials knows the strengths and limitations of the tools and building materials used to construct their software product (e.g. algorithms, programming languages, code libraries, etc.).

> *"If you write in Java, you're probably not going to have performant code... It's just the constricts you're given in Java... language is like a tool... a good developershould be able to realize that a certain language is not the right tool for that particular job."–Windows developer*

49) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Knowledgeable about software engineering processes

A developer that is knowledgeable about software engineering processes knows the practices and techniques for building a software product (e.g. unit testing, code reviews, Scrum, etc.): their purposes, how to do them effectively, their cost in time and effort, and when best to use them.

50) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Grows their ability to make good decisions

A developer that grows their ability to make good decisions builds their understanding of real world situations, identifies alternative courses of action, projects likely outcomes, and estimates the values of the outcomes.

"When you're right, evaluate it: why were you right? were you lucky? ...When you're wrong, do the same thing: was it bad luck? or was it bad insight? ...Correcting things as you go... you'll soon be operating on theories about how things should work... rework that theory until you converge at something that's functional." -Xbox developer

51) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?
( ) 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 ( ) I do not know

---

Interacting with others

In this section, we ask about 18 attributes of a developer's interactions with others. We'll describe an experienced developer---whose primary responsibility is to develop software---with the attribute, along with a supporting quote. Please judge the importance of that attribute for being a great developer, based on developers that you've worked with in your career.

Creates shared understanding with others

A developer that creates shared understanding with others molds another person's thinking of the situation: tailoring the communication to be relevant and comprehensible to the other person so that the other person can incorporate the information into their thinking.

"Understand how to most compellingly relate the value of that abstraction... to each person in the communication chain: their peers, as developers, their testers, their PMs, their designers, their management. Or if they were to speak at a conference or do demos or interviews... empathize with your audience, whether they are groups or individuals, in order to get them to get it..." -Windows developer

52) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Creates shared success for everyone

A developer creates shared success for everyone involved (i.e. win-win situations). They engage with others to decide on actions that is beneficial to everyone---not just themselves---commonly involving establishing a common big picture or long-term goals that everyone can buy into.

> *"...find the common good in a solution, and be able to say: 'I'm pushing for a solution, here's the value for me, and also here's the value for you.' Understanding their concerns to the point where you can actually have them saying, 'Yeah this is the right thing to do. This is the right thing to approach and go with.' Even though you're still accomplishing the goals you want, they're feeling like they're winning. It's a win-win situation." - Windows developer*

53) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Creates a safe haven for others

A developer creates a safe haven for others, where others are not afraid of being blamed for mistakes, empowering others to do what they feel is right, and to learn and grow.

> *"I think failing is good, if you learn something from a failure, that's a wonderful sort of thing. I don't even think of failing as taking a risk; that should just be part of your normal learning experience... If you're afraid of getting smacked upside the head because you made a failure, you're taking a smaller risk there." -Office developer*

54) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Honest

An honest developer is truthful: not sugar coating or spinning the situation for their own benefit. They provide credible information and feedback that others can act on.

> *"'You know what? I know that this person always speaks the truth.' ...they say whether something is good or bad ...whether or not something was successful that they did. They're not trying to paint a too rosy picture... when they say something is good, I will totally believe them because they are not trying to misrepresent something or make them look better." -Windows Services developer*

55) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Integrates understandings of others

A developer that integrates understandings of others can combine and integrate the knowledge of others---especially when there are multiple people, each with their own understanding of the situation---into a more complete understanding, noticing and asking questions about the gaps.

> *"If they say something that doesn't really line up with your intuition... ask questions and try to figure out where the discrepancies lie... internalize it and connect it with the way you think about things... incorporated into your own; mesh it with you own knowledge base." -Xbox developer*

56) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Well-mannered

A well-mannered developer treats others with respect: not obnoxious about titles, accolades, or knowledge.

57) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Is a good listener

A developer that is a good listener effectively obtains, comprehends, and understands others' knowledge about the situation.

> *"Being a good listener is important: you're really hearing the other person's concerns and opinions." -Windows developer*

58) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Does not make it personal

A developer that does not make it personal avoids personal biases. They act and react based on fact and reason, avoiding dysfunctional behaviors based on personal feelings and perceived slights.

> *"You can have a very good discussion... it never gets personal. Oh, this is your idea, and it's good or it's bad. It's all very professional." -Server & Tools developer*

59) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Mentoring

A developer that is mentoring teaches, guides, and instills knowledge to others, helping others---often new team members---to improve and to be more productive.

> *"...[He's] seen stuff that you haven't seen yet, and he's willing to share his knowledge. The kind of people that horde their own knowledge, I have no time for that. It's great that they have the knowledge and they can be successful, but we're a company, we're trying to survive, let's spread some of that good knowledge around." -Office developer*

60) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Challenges others to improve

A developer that challenges others to improve, challenges others to take action (e.g. doing something new or taking on more responsibilities), expanding others' limits and capabilities.

