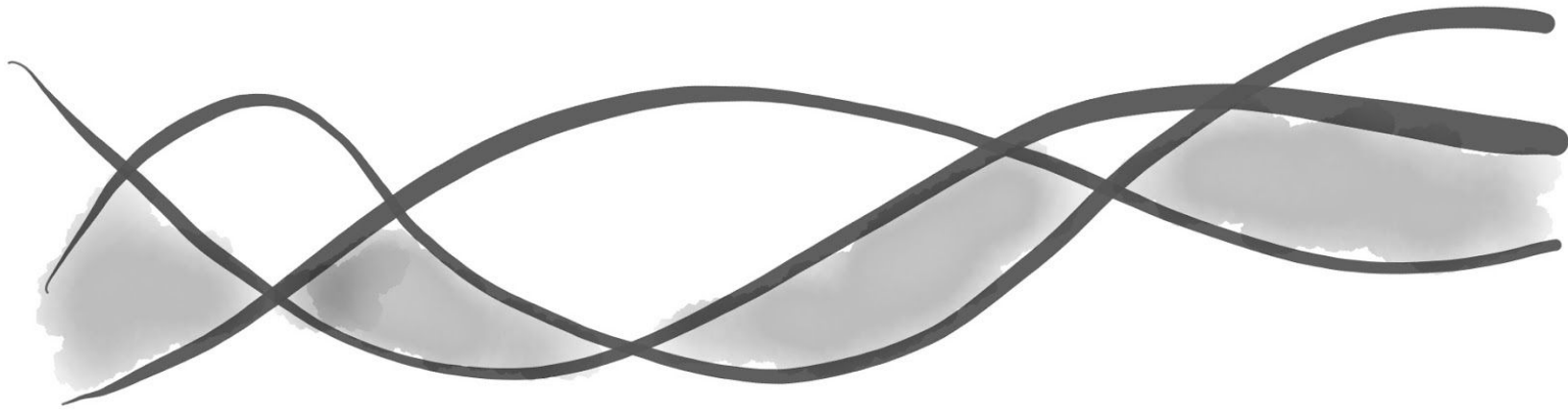
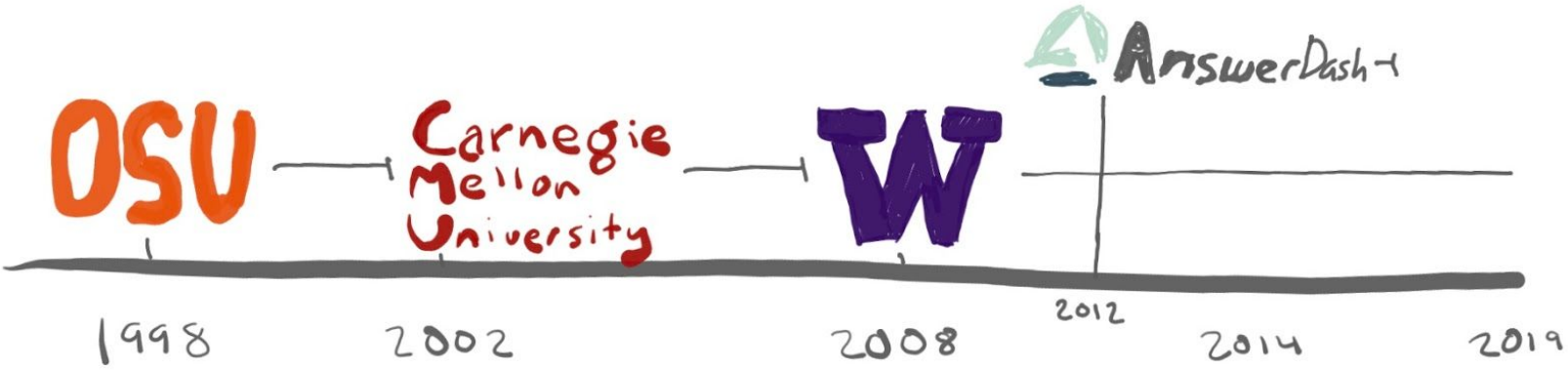


Materials for Promotion to Full Professor

Amy J. Ko, Ph.D.

Associate Professor, The Information School
Adjunct Associate Professor, Computer Science & Engineering
Program Chair, Informatics
University of Washington, Seattle, USA



Overview

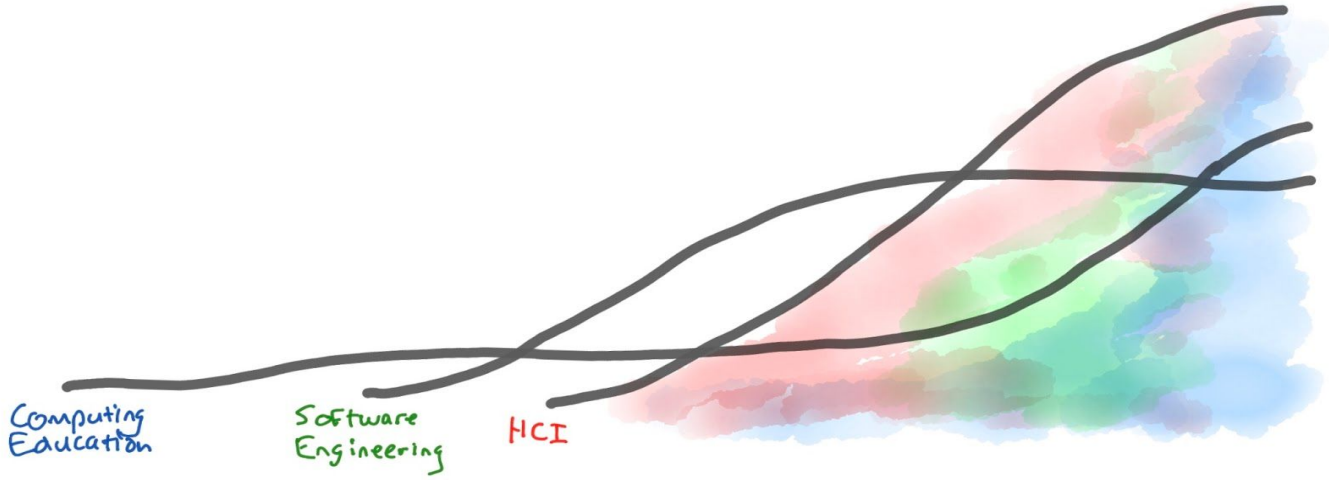
I wrote my first computer program in 1991 at the age of 11 in 5th grade. My math teacher told us to transcribe a series of "put" statements from a handout into an Apple II. I ran the program, I saw the computer render a cute little duck, and I was hooked. If this machine and the commands I gave it could create a picture, *it could do anything*. I spent the rest of my youth transfixed by computational making: I designed games on my TI-82 graphing calculator, I wrote text adventures in Pascal, and I made 3D rendering libraries in C. Computers weren't hardware to me, they were a medium for expression.

