Discovering Your Software Umwelt

REBECCA WIRFS-BROCK, Wirfs-Brock Associates, USA
ALLEN WIRFS-BROCK, Wirfs-Brock Associates, USA
JORDAN WIRFS-BROCK, Whitman College, USA

We apply the biological-behavioral concept of an *umwelt*, which is how an organism perceives and acts within its environment, to the practice of software development. By writing narrative descriptions of our own software umwelts and iteratively discussing and analyzing them, we develop prompts that can elicit reflection on how and why we relate to software in the ways that we do.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Social and professional topics → Computing profession**; *Informal education.*

Additional Key Words and Phrases: software development practices, software development careers, umwelt theory, reflective practices, cultures of programming

ACM Reference Format:
Rebecca Wirfs-Brock, Allen Wirfs-Brock, and Jordan Wirfs-Brock. 2024. Discovering Your Software Umwelt. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3689492.3689815

Software Developers Are People

As software developers[1], we each hold unique views of what creating software entails and how to best approach software design and programming. Maintaining an online transaction processing system running on an IBM mainframe is different from building the front-end of a web application; writing a Linux device driver is different from wrangling a dataset to create a visualization; designing flight control software for a modern airliner is different from scripting a non-player character for a video game. These scenarios differ in more than their application domains: The people working on these projects build software with different programming languages and different tools. They use different development processes, adhere to different programming paradigms, and speak with different jargon. Further, someone doing these activities professionally will encounter different wage scales and follow different career paths. Software developers operate within complex *environments* constituted of *cultures* [7]—people and practices—and *tools*. To understand software development, and who we are as software developers, we need to understand how people work within their environments. This essay explores the concept of a *software umwelt*—and how it is formed, shaped and tuned over time—as a way for software developers to understand themselves and to navigate the complex and ever-changing environments in which they work.

[1]In this essay, the term "software development" encompasses all roles and disciplines that contribute to the study and creation of computer programs and systems. We treat "programming" and "software development" as synonyms and use them both throughout this essay.

Authors' Contact Information: Rebecca Wirfs-Brock, rebecca@wirfs-brock.com, Wirfs-Brock Associates, Sherwood, Oregon, USA; Allen Wirfs-Brock, allen@wirfs-brock.com, Wirfs-Brock Associates, Sherwood, Oregon, USA; Jordan Wirfs-Brock, wirfsbrj@whitman.edu, Whitman College, Walla Walla, Washington, USA.

The concept of *umwelt* was first formulated by the German zoologist, Jakob von Uexküll, who proposed that humans, as do all living beings, experience life in terms of species-specific, spatio-temporal, "self-in-world" subjective reference frames [6]. In German, umwelt means "environment" or "surroundings." According to von Uexküll, there are two aspects to our umwelt: (1) our unique perception of the world and (2) the actions we can take to affect that world. Our umwelts are personal and unique. What we see is not necessarily what we perceive: How we act on our perceptions is partly based on the significance we assign to them. For example, many primates, such as chimpanzees, will recognize a rope lying on the ground that is attached to visible but inaccessible food. They will pull the rope to retrieve the food. In contrast, tree-dwelling gibbons, when on the ground, ignore such a rope; but if that same rope is hanging from the top of their cage, gibbons will pull it to get the food [1]. Apparently, a gibbon's umwelt has an arboreal focus that filters out a rope on the ground. Gibbons are biological creatures, so are people—including programmers.

Our umwelts are multi-faceted. People operate in many contexts, such as home life, work life, and public life, where we perceive and act upon different things. For each of these contexts, we have our own specialized sub-umwelt. Software development, whether it happens in professional settings, classrooms, open-source projects, or elsewhere, is one of these contexts. Thus, every software developer has a unique software umwelt which is part of their more general umwelt. In this essay, we explore the ways we might examine the proverbial ropes in our software umwelts, the ones we recognize as well as the ones in our blind spots, in order to be better software developers.

To Develop Software Is to Adapt

Our programming habits, as well as our software design values, heuristics, and practices seep into our umwelts through our experiences [9]. Within the broader discipline of software development, we form communities based upon beliefs and practices—communities that sometimes have major conflicts about what is a more intuitive or appropriate way to work. Controversies sometimes divide the worlds of software development: Some things that a programmer of a dynamic object-oriented language considers to be best practices would be considered undisciplined or even dangerous by a programmer from the purely functional programming world. Why is this? How is it that we come to develop such strong feelings about core beliefs and values that drive our software practices? And are there ways for us to reshape our beliefs if they have become counter-productive? We assert that noticing and describing our own software umwelts can help us more deeply understand our relationship to these different software communities and might give clues as to how we might navigate between them.

From the transition from mainframe computers to personal computers, to the internet, and now to the use of large language models as generative tools to write code, software development has always been about change—in our tooling, our programming languages, our deployment platforms and design processes. When we encounter something unfamiliar in our environment or enter a new environment, our umwelt can no longer be a reliable guide. It may even mislead us. In such situations, only our untuned perceptive capabilities are available to us. We don't know what's significant nor are we adept at determining appropriate actions to take.

While our umwelt dictates our perceptions and constrains our actions, it isn't static. Through interactions with our environment, we learn and adapt, and as we do, our umwelts grow and evolve with us. The concept of umwelt may be a useful model to help programmers navigate inevitable change—but only if we can be aware of our umwelts first.

How Do You Spot a Software Umwelt?

The trouble with umwelts is that it is hard for us to perceive them: How do we go about seeing our own eyes, or smelling our own noses? If everything we do is filtered through our umwelt, then we can never truly step outside of it.

So is the task of trying to sense our own umwelt a futile exercise? Maybe, but in this essay we are going to try. Just as pair programming helps us understand our own coding habits and thought processes, we posit that our umwelts might become evident when they bump up against those of other software developers.

To explore this idea, we the authors embarked on a collaborative journey to understand our own, and each other's, umwelts. We are closely related: Rebecca and Allen are married, Jordan is their daughter. We have had many discussions about software design and programming—over beer, over the phone, in the car, in our easy chairs—where we told stories about how we got involved with programming, how we think about it, and what we love about it and what irks us. We were intrigued by our differences, and our discussions were interesting (to us), but unfocused. So we decided to use the concept of umwelt as a lens to observe and reflect on our prismatic approaches to software development, by noticing and describing how our respective experiences have shaped our umwelts.

We tried an experiment: We would each write a reflective narrative discussing the emergence and evolution of our unique software umwelts. Because of umwelt theory's emphasis on individuals' unique perceptions and resulting actions, we can compare the effects of our experiences on our different umwelts. To begin this process, each of us independently wrote a narrative description of our own software umwelt by responding to a prompt that includes these questions: *How do you perceive software development? What values do you assign to those perceptions? How do you approach a programming problem? How did you learn to develop software the way you do? Why do you act the way you do and not in other ways?*

We refer to these narratives as our *software umwelt stories*—and we will excerpt them throughout this essay to illustrate our ideas. These stories were just the starting point. Next, we read each others' stories and commented on them, asking questions in places (*"Does the distinction between between applications programming and systems programming still exist as such?"*) and reacting with surprise in others (*"So interesting that you had to 'translate' Smalltalk concepts to other languages to conceptualize them! Can you expand on this?"*). It is through this living discourse that each of our own software umwelts—and the nuances that make them unique—emerged.