> *"... the way he communicates implies that he believes that you can do it. There's this shared confidence: it's like he's done it and so you can do it... he has to be able to spark your imagination and your sense of self confidence for you to boot strap yourself up to being a productive developer." -Windows developer*

61) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Walks-the-walk

A developer that walks-the-walk is an exemplar for others: being a great developer themselves, letting others see their actions, and inspiring others to follow.

> *"...I would like to model myself against that [developer's] behavior. It inspires me to do the same thing..." -Ad Platform engineer*

62) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Manages expectations

A developer that manages expectations sets forth what they are going to do and by when, updates expectations (e.g. explaining impacts and implications of unexpected problems), and then delivers on promises.

63) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Has a good reputation

A developer that has a good reputation has the belief, respect, and confidence of others. They have a track-record of success such that they are trusted with current and future decisions.

> *"... it's because I trusted [him]. I've seen his previous work. I knew about it. I've seen him probably make other recommendations that turned out to have good outcomes... You have to build up that reputation and that trust through your years... so that when you make that recommendation, they go, I am going to listen to him." -Windows Services developer*

64) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*

( ) 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 ( ) I do not know

---

Resists external pressure for the good of the software product

A developer that resists external pressure for the good of the software product will articulate and advocate actions that are for the good of software product (e.g. not doing last minute features or slipping the schedule for bug fixes), being firm against outside pressures (e.g. management, partner teams).

> *"If what they're asking him to do jeopardizes something else, he'll say no. He can stand up and be brave about it. He might come back and say 'Well, we can think about this and try to plan it the right way for next time around, but right now we'd just be bolting it in and asking for more trouble.'" -Windows developer*

65) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Trades favors

A developer that trades favors builds personal equity with others, such that the developer can call upon others to do them personal favors.

> *"It's you returning a favor here and there... someone goes above and beyond to help somebody else out, and then somewhere down the road that person has that extra good will to come help you out." -Windows developer*

66) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Personable

A personable developer is a person that others enjoy interacting with; they establish good personal relationships with others.

> *"... one of the characteristics I look for in every person that I get, coder or not, but definitely if it was a coder is, 'Can I have a beer with this guy?'... That's important, because if I can't then we can't really work together." -Servers & Tools developer*

67) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Asks for help

A developer that asks for help will find and engage others with needed knowledge and information. They know the limits of their knowledge and supplement their knowledge with the knowledge of others.

> *"Without asking for help, you won't learn anything in big company like Microsoft... To get to the right thing, you are dependent on so many people... You should not be afraid, just go, reach out to people, 'Tell me this thing.'" - Windows Services developer*

68) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Does due diligence beforehand

A developer that does due diligence beforehand searches for and examines available information (e.g. documentation, code samples, wiki, etc.) before engaging. They are prepared when they discuss situations and do not waste others' time.

> *"I don't respect people who don't do their homework... they don't read the MSDN article, they don't download the SDK, they don't read the help files, they don't read the sample code... they just shoot off an email to the distribution list..."' - Windows developer*

69) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Software Product

In this section, we present 9 attributes of the software and designs that a developer produces. We'll describe an experienced developer---whose primary responsibility is to develop software---with the attribute, along with a supporting quote. Please judge the importance of that attribute for being a great developer, based on developers that you've worked with in your career.

Elegant

The developer can produce elegant software: intuitive (i.e. minimum complexity) design solutions that others can understand.

> *"...very 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." -Windows developer*

70) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Creative

The developer can come up with creative solutions: novel and innovative solutions based on understanding the context and limitations of existing solutions.

> *"...a traditional solution ...usually with solutions we often have constraints. Being creative is... take these constraints, take the difficult circumstance, and actually make it into something that could still work, but without a huge complex overhead." -Windows Services developer*

71) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Anticipates needs

The developer produces software that anticipates needs---problems and needs not explicitly known at the time of creation---based on their knowledge and understanding.

> *"He was really good at coming up with examples of how people might want to use technology... How would you maybe change your design with that in mind? Or that we might have to accommodate inter-operating with that technology in the future?" -Windows developer*

72) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Makes informed trade-offs

The developer makes informed trade-offs in their software (e.g. code quality for time to market), meeting critical needs of the situation.

73) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Pays attention to coding details

The developer produces software that pays attention to coding details, including error handling, memory consumption, performance, and style.

74) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Fits together with other pieces around it

The developer produces software that fits together with other pieces around it, such as environmental constraints, complementary components, and other products.

75) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Evolving

The developer can produce software designs and architectures that are evolving: structured to be effectively built, delivered, and updated in pieces.

76) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Long-termed

A developer that is long-termed considers long-term costs and benefits in producing software and designs, not just short-term gratification.