In college, however, I learned that to most of my peers, computers were something very different. They were a foreign, impenetrable way of thinking. As I flourished in learning, my peers struggled and quit. And those that persisted saw themselves as imposters: I remember a friend confiding at graduation, "*I still have no idea where to start with a blank source file.*"

When I graduated in 2002, these stories were happening at the scale of tens of thousands of CS majors. Almost two decades later, these same stories are happening at the scale of a 100 million. I've spent these past two decades trying to prevent these stories of failure. I've done this through **research**, deepening our understanding of programming as a skill and inventing tools that improve how people program. I've done this through **teaching**, mentoring thousands of students in how to design and engineer software and mentoring other teachers to do the same. And I've done this through **service**, creating, evolving, and expanding academic programs that support learning about computing, while helping to mature academic discourse on programming.

Pursuing this work has required me to be highly interdisciplinary. I draw upon computer science, psychology, learning sciences, education research, and design. I participate in three largely non-overlapping academic communities: **human-computer interaction** (where I publish novel ways of supporting programming), **software engineering** (where I've deepened our understanding of programming in organizations), and most recently, **computing education** (where I've contributed new conceptions programming and broadened our understanding of online and informal learning). I've taken these discoveries outside academia: I co-founded a startup that is reaching millions of consumers, my inventions have shaped a dozen widely used software development tools and learning technologies, and I've recently worked with policymakers to shape Washington state and federal law on computing education. I've also organized communities of hundreds of CS education advocates in Washington state and have used social media to share my ideas and perspectives more broadly with academia and industry.

At the University of Washington, appointment to the rank of professor requires *"outstanding, mature scholarship as evidenced by accomplishments in teaching, and in research as evaluated in terms of national or international recognition."* I believe that my research, teaching, and service have reached this maturity and recognition and so I am seeking promotion to the rank of Professor. In the rest of this document, I'll detail these accomplishments and share my vision for the next phase of my academic career.



Research

Since software was first invented, the world has been both captivated by its power to reshape society, but confused by complexity. My research lies at the nexus of these two forces, aiming to clarify to humanity how software works, and in doing so, empower them to use it. Some of my research has focused on **software developers** and the tools they use to create software. Some has focused on **end-user programmers** such as scientists, designers, and other professionals who are using code to solve just part of a larger problem or to express themselves. And most recently, my work has focused on **learners** who want to learn to code, either in school, at work, or in life.

My discoveries come in three forms: 1) **theories** about what programming is and why it is hard to do; 2) **empirical studies** that test and refine these theories; and 3) **new tools and teaching methods** that improve how people learn about and create software. In training, my epistemology is post-positivist, in that I embrace objectivity and the scientific method, while recognizing that my background, knowledge, identity, and values influence what can be observed. In practice, however, I am an epistemological pluralist, embracing the validity of multiple ways of knowing, and always learning new methods with which to express that pluralism. This isn't always compatible with the epistemologies of my academic communities.

Because my interest is in programming, and programming is relevant to many aspects of computing, I publish in many areas of computing. I first began publishing novel systems for interacting with code In **human-computer interaction**. Next, I began publishing studies of how software developers

reason about code individually and in teams in **software engineering**. Most recently, I have published In **computing education**, deconstructing programming as a *skill*, inventing new methods of teaching these skills, and uncovering structural barriers to accessing this teaching. In all of these fields, I publish in top journals and conferences, as well as more focused but less prestigious venues.

Over my past 11 years as faculty, I have done most of my research in collaboration with **18 doctoral students** and **36** undergraduates (some of whom have gone on to pursue doctoral studies at UW, Northwestern, Syracuse, and CMU). To support my time and the time of my students, I have collaboratively raised over **$11 million** in competitive NSF grants and **$100K** in gifts from Google, Microsoft, and Adobe.

To date, my work has been recognized internationally across all of these fields by **3 most influential paper awards**, **6 best paper awards, 4 best paper nominations**, **5,600+ citations** (h-index 36), **56K+** downloads from ACM digital library, **3 invited keynotes** at premier conferences in my fields, and invited research talks at top institutions such as MIT, Stanford, Michigan, Northwestern, ETH Zurich, Microsoft, Adobe, Intel, and IBM. Some of my academic impact has been outside of my core fields, in programming languages, databases, operating systems, education research, learning sciences, and game science. My research has also been recognized in industry and nonprofits, including 3 patents, both direct and indirect impact on commercial software development tools and learning technologies, and direct impact on the curricula used by millions of software developers and learners.

