A Three-Year Participant Observation of Software Startup Software Evolution

Andrew J. Ko
The Information School, DUB Group
University of Washington, Seattle
AnswerDash, Inc.
ajko@uw.edu

***Abstract*—This paper presents a three-year participant observation in which the author acted as CTO of a software startup, spanning more than 9,000 hours of direct experience. The author's emails and diary reflections were analyzed and synthesized into a set of nine claims about software engineering work. These claims help shape software engineering research, practice, and education by provoking new questions about what makes software engineering difficult.**

Keywords- management, human factors, project management.

I. INTRODUCTION

In past decades, there has been much progress in studying how developers work. We understand many of the questions they ask [34], the information they need [19], and many other factors that affect their productivity [7,23,24]. These discoveries have informed the design of many tools and processes that may positively affect practice [13,27].

These discoveries, however, are heavily biased toward the *observable* activities that developers perform, overlooking developers' internal, emotional, and cognitive experiences at work. For example, what does it feel like to be an engineer day to day? Which parts of the job are exhilarating and which parts are dull? What role do developers' social experiences with their teammates and managers have on their work? How do these experiences change and evolve over time, especially as developers learn and an organization evolves?

These aspects of developer experience are important for many reasons. Emotion, for example, is a dominant factor in decision making [26], and by not studying its role, researchers may overlook large factors that shape software development. Longitudinal studies of experience are also important as many challenges in software engineering occur over years. Investigating experience may also reveal new opportunities to innovate in tools, process, and education.

Unfortunately, experience over time is difficult to observe. Surveys and experience sampling can get at some aspects of experience, but often lack depth. Interviews offer depth, but require developers to recall their experiences, leaving data subject to memory bias. Diary studies can capture depth over time [12], but most developers are unlikely to spend significant time writing about their experiences over the months or years. Moreover, in all of these methods, developers are unlikely to share the more emotionally challenging aspects of their work, masking potentially powerful factors that shape their daily work.

One way to observe developer experience longitudinally is for *researchers* to engage in engineering themselves, reflecting on their own experiences. For example, in 1975, Brooks reflected on his industry experiences in *The Mythical Man Month*, presenting over 200 testable propositions about software engineering [7]. Similarly, in 1989, Knuth published the *Errors of TeX* [18], a diary study documenting and reflecting on the 850 errors that he made over a decade of work. Each of these works provided rare glimpses into programming, project management and software maintenance, informing research on software engineering.

In this paper, I report on a similar self-examination, describing my three-years as CTO and co-founder of a venture-backed software startup in Seattle. To study my experiences, I wrote daily in a personal diary and archived over 15,000 emails exchanged with my co-founders and employees. Both data sources focused on my experiences as a founder, executive, manager, and developer as our company and software evolved. I then analyzed my diary and emails, deriving nine novel claims about software engineering to be tested in further research. In the rest of this paper, I describe my method in detail and then present my claims.

II. METHOD

Knuth's study and my study are both examples of *participant observation*. This is a method long used in cultural anthropology in which the researcher is both an observer *and* a participant in some activity over time [16]. Participant observations have the unique strength of describing complex aspects of cognition, social interaction, and culture over time, and can be used to improve business from within [37]. But they also have limitations: they require introspection, which is subjective [30]; they represent a single perspective; and they can suffer from the *observer-expectancy* effect, in which the presence of an observer influences other participants' behavior [33]. These limitations are usually mitigated through 1) triangulation of other data sources, and 2) transparency about the observers' biases and beliefs, allowing the reader to better interpret the subjective observations [2].

I used two sources of data: my personal diary and emails exchanged with my coworkers. I then also asked my co-founder and my VP of Engineering to read this paper to corroborate my perspectives, noting when their experiences differed. These two sources of data aimed to go beyond my individual perspectives to better reflect the perspectives of three key executives. This ensured that I used more than my diary, emails, and memory to reconstruct events.

To increase transparency, here I describe the many biases that influenced my data and analysis. I am a researcher, versed in studies of human aspects of software engineering, and so I

brought to my observations a broad knowledge of human factors in software engineering research. I have done several studies of software companies that have shaped my perceptions of how software compares work. My identity as a researcher shaped the decisions I made as a CTO, biasing them toward evidence over intuition. I have a background in Human-Computer Interaction research, which predisposes me to sociotechnical explanations of phenomena. I had no prior experience in professional software engineering and so many of my observations may have emerged from my learning. Finally, because I live in Seattle, I have many peers who work in the software industry. My incentives in participating in the company were also mixed: I wanted to build a successful business, but as a researcher, I also wanted to understand how our business evolved.

My data collection across the three years was frequent and in situ, as is best practice [2, 12]. I wrote daily on everything I observed, including decision-making by myself and others, management choices, the evolution of our product and engineering management, our technology stack, our product's architecture, the role of tools, and the role of research. My focus was biased toward events that caused me negative emotions such as stress, anxiety, frustration, boredom, and confusion, and toward phenomena I was curious about, such as software process, design, product management, and engineering management. I amassed over 15,000 emails between me, my co-founders, my employees, and our customers, many of which included tense exchanges about challenging engineering, management, and business problems about which my co-workers and I had differing views. My diary and emails ultimately spanned 9,000 hours of work from December 2012 to December 2015.

III. The Origins of the Business

The kernel of the business was my National Science Foundation CAREER grant, awarded in 2009. In the grant, I observed that software help requests by end users are fragmented, duplicated, and disjoint online because they are inconsistently expressed. If we could structure help requests so that users expressed similar problems in similar ways, we could both retrieve help more reliably, but also provide aggregate data about bugs, usability issues, and feature requests. My Ph.D. co-advisee Parmit Chilana led this work, inventing a Q&A retrieval technique called *LemonAid*, with some help from me and her other co-advisor Jacob O. Wobbrock (hereafter "Jake"). The technique allowed end-users to select user interface elements in the website they wanted help with (e.g., text, images, buttons). Our system then retrieved Q&A that were most relevant to their selection [8]. Across four deployments to public web sites owned by the University of Washington (UW), we saw significant rates of usage and successful answer discovery and many users said they preferred it over all other forms of help [9].

