Appendix to:
What Makes a Great Software Engineer?

March 7, 2019
Technical Report
MSR-TR-2019-8

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Appendix to:
What Makes A Great Software Engineer?

Paul Luo Li[*+], Andrew J. Ko[*], Jiamin Zhu[+]

Microsoft+
Seattle, WA
{pal,jiaminz}@microsoft.com

The Information School*
University of Washington
ajko@uw.edu

APPENDIX DESCRIBING ATTRIBUTES OF GREAT SOFTWARE ENGINEERS CORRESPONDING TO (Li, Ko, & Zhu, 2015)

Our analysis (Li et al., 2015) 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 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.

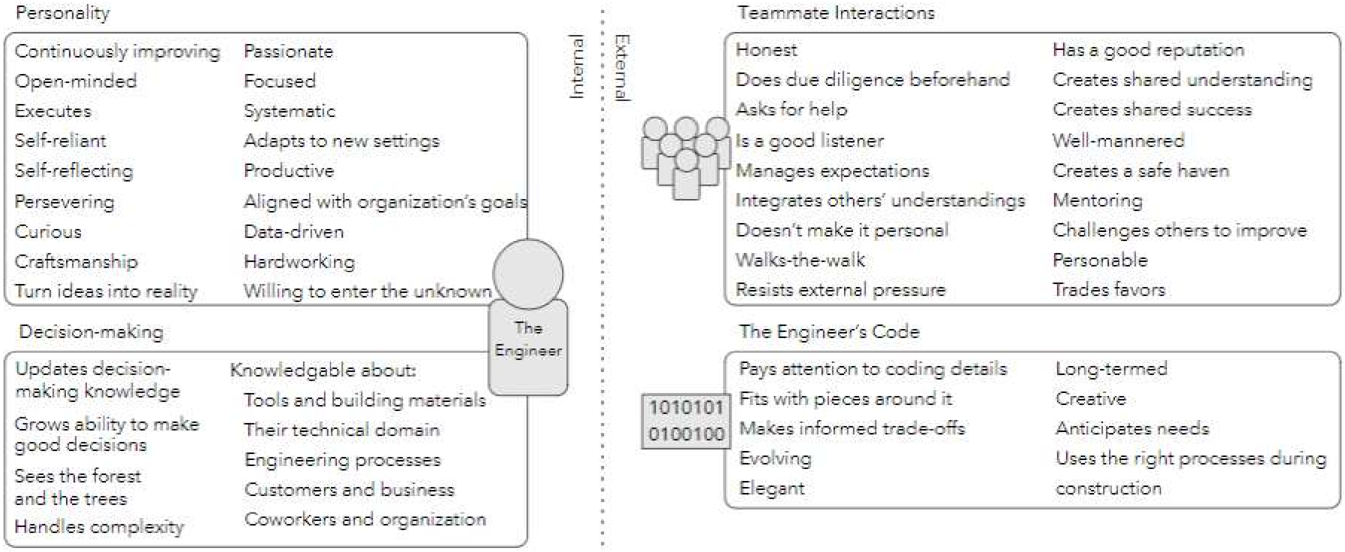

Figure 1. Model of attributes of great software engineers

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

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), 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.

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.

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.

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*

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.

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.

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*

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.

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*

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*

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 and productive; 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*

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*

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*

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.

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, DeLine, & Venolia, 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.

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). 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*

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. 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*

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*

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*

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*

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.

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*

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).

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. ‘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, Zubrow, Goldenson, Hayes, & Paulk, 1997)).

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.

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, 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, Hiew, & Murphy, 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, Kim, & Zimmermann, 2009)) are all important to the bug triaging/assignment process of software engineering teams.

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. 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.

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*

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.

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*

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 (Radermacher & Walia, 2013) (Cruz, da Silva, & Capretz, 2015). In our discussions, we deconstruct these four concepts into their constituent attributes and then examine each attribute separately.

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, Capretz, & Campbell, 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—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. 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*

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*

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*

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.

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; 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*

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*

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*

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*

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*

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). 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*

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*

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.

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*

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. Therefore, resisting the desires/wishes of the organization could not be good for the software product.

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*

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 well-mannered: 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. Well-mannered was the best known and easily identified attribute among our informants; even software engineers who did not discuss this attribute immediately recognized well-mannered 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, 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*

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*

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*

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.

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*

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*

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 or anticipates needs—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—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*

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*

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*

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*

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*

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*

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. 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.

BIBLIOGRAPHY

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?

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).

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).

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering* (2nd ed.). Addison-Wesley Professional.

Clark, H., & Brennan, S. (1991). *Perspectives on Socially Shared Cognition*. American Psychological Association.

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).

Edmondson, A. (1999). Psychological safety and learning behavior in work teams. *Administrative Science Quarterly*, *44*(2), 350–383.

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.

Iqbal, S. T., & Horvitz, E. (2007). Disruption and recovery of computing tasks: field study, analysis, and directions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 677–686).

Jeong, G., Kim, S., & Zimmermann, T. (2009). Improving bug triage with bug tossing graphs. In

Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 111–120).

Joint Task Force on Computing Curricula. (2014). *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. *ACM Curricula Recommendations*.

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*. Carnegie Mellon University.

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., 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 *Proc ICDMW '13*.

Krishnamurthi, S., & Felleisen, M. (1998). Toward a formal theory of extensible software. *ACM SIGSOFT Software Engineering Notes*, *23*(6), 88–98.

Li, P. L., Ko, A. J., & Zhu, J. (2015). What Makes A Great Software Engineer? In *Proceedings of the 37th International Conference on Software Engineering*.

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 ed.). Microsoft Press.

Myers, C. R. (2003). Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Physical Review E*, *68*(4).

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

Perlow, L. A. (1999). The Time Famine : Toward a Sociology of Work Time. *Administrative Science Quarterly*, *44*(1), 57–81.

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. 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*. O'Reilly Media.

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).

Schraw, G. (1998). Promoting general metacognitive awareness. *Instructional Science*, *26*(1–2), 113–125.

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.

Microsoft Research. Technical Report. MSR-TR-2019-8

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.

Zachary, G. P. (1994). *Showstopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. Free Press.