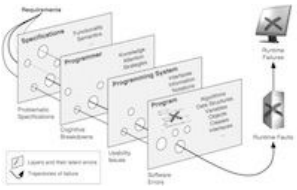
To summarize my research contributions, I'll describe three areas of focus:

1. Deconstructing programming
2. Making programming easier to do
3. Making programming easier to learn

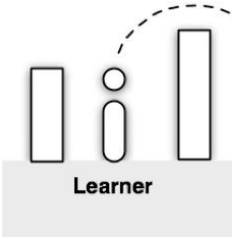
I will also discuss what I describe as "service" research, which includes literature reviews, books, and other writing that serves to advance my academic communities, but does not directly advance my research focus. Throughout, I will interleave discussions of the impact of my work. For citations, I will use the labels found in my CV (e.g. "C.3" refers to my third peer-reviewed Conference paper, "J.6" to my sixth journal article).

Focus 1: Deconstructing programming

At the heart of my curiosity about programming is a definitional question: *what is it* [W.6]? Since first starting research in 1999, I have investigated this question from many perspectives, contributing a human and sociotechnical conceptualizations of programming. My work is distinct from the more strictly technical view of programming across computer science.

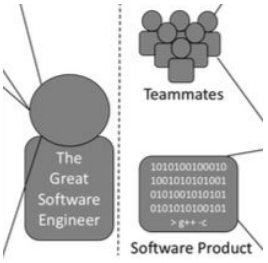


Programming as expression of intent. My earliest theories investigated the gap between a programmers' goals and the errors they make [C.3, C.4, J.2]. This work applied theories of human error to programming, arguing that errors vary in their origins: some emerge from slips in routine skills, some emerge from good rules applied in the wrong context, and some emerge from higher level errors in reasoning. This award-winning work has shaped several efforts to design developer tools that prevent programming errors.



Programming as human-computer interaction. Another theme in my work has considered the programming languages, platforms, and tools as *interfaces to be learned* [W.6]. For example, I've deconstructed the barriers to learning APIs and programming languages [C.5] and with my student Kyle Thayer, defined what API knowledge constitutes [S.6, J.10]. This award-winning work has been the foundation for many innovations in developer tools.

Programming as problem solving. My work with my student Dastyni Loksa has also conceptualized programming as a *self-regulated orchestration of tasks* such as problem interpretation, searching for analogous problems, searching for solutions, evaluating solutions, implementing solutions, and evaluating solutions [C.39, C.40]. These ideas have recently impacted how teachers are viewing what they are teaching, leading to new methods for teaching process. This includes shaping the instructional design of Code.org's AP CS Principles curriculum, which is currently under revision.

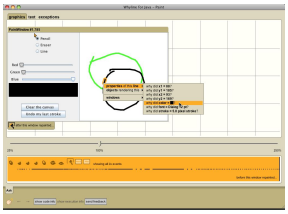


Programming as skills. With my student Paul Li, I have framed programming as a set of personality attributes and sociotechnical skills [C.37, W.8, J.11], uncovering the information needs that generate the need for these social skills in organizations [C.12, C.13, C.46]. These works have impacted NYC public school curriculum, they have provided the foundation for new theories of software engineering expertise, and they have reached more than 50,000 professional software engineers through an ACM learning webinar.

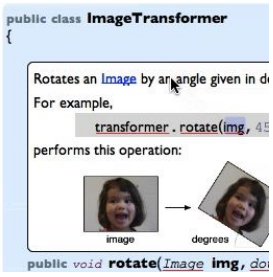
Programming as design. With my student Parmit Chilana, I have also studied programming as a process of gathering data about the world's needs, interpreting those needs, and translating those needs into computation. I studied this through the lens of *bug reporting*, in which developers navigate tensions between what they have created and what the world wants [C.9, C.21, C.25, C.26, C.38, W.3, W.4, W.5, S.5, S.7, N.9, N.13]. This work contributed a view of software engineering as inherently sociotechnical. Through a series of invited talks, this work has shaped the bug triage tools and practices at Microsoft, Amazon, Adobe, ABB, and IBM.

Focus 2: Making programming easier to do

I've spent much of my career inventing tools that streamline challenging programming tasks, particularly debugging. My key contribution has been inventing *interactive tools that combine human and machine intelligence* to outperform purely human or automated approaches.



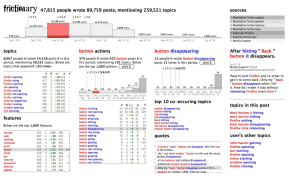
Debugging is the process of identifying the portion of a program causing some unwanted program behavior. Debugging is hard because it requires someone to search an incredibly large space of program execution events for a small number of unintended actions. My award-winning work on this problem observed that what makes debugging slow is that most developers begin with a *guess* about what is causing a failure, and most developers' guesses are wrong. I invented a set of algorithms and interaction paradigms (embodied in a tool I called the *Whyline*), which instead have developers begin with the *program output* they know to be wrong, then have the computer automatically identify the chain of causality that caused that faulty output. By presenting it for inspection, developers could more rapidly isolate the defect than a manual search [C.6, C.15, C.17, J.4]. The ideas in the Whyline and its follow-up work (including collaborations I have done on debugging machine learning [C.18, C.19, J.5], and adaptations for the web with my student Brian Burg [C.29, C.34]) have shaped debugging tools developed by Microsoft, Apple, Google, Mozilla, IBM, and Adobe, as well as many debugging tools in research.



Program understanding is the process of understanding how a program is built, so that it can be changed, enhanced, or repaired. My innovations in this space have focused on changing how programs are *navigated*. I have contributed award-winning discoveries about how tools shape how developers search and browse large programs [C.8, J.2, J.3, J.6], how tools can

streamline this navigation by presenting fragments of programs relevant to a specific concern [W.1], and how programs can be edited to combine the benefits of text-editors with structured editors [S.2, C.11]. These contributions have directly shaped the design of widely used professional IDEs such as Eclipse, as well as educational IDEs such as Scratch and Code.org's CodeStudio.