Parmit presented her research while interning at Facebook in October 2010. During her visit, the company asked how the retrieval worked. Parmit declined to answer and returned to UW to discuss the intellectual property implications with our technology transfer office. Intrigued by the prototype, the tech transfer staff connected us with an "entrepreneur-in-residence" to discuss opportunities. He encouraged us to consider commercializing. After almost a year of deliberation (including whether Parmit would finish her Ph.D., whether Jake would spend his sabbatical raising financing for the company, and whether I would moonlight the year before going up for tenure), Jake and I decided to spin out a business and temporarily leave the university, while Parmit decided to focus on her academic job search. I became CTO because of my substantial experience with web development and Jake took the CEO role to focus on strategy, customers, and financing. We founded the company on September 24th, 2012.

To begin, Jake and I started talking to customers with the help of the entrepreneur-in-residence, who we hired as a business development consultant. Our first sales pitch was with a small startup looking for a contextual help solution. We had no product, and so our pitch relied on a video of the *LemonAid* prototype. The startup loved it and wanted to launch in January 2013. It was December 14th, 2012.

I only had two weeks to build a production-ready alpha. I did not have time to learn the modern web technologies at the time (*Backbone*, *AWS*, *node.js*, *Ruby*, *Postgres*, etc.), and so I started with the technology stack I knew: *Linux*, *Apache*, *MySQL*, *PHP*, and *jQuery*. By January 4th, 2013, I had a deployable alpha and we launched on that customer's site. We gained several additional customers in subsequent months.

Feature requests swamped the 15 hours per week that I had dedicated on top of my full-time faculty work. I quickly saw the consequences of my dated technology stack, with poorly logged error messages in *PHP*, callback spaghetti in *jQuery*, and the burden of server administration. Responding to feature requests meant taking on more technical debt in service of more sales pitches, more fundraising pitches, and better product customer fit. In my mind, all of this debt would be worth accruing if we could close funding, so we could hire a larger engineering team to build a better infrastructure.

These investments eventually paid off. We closed $2.54 million in venture capital on December 2, 2013 and began hiring. Despite my fears of my rushed implementation deterring new hires, many candidates viewed it as an opportunity to build a modular, scalable 1.0 product. And with our team, that's exactly what we did, releasing on modern infrastructure with a better architecture. As our team grew and Jake's responsibilities as CEO expanded, I took on the role of product manager, engineering manager, and architect for the next two years. In my last year, I handed the role of engineering manager to a new VP of Engineering.

After about three years as the company's CEO and CTO, Jake and I left the company, hiring new leadership to scale the business. At the time of my departure, the company had seven developers, a VP of engineering, a designer, three sales and marketing staff, a customer success manager, an admin, and a CEO. Our implementation at the time ran dozens of services hosted on Amazon Web Services spanning hundreds of thousands of lines of code written in *PHP*, *Python*, *JavaScript*, *HTML*, *CSS*, *LESS*, *Java*, and *Common Lisp*. The team ran continuous integration, released daily, had an extensive test automation infrastructure, performed code and design reviews, and followed many other modern practices for building secure, scalable web-services.

IV. CLAIMS

My three years of experience amounted to over 100,000 words in my diary and emails. To analyze this data, I first chronologically read every word of my diary and emails, identifying trends in my experiences and reflections. I noted each trend in the form of a claim that I believed applied to all of my experiences (e.g., "Decisions lack sufficient evidence." or "Debugging was straightforward."). From this first read I derived 80 claims. I then consolidated, reconciled and synthesized the claims into a final set of 9 claims. I then re-read the diary and emails, tagging each one with one or more of the claims. I then drafted a description and illustration for each claim, using stories from my data as supporting evidence. Finally, I sent drafts of each claim to Jake and my VP of Engineering to look for experiences or interpretations they did not agree with. This resulted in several small revisions to details in the reporting, but no changes in the claims themselves, as both found the claims described their experiences. The text in the rest of this section represents the final synthesis and triangulation of these nine claims.

A. *Claim 1: Software engineering is structured by individual and social decision-making under uncertainty*

Many studies, including some of my own, frame software development as an activity of information seeking, arguing that developers have questions about code [22,34], their coworkers' activities [23,19], and the rationale for a product's functionality [19], and their ability to answer these questions affects their productivity [19].

While information seeking was a fundamental part of my experience, it was by no means the most salient aspect of my time. Both early in the company and three years in, it was the decisions I made (alone and with others) that structured my work and the work of my engineers. Including the hundreds of thousands of lines of code that I wrote, the hundreds of mockups I designed, the processes I devised and the systems I structured for managing information, I estimate there were nearly a quarter million decisions—about 25 per hour, 3,000 hours of work a year.

These decisions varied widely in their scope and significance. They included choices such as: *What's an extensible way to structure this data? How can I automate this test? Is this bug important enough to fix right now? How do I convince this customer that their feature request is a bad idea? How do I convince my CEO that this feature is critical to growth? How do I get my engineers to make this deadline? Do these stand-up meetings have sufficient return on investment? Where does this product need to be before the end of this financial quarter? Should I go home to my family or finish these board deck slides for tomorrow?* Each decision led to information needs, but the information was only in service of informing decisions.

Decisions are among the more difficult things that human beings do, and so we satisfice [35]. This was necessary not only because I rarely had all of the information necessary to enumerate or weigh alternatives, but I also rarely had an accepted utility function with which to *evaluate* alternatives. Take, for example, the decision of whether to fix a bug. My team once found a particularly nasty data loss defect that, to fix, required significant re-architecting, while delaying progress on other important feature work. Some engineers preferred to decide on principle, viewing data loss as inherently unacceptable. Others were pragmatists, focusing on how frequently the bug was encountered and how severe the loss of data would be to customers. Others focused on return on investment, asking what the company would gain and what it would lose if we delayed the fix. All of these were legitimate decision frames, so the only way to resolve the conflict was for me to decide using my authority. This is similar to the decision making observed in open source projects, in which speculation, anecdote, and overgeneralization are common as rhetorical devices, but authority ultimately wins [20].