Our process draws on philosopher Donald Shön's idea of reflection-in-action. Shön observed that, "knowing is ordinarily tacit, implicit in our patterns of action and in our feel for the stuff with which we are dealing…our knowing is in our action" [8]. He suggests that we might tap into that tacit knowledge by engaging in surprise—by making familiar actions strange again so that we might re-examine them with fresh eyes. By telling our software umwelt stories—which included important events, origin stories, and other moments that stood out to us (e.g. caused us to be surprised)—we gained insights into how our experiences shaped our umwelts. By reading and asking questions about each others' stories, we came to deeper insights. What seemed banal to a narrative's author proved to be an interesting or surprising aspect of our umwelt formation by another.

Through this discourse we identified elements of a software umwelt story that seem most foundational, and turned those elements into threads. What follows are the threads that we found in our software umwelt stories. (Our full individual narratives and the common prompt we each started from are included as appendices to this essay.) With each thread, we've included a new prompt that we think will be useful to readers who want to repeat this exercise with their own software development team or class.

Umwelt Origin Stories

Prompt: *What is your earliest memory of writing code? What type of device did you use? What activities did you do? Who else was involved in this memory?*

Today, computing devices are ubiquitous and most children unconsciously incorporate the existence of such devices into their general umwelt. But there is a difference between using these devices and trying to manipulate them to create something new. We think that how and when someone first becomes interested in using computing technologies as a creative tool influences how they relate to those devices later on in life. Thus, reflecting on how we came to first understand programming can be a crucial foundation for understanding the rest of our software umwelt.

As we wrote our own software umwelt stories, we reflected on the origins of our relationships with computers and programming as an activity. Allen learned machine language programming in the 1960s at a summer course between the 8th and 9th grades on a Burroughs 205, a mid-1950s room-sized vacuum-tube decimal computer whose main memory was a magnetic drum. He found coding to be radically different than other educational experiences:

> **Allen**: …I had many of the available science toys—chemistry sets, electronics kits, etc. but I had found them frustrating. They mostly provided recipes for experiments using the provided components. But, they didn't teach me how to come up with recipes for my own science experiments. Programming satisfied that itch. *Software was to be my creative media.*

Rebecca didn't lock on to a lifelong interest so quickly. When she encountered computing as part of a college job, she became curious about how computers worked and signed up for a FORTRAN programming class.

By the early 1990s, as Jordan was growing up, computer-based devices were more common and computing enrichment programs were available for children. Living with parents whose professional lives revolved around computing, they gave her small design problems to work out using CRC (Class-Responsibility-Collaborator) cards. Her parents told her this story many times, until it became part of family lore:

> **Jordan**: …I imagine that I drew simple pictures to go along with the names of the classes and their responsibilities, and that I would discuss the solutions with my parents, who would weave in software design nuances so subtly that I didn't notice…I also remember creating interactive stories using Hypercard in a summer camp…But as I got older, I left programming behind and focused on other interests: hiking, geology, creative writing.

We each have distinct software origin stories: Allen is the precocious adolescent discovering and compulsively pursuing a compelling interest. Rebecca is the curious college student learning that programming can satisfy an itch to understand more about what computers can do for us. Jordan is the computer-literate millennial initially trying to find a path distinct from her parents before (as we will see) eventually meandering back to software development. These software developer origin stories have implications that ripple through our umwelt development. As we grew beyond these early beginnings, we carried their traces forward with us.

Nurturing Our Umwelt

Prompt: *How did you learn more about programming? What did you learn about yourself in the process? What were the "sticky" things you learned and why?*

After being introduced to a new environment—in this case programming—our umwelt grows as we encounter more features and develop more perceptions and skills related to that environment. New skills may come from formal training, self-directed study, reference materials, other people, or hands-on experience. The skills and the style of their acquisition impacts our values and our beliefs about ourselves as programmers.

Since her first programming class piqued her interest, Rebecca continued to take programming classes (along with some computational theory) and ended up graduating with a double major in computer science and cognitive psychology.

> **Rebecca**: I found I liked writing code because the amount of effort that I put in paid off in direct results. Also, I could work on my programs as much as I wanted until I got the structure and algorithms "right." I liked tweaking code…

She found programming to be much more personally satisfying, and less stressful, than, "proving theorems in Calculus, where I had to get the 'right answer' at first crack." In her practicum course, she wrote a small business application in COBOL for the computer center. Wow! With a little effort you could write programs that people actually use.

Allen credits the summer course using the Burroughs 205 for teaching him fundamental concepts of programming:

> **Allen**: All computers can do is carry out commands to store, retrieve, and simply manipulate numbers. Any information—including a computer's commands—can be encoded as grouping of numbers. Programming is all about identifying the informational part of a problem, choosing an encoding, and writing sequences of commands that manipulate the encoded information; most programs involve loops. You will spend a lot of time debugging your programs.
>
> *I still apply these concepts—in more sophisticated forms—when I program. They became fundamental tenets of my software development umwelt.*

After that first course Allen, as a high school student in 1966, didn't have access to physical computers; but that didn't stop him from continuing to learn programming. He found books on programming at the public library and he discovered that,

> **Allen**: [I] could walk into the local IBM field office and order complete manual sets for any IBM computer...I spent the next couple years reading manuals and writing programs on paper for desk execution. *I learned that programming was an intellectual activity that I could do anywhere—even if I didn't have access to a physical computer.*

In contrast, Jordan wanted to put "intellectual distance between myself and and my parents" and didn't further engage with programming until it was required by her undergraduate major:

> **Jordan**: [I] bemoaned the required programming classes I had to take in my aerospace engineering degree—until I realized that I enjoyed them vastly more than studying fluids and thermodynamics. Still, computer programming felt cryptic, something that strayed just beyond the edge of my comprehension. I could scrape together passing assignments with a bunch of help, but they never truly made sense. It wasn't until several years later, after countless self-paced courses, guided workshops, and half-baked personal projects, that things finally clicked.

What makes computing a "sticky" interest that leads us to develop a distinctive software umwelt? Perhaps this is a matter of self-perception, positive experience, supportive environment, and ongoing interest. Many kids go to computer camp or take intro programming classes and don't transform into programmers, even casual ones. Some people learn a first programming language and stop there. The environment in which our software umwelt grows and

develops is somewhat a matter of chance, but it is crucial to how we relate to software as a discipline: Is it supportive or competitive? Off-putting or welcoming? Engaging or boring? Is it fun?