Troubleshooting. Unlike many computer scientists, I view the *use* of software as a kind of programming—after all, manipulating the operating environment, configuration, and inputs given to a program can be just as confusing to software users as manipulating a program's code. My work has explored a vision in which software *explains itself*, so that users can learn models of how software works to help alter its behavior. Some of my work has generated these explanations automatically [C.10]; other work with my student Parmit Chilana explored crowdsourced explanations via help systems [C.27, C.31]. I took these innovations to market with my co-inventors Jacob Wobbrock and Parmit Chilana into a venture-backed startup called AnswerDash; it's products have reached 100's of millions of consumers, and its core innovations have been replicated by competitors at Oracle, Salesforce, and Zendesk.



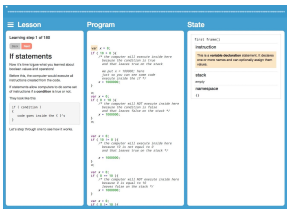
Verification. Discovering defects in programs is non-trivial. My research has taken the novel approach of investigating interactive and crowdsourced ways of detecting defects, leveraging developer knowledge that tools do not have. My work on *FeedLack* explored the detection of usability problems [C.24] and has inspired new automated forms of usability detection at Google. My work on *Frictionary* explored the mining of frequent software problems from help forums [N.12], and has inspired new data analytics tools in customer support tools. I have also explored detecting defects in dynamically typed languages by monitoring for anomalies in naming [C.20] and units of measurement [S.1].

Focus 3: Making programming easier to learn

My recent focus has been on how to help people *learn* to program. Whereas most prior work on computing education has focused on specific educational contexts (e.g., higher education programing courses, K-12 classrooms), my work has primarily focused on basic questions about learning programming, particularly in informal contexts.

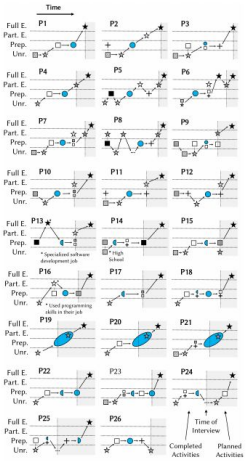
Learning to code online. With my student Michael Lee, I have shown that one of the central challenges in learning online is paying attention to the right information, because of the absence of personalized feedback, clear learning trajectories, and structure [C.47]. I've discovered key design choices that can guide learners' attention and keep them engaged. These include using anthropomorphized representations of compilers [C.23] and data [S.8], incorporating formative assessments [C.30], and predicting abandonment based on features of learning activity [C.41]. This work culminated in a game called *Gidget* [C.32], which has been played by more than 100,000 people worldwide. We demonstrated that Gidget not only rapidly shifts adult attitudes toward learning to code [C.33] but also produces superior learning outcomes to less guided experiences [C.36]. This award-winning work has directly impacted the Code.org's CodeStudio and Apple's Swift Playgrounds, which have been used by tens of thousands of teachers and hundreds of thousands of K-12 students across the United States.

Learning programming languages. Much of learning to code is about learning the syntax and semantics of programming languages. With my students Greg Nelson and Benjamin Xie, I have advanced foundational perspectives on what programming language knowledge is [C.42], how to sequence the learning of programming language knowledge [J.8], specific strategies for enacting knowledge of the evaluation rules of a language [C.54], and new approaches to measuring programming language knowledge in more valid ways [C.58, C.59]. These discoveries have broad implications for how programming languages are taught in K-12, higher education, coding bootcamps, in professional settings, and online. My work has already directly impacted the curriculum, pedagogy, and lesson plans of Code.org's instructional materials.



Programming problem solving. Equally important to programming skill is the ability to orchestrate the many activities involved in programming (editing, modifying, testing, and debugging code). With my student Dastyni Loksa, my work has led to several discoveries about the importance of promoting *self-regulation skills* to help learners structure their programming process [C.35, C.39, C.40, W.7], as well as explored a new vision for *explicit programming strategies* [J.9], which encode tacit problem solving expertise as partially formal, partially informal procedures that learners can follow [C.56]. Work in review has shown that some strategies are capable of helping novices achieve expert performance by following expert strategies.

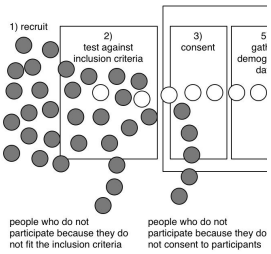
Broadening participation in computing. The world is full of barriers to learning to program. While other researchers have documented these inequalities in formal systems of education, my research has investigated sources of inequity in *informal learning* contexts. This include studies of high-privilege students' lifelong paths towards and away from computer science and the informal supports that facilitated their journeys [C.16], the structural barriers to CS transfer students accessing these same pathways [C.50], the parallel inequities being created in coding bootcamps pursued by adults [C.43], and the role of mentorship and family in supporting chronically underrepresented groups in discovering computing [C.44, C.52, C.53]. I've also recently begun to explore the structural barriers to educating future developers about how to engineer accessible software, including a national survey of CS teachers' perceived barriers to teaching about accessibility [C.55] and an exploration of ways to train CS teachers about how to teach accessibility [C.57]. Some of this work has been read widely, including on Reddit by over half a million readers.



Service scholarship

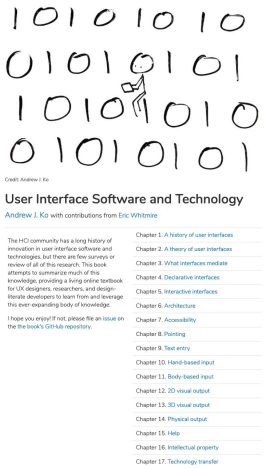
While most of my work has focused directly on programming, I've also pursued two parallel threads of "service" scholarship, which I pursue out of a demonstrated need in my research communities.