When we sought information to inform decisions, there were rarely clear answers. We could not know how frequently the data loss was occurring because we had no logs of that feature's use. We did not know how customers would react and did not want to risk the loss of trust that would come in asking them. And trying to compute the return on investment of the fix was near impossible: if the loss turned out to matter greatly to a customer, would we lose just them, or would we lose other customers too? Because of the degree of uncertainty, I began to view my role as shielding engineers from uncertainty, so they could focus on code-level decisions.

Uncertainty had several implications. For example, although we logged usage, carefully archived customer requests and feedback, and leveraged the state of the art in web site analytics, this data was of limited use because of the uncertainty of other factors for which we had no evidence. For example, logging suggested that customers were rarely going to our analytics dashboard. Whether this mattered depended on how much value customers were getting from their infrequent visits. We could not easily know this and so we relied on our expertise to guess. It was hard to say whether this reliance on expertise was negative: by the time we found out whether our facts or expertise were right, it was too difficult to trace the origins of our decisions.

Trust was another significant factor in decisions. When I solicited opinions, I needed to factor in how much I believed them. *Were the CEO's beliefs about our product's traction well informed? Did my junior engineer correctly parse our Apache logs for traffic to that page? How much faith do I put in my head of sale's interpretation of that customer phone call?* And of course, the other executives asked the same questions about me, learning that I was prone to weighing data over expertise, bottom line over technical and design factors, and being better at strategic matters than tactical matters. The constant need to trust and be trusted was an extra layer of relationship management that I did not expect.

B. *Claim 2: Product design is a power struggle between diffuse sources of domain insights*

Modern perspectives on software design put end users and customers at the center. For example, user-centered design focuses on users' goals and tasks, and then iteratively designs and evaluates experiences to support those tasks (e.g., [6]). Agile and XP begin from the premise that requirements cannot be determined at the beginning of a project, and so continuous

customer and stakeholder involvement is critical to designing an acceptable product [14]. *The Lean Startup* [32] espouses similar ideas, encouraging the release of "minimum viable products" that provide value, but also opportunities to learn more about customer needs.

While I found these perspectives helpful, they mostly described how and why to *gather* insight, saying little about the process of *synthesizing* it into action. Synthesis was made difficult because insights were usually diffuse, conflicting, and sparse, requiring me to assemble a complete product vision from only fragments. Current customers expressed feedback through bug reports, feature requests, and technical support. Prospective customers expressed feedback through sales objections and competitor comparisons. Domain experts (primarily Jake and myself) had opinions about product value. Our board provided tactical and strategic recommendations about requirements prioritization. The market constrained and shaped which requirements were important, viable, and differentiating. And of course, engineers had strong opinions about what they did and did not want to build. As our product manager, synthesizing these disparate perspectives forced me to decide which sources to value and when.

This synthesis was further complicated by the varying *power* that these different sources held. Because Jake and I had studied contextual help so extensively, we viewed ourselves as the experts and therefore held considerable power within the company. Customers, however, had vastly more power than even us, as they were the ones making the buying decisions. Moreover, while we viewed customer support as an opportunity for insight and competitive advantage, they viewed it as a cost-center [8]. This misalignment between our expertise and our power meant that many decisions were a battle between the end users who needed answers (who we represented) and the champion inside our customer's company. We almost always needed to let customers win.

To regain power, we used two strategies. When we gave a customer control over how a feature worked, we would only give them as much as they demanded, and fought tirelessly to get them to see their product from their customers' perspectives. We knew that every bit of design control we lost meant a poorer experience for end users, which ultimately meant our product would produce less value for our customers. Our second strategy was longer term customer education, in which we used our sales and marketing efforts to change how customers viewed customer service, educating them about best practices and building our "thought leadership" in industry. This was slow and costly work, but necessary to eventually sell what we viewed as the best possible user experience in our customers' products.

C. *Claim 3: Translating a value proposition into code requires both planning and persuasion*

Product management typically comprises three activities. First, it involves managing the flow of engineering work by triaging incoming requests and assigning work to engineers [14]. Second, it involves maintaining a product roadmap to organize which work will happen in the future and when [14]. Third, and most importantly, it involves establishing "product-market fit" by defining and refining a *value proposition* that expresses why a product would be valuable to customers [28] and then ensuring that the product roadmap always arranged to test and refine this value proposition [32].

In my experience, refining our value proposition was the most important but invisible work that I did in validating our business. It was behind every feature or fix I prioritized; it shaped our marketing materials sales rhetoric; it was deeply embedded in our product's code; and it was behind every work item I assigned to an engineer. This refinement, however, was also a surprisingly complex social process, especially as it related to engineers. This was for many reasons. The first was that keeping my engineers' *understanding* of our value proposition consistent and up-to-date—achieving what Brooks might call "conceptual integrity" [7]—required constant communication of design rationale at low levels of granularity. It was not enough for engineers to know that we reduced support tickets and increased sales conversions, for example; they needed to know how the "hidden" Q&A state they were adding affected our value proposition so they could fully express that value through their code-level design decisions and keep new decisions consistent with existing ones. I therefore spent much of my time explaining and justifying design decisions to engineers. This forced me to have a well-reasoned, coherent idea of the value of each particular feature or change, and ensure that these smaller rationales were well aligned with the overarching and evolving value proposition.

Occasionally, engineers added friction improving our product's value proposition by weighing some software qualities higher than customer value. When we deferred fixing low-risk security vulnerabilities in order to accelerate a feature's release, for example, many of the engineers found this borderline unethical, arguing that it would be better to delay release and risk losing customers than to release something others would view as insecure. When we released user interfaces with subpar usability, our front-end engineers and designers felt similarly, struggling to accept a tradeoff between the company's bottom line and their design principles. Therefore, product management was as much about leadership and persuasion as it was about optimizing product-market fit.

Another challenge was balancing the *kind* of work that I assigned engineers. They wanted work that was technically interesting and would develop their skills, but the business needed them to focus on increasing product value, and these tasks were often straightforward and boring. For example, we once wrote a "whitelabeling" feature that amounted two lines of code, but made us tens of thousands of dollars per year per customer (as it allowed customers to hide our logo and preserve their brand). This was neither interesting or challenging to implement, but it was highly valuable to the business. In contrast, I once assigned a refactoring of a key data structure that took two weeks. It provided no product value, other than removing a barrier to a valuable feature, but it raised many fascinating architectural issues. The lack of correlation between *interesting* and *important* work meant that our roadmap was rarely a pure effort to increase product value, but a delicate balance of product work and developer nurturing.