Our beliefs about who we are and what we are capable of doing are key factors in how persistent we are at engaging with software development. By coming to understand what nurtures us, we can prepare for our unique software development journey. Not everyone is the same. Are you someone who learns best by interacting with others? If so, are you around the right people? Are you moetivated by self-learning? Are you determined to figure things out on your own? If so, are you encouraged to tinker and experiment? These characteristics and the situations you find yourself in can pull you in or repel you from software as a lifelong interest.

Putting Our Umwelt to Work

Prompt: *When did you first feel competent/confident in your programming skills? What things did the "real world" teach you that you hadn't studied? What was the first software project you worked on that made you feel proud, and how did it unfold?*

Our umwelt moderates our perceptions and influences our actions. In some situations, our umwelt development may be driven by our need to take action in a new environment. Alternatively, our umwelt may be developed through training in preparation for dealing with an anticipated environment. As described up to this point, our software development umwelts were mostly developed through training. Ultimately, we each needed to face the real world of software development.

While in high school Allen found various ways to work with real world computers. He got access to programming environments on various timesharing systems and had free-range access to a PDP-8 minicomputer at the local science museum.

> **Allen**: After high school I found a variety of programming jobs and eventually was credential motivated to get a CS degree . . . At that time a common division of programming labor was between applications programming and systems programming. *I soon realized that I was a systems programmer*—someone that builds "pieces of infrastructure that integrate multiple interacting parts, and sit underneath application code" [3].

Allen spent nearly fifteen years at Tektronix as a Software Engineer ultimately specializing in programming language design, implementation, and tooling.

Rebecca, armed with a newly minted CS degree, landed her first programming job as an Analyst Programmer II for the city of Vancouver, Washington. But she quickly moved on to something more challenging.

> **Rebecca**: I spent 3 short months writing thousands of lines of water and sewage billing code in COBOL. . . No one read or tested my programs—I was given hand written specifications which I took as a starting point. I penciled in my programs on IBM COBOL coding forms and gave them to our keypunch operator to type up.

Rebecca was comfortable in this software environment, as her COBOL programming skills she had acquired in school transferred directly. But no one cared about her code: They didn't even read it. She thought that if this was all there was to was the life of a COBOL programmer, there wasn't much more to learn. Rebecca didn't want to program in a vacuum: She wanted to work with people who programmed as a team. She wanted to grow her programming skills and liked learning new things, so she sought more challenges. After a few months writing COBOL Rebecca was recruited for a Software Evaluation Engineer job at Tektronix and jumped at the opportunity. She tested graphics libraries, terminals,

and an early graphics workstation. A year later she became a full-fledged Software Engineer I and was assigned to single-handedly build a linking loader that supported multiple microprocessor architectures. She was routinely assigned to design and build things that were completely new to her.

> **Rebecca**: Over the next few years at Tektronix, I held engineering positions with increasing responsibility. I was lead developer for a networked file server…I led a fast-track firmware development effort for a low-cost graphics terminal. We wrote functional specs and created structured design documents. We incrementally developed and tested our code, and held frequent code reviews. This all was just standard practice.

In a rapidly growing tech company, with lots of new products being developed, there are plenty of opportunities to grow a software development umwelt. Through her experiences, Rebecca learned the discipline of an engineering process. She learned how to invent. And imprinted into her umwelt was the belief that she was capable of developing new and innovative software in areas where she had no prior expertise.

After several years without software development in her life, Jordan discovered the need to reactivate her vestigial software development umwelt. She earned a master's degree in data-focused journalism and began working as a data curator in the non-profit sector. Here, she ended up as product manager for a complex software development project—a role in which she felt out of her depth.

> **Jordan**: Thankfully, the lead software engineer I worked with was one of the most patient, generous people I have ever met. As he taught me what an agile development process was and how to run one, he was never condescending.

Within this software team, Jordan felt safe enough to learn and grow—which is not the case in all teams. This experience helped her see herself as someone who is capable of creating software: On this project, "I didn't write a single line of code, yet I felt like I understood the software thoroughly." After that project, Jordan worked as a data journalist for a public media news organization. Armed with the knowledge that she could comprehend code other people had written, she was excited to expand her programming skills on her own to do data analysis and visualization—something she may not have had the confidence to do without her prior experience of leading a complex software project.

Early in our careers, our umwelt may be less developed and readily adaptable to a variety of software development cultures. Unless we land in a newly formed organization or a fledgling project, we'll have to fit into a pre-existing software development culture. Our umwelt will eventually need to conform, adapt, or reject that culture. As our umwelt further develops, our cultural fit will become a bigger factor in our future career choices.

We might land a programming gig, writing small, standalone programs. We might drift into significant programming activities as a vital side-effort in support of our "real" job. We might develop business applications or find ourselves immersed into an engineering culture where we are expected to learn new skills while designing and building complex software products and systems. But where we find our initial niche is largely a matter of the interests we decide to pursue (and whether our skills match that particular environment), and happenstance. But if you have an awareness of your developing software umwelt and your values, your next actions might be more purposeful.

Refining Our Umwelt Through Practice

Prompt: *Describe how your relationship to software changed as your expertise grew. Did you become a generalist or specialist? How did your software development values and practices change?*

If you work in a relatively stable environment, you may find yourself settling into a development rhythm. Your software development umwelt grows and deepens as you gain proficiency. As you take on new software development efforts, you will stretch your umwelt to acquire new knowledge and skills. You may encounter new-to-you programming languages and development tools. You may develop your own sense of style.

Allen found that he liked systems with "intuitive understandable command names and contextual defaults"—a preference that predates the development of GUIs. He disliked systems that used obscure notations or required the user to repetitively tell the system things it should already know.

> **Allen**: This developed into a personal design aesthetic that was incorporated into my umwelt: *Software is created by humans and exists to support humans and human activities. The design and implementation of software should be approachable, understandable, consistent, and comfortable to the humans that will interact with it.* This aesthetic relates to both the internal design of software and to the design of software interfaces, both human interfaces and the interfaces between software components.

When Allen and Rebecca joined Tektronix, its corporate motto was "Committed to Technical Excellence."[2] The manifestation of that model in the Tek engineering community made a strong impression on both of them.

> **Rebecca**: Tektronix was an engineering company where the "next bench" philosophy was practiced—build something of high quality that your peers would use.
>
> Tektronix had lots of internal processes for designing and engineering hardware products…
> most of the processes and various design documentation requirements we used were imposed on us software developers with very little adaptation.
>
> My software development umwelt after 5 years at Tektronix as a Software Engineer was largely shaped by my experiences working with highly motivated peers who held themselves to high standards. *We worked hard, we were supported by our management, and were expected to specify, design, and build products (and the software embedded in them) from the ground up.*

Rebecca internalized the values that were formative to Tektronix's engineering culture and grounded their practices. She came to deeply value producing quality software.

At the non-profit, Jordan "successfully" delivered a complex software project, a community data storytelling platform, on time and on budget. But she learned a hard lesson:

> **Jordan**: …we delivered what we said we would deliver. At launch, our platform was beautiful, easy to use, and well-built. But…we delivered something that people didn't actually want…We should have paused to do more exploratory research on the needs of our community members and stakeholders…but at the time I didn't know to ask for it.