> *"If you packaged up a bunch of isolated, fragmented, short-term solutions together, what do you get? Not something great... long-term vision and say, 'We make decisions not based on the immediate problem. We make decision based on some long-term goal and some real principles we follow.'" -Corp Dev developer*

77) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

---

Uses the right processes during construction

The developer that uses the right processes during construction using the right processes (e.g. unit testing and code reviews) to construct their software and designs, in order to deal with potential problems.

78) If an experienced developer---whose primary responsibility is developing software---did not have this attribute, could you still consider them a great developer?*
( ) 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 ( ) I do not know

_____

Did we miss anything?

79) Finally, may we contact you learn more about your answers? (Optional)
( ) Yes

80) Did we missed any attributes of great developers that you've worked with? If so, what are the attribute(s) and how have they been important, in your experience? (Optional)
_____
_____
_____
_____

_____

To receive the report and be entered into the raffle

81) (Optional) To receive the findings, as well as to be entered into the drawing for one of two $75 Amazon gift certificates, please provide your email below. Your email will be not be associated with your answer, will not be shared, and will be deleted once the drawing is completed.
_____

_____

Thank You!

Thank you for taking our survey. Your time and input is greatly appreciated.

_____

APPENDIX C: INTERVIEW SOLICITATION EMAIL FOR EXPERT NON-SOFTWARE ENGINEERS

From: Paul Li
Sent: Sunday, June 7, 2015 12:57 PM
To: [an expert artists]
Subject: Understand artists and developers

Hi [Name]

I'm working on a research project with the University of Washington and Microsoft Research aiming to understand attributes of great developers, which includes examining insights and opinions of non-developers working on engineering teams, like yourself.

Your 'artist' role is one of the ones I'm particularly interested in. I know little about what you do, your interactions (if any) with developers, and your perceptions about the great developers that you've worked with.

So, I was wondering if you might have an hour free for me to interview you (anonymously), to learn about what you do and to get your take on the subject.

Thanks
Paul

VITA

Paul Luo Li

Senior Data Scientist
Microsoft

Research Area

My research interests are in software engineering expertise: the distinguishing attributes of great software engineers and why those attributes are important in the real-world engineering of software.

Education

2016	**Ph.D. in Information Science** University of Washington What Makes a Great Software Engineer? Andrew J. Ko (Chair), David Hendry (Washington), Andrew Begel (Microsoft), Charlotte P. Lee (Washington)
2007	**M.S. in Software Engineering** Carnegie Mellon University
2001	**B.S. in Mathematics – Actuarial Statistics** University of Virginia

Professional Experience

2015-present	Senior Data Scientist Microsoft
2007-2015	Program Manager Microsoft
2005-2006	Data Analyst IBM
2005-2005	Software Researcher ABB
2004-2004	Research Intern

IBM

2003-2003 Research Intern
Avaya

Patents

2011 Network Hang Recovery
Patent No. 7,934,129
Paul L. Li, Andrew J. Lagattuta, Matt Eason, Baskar Sridharan, Abdelsalam Heddaya, Stephan Doll

Publications

What Makes a Great Software Engineer?
Paul Luo Li, Andrew J. Ko, Jiamin Zhu (2015)
International Conference on Software Engineering: 700-710

Characterizing the differences between pre- and post- release versions of software
Paul Luo Li, Ryan Kivett, Zhiyuan Zhan, Sung-eok Jeon, Nachiappan Nagappan, Brendan Murphy, Andrew J. Ko (2011)
International Conference on Software Engineering: 716-725

Reliability Assessment of Mass-Market Software: Insights from Windows Vista®.
Paul Luo Li, Mingtian Ni, Song Xue, Joseph P. Mullally, Mario Garzia, Mujtaba Khambatti (2008)
International Symposium on Software Reliability Engineering: 265-270

Estimating the Quality of Widely Used Software Products Using Software Reliability Growth Modeling: Case Study of an IBM Federated Database Project
Paul Luo Li, Randy Nakagawa, Rob Montroy (2007)
Empirical Software Engineering and Measurement: 452-454

Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc.
Paul Luo Li, James D. Herbsleb, Mary Shaw, Brian Robinson (2006)
International Conference on Software Engineering: 413-422

Predictors of customer perceived software quality
Audris Mockus, Ping Zhang, Paul Luo Li (2005)
International Conference on Software Engineering: 225-233

Forecasting Field Defect Rates Using a Combined Time-Based and Metrics-Based Approach: A Case Study of OpenBSD
Paul Luo Li, James D. Herbsleb, Mary Shaw (2005)
International Symposium on Software Reliability Engineering: 193-202

Finding Predictors of Field Defects for Open Source Software Systems in Commonly Available Data Sources: A Case Study of OpenBSD
Paul Luo Li, James D. Herbsleb, Mary Shaw (2005)
IEEE International Software Metrics Symposium: 32

Empirical evaluation of defect projection models for widely-deployed production software systems
Paul Luo Li, Mary Shaw, James D. Herbsleb, Bonnie K. Ray, Peter Santhanam (2004)
ACM SIGSOFT International Symposium on the Foundations of Software Engineering: 263-272