D. Claim 4: Debt is both technical and cognitive

Ward Cunningham's metaphor of *technical debt*—the degree to which an software implementation fails to reflect a team's best understanding of the problem they are solving [11]—has become a powerful tool for thinking about software evolution [3, 25]. It was a powerful tool for me as well—so powerful, in fact, that I began to see many other kinds of debt.

For example, I regularly encountered *comprehension debt*, which reflected the extent to which engineers' mental models of our implementation's behavior was out of sync with its actual behavior. Our comprehension debt was low enough that there was usually at least one person who understood a component, ensuring that they could efficiently repair and enhance it. There were many cases however where there was *only* one person. When they were on vacation and we encountered a defect in the component, the team scrambled to decide who had enough familiarity to diagnose and fix it. Our strategy for preventing comprehension debt was to cross-train engineers on components, ensuring that there were always multiple people who could work on every part of our implementation. Some of this cross-training occurred during onboarding, where I would teach new hires the architecture of our product. Code reviews also cross-trained, ensuring every change made it into at least two heads.

There were cases where no one understood a component. Some of the code I wrote in the early days of the company had not been read by anyone, even me, for years. One day, an important customer reached out about an Internet Explorer issue with our user interface selection functionality. When I went back to debug it, I had to re-comprehend thousands of lines of JavaScript I had not seen in two years, and so it took me a day instead of an hour to diagnose and repair the issue. In other cases, components were so highly coupled, no engineer felt they owned it, creating a tragedy of the commons [17]. For example, at one point there was a server-side PHP script that had been appropriated for so many diverse uses, no one felt like the mess was theirs to clean up. Consequently, no one spent the time to understand the mess, resulting in comprehension debt that posed future maintenance risks.

I also observed *design rationale debt*, where no one remembered *why* a component behaved as it did. For example, early in the 1.0 of our database schema, we decided to allow questions to optionally be retrieved only through selection and not via text search. Two years later when we were refactoring the schema, the engineer performing the refactoring understood the column functionally, but he did not know why it was there, how important it was to customers, or which customers were dependent on it. Even though I was the one who added the feature, I could only vaguely remember why we thought it was so critical at the time. Because I could not recall this rationale, removing the feature risked breaking an undocumented customer requirement. Paying off this debt required reconstructing the rationale for a decision, documenting customer dependencies that were not visible in code or data, and reevaluating the rationale.

The only way to prevent design rationale debt was to document rationale upfront. This preemptive investment posed all kinds of questions: where to store it, how to make it easy to find, and how much time to invest in writing it. We settled on a practice of writing rationale comments tagged with unique identifiers so that crosscutting decisions could be easily searched.

I also observed *planning debt*, which occurred when developers privately maintained plans for improving software architecture. For instance, in my diary, I frequently reflected on ways of improving my code, and then realized that the *only* place these plans were documented was in my research diary. I asked my engineers where they kept their plans, and most said three things: 1) their memories, 2) handwritten notes on their desk, or 3) as partially implemented (and therefore opaque) changes in code. The cost of this debt was that when a component changed owners, there was no easy way for the new owner to learn about those plans and complete them.

These three types of debt—comprehension, rationale, and planning—concerned the cognitive gaps between developers' mental models of what code does, why it does it, and how it should do it differently. This suggests that there are actually many pairwise gaps between product, code, and developers: product and code (technical debt); code and mind (comprehension debt); mind and product (rationale debt); and architecture and code (planning debt). I saw all of these types of debt lead to costly miscommunication and defects.

E. Claim 5: Effective developers are patient teachers, rapid learners, and prosocial communicators

Although prior work makes it clear that engineers need to be effective communicators [5, 24, 31], my time managing a team of up to seven engineers made it clear precisely what "effective" means in practice. Nowhere was this more salient than in the onboarding of new engineers, which required existing engineers to teach, new engineers to learn, and all engineers to productivity communicate.

First, existing engineers had to be effective *teachers* in order to rapidly transmit all of the knowledge new engineers needed to be productive. They had to teach the architecture of our implementation, the reasons for our technology stack, the languages that the new engineer might not be familiar with, the toolchain we used to build, test, and debug, the deployment practices we followed, the issue tracker workflows we used, the processes we used to interface with product, design, sales, marketing, and support expertise in our company. And after they had on-boarded, new engineers needed to become teachers themselves as they took ownership of some component from someone else and had to teach others about it. On some days, I found myself teaching the whole day, more than I had ever taught on a normal day in faculty life. My engineers regularly spent an hour or two each day teaching each other, whether informally as part of adopting a new library, or formally in code reviews or on-boarding.

Of course, engineers had to be great independent *learners* as well, spending time in discussion forums, documentation, blogs, and Q&A sites. To learn from other engineers, they also had to be good at listening, asking good questions, knowing when to ask for help, and being self-aware enough to know when they were stuck because of a problem or stuck because of a knowledge gap. This was made easier by our collocation; when I heard a developer let out a big sigh, I could just turn

around from my desk and prompt an engineer about how their work was going. This was a good opportunity for engineers to raise questions they were stuck on or surface some confusion about our architecture. At some point, I became explicit about maintaining a culture of learning, because it became clear that one of our engineers was subscribed to the "geek gene" hypothesis [1], pigeonholing some engineers as just "not getting it." Our culture was a buffer against his bias.

Because learning and teaching were so critical to engineering progress, the ability to communicate in encouraging, constructive ways was also paramount. This prosocial behavior [15] was difficult for some of our engineers. For some, it was a language barrier, where English was their second or third language: there were idioms, metaphors, and even pop culture references that they did not understand, which isolated them from the rest of the team in both work and non-work related conversations. For others, it was the lack of listening skills: some were so resistant to feedback and instruction, other engineers became hesitant to even try to teach them, which isolated them from important knowledge they needed to progress on their work. For others still, personality was the problem: if they were quiet or reserved, they were less likely to ask for help from anyone, which limited their productivity. In other cases, gender and seniority warped communication channels, with some female engineers going to trusted coworkers, rather than expert coworkers, and some engineers more willing to listen to male engineers than female engineers.