A couple years later, while working as a data journalist, Jordan, who had first programmed using Hypercard, discovered that a live programming environment was the key to her embracing coding.

> **Jordan**: …I began dabbling with Python Jupyter Notebooks. For the first time in my life, I loved writing code. I loved being able to tweak a single line of code and see the results immediately in a data frame

[2] A few years later, the motto was changed to "Committed to Excellence." Some employees from that era believe this was the start of Tek's decline.

> or data visualization. I felt like I had agency and power, and I understood what I was doing. Yes, I still faced extreme frustration, but it felt worth it because I could see what I was working toward.

Experiences can lead to discovering both what we like—which isn't necessarily the same as what we are good at—and forming lasting interests. Our experiences not only contribute to growing our software umwelt, they also shape our beliefs about ourselves as people and programmers. If our experiences are positive, we are likely to seamlessly integrate many of the values, practices and perceptions of the programminmg cultures we are part of into our own umwelt.

Umwelt Pivots

Prompt: *Describe a significant life or career pivot. Why did you make it? How did this change your views on software?*

The term "half-life of knowledge," coined by economist Fritz Machlup [5] denotes the time it takes for half of the knowledge in a profession to be superseded. To stay current in any field, we must acquire new knowledge and integrate it into our umwelt. In 2008, Philippe Kruchten took a stab at estimating the half-life of software engineering [4]. He examined the 1988 issues of *IEEE Software* and informally assessed which ideas were "still important today or at least recognizable." As a result of this exercise, he conjectured that the half-life of software engineering ideas to be around five years.

Certain changes in your software development environment may appear inconsequential in the moment. For example, taking on a different role for the duration of a project or moving to a development team which follows new-to-you practices. However, they can subtly grow your software umwelt in ways that open up the possibility of making significant pivots.

After a short stint programming in COBOL, Rebecca pivoted to a software testing role in an engineering company, and the following year made another pivot to a software engineering role. She didn't hesitate as she was eager to grow her design and engineering skills. These pivots were easy to make as she didn't have a lot invested in being a COBOL programmer or a tester. Changes of scenery—or if you're fortunate, inspiring glimpses into an entirely new and intriguing software development umwelt—can also lay the groundwork for more significant pivots.

If startling enough, exposure to new ideas might even reframe your worldview of software development, and lead you on an entirely new career path. In 1978, Allen attended a talk by Alan Kay titled "Don't Settle for Anything Less." Kay showed filmed examples of early versions of Smalltalk and other systems and applications developed at Xerox PARC:

> **Allen**: Kay's vision aligned with my software design aesthetic…how could I contribute? I transferred into my employeer's advanced development "labs" in a role that today might be called a Research Software Engineer. A couple years later I had the opportunity to try to make Smalltalk practical for use on affordable personal computers [2]. At first this seemed like an iffy possibility. Smalltalk had been developed running on personal supercomputers not on commodity microprocessors. After several false starts I was successful…

Allen went on to be the software architect of a family of Tektronix Smalltalk workstations, and Smalltalk and object-oriented programming became his career focus for the next twenty years.

We carry our existing umwelt with us. Either we transform our existing perceptions and actions into ones more suitable for our new environment, or we discount them. We'll certainly have to tune our perceptions to pick up on relevant new cues and then learn how to act on them:

> **Rebecca**: . . . along came Smalltalk. . . unlike any programming environment I'd encountered: immersive, graphical, and utterly alien. Its syntax was simple, but the terminology for Smalltalk programming concepts were foreign. I had to translate Smalltalk terms to roughly equivalent programming constructs familiar to me before I could even start to understand what people were talking about. . . What exactly was a Smalltalk application? What were good ways to think about structuring programs? When should I define my own classes and when should I use existing ones?

When we do make major pivots, we can't simply flush out our existing software development umwelt and start over. Our brains don't work that way.

The programming culture you find yourself in can significantly impact the trajectory of your pivot. Fortunately, Rebecca worked in a collaborative engineering culture: A fellow engineer read her first awkward Smalltalk code and offered welcome advice. Programmers, experimenting with ways Smalltalk could be used in Tektronix products, freely shared their insights with her along with their code.

Over the next several years Rebecca immersed herself in Smalltalk, leading Tektronix' Color Smalltalk product development effort. She came to value carefully designed class libraries:

> **Rebecca**: I came to appreciate the effort it takes to design frameworks . . . Developing libraries for others' use demands attention to detail, code comprehensibility, design coherence, and utility. But people won't know how to use your libraries unless you present a coherent model of how your classes and objects work together and demonstrate how to use them. Working code isn't sufficient.

Successfully making a large pivot requires letting go of certainty. It takes time and concerted practice to adapt your umwelt to a new environment. Once you do, you will have gained your footing, know how to think and act in that environment, and feel comfortable:

> **Rebecca**: In retrospect, my Smalltalk experiences have had a huge impact on my software development umwelt and the trajectory of the rest of my software development career. . . I grew to understand how productive Smalltalkers thought, I realized that there were different schools of thought about how to approach object-oriented design. I found that initially thinking in terms of the roles objects play, their "responsibilities" if you will, and how they interact led to significantly different design solutions than if you first focused on how objects responded to events, or what states they were in, or their internal data structure. This led to my inventing responsibility-driven design, teaching numerous design courses, and writing two books about object design.

One of Jordan's pivots began while she was working as a data journalist in a radio organization. She struggled to bring data to life in radio stories, and became frustrated saying, "Check out our website to see visualizations of the data in this story"—something she knew radio listeners would rarely do. She discovered data sonification, which is the practice of communicating data through sound, and began tinkering with it when, "I learned about a Python package. . . to turn time series data into MIDI notes by mapping quantity onto pitch. . . Here, code was a means to creative expression."

Some pivots are hiding in plain sight, and we don't know that we are experiencing them. At the time, this seemed like a fun, one-off project—but Jordan would return to data sonification several years later, while getting a PhD in Information Science. Jordan left the field of data journalism, and returned to grad school, because a seed that had been planted earlier—the civic technology project that didn't meet the needs of the people it aimed to serve—grew and followed her as her career progressed:

> **Jordan**: I loved being a data journalist: In this role, I could combine my love of storytelling with my love of data to find insight and meaning. And I was constantly learning new coding and technology tools to help me do my work, which I found immensely satisfying. Yet I sensed something was missing: Our team spent all of our efforts on reporting and communicating stories, yet we spent almost no time understanding our audiences. What issues did they care about? Were they understanding the complex energy topics we tried to explain? I left journalism to pursue a PhD in Information Science with [a human-computer interaction] emphasis, so that I could learn how to understand people and how they interact with information and technology.

There, her two umwelt pivots collided:

> **Jordan**:…I created many more data sonifications using all kinds of tools…programming languages, no-code tools, I even developed my own techniques to make "lo fi" sonifications using my voice and body…I found the range of tools just as interesting as the sonifications one could produce with them. I also saw coding as a practice that integrated into a constellation of other ways of creating with data and computation—including algorithmic expression that did not require a computer at all.