The first thread is *methodological*, including novel research methods for studying the human performance of programming [C.1, J.1, J2], deconstructions of the complexities of running controlled experimental evaluations of programming interventions [J.7], systematic design processes for designing empirical studies [B.3], and epistemological arguments about the competing demands of furthering design and furthering theory in applied disciplines such as HCI, software engineering, and computing education [C.48]. I am deeply fascinated by the endless challenges in the pursuit of knowledge, and how we do so specifically in studying programming. My methods contributions in this space have generally become widely used and widely cited; my epistemological works have generated controversy, social media debate, conference panels, and rebuttal papers.



My second thread of service scholarship is in publishing surveys of research literature. I view these as critically important to progress in research, accelerating the learning of newcomers to my fields. I have summarized the

state of the art in end-user programming [J.6], informal computing education [B.4], and tools and technologies for learning to code [B.5]. I have written book chapters summarizing scientific evidence about tools, testing, and productivity for professional developer audiences [B.1, B.2, B.6, B.7, B.8]. I have also authored three online books summarizing research literature on human and collaborative aspects of software engineering, user interface software and technology, and HCI and design methods. Each of these online books are living documents: people reading and using them from around the world provide feedback and I respond with frequent revisions.



Future work

Looking forward, I'll need to make some tough choices about which communities to invest in. I love my first community of HCI for its interdisciplinary breadth and its orientation toward change, but programming continues to be a niche topic in the field. I value my second community of Software Engineering for its fascination with the rich complexities of programming, but the field continues to be more interested in tools than developers themselves. Therefore, while I have no intent to leave these fields entirely, I think the best way to deepen my scholarship is to more fully commit to the field of computing education. I want to help the field deepen it's scholarship, broaden its impact, and establish its role CS departments, Information Schools, and Colleges of Education, all which require focus.

With this focus, I want to investigate the foundations of CS teaching and learning. What is programming? How do people learn it? How can we teach it? How can we teach teachers to teach it? And what role can technology play in teaching and learning? As old as the field of computing education is, its lack of theoretical foundations makes it difficult provide guidance to CS teachers about how best to support learning. I want to contribute these foundations, as much of my recent work has begun to do [C.48, C.60, J.8, J.9, J.10].

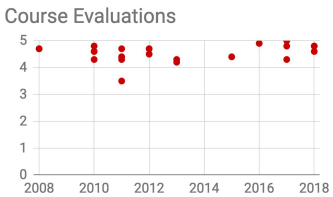
At the same time, I expect to pair these theoretical contributions with the very practical matters of everyday learning to code. How can we develop robust knowledge of APIs? How can convey strategic knowledge for problem solving? How can we support CS teacher learning, shaping their pedagogical knowledge for the ever increasing range of computing concepts they might teach? If we don't answer these questions, we will continue to have a world in which a privileged few can harness the power of computing.

Teaching

From the age of 7 until I went to college, I watched my mom teach 5th grade, in awe of her knowledge, her ability to manage a classroom, and her devotion to mentoring, coaching, and caring for each of her students. These early experiences with teaching inspired me to become a teacher long before I wanted to do research. I begin tutoring in 5th grade and throughout high school, then taught multiple CS courses as the instructor of record as an undergraduate. My role as a teacher is not only fundamental to my role as a professor, but it is also part of my identity.

I've always followed a few basic principles in my teaching: *engage everyone*, *mentor all who want it*, and *reflect on my practice*. While most of the students I teach are at UW, many are not, and so below I detail every group I teach and mentor and how I've tried to follow these principles in each.

Teaching in higher education



My primary focus in undergraduate education has been on three courses. My first course was our required **Design Methods** course, which covers HCI and Design foundations. I've taught this course for a decade, and have carefully crafted a reusable curriculum and a free web-accessible book on design methods to support it. Students enjoy this course immensely. On our student evaluation of teaching scale (a median of four prompts about the quality of the course content and instruction), I've scored a median of 4.6/5.0, never lower than a 4.3, and twice a 5.0/5.0. Many core and guest faculty have reused

my curriculum and performed similarly well. One student recently commented on one of my blog posts about teaching:

"I've used the principles learned in Andy's course every day of my software engineering career. Co-workers, friends, and family can blame him for my constant polite and impolite meanderings on design..."

The second course I designed was a software engineering elective I titled **Cooperative Software Development**. This course was the culmination of years of reflection on the *purpose* of a software engineering class for future software developers. After all, according to my research, most new developers learn most of what need on the job, and never stop; what should we teach them in college, if anything? After two years on leave as a CTO and engineering manager, I decided that the most lasting and foundational ideas were *not* dominant ideas in software engineering research such as formal verification, but rather the human aspects of software development: communication, coordination, collaboration, and comprehension. I wrote a book synthesizing the research literature on these topics and designed a course that engaged graduating seniors in a quarter-long project in which they immerse themselves in these challenges, learning to recognize and recover from failures in these skills, both in their own teams and others. All three times that I've taught the course, I received a 4.8/5.0 average, with alumni reporting that it's been the most important course they took at UW, helping them "parse" the organizations they join, the managers they work under, the processes they work in, and the teams they eventually lead.

The third and most recent undergraduate course I designed was my iSchool's **Intellectual Foundations of Informatics**, our introduction to information science for freshman and sophomores. Preparing for this course was an immense intellectual challenge, as I don't come from information science and haven't read most of its seminal work. I spent 6-months prior to teaching reading everything my colleagues recommended and analyzing the prior course materials of all of our instructors. Prior offerings had more of a survey feeling, with every day covering a different topic, but little to stitch together the core ideas in each. I set out to devise a single coherent narrative across 10 weeks, bringing together the many disparate disciplines in the iSchool into one vision. My first offering was a success: students scored me a 4.8/5.0, and said,

"Andy has been the best professor I have encountered in my college career. He is knowledgeable, friendly, patient and open to others' ideas. He shows that he cares about his students' thoughts, opinions and ideas... He balances speaking professionally and casually to students. He also does a

superior job of providing examples that are relevant or interesting to the class population (i.e. millenials/Gen Z-ers).