As an engineering manager, I spent much of my time establishing, streamlining, or repairing these lines of communication, working around personality or interpersonal biases so that each engineer had all of the information and knowledge they needed from other engineers to make progress. Doing this well meant knowing my engineers well: their personalities, their biases, who they did not like, their career goals, and the dynamics of their life at home. I needed to maintain a mental model of the social network of my team and the lines of communication that did and did not exist, so I could ensure that all of the information necessary for moving some issue forward would make it to the proper subset of the team, intact, and consistent across each individual's mind.

F. Claim 6: Quality-driven management requires trust

As an engineering manager, I adopted a team-level view of productivity from Agile, tracking *velocity* (the number of work units completed per unit time [4]). This helped me estimate the capacity of my team to do work in the future and identify bottlenecks. I discovered, however, that my velocity-centric view of productivity was very different from those of my company's other executives. This was particularly true for Jake, my co-founder and CEO, whose preference was to manage the *outputs* of the team, giving engineers deadlines and expecting them to meet them, even if it meant working 12+ hour days. This was the culture he came from in 1990's Silicon Valley and it was consistent with many of the corporate cultures in Seattle. From his view, if it was good enough for Amazon, Facebook, and Google, it was good enough for us.

I understood the pressure he was under: deadlines helped him forecast and allowed him to communicate confident plans to our board. So, we tried his way first. Unfortunately, when we were deadline focused, we missed deadlines and shipped defective code. This frustrated my engineers, who felt like they had to achieve both speed and accuracy. It also led to tense exchanges between me and Jake, as in this email snippet from Jake to me:

> *"I need you to appreciate that, yes, due to many many factors, some of which were outside your control (heartbleed, server port, additional hire), we have now fully slipped. Repeat after me: 'I, Andy Ko, have slipped my V1 ship date. I, Andy Ko, am not being hard enough on my team. And I, Jake am not being hard enough on you, Andy, either.'"*

I discussed this tension with my engineers. They valued the pressure to ship as a motivator, but also felt demoralized by having to compromise on quality. I returned to Jake and conveyed to him that my engineers were fully committed, on task, and doing good work, but that there was just too much work to make the deadlines we had self-imposed.

I countered with a management model focused on *inputs* and *quality* rather than outputs and deadlines. As usual, we would select high-value issues for our roadmap, but rather then setting hard deadlines, I would ask for a fixed amount of engineers' time each week and regularly provide estimated ship dates. This allowed engineers to balance work and life, while giving everyone else in the company a tentative date to work against. It also allowed the engineers to focus on writing correct, maintainable code. Most of the engineers were satisfied with this model (except for the few that wanted deadlines as extrinsic motivation; for them, I imposed internal deadlines to keep them moving).

While this model significantly improved quality, it caused other problems. Jake and the rest of the company felt even more powerless to plan. My shifting release dates led to distrust in my estimates. Many in the company perceived the engineers' fixed-length work week as a lack of commitment to the business, since they were used to engineers staying late. Worse yet, Jake could not understand how we were still shipping bugs:

> *"If we're focused on quality, then why are we seeing bugs like this? How can a 10-day search bug go uncaught, when the very last project, if I understand, was to write tests to ensure the proper functioning of search itself? On the face of it, you have to see that from my view, this seems ridiculous. It makes me question everything about this 'release when we're ready' approach."*

This misconception of test coverage led other executives in the company to make poor diagnoses: *Are the engineers not putting in the time? Are they lazy? Are they just bad engineers?* My engineers felt bad enough about shipping defects. This mistrust compounded their guilt, only making them more careful, which pushed release dates further out.

I tried to improve it by making engineering work more visible: I invited everyone to our product roadmap so they could get update notifications. Our new VP of Engineering adopted Slack, making commits, builds, and collaboration more visible. These strategies helped in small ways, but I still had to regularly reassure my executives that engineers were on task, moving fast, and quality-focused.

G. Claim 7: Schemas structure work

The role of data and its relationship to decomposition has a long history in software engineering research. Most notably is Parnas' principle of information hiding [29], isolating decisions that are likely to change from each other to minimize the cost and consequences of changes. Such ideas have since matured greatly into the area of software architecture [36], providing a robust vocabulary of patterns, styles, components, and connectors, helping to describe the flow of information through code.

These architectural ideas were highly relevant in the decisions I made during my three years, particularly at the database schema level. I observed that while the representation of our data was highly decoupled from the other layers in our applications, the *semantics* of the data was not and therefore had to be maintained separately at every layer. This meant that adding or removing database columns was trivial (only requiring the addition or removal of related functionality), but that *changing* a data type was costly and error prone. For example, to support a popular customer request, we once changed a Boolean column to a three-state enumerated type. This required us to find every downstream reference to that type, including not only manipulations and computations of these Booleans, but also complete redesigns of user interfaces to reflect the new semantics.

While the technical work from these semantic changes required careful work, the internal and external business implications were even more complex. Data migrations that changed the semantics of our customers' data were halting, monumental changes that required careful planning by our customer success team, our account managers, and our support team to communicate the changes, anticipate concerns, update help content, and retrain every non-engineer in the company about the new semantics of the data. We rarely executed these semantic schema changes without crisis, because we had to uncover the dependencies in our business processes, our customers' processes, and our own employees' processes manually. As we matured our planning and release processes, it became smarter to assume that every employee and every customer needed some kind of retraining before a change began.

Changes in data semantics also had implications for our employee's work. For example, one of the features that I had designed early on was a component that would use basic part-of-speech tagging to automatically tag incoming questions with key noun and verb phrases. These tags, which we represented as a table in our database, were then used to optimize retrieval, while giving customers direct control over where questions would and would not be retrieved. I had also built into the schema the ability to group synonymous tags, in case we decided that higher recall was important in our retrieval algorithms. What I originally viewed as a clever product design decision translated into hundreds of hours of engineering and non-engineering work time. Back-end engineers had to constantly work around the tag grouping schema that was hardly used; front end engineers had to design user interfaces that hid the grouping indirection in the schema. And removing the grouping table, because of the sheer volume of code that depended on it, would have required massive re-architecting, eliminating all of the logic to operate on tag groups that was necessary to operate on tags. Our sales and success teams, fearing customers seeing the false positives in our part of speech tagger, spent hours cleaning up every tag to minimize the chances of bad experiences before account launch. This one schema choice and its downstream costs the company hundreds of person hours over my three years in charge, but ultimately provided limited product value.