The culture you are currently in makes career pivots more or less difficult. It can be hard to see a pivot when you are in the middle of one. You don't know which pivots will be, well, pivotal, and which will just be bumps in the road. However, a COBOL programmer isn't fated to be a COBOL programmer. A software engineer isn't destined to develop embedded software. A data journalist can dive into the culture of academia and find her research niche. If you are itching to try new technologies, tooling, or practices, you can find opportunities to acquire new skills and make significant pivots throughout your life as a programmer.

Distilling Our Umwelt

Prompt: *What are the core values involved in how you develop software? What makes you unique as a software developer?*

We undertook an experiment to find out if writing and comparing reflective narratives of our personal journeys as software developers could enable us to step outside ourselves and perceive the important details of our software umwelts. So, was this experiment successful?

We think so.

We gained insights into our personal software development umwelts that we previously lacked. We teased out common threads that run through our unwelts and that are likely present in other software developers' umwelts. But, the concrete details of our unwelts—how they filtered our perceptions and influenced our actions—were scattered throughout our thousand-word narratives. It was all too easy to lose ourselves, and hard to answer the question, "What *is* your software umwelt?"

So, to continue our reflective journey, we gave ourselves one final challenge: to distill our software umwelts down to their cores, to craft a kind of personal software developer mission statement:

Rebecca: The foundation of my software umwelt is curiosity. It drives me to explore new things and immerse myself in new environments. I can be uncomfortable and uncertain of my abilities, while at the same time believing in my ability to successfully navigate new environments. Becoming comfortable with being uncomfortable leads me to find ways to adapt, grow, and stretch my ability to perceive and act. I never hesitated to research design options or ask other developers questions. My ego isn't tied to figuring things out myself. I'm good at pinpointing key aspects in new

environments, processes, or practices and explaining them to myself and others. My outsider's perspective, along with my design values and practices offer unique and valuable insights. I can zoom out to establish the bigger picture of the overall system architecture, or dive into details and tweak the software design, processes, techniques, or practices. And then as a communicator, I write papers, essays, and books.

Allen: Software is my creative media and programming is an intellectual activity that can be done even in the absence of a physical computer. I'm a systems programmer. I value technical diversity, expect ongoing evolutionary change, and relish radical innovations. Lateral thinking is essential for software innovation. I believe that software exists to support humans and human activities. It should be approachable, understandable, consistent, and comfortable to the humans that will interact with it. I've come to understand that programming is model building. I can build software models of real and imaginary worlds. It's all the same to a computer. Programs are built from layers of abstractions that are ultimately rooted in the numeric manipulation capabilities of computers.

Jordan: My software umwelt is tightly coupled with my familial umwelt, and is a matter of legacy and identity. It has resulted in pride and curiosity. Writing software requires a type of thinking that we don't often encounter in other dimensions of our lives. We need to empathize with how other people relate to programming. There are many ways to program, and embracing them leads to broader and deeper perspectives and ways of doing things. Software development is a sociotechnical system. I'm always looking at systems, from a standpoint of trying to figure out what people and tools are required to navigate them. But the real power of software is that we can make things that people use—technologies that change the world. But how do we know that we are making something people want? That they need? That doesn't harm them? To consider these questions, we must always remember to to pull back and see the bigger picture.

By revisiting and reflecting on your own story, refining and rewriting it, and doing so alongside other people, you can identify what is truly important to you when it comes to software development. By sharing your stories with others, you gain a more nuanced understanding of your umwelt and where your umwelt is similar and different from others.

Join Us in Writing Your Own Software Umwelt Story

The exercise of discovering your own software umwelt is hard. It requires effort and intentionality. But it is worth it. We invite you to join us to reflect on your own software umwelt, and—after flailing through this activity—we offer some advice:

- Use the prompts we've provided as starting points. Modify them to fit your history, circumstance, and what you hope to learn.
- Start with descriptions of memorable events. Then, read what you've written and reflect on their meaning. Why did these experiences stick with you? What values are embedded in the details that you fixated on?
- Do this with a buddy—or two, or three. Build yourself a reflective community of people you trust and feel connected to.
- This is ongoing work that is never done. Step away from it, then come back. Alternate between modes of inquiry (writing, discussing, reading).

Reflective writing includes both descriptions of memorable events and reflections on their significance. It is subjective and personal. You may not have done this kind of work before, so be kind to yourself as you start writing about your experiences and examining your beliefs, values, attitudes and assumptions that form the foundation of your umwelt.

This essay is a starting point. We hope that software developers, researchers, students, and programmers from all walks of life might use these prompts to uncover, reflect on, and expand their own umwelts. When you do, please share them with us—we can't wait to perceive them.

Appendix

A Full Prompt and Narratives

Prior to writing this essay the authors decided to try an informal experiment. We would each write a self-reflective narrative discussing the emergence and evolution of our unique software development umwelts. To get started on a common track we agreed upon following the following **Co-author Narratives Prompt**:

The concept of umwelt was introduced by the German zoologist, Jakob von Uexküll. According to von Uexküll, we—as do all organisms—experience life in terms of species-specific, spatio-temporal, "self-in-world" subjective reference frames or umwelt. (from Wikipedia) Our umwelt filters our perceptions. Subsequently, how we act is based on those perceptions and the significance we assign to them.

When we encounter an unfamiliar environment, only our "untuned" perceptive capabilities are available to us. We don't know what's significant nor are we skilled at determining what actions may be appropriate. It is through repeated interactions with our environment, that we learn, fine tune our perceptions, and grow our skills at producing desired effects.

So, what do we know about software developers' umwelts? How do personal experiences affect the formation of our software development unwelts? How do they subsequently impact and shape software development careers?

Write 1500 words or so about your software development umwelt and its formation. Potential points to discuss:

- *How do you perceive software development?*
- *How do you interpret a programming problem?*
- *What values do you assign to those perceptions?*
- *How do you approach a programming problem?*
- *How did you learn to develop software the way you do?*
- *Why do you act the way you do and not in other ways?*
- *Were there formative experiences that imprinted on your umwelt?*

A.1 Rebecca Wirfs-Brock's Umwelt Narrative

My introduction to computing was at the University of Oregon's Test Scoring Center where I had a part-time job ensuring students' multiple-choice exams were properly marked. They were read by a machine onto tape that was then processed at the computer center. The output of the test scoring program was a listing of the students' test scores and some statistics.

Curious about that test scoring program, I signed up for a FORTRAN programming class where I typed my WATFIV programming assignments onto punch cards. I turned those in and picked up a listing of my results a few hours later. I was lucky to get a couple of execution runs per day. I got a printed trace of compiler errors (and usually, just the first one that tripped up the compiler and sent everything else south). Once my program compiled, I got (erroneous) output to look at…so debugging took a lot of effort. I found I liked writing code because the amount of effort that I put in paid

off in direct results. Also, I could work on my programs as much as I wanted until I got the structure and algorithms "right." I liked tweaking code.