I summarized the big ideas in the course in a widely read blog post.

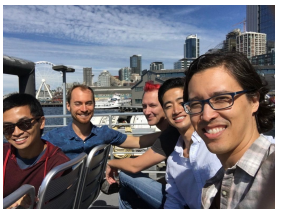
I also regularly teach a core course in our Masters in HCI and Design titled **User Interface Software and Technology**. Jeff Heer, a colleague in UW CSE, taught the original version as a user interface programming course. This is what many HCI masters programs teach, but I found the diversity of student's prior knowledge made such content too hard for students without a CS degree and too easy for students with a background in CS. Instead, I designed a first-of-a-kind course that covers the past, present, and future of user interface technologies from a conceptual perspective. Because such a course has (to my knowledge) not been taught, I wrote another web-accessible book to support the class, which synthesizes the HCI literature on user interface technology. In the first iteration of the course, students reported it was the best class in the program, and that I was the best teacher in the program (I received a 4.8/5). One student in their evaluation said:

"The course content was phenomenal! Super interesting and practical to consider interfaces from a theoretical standpoint. Exactly what budding UX designers like us need to be able to do."

In a mid-quarter evaluation of my second iteration, an independent evaluator who has evaluated hundreds of courses said:

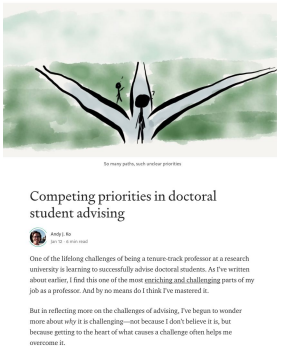
"*I didn't think there was any room for improvement from last year, but somehow you found it. The course is nearly perfect; I have no recommendations for improvement. In fact, I've been performing evaluations of other classes these students are in, and many have suggested that other faculty teach more like Andy.*"

Mentoring junior researchers



While I primarily view **doctoral student** advising as research, I frame it here as a core teaching activity, since I regularly reflect on it as teaching on my blog. Most of my research is a direct collaboration with one or more of my doctoral students, and more often than not, one or more undergraduate researchers. To date, I have graduated four students, two of whom are now tenure-track faculty (Parmit Chilana at Simon Fraser University CS and Michael Lee at New Jersey Institute of Technology in Information Systems), one a principal data scientist for Microsoft's Windows team (Paul Li), and one an architect for Apple's WebKit developer tools team (Brian Burg). I'm currently advising seven students, four of whom will graduate in the next two years. In addition

to teaching my own doctoral students, I also quite enjoy mentoring other people's doctoral students. I have served on 28 other students' dissertation committees. I have also organized 5 doctoral consortiums in the past 10 years (three at the IEEE Symposium on Visual Languages and Human-Centered Computing and two at the ACM International Computing Education Research Conference), and served on an additional two.



I also spend considerable time mentoring **junior faculty.** This includes six faculty in the Information School, three faculty in Computer Science & Engineering, and another dozen faculty at other universities that have formal mentoring programs that engage their faculty with outside mentors. I also regularly participate in the *International Conference on Software Engineering New Faculty Symposium*, speaking on research, teaching, advising, and fundraising. I have also recently started an informal annual gathering of new faculty at the ACM International Computing Education Research conference. I also blog regularly about faculty life, including over 30 essays that have been viewed over 100,000 times, on topics such as doctoral student advising, course design, time management, career planning, and dealing with rejection. These have led to interviews on podcasts such as Geraldine Fitzpatrick's Changing Academic Life.

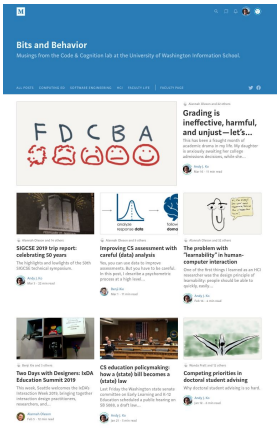
Teaching in high school

Since 2016, I've taught for UW's **Upward Bound** program, which reaches low income and/or first-generation college students from the south Puget Sound region. These students are typically recent refugees from other countries or Seattle natives in high poverty school districts. I usually teach a 6-week computer science class to 15-20 students in the summer, and recruit the help of undergraduate teaching assistants. In addition to teaching core computing topics, I also make an explicit effort to develop mentoring relationships with students. Of the 45 students I've taught so far, 12 (all now in college), still reach out to me for advice and to celebrate their achievements.

Teaching the public

I regularly teach the public—both software developers and CS teachers—by writing **books** that synthesize large bodies of work. I strongly believe that digital libraries of PDFs are *not* the ideal entry point for practitioners to learn.

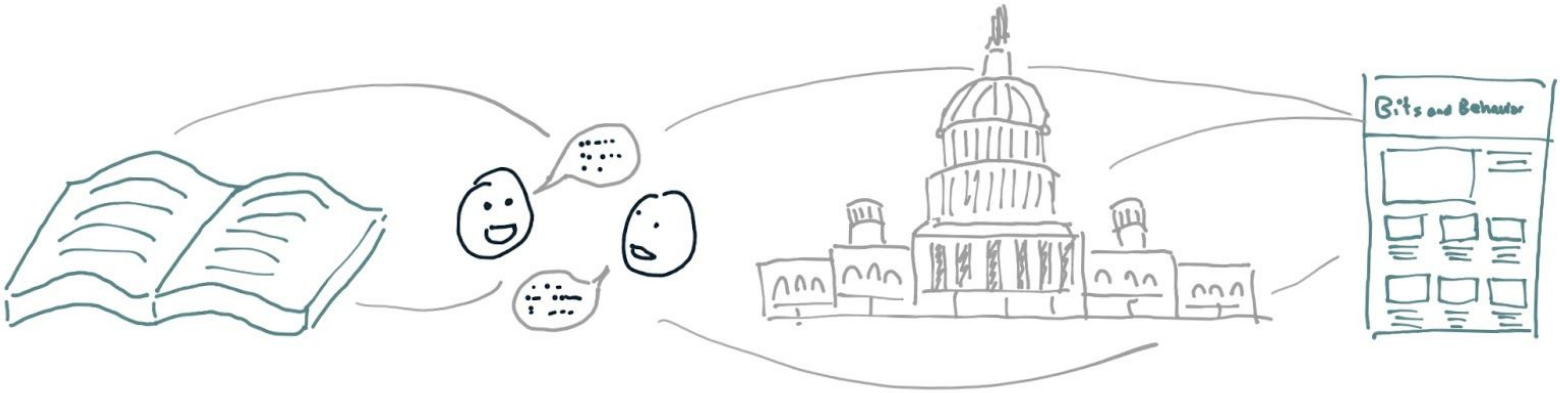
Rather, academia must synthesize our bodies of knowledge into more accessible forms. Therefore, in addition to the three free online books on design methods, software engineering, and user interface software and technology I mentioned earlier, I have also co-authored a survey of research on data science in software engineering (*Perspectives on Data Science for Software Engineering*), a survey on software engineering productivity (*Rethinking Productivity in Software Engineering*), and an edited volume surveying the entire history of computing education research (*Cambridge Computing Education Research Handbook*). I have also begun organizing a new book on Accessible Computing in collaboration with Jeff Bigham and Richard Ladner, to give developers and CS teachers the necessary foundations for creating accessible software.



Also teach the public by **blogging**. My audience includes academics, software developers, CS teachers, designers, and policymakers; my two primary topics are programming and academia. To date, my writing has amassed about 300,000 reads over 2 years, including 1,500 regular followers on Medium and the distinction of a top writer in Medium's Education channel. This growing audience has led to numerous opportunities for even broader communication to the public, including participation on podcasts (e.g., Software Engineering Daily, which reaches hundreds of thousands of developers), and frequent conversations with policy makers, CEOs of local startups, and administrators of public school districts across the west coast.

Next steps

In the coming years, I'm particularly interested in teaching **K-12 CS teachers**. Teachers are our most powerful leverage point in society; if we want a future in which society is ready to harness the power of computing confidently but ethically, we *must* prepare hundreds of thousands of excellent teachers who feel confident teaching about programming, information, and computing. I'm working closely with the UW College of Education and several regional universities in Washington state to build pre-service CS teacher education programs to do just this. I want to devise the curriculum for these programs, create evidence-based instruction informed by my community's research, and then teach these classes. If I can succeed at this, and replicate that success across the world, we will create an impactful, sustainable infrastructure for developing future generation's knowledge of computing and information.



Service

Service is synergistic with my research and teaching: it expands my network, helps me recruit students, opens new fundraising opportunities, and shapes my reputation. Since pivoting to computing education research, I've found even more synergy, as the service I do often has direct impact on my teaching and research, while also creating a direct channel for sharing my discoveries. Therefore, I've purposefully supplemented my normal service goals of supporting peer review and academic committees with larger, long-term investments at UW and beyond academia. Below are the achievements I'm most proud of.

Higher education administration

Program chair of Informatics. Since 2016, my service contribution to the UW iSchool has been leading our undergraduate program, *Informatics*. As my school's largest and most visible program, and one of the most popular majors on campus, it is no small amount of work. Therefore, a necessary goal has been streamlining operations of the program to ensure that both future chairs can sustainably integrate the role into their academic lives. However, my goals have been more than operational. I overhauled admissions to eliminate structural inequities while also streamlining review, increasing the number of transfer students, women, and students of color who we admit by 20-30%. I launched a new minor (now the most popular minor on campus). I streamlined the recruiting, hiring, and mentoring of guest (adjunct) faculty, helping to more deeply integrate them into our school's academic community. I have defined and communicated a unified vision of our curriculum and regularly share this

vision with other iSchools that do not yet offer undergraduate degrees (most recently with the University of Illinois at Urbana-Champaign). I've also championed cultural changes, including community norms that emphasize inclusion, replacing biased summative teaching evaluations with more formative mid-quarter feedback, and practicing a radical transparency in communicating with students and the public about admissions, curriculum, student experience, and diversity, primarily through social media such as Reddit "Ask Me Anything" posts and blog posts about diversity.

Program chair of the Masters in HCI+Design. In 2015, I chaired our MHCI+D program. At the time, it was in transition, needing space, a new director, a new lecturer, and a renewed agreement (between the iSchool, Human-Centered Design and Engineering, Computer Science & Engineering, and the School of Art + Art History + Design). In one year, I succeed at all three, successfully recruiting the excellent designer Michael Smith as our director, secured a lease for a dedicated design studio space for 35 students and 4 staff, and forged an agreement with all four units that ensured the program had sufficient faculty teaching load to teach its core courses. The program now attracts nearly 700 applicants a year for 35 slots, competing directly with other top masters programs globally. I continue to support its administration informally.

Community organizing and policy

Organizing the Puget Sound CS education community. Despite being one of the most active regions in the U.S. for CS education, it is also one of the *least* organized. Dozens of K-12 CS teachers have never met each other, CS faculty in higher education have rarely met CS education researchers like myself or my students, and product teams at places like Microsoft, as well as not-for-profits like Code.org, rarely talked to each other. To address this, in 2017 I started a quarterly meetup, which, to date, has brought together 500+ champions of CS education in our region. Not only has this created a more vibrant network of students, teachers, researchers, product designers, and policy makers, but it has translated into structural change, including a new CS advisory board in Seattle Public Schools (for which I am the secretary).

Leading K-12 CS education policy and infrastructure in Washington state. Much like Puget Sound, Washington state CS education activities have been disconnected, but much of the state is also underserved. This has been

problematic for many reasons: many students transfer between universities in our system and senators and representatives in our state legislature have no clear group to consult for K-12 education policy guidance. To fix this, in 2018 I created a statewide advocacy team as part of the NSF-funded Exploring Computing Education Pathways project. I brought together the head of Microsoft TEALS, the Office of the State Superintendent of Public Instruction, and a community leaders from central, eastern, and southwestern Washington who have been passionately organizing their own regions of the state. In just a year, my team has built a network of universities interested in creating new pre-service teacher education programs for aspiring K-12 CS teachers and created partnerships with TEALS, Code.org, and Washington STEM to lobby our state legislature for new K-12 CS education policy for the entire state. We have successfully passed two bills, one requiring reporting from schools about CS education activities, and one requiring all public high schools in the state to offer at least one CS elective. I'm the key expert in helping the state implement and enforce these regulations, ensuring equity is at the center.

International service

A central part of my service is **organizing and reforming peer review**. I am an Associate Editor for premier journals on computing education (ACM Transactions on Computing Education) and software engineering (IEEE Transactions on Software Engineering). I've served as senior members of premier conference program committees (ACM CHI, ACM UIST, ACM/IEEE ICSE, and ACM ICER). I've also helped modernize peer review at the SIGCSE conference, separating research papers and experience reports and writing the research track's review criteria. Throughout all of this, I have won multiple exceptional reviewer awards for insightful and constructive on-time feedback.

I've also played the role of a consultant and mentor in **national and international K-12 CS education advocacy**. I helped Google Education shape their K-12 CS education grant funding initiatives. I'm partnering with a coalition of Google engineers to modernize Software Engineering Education in higher education. I've joined as co-PI of AccessComputing, a program that lowers barriers to students with disabilities accessing CS; my role has been to start a national effort to expand what CS faculty teach about accessibility in K-12 and higher education (including a national survey, the creation of faculty professional development on accessibility, and editing a new textbook on accessible computing). I've joined the international *ACM Education Advisory*

Board, helping to develop ACM policy and frameworks, lately around ethics and accessibility. I've also supported the Computing Research Association, serving on its Outstanding Undergraduate Award Committee, co-authoring a CRA white paper on computing education research, and attending the Snowbird conference to educate CS chairs and deans about computing education. I also maintain a widely visited FAQ on computing education research, which thousands of researchers have used to learn to learn about the field and find collaborators and advisors. I have also regularly consulted with U.S. congressional staffers on national CS education policy.

Next steps

My future service goals all involve creating **sustainable academic infrastructure** that matures and grows the field of computing education research. I want to create pre-service CS teacher education programs at UW and across the state. I want to modernize the peer review in computing education research conferences and journals. I want to build a pipeline of outstanding computing education doctoral students and faculty. I'll know I'm successful when all CS departments and Colleges of Education view computing education as a core part of their disciplines, when there's a diverse pipeline of future tenure-track faculty for the field, and when there are sustainable programs at NSF to fund research on the topic. I'm committed to spending the rest of my career making this happen.

Diversity

I see diversity is inseparable from research, teaching, and service. After all, embracing diversity, and designing structures to support and celebrate it, is not about *supplementing* these activities, but *fundamentally changing* how we do these activities. That said, I believe it is important to address diversity explicitly when evaluating academic careers, and so here I briefly highlight the role that diversity places in each of my areas of responsibility.

Research. Up until 5 years ago, diversity was not a major component of my research. I was concerned with software defects, productivity, and other practical considerations around the design and construction of software. However, when I pivoted to computing education research, diversity quickly became both a key motivation for my work, but also an explicit phenomena of study. My work has since focused on 1) who is learning, who is not, and why; 2) barriers to accessible design in higher education; and 3) pedagogy and educational technology that can equitably serve learners of varying motivation, identity, and prior knowledge. I credit the fields of education research and learning sciences for giving me the language, concepts, and literature to pursue this work, and the UW iSchool for signaling that diversity-informed research would be celebrated, supported, and desired.

Teaching. All of my teaching begins from the goal of equitable outcomes. I have always viewed every grade I assign that is less than a 4.0 as partially my failure as a teacher. Therefore, whether I'm teaching high school youth, undergraduates, masters students, doctoral students, or the general public,

my goal has always been to invent more inclusive and engaging ways of teaching. For example, the books I maintain online are in standard HTML rather than print so that anyone, regardless of their physical abilities, can access them via computer. I designing inclusive pedagogy that doesn't require stigmatizing students with learning disabilities. I design activities that empower students to bring their funds of knowledge to class, rather than relying on my ideas from my own culture, to ensure that my students see a role for their ideas in their learning. And I mentor novice teachers in higher education to ensure they feel empowered to enact the same values.

Service. In my role as an administrator, my objective has been eradicating structural inequalities that erode diversity in academia. I eliminated GPA as a factor in our undergraduate admissions, as it is highly biased against transfer students and first-generation college students who for many reasons unrelated to their knowledge and skills, do not achieve comparable grades as more conventional students in higher education. I added implicit bias training and explicit criteria to our admissions process to ensure committee members would focus only on the criteria we have identified. I devised a review process that is asset-based, focusing on criteria that allow applicants to be admitted via multiple dimensions of success, not just grades or writing. I implemented a second annual round of admissions to support transfer students and interest changers who find our program late. I added a tutoring program to help students who have less prior knowledge in programming than students with more preparatory privilege. I added new courses to support freshman direct admits and transfer students, giving them some degree of community as they enter the program, following evidence that these groups often lack the social support necessary to thrive in their first year. And of course, I'm expressing many of these same values in my efforts to create state policy and programs on K-12 CS education, albeit at a much larger statewide scale.