This surfaced a frustrating paradox: early data schema decisions were highly consequential to the implementation of every layer in our architecture and were the most difficult decisions to change because of the massive set of downstream code, company, and customer dependencies. And yet schema decisions often had to be made well before we had any significant requirements certainty.

H. Claim 8: Coding is easy (when dependencies are known)

Most of the software development work we did was straightforward. There were engineering activities, however, that became complex or impossible. In nearly every case, this was due to unknown or unknowable dependencies on code that required time-consuming and error-prone manual efforts to fully uncover and understand. In this section, I discuss common developer activities and which types of hidden dependencies turned them from easy to intractable.

Coding was the least difficult developer activity. We hired engineers that were capable, quick programmers. Even our most junior engineers learned new languages and tools quickly. Because our engineers were aware of the error-prone aspects of languages—especially with the constant feedback of the linters we incorporated into our builds—they learned to avoid risky aspects of languages in favor of their safer, less error-prone semantics. When these semantics were *not* known, however, defects became almost intractable to localize, because developers did not know to look for these unexpected, error-prone semantics. These semantics were, in effect, hidden dependencies on execution.

Our own specifications were incredibly valuable, but hard to find, making the design dependencies they contained difficult to account for. This caused developers to overlook requirements and constraints, leading them to break functional requirements or other dependencies that were poorly documented. We mitigated some discoverability problems by writing most of our specifications as source-level documentation (formal when possible). For example, we chose a declarative REST API framework that was self-documenting, we described data schema semantics and rationale in table and column comments, and we wrote pre- and post-conditions for functions. When developers saw these specifications, or better yet, when tools enforced them, they were invaluable. When they were hidden from developers' work, they were useless.

We never had enough test coverage, but this did not cause many failures. When it did, these were failures we not likely to have anticipated, because they arose not from unexpected input, but from assumptions or defects in code that were not visible. We mitigated these severe failures by investing heavily in server-side and client-side monitoring, which

meant we usually deployed a fix within an hour of an alert. That is not to say that testing did not matter—we wrote many tests for critical features, as well as did significant manual testing—but the hard parts of testing concerned the inputs and states we had *not* anticipated, not the ones we had.

Debugging was routine and time-consuming, but only occasionally difficult. When it was difficult, it was almost always the lack of reproduction steps that posed the challenge, not fault localization or repair. This usually manifested as a vague report from an important customer, which forced us to imagine the space of possible inputs that might cause the failure. In some cases, we did not even have a clear description of the failure. As in other activities, it was our team's inability to see the inputs and state on a customer's machine that made the work difficult.

The team frequently refactored to pay down technical debt, keeping our implementation aligned with our product's value proposition. Because we used primarily dynamic scripting languages, it was error-prone, conflict-prone, and time-consuming, especially in components with numerous dependencies, none of which were visible statically. Because refactoring dynamic code was so difficult, it was something we avoided doing until necessary, which often artificially delayed important changes.

Documentation was another source of hidden dependencies. Prior work has found that documentation is often incomplete, ambiguous, and unexplained [38]; I found that it was specifically missing two types of details. First, APIs, frameworks, and libraries would often have overarching *design patterns* in their use, but documentation rarely explained them. We adopted Facebook's React, for example, and spent hours with its documentation, but most of our engineers only really understood its core design patterns after attending a local React class sponsored by Facebook in their Seattle office. The second type of missing detail were *runtime semantics*, such as usage rules and runtime properties such as performance, testability, and debugger support. Design patterns and runtime semantics were occasionally available in a blog post or StackOverflow answer, but rarely in official documentation.

Infrastructure posed one of the greatest hidden dependencies: the future demand on a web service. This required significant "DevOps"—the set of activities involved in building and maintaining 24/7 web services—to help prepare for unanticipated service volume. We used third-party, cloud-based solutions for everything that we could, both to save money, to focus our time on our own product, and to simplify scaling, but this required substantial system administration expertise to build, maintain, and scale.

Security also posed many hidden dependencies. Secure coding itself posed unanticipated inputs, requiring engineers to use tools to anticipate these inputs. But security was even more challenging when it involved responding to *other's* efforts to explore our vulnerabilities. Enterprise customers, for example, often ran much more extensive automated security audits, which would send us long lists of false positives for us to review, and this would occur every time we encountered a new customer with the same tool. Freelance security consultants would also regularly do penetration testing on our infrastructure in order to extract some form of compensation. This created unexpected and usually unhelpful work.

Among all of these activities above, the most difficult, error prone tasks had one common feature: they had dependencies that could not be easily discovered because there were not observable, recorded, or planned. Whether it was missing reproduction steps, undocumented API usage rules, unexpected traffic, or dynamic dependencies that were statically hidden, when a developer could not see it, they could not easily plan for it, act on it, or resolve it.

I. Claim 9: Research impact requires perfect timing and minimal risk

One of my motivations for commercializing our research was to understand why technology transfer is so rare. Some researchers have begun to investigate the barriers. For example, engineers view most software engineering research innovations as worthwhile [27], but there is still a wide gap between research prototypes and whole products [10]. Moreover, many engineers have strong beliefs based from personal experience that cause them to discount research findings [13]. As CTO, I was responsible for adopting new technologies and leveraging empirical evidence. What other barriers to technology transfer did I observe?

After three years, the barriers were so numerous, I began to wonder how innovations *ever* become part of practice. First, public research innovations had a very limited value to our business. Because we had published our work and described how to replicate it, we had actually rendered our intellectual property useless, and so our board encouraged us to generate new private IP as soon as possible. And yet, investing in innovations as a startup was risky and hard to justify. I never found enough time to do the work we *knew* we had to do, let alone work that might hypothetically valuable. The company essentially had room for only one risk, which was the original risk it took when we founded it: testing its value proposition.