After FORTRAN, I took more computer courses and graduated in 1975 with a dual major in computer science and psychology. Most of my computer sciences courses were programming courses. To me programming was much more satisfying than, say, proving theorems in Calculus, where I had to get the "right answer" at first crack.

In addition to writing in FORTRAN, I wrote in PL/I, 360 Assembly Language, and COBOL (all on punch cards). I also wrote some BASIC on the PDP-10 timesharing machine where access was limited. I also completed a senior practicum where I completed a programming project written in COBOL. Oh yeah, I took some courses in computing algorithms (we read Knuth) and logic.

I landed my first professional programming job as an analyst programmer II (I had a CS degree!) for the City of Vancouver, Washington. I introduced our five person COBOL programming shop to the structured programming techniques I had learned at school, and the use of perform statements, by creating a program that executed the story of Goldilocks and the Three Bears. I spent 3 short months writing thousands of lines of water and sewage billing code in COBOL (no one else read or tested my programs—I was given hand written specifications which I took as a starting point). I penciled in my programs on IBM COBOL coding forms and gave them to our keypunch operator to type up.

In January 1976, seeking more challenges, I took a Software Evaluation Engineer job at Tektronix. For the next year I tested a variety of graphics libraries, terminals, and an early graphics workstation. I read as much of the code I tested as I could and found several bugs simply by reading code. I also wrote test specs and test code.

Wanting to do more significant programming, I applied for a job as a Software Engineer I in the newly formed Microprocessor Development Products division in 1977. (At that time, Software Evaulation Engineers were a pay grade lower than Software Engineers). Over the next year I designed and wrote a download program in 2650 assembly and a universal linking loader in PL/M. I was the lone engineer on the linking loader.

Knowing nothing about linkers or loaders, I researched how linkers worked before designing the features to support different microprocessor instructions sets and architecture in a single program. Based on descriptive meta-data, my linking loader would allocate memory, resolve address references, and rewrite the binary file according to the "rules" specified in file headers. The linking loader could even process multiple files for different processors into a single relocated binary file (why we thought people would want this capability still puzzles me).

While I designed and programmed this universal linking loader, a small team developed a universal macro assembler. We also developed a common utility library that was shared between the assembler and linker. This was a fast-paced product development effort for Tektronix. We worked on a very tight schedule transforming an acquired product into a high quality Tektronix product line, turning out multiple product releases in a little over a year.

Tektronix was an engineering company where the "next bench" philosophy was practiced—build something of high quality that your peers would use. My colleagues were, for the most part, a highly motivated bunch of new hire recent grads. Our team also included one or two more experienced programmers and a couple of engineers without college degrees. Tektronix had lots of internal process for designing and engineering hardware products but (at least in this part of the company) very little experience with software. So, most of the processes and various design documentation requirements we used were imposed on us software developers with very little adaptation.

Over the next few years at Tektronix, I held engineering positions with increasing responsibility. I was lead developer for a networked file server (we basically wrote our own operating system, drivers, and file management system code from scratch) in Pascal. In 1981, I led a fast-track firmware development effort for a low-cost graphics terminal (written

in PL/M). We wrote functional specs and created structured design documents. We incrementally developed and tested our code, and held frequent code reviews. This all was just standard practice.

Up until this point in my career I had briefly written billing software (my COBOL programming stint) and embedded systems software.

And then, along came Smalltalk which was the main programming language for the Tektronix 4404 Workstation. I joined the AI Machines Product group right after it was formed as a Principal Software Engineer, but shortly was recruited to manage the software team until the first product introduction (a little over a year—again, this was a fast-track product development jointly with Tek Labs folks). My engineering management position didn't afford me much time to program in Smalltalk, yet I had a keen interest in grokking it. I wanted to become a Smalltalk developer and engineer products using Smalltalk.

Smalltalk was unlike any programming environment I'd encountered: immersive, graphical, and utterly alien. Its syntax was simple, but the terminology for Smalltalk programming concepts were foreign. I had to translate Smalltalk terms to roughly equivalent programming constructs familiar to me before I could even start to understand what people were talking about.

Objects were instances of classes. Classes defined an object's methods and variables. Classes and objects didn't have any analogs I could draw upon. However, methods seemed a lot like procedures. Objects were instantiated, that is allocated. Objects respond to messages. Messages seemed roughly equivalent to function calls.

To write a Smalltalk application (whatever that meant), I typically wrote statements in a workspace. I then highlighted that code and executed it. Were those statements then my program?

I didn't need to create my own classes; I could do a lot using the classes that came with the Smalltalk environment. I could freely add methods to existing classes too (although that was frowned upon by those with more Smalltalk programming experience).

I still had several questions: What exactly was a Smalltalk application? What were good ways to think about structuring programs? When should I define my own classes and when should I use existing ones? Reasonable answers to these questions may seem obvious to today's object-oriented programmer, but they weren't so settled in those days.

Most of the early Smalltalk programmers worked in Tek Labs. They were tasked with demonstrating potential applications of Smalltalk in Tektronix products. I looked to those folks for good programming practices. How did effective Smalltalkers think? My first Smalltalk code was an ugly memory hog. Not knowing better, I extensively used blocks (closures) to define font families that could be lazily loaded from external files. Fortunately, a fellow engineer read my code and offered helpful suggestions for restructuring it to be more understandable and performant.

Over the next several years programming in Smalltalk, I came to value carefully designed class libraries. I rejoined the engineering team and led Tektronix' Color Smalltalk product development. My design ideas and code were always reviewed. Since our Smalltalk code was visible to other programmers, we wanted it to be exemplary.

I came to appreciate the effort it takes to design frameworks (for color representations, graphic hardware memory, color matching, color graphics, use of color in Smalltalk windows, etc.). Developing libraries for others' use demands attention to detail, code comprehensibility, design coherence, and utility. But people won't know how to use your libraries unless you present a coherent model of how your classes and objects work together and demonstrate how to use them. Working code isn't sufficient.

In retrospect, my Smalltalk experiences had a huge impact on my software development and the trajectory of the rest of my software development career. Through my Smalltalk engineering experiences, I learned not only how to create extensible code and frameworks, but also how to communicate the important aspects of a software design and

the object design process. As I grew to understand how productive Smalltalkers thought, I realized that there were different schools of thought about how to approach object-oriented design.

I found that initially thinking in terms of the roles objects play—their "responsibilities" if you will—and how they interact led to significantly different design solutions than if you focused first on how objects responded to events, or what states they were in, or their internal data structure. This led to my inventing responsibility-driven design, teaching numerous design courses, and writing a book about object design. Over the next ten years, I refined my object design approaches as I refined my perceptions of object-oriented software development. I saw different patterns of object interactions and distributions of behaviors (this led to my idea of object role stereotypes and control styles, and a new object design book). I enjoyed figuring out (perceiving) patterns of good designs and software development practices I discovered in my work and colleagues, and then communicating them to broader audiences.

As a software design and architecture consultant I have worked on several multi-year projects employing various technologies. I have worked with developers, business analysts, domain experts, architects, engineers, and managers. I could always dive into programming. I felt at home working with software engineers. But I didn't want to be "slotted" into only the role of object design guru. I became more comfortable with being uncomfortable in those new software environments and problem domains I encountered. My past experiences had taught me that I was capable of learning, and adapting and adjusting my umwelt, even though it wasn't easy or comfortable. I came to realize that my outsider's perspective, was an advantage. Along with my design values and practices, I could offer unique and valuable insights. I became adept moving around between and among software development roles, adding value where needed. I could zoom out to establish the bigger picture of the overall system architecture, or dive into details and tweak the software design, processes, techniques, or practices. I have spent nearly twenty years writing patterns about all aspects of software development.

Throughout my consulting career, I enjoyed "shifting" between different perspectives on complex systems while learning about new technologies and problem domains and details. I enjoyed working with people with diverse backgrounds and interests. I found myself (still do) gravitating towards helping technology leaders explore design innovations, discovering new ways to think about software design and modeling, and experimenting with effective ways for others to communicate their design practices and personal design heuristics.

A.2 Allen Wirfs-Brock's Umwelt Narrative

(In the narrative below I have used italics to highlight what consider to be the key elements of my personal software development umwelt.)

I first learned to program in the mid-1960s at a two week summer course between the 8th and 9th grades. I learned to write machine language programs for a Burroughs 205, a mid-50s vintage room-size vacuum tube decimal computer whose main memory was a magnetic drum. It used punch cards and a line-printer for I/O.

I learned some fundamental concepts of computing during those weeks:

- All computers can do is carry out commands to store, retrieve, and simply manipulate numbers.
- Any information—including a computer's commands—can be encoded as grouping of numbers.
- Programming is all about identifying the informational part of a problem, choosing an encoding, and writing sequences of commands that manipulate the encoded information.
- Most programs involve loops.
- You will spend a lot of time debugging your programs.

I still apply these concepts—in more sophisticated forms—when I program. They became fundamental tenets of my software development umwelt.

Why did these early concepts have such an impact? Was it because early experiences are so sticky or because there is something about the "early days" of software? It's probably both. I think that early experiences were very important in forming my programming umwelt. And that in those days the world of software was much smaller and less complex so it was easier to lock on to fundamental concepts.

In grade school, I had been "good at math," interested in all areas of science, and an avid reader of imaginative science fiction. I had many of the available science toys—chemistry sets, electronics kits, etc. but I had found them frustrating. They mostly provided recipes for experiments using the provided components. But, they didn't teach me how to come up with recipes for my own science experiments. Programming satisfied that itch. *Software was to be my creative media.*

After that first course I no longer had access to the venerable Burroughs 205, but I did have access to the public library where I found books about a wide variety of computers and programming languages. I also discovered that I could walk into the local IBM field office and order complete manual sets for any IBM computer and they would be shipped to my home—for free. I spent the next couple years reading manuals and writing programs on paper for desk execution. *I learned that programming was an intellectual activity that I could do anywhere—even if I didn't have access to a physical computer.*

I eventually found ways to get some limited computer access to actually run programs. I used Dartmouth Basic on an early commercial timesharing service. I also had some access to a Digital Equipment PDP-10 timesharing system and an IBM system that ran an interactive version of PL/I. Through the local science museum I got free-range access to a PDP-8 minicomputer. This was my first experience with a small single-user computer. I wrote lots of small programs (there are no large programs on a 4K main memory machine) in assembly languages.

At this same time, integrated circuit technology was revolutionizing computer hardware. New and more powerful, yet less expensive, computers were announced in the trade press almost weekly. Yet this was still the era where each computer brand and model had its own unique instruction set architecture and typically its own unique control program/operating system and programming languages (or at least unique dialects). I found myself becoming a connoisseur, from a programming perspective, of computing systems. This also became deeply ingrained in my computing umwelt: *I value technological diversity. I expect constant evolutionary change. I relish unexpected technical innovations.*

After high school I found a variety of programming jobs and eventually was credential motivated to get a CS degree. I was exposed to a variety of application domains and styles of programs. At that time a common division of programming labor was between applications programming and systems programming. *I soon realized that I was a systems programmer*—someone that builds "pieces of infrastructure that integrate multiple interacting parts, and sit underneath application code" [3]. Systems programming encompasses operating systems, programming languages, networking, databases, graphics libraries, and much more. I did work in a number of these areas but through chance and interest I ultimately specialized in programming language design, implementation, and tooling.

Over the first decade of my programming life I found that there were characteristics for software systems and tools that I liked or disliked. I liked the "Do What I Mean" command style of the PDP-10 operating system with its intuitive understandable command names and contextual defaults. If I type "run foo" it would execute "foo.exe" if it existed. Otherwise it would figure out what programming language "foo" was written in, compile and/or link it if necessary, and then execute it. I disliked systems with obscure notations or that required me to keep track of details and repetitively tell the system things it should already know.

This developed into a personal design aesthetic that was incorporated into my umwelt: *Software is created by humans and exists to support humans and human activities. The design and implementation of software should be approachable, understandable, consistent, and comfortable to the humans that will interact with it.* This aesthetic relates to both the internal design of software and to the design of software interfaces, both human interfaces and the interfaces between software components.

The mid-1970's saw the rapid emergence of what at first was known as "home computers." At first these looked and functioned like scaled down mini-computers. The idea of having my own computer at home was exciting but what exactly would I do with it? I routinely programmed much more powerful computers at work. Maybe all I really needed at home was a good terminal? I got my answer when in March of 1978 I attended "The SECOND West Coast Computer Faire" and attended a banquet presentation by Alan Kay titled "Don't Settle for Anything Less." Kay explained the concept of a personal computer and his vision of the Dynabook. He showed filmed examples of early versions of Smalltalk and other systems and applications developed at Xerox PARC. *It opened my eyes to the possibility of creating radical software that benefited individuals rather than organizations.*

Kay's vision aligned with my software design aesthetic but I was a systems programmer, how could I contribute? I transferred into my employeer's advanced development "labs" in a role that today might be called a Research Software Engineer. Initially I did so by developing a compiler for the next generation of microprocessors that would be used in future personal computers and prototyping some early IDE prototypes. But a couple years later I had the opportunity to try to make Smalltalk practical for use on affordable personal computers [2]. At first this seemed like an iffy possibility. Smalltalk had been developed running on personal supercomputers not on commodity microprocessors. After several false starts I was successful but it had required approaching various aspects of the implementation differently than the original PARC developers. *This taught me that lateral thinking was essential for software innovation.*

These events all occurred fairly early in my long software career. I subsequently worked on many projects in various senior technology roles—principle engineer, software architect, founder, CTO, contributor to important language standards, HoPL paper author, etc. Some roles were entrepreneurial and some were in large organization. They were all innovation focused. My choice of projects and the technical contribution I achieved were all greatly influenced by the values and behaviors—my umwelt—that I developed over those early years.

A.3 Jordan Wirfs-Brock's Umwelt Narrative

When I was a child–I can't tell you how old, exactly–my parents would give me hypothetical object-design problems that I would solve by drawing CRC (class-responsibility-collaborator) cards with a number two pencil. I cannot disentangle the memory of this event from the stories my parents told me about it: I imagine that I drew simple pictures to go along with the names of the classes and their responsibilities, and that I would discuss the solutions with my parents, who would weave in software design nuances so subtly that I didn't notice, but I have no idea if any of this is true.

I also remember creating interactive stories using Hypercard in a summer camp, and my mom showing me the line in the acknowledgements of her first book where my name appeared. These memory fragments tell me that software design has been present in my life since my earliest days. Thus, my software umwelt is tightly coupled with my familial umwelt, and is a matter of legacy and identity. Early in my life, when I was rebellious, it resulted in aversion; later, as I value family legacy more, it has resulted in pride and curiosity. I have brought these emotional orientations to how I perceive coding (which has been productive and counterproductive at times).

But as I got older, I left programming behind and focused on other interests: hiking, geology, creative writing. As an undergraduate student, I wanted to put intellectual and geographic distance between myself and my parents, so I moved

some 2,500 miles away and bemoaned the required programming classes I had to take in my aerospace engineering degree—until I realized that I enjoyed them vastly more than studying fluids and thermodynamics. Still, computer programming felt cryptic, something that strayed just beyond the edge of my comprehension. I could scrape together passing assignments with a bunch of help, but they never truly made sense. It wasn't until several years later, after countless self-paced courses, guided workshops, and crappy half-finished personal projects, that things finally clicked. I could finally write code–crappy code–that could do what I wanted. One of the experiences where I felt like I truly enjoyed coding was an online course that I took many years after I graduated from college. The assignments felt like interesting puzzles to solve, rather than stressful problem sets that I needed to get done on a deadline. Also, my partner and I took the course at the same time, and even though we didn't work together, we enjoyed comparing our approaches and discussing the assignments together. The fun, social, challenging, problem-solving nature of coding when I was a kid with Hypercard was back.

Writing software requires a type of thinking that we don't often encounter in other dimensions of our lives—this makes it both immensely frustrating and immensely rewarding, and means that it will "click" for some people and seem strange for others. There is no value judgment here, and we need to empathize with how other people relate to programming. There are many ways to program, and embracing them, rather than alienating them, will ultimately lead to broader and deeper perspectives and ways of doing things.

After getting a master's degree in journalism, focusing on how to use data to answer questions that matter to people, I found myself working for a foundation in the nonprofit sector. There, in the first few weeks on the job, I helped write a grant for a civic technology software project: A community data storytelling platform. We ended up getting the grant, and assembled a small, scrappy but mighty team of contract workers that included a project manager, a software engineer, and a UI/UX designer. In one of our first team meetings, the software engineer asked, "Who is the product manager/product owner?" I didn't even know what that meant, but I somehow fell into that role. I found myself liaising between internal and external stakeholders, as well as managing our development team and shaping the vision for what the product should ultimately be.

Thankfully, that software engineer was one of the most patient, generous people I have ever worked with. He taught me what an agile development process was and how to run one. He taught me how to write user stories and bug reports, and explained unit tests and integration tests to me–all without a hint of condescension or frustration with my lack of experience. I didn't write a single line of code, yet I felt like I understood the software thoroughly. Through this process, I learned that things that seem simple from the outside as a user (creating an account, resetting a password) can require dozens of decisions. I don't know how I didn't buckle under the decision fatigue.

To program, you need to learn how to understand a human-constructed formal system and operate within it. As software as a field developed, these systems have gotten more complex, and are now so complex that no one person can understand them. A similar progression has happened with projects. Complex software projects require collaboration and teams. Thus, part of my software umwelt is always looking for the system, from a standpoint of trying to figure out what other people and tools are required to navigate the system.

Astonishingly, in 1.5 years we delivered a complex project on time and on budget. And we did so because we focused on consistent, incremental software development: long-live the daily grind. And, we delivered what we said we would deliver. At launch, our platform was beautiful, easy to use, and well-built. But, those things did not constitute success because we delivered something that people didn't actually want. In hindsight, I now know that we should have paused to do more exploratory research on the needs of our community members and stakeholders. That is something we

certainly could have done; but at the time I didn't know to ask for it. (Years later, I would end up going back to graduate school to get a PhD in Information Science with an HCI emphasis, focusing on design research methods.)

I perceive software development as a sociotechnical system. Building software for the sake of building software can be a rewarding and stimulating intellectual exercise. But the real power of software is that we can make things that people use—technologies that change the world. This comes with a great burden: How do we know that we are making something people want? That they need? That doesn't harm them? Software is made by people for people, and it has complex implications for people and society. It may be difficult to perceive these as we are in the weeds of creating software, but we have to try. We have to pull back and see the bigger picture. This is hard, but it is the primary focus of my umwelt. [And I fully acknowledge that other people's umwelts may have a different focus – that is fine ;)]

Throughout this time, the other major part of my job was doing data analysis and visualization for local non-profit organizations. The main tools I used were MS Access, ArcGIS, Excel, and Adobe Illustrator, but I took small forays into coding: Looking under the hood at the SQL coding in Access when I get frustrated with the visual interface; dabbling with d3.js to make some interactive visualizations.

After this, I worked as a data journalist for several years, trying to make data about the energy industry—from household energy consumption, to the cost of cleaning up abandoned oil and gas wells, to the rapidly dropping cost of solar power—approachable for general audiences. During this time, I began dabbling with Python Jupyter Notebooks. For the first time in my life, I loved writing code. I loved being able to tweak a single line of code and see the results immediately in a data frame or data visualization. I felt like I had agency and power, and like I understood what I was doing. Yes, I still faced extreme frustration, but it felt worth it because I could see what I was working toward.

As a data journalist working in radio, I often got frustrated with saying, "Check out our website to see visualizations of the data in this story." Seeking out a way to bring data to life in the radio, I discovered techniques for communicating data with sound. This is a field called data sonification—but when I was first discovering it, I had no idea that this term existed, which made it nearly impossible to search for and find resources. Luckily, at a conference I learned about a Python package another journalist, Michael Corey, had created to turn time series data into MIDI notes by mapping quantity onto pitch, called MIDITime. I knew enough Python that I could remix and extend the code to make my own data sonifications. Here, code was a means to creative expression. I also thought it was so cool that someone had made a creative tool that other people could use, and aspired to do the same.

Over the next several years, I would create many more data sonifications using all kinds of tools: programming languages (Sonic Pi became my favorite, but I also played with Supercollider and Pure Data) no-code tools (Adobe Audition, TwoTone, DataSonifyer), even developing techniques to make "lo fi" sonifications using my voice and body. Unlike most sonification practitioners, who pick a tool and learn it deeply, I preferred to dabble. I found the range of tools themselves, their quirks and nuances, to be just as interesting as the sonifications one could produce with them. I also saw coding as a practice that integrated into a