Our company's own innovative features were also harder to sell because they required customers to learn new ideas, develop new processes, and take the potential value of the innovation on faith. For example, our "object search" technology, which was the key innovation in the original research [8], was rarely the reason that customers bought our product. In fact, it was often a liability. The sales team had to understand it (which they often did not), they had to teach it to prospective customers (in about 1 minute as part of a larger 30- to 60-minute product pitch), and customers had to believe that it would provide value that outweighed the risks and costs. Their perceived risks and costs were always exaggerated—we had plenty of evidence that companies could easily adopt the service, including research evidence [9]—but this data did not matter to customers. They asked, "*It's great that it worked for that other company's customers, but what about mine?*"

When I evaluated new developer tools for adoption internally, I behaved just as our customers did. In fact, there was a day when I needed a way to extract trending descriptions of problems out of technical support requests in order help our customers find "gaps" in their Q&A content. As it turned out, I had earlier invented an algorithm that solved

this exact problem [21]. I re-read my publication and began analyzing the gaps between the research paper and our product need. *Would it support multiple languages? How does it scale? What kinds of false positives occur? Was the content in the support tickets we were mining similar enough to the discussion forums the technique was tested on? How would it handle noise?* After an hour, I gave up: without a plug-and-play solution, there was too much work and risk in converting my own invention into product, no matter how perfectly the research fit the situation. I had a long list of more valuable ways to invest my time.

These constraints also applied to non-technical research discoveries. I frequently cited evidence from empirical studies of software engineering, summarizing findings and recommending policies based on the evidence. But each time I shared these discoveries, I received the same (reasonable) response: *Would that happen in our case? Why should I believe that when it was only based on 20 engineers?* In the end, our executives' prior beliefs usually heavily outweighed the small bits of evidence from studies.

Despite these barriers to innovation, many of the beliefs that our engineers and executives held ultimately did come from research. They were beliefs embedded in tools and processes engineers had learned in school. They were beliefs from business and management books that were informed by long bibliographies of academic research. The difference was the *context* in which our employees had learned them: as a CTO, I was offering recommendations as an executive and a manager, which is a context that affords debate, disagreement, and dissent. Had I been communicating in a classroom as a teacher, at a conference as a speaker, or through a textbook as an author, I suspect there would have been a higher level of deference to authority. (This is not to say that innovations and evidence in software engineering research are always valid. In many cases, our evidence base and technologies are immature and not yet ready for adoption).

V. Threats to Validity and Discussion

In this paper, I have presented nine claims that attempt to capture my experience as CTO of a software startup:

- Software engineering is structured by individual and social decision-making under uncertainty
- Design is a power struggle between diffuse sources of domain insights
- Translating a value proposition into code requires both planning and persuasion
- Debt is both technical and cognitive
- Effective developers are patient teachers, rapid learners, and prosocial communicators
- Quality-driven management requires trust
- Schemas structure work
- Coding is easy (when dependencies are known)
- Research impact requires timing and minimal risk

These claims portray software engineering as a technical activity besieged by uncertainty, speculation, power imbalance, code and cognition misalignments, and information exchange and decision making warped by social factors. This view suggests that software engineering is far more complex than we often admit when we practice it, teach it, or attempt to disrupt it through new software engineering tools. It demands that we increase efforts in both academia and industry to move beyond purely technical conceptions of how we create software, more deeply studying the intricate social, cognitive, and emotional dynamics of this inherently sociotechnical work.

As with any empirical study (including the self-reflections by Knuth [18] and Brooks [7]), I cannot say with certainty that the nine claims are internally valid. I do not know that they faithfully represent the cause and effect in my company or even in my own behavior. As I stated earlier, everything I have reported is filtered through my subjective biases as a practitioner and a researcher. My reactions to software engineering work were a product of what I observed and what I found salient about these observations. My data was biased by what I felt compelled to write about in my diary on a daily basis. This was partly a product of my interests in social interactions in terms and also the emotional experiences I had during a very intense three years of work. My biases and experience also meant that I did not deeply reflect on my experience writing hundreds of thousands of lines of code, because I found most of that work routine and uninteresting.

As with any diary study, this one is also limited by the *observer-expectancy* effect: my process of reflection likely changed how I worked and how I structured our business. This is both a strength and a weakness: my involvement may limit the generalizability of the nine claims, since my decisions were guided by my reflective practice, but it was also a unique opportunity for me to test the validity of the theories I was developing through process, policy, and interactions, and then observe the efficacy of these embodiments in practice. This may limit the generalizability of the data to organizations that have some number of strongly reflective people in positions of leadership, and may not generalize to organizations who lack reflective practitioners (or have them, but in positions with little influence).

Although my CEO and VP of Engineering read this paper and agreed with my interpretations with respect to their own experiences, had they kept their own diaries, they might have arrived at different claims because of their unique perspectives. Therefore, the claims I present are not the *only* claims to be made, but a small sample of other potential truths that may have only been visible to other employees in my company. Therefore, the claims should be viewed with a healthy skepticism, as ideas to investigate in future work, and not verified truths (much like the many testable but unverified claims in Brooks [7] and Knuth's [18] works).

In addition to not knowing the *internal* validity of the claims, I also do not know their *external* validity, as our Seattle-based, venture-backed, software-as-a-service web startup is unlike many other types of software organizations. For instance, because our company was a startup, we were more focused on validating our product's value proposition than maintaining an already successful product. More mature software projects may spend far less time worrying about product value propositions, as they have already proven value. Unlike many software products, ours involved user interfaces,

information retrieval and extraction, and analytics; companies that build other types of software likely experience different challenges. Our company was also in Seattle, Washington, USA, which likely has a different work culture from that of other regions in the world. It is therefore possible that many of the nine claims are *only* true to my company. Further research will be necessary to understand how well they generalize to other settings.

Given these threats to validity, developers reading this should decide whether it is true to their experience, and continue to discuss these claims further in public forums. I hope researchers will test these claims, attempting to refute them through harder evidence, and seeking to understand when they are true—and more importantly, why. Educators should take these claims, and perhaps even this paper, and share it with their students, giving them a glimpse of the types of experiences that their students might have in the software industry. These may shape how they see their work, and more importantly, how they structure others work, as they get promoted and begin managing their own teams.

Only with continued reflective practice in academia and industry will we hope to eventually transform the art of software engineering into a science. I hope this paper encourages more of this reflective practice, and more sharing of it in public ways.

ACKNOWLEDGMENT

Thank you to my colleagues Jacob O. Wobbrock, Thomas LaToza, Parmit Chilana, and Kevin Knoepp for their extensive feedback on early drafts of this paper. This work was supported in part by the National Science Foundation (NSF) under Grants 1314399, 1240786, and 1153625. Any opinions, findings, conclusions or recommendations are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] A. Ahadi & R. Lister (2013). Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant? *ACM ICER*, 123-128.

[2] P. Atkinson & M. Hammersley (1994). Ethnography and participant observation. *Handbook of Qualitative Research*, 248–161.

[3] R. Bavani (2012). Distributed agile, agile testing, and technical debt. *IEEE Software*, 29(6), 28-33.

[4] K. Beck (2001). Manifesto for Agile Software Development. Agile *Alliance. http://www.agilemanifesto.org/*, retrieved Dec. 3rd, 2015.

[5] A. Begel, & B. Simon (2008). Novice software developers, all over again. *ACM ICER*, 3-14.

[6] H. Beyer & K. Holtzblatt (1997). Contextual design: defining customer-centered systems. Elsevier.

[7] F. Brooks, Jr. (1995). *The mythical man month*. Addison-Wesley.

[8] P.K. Chilana, A.J. Ko, & J.O. Wobbrock (2012). LemonAid: selection-based crowdsourced contextual help for web applications. *ACM CHI*, 1549-1558.

[9] P.K. Chilana, A.J. Ko, J.O. Wobbrock, & T. Grossman (2013). A multi-site field study of crowdsourced contextual help: usage and perspectives of end users and software teams. *ACM CHI*, 217-226.

[10] P.K. Chilana, A.J. Ko, & J.O. Wobbrock (2015). From user-centered to adoption-centered design: A case study of an HCI research innovation becoming a product. *ACM CHI*, 1749-1758.

[11] W. Cunningham (2009). Ward explains the debt metaphor. *c2.com/cgi/wiki?WardExplainsDebtMetaphor*, retrieved Aug 5, 2016.

[12] M. Czerwinski, E. Horvitz, & S. Wilhite (2004). A diary study of task switching and interruptions. *ACM CHI*, 175-182.

[13] P. Devanbu, T. Zimmermann, & C. Bird (2016). Belief & evidence in empirical software engineering. *ICSE*, 108-119.

[14] T. Dingsøyr, T. Dybå, & N.B. Moe (2010). *Agile software development: current research and future directions*. Springer Science & Business Media.

[15] N. Eisenberg, R.A. Fabes, & T.L. Spinrad (2007). Prosocial development. *Handbook of Child Psychology*.

[16] R.M. Emerson, R.I. Fretz, & L.L. Shaw, (2001). Participant observation and fieldnotes. In P. Atkinson, A. Coffey, S. Delamont, J. Lofland, & L. Lofland (Eds.), *Handbook of Ethnography*, 356-357.

[17] G. Hardin (1968). The Tragedy of the Commons. *Science*, 162 (3859): 1243–1248.

[18] D.E. Knuth (1989). The errors of TEX. *Software: Practice and Experience*, 19(7), 607-685.

[19] A.J. Ko, R. DeLine & G. Venolia, G. (2007). Information needs in collocated software development teams. *ICSE*, 344-353.

[20] A.J. Ko & P.K. Chilana (2011). Design, discussion, and dissent in open bug reports. *iConference*, 106-113.

[21] A.J. Ko (2012). Mining whining in support forums with Frictionary. *ACM CHI, Extended Abstracts*, 191-200.

[22] T.D. LaToza & B.A. Myers (2010). Developers ask reachability questions. *ICSE*, 185-194.

[23] T.D. LaToza, G. Venolia, & R. DeLine (2006). Maintaining mental models: a study of developer work habits. *ICSE*, 492-501.

[24] P.L. Li, A.J. Ko, & J. Zhu (2015). What makes a great software engineer? *ICSE*, 700-710.

[25] E. Lim, N. Taksande, & C. Seaman (2012). A balancing act: what software practitioners have to say about technical debt. *IEEE Software*, 29(6), 22-27.

[26] Lowenstein, G., & Lerner, J.S. (2003). The role of affect in decision making. In R. Davidson, K. Scherer, & H. Goldsmith (Eds.), *Handbook of Affective Science*, 619-642.

[27] D. Lo, N. Nagappan, & T. Zimmermann (2015). How practitioners perceive the relevance of software engineering research. *ACM FSE*, 415-425.

[28] A. Osterwalder, Y. Pigneur, G. Bernarda, & A. Smith (2015). *Value proposition design: how to create products and services customers want*. John Wiley & Sons.

[29] D.L. Parnas (1972). On the criteria to be used in decomposing systems into modules. *CACM*, 15(12), 1053–58.

[30] E. Pronin (2007). Perception and misperception of bias in human judgment. *Trends in Cognitive Sciences*, 11(1), 37–43.

[31] A. Radermacher, & G. Walia (2013). Gaps between industry expectations and the abilities of graduates. *ACM SIGCSE*, 525-530.

[32] E. Ries (2011). *The Lean Startup*. Crown Books.

[33] M.S. Schwartz & G.C. Schwartz (1955). Problems in participant observation. *American Journal of Sociology*, 60(4).

[34] J. Sillito, G.C. Murphy & K. De Volder (2006). Questions programmers ask during software evolution tasks. *ACM FSE*, 23-34.

[35] H.A. Simon (1956). Rational choice and the structure of the environment. *Psychological Review* 63(2), 129–138.

[36] R.N. Taylor, N. Medvidovic, N., & E.M. Dashofy (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing.

[37] I. Treitlier (2014). Backyard ethnography: Defamiliarize the familiar to transform business. *International Journal of Business Anthropology*, *5*(1), 93-105.

[38] G. Uddin & M.P. Robillard (2015). How API documentation fails. *IEEE Software*, 32(4):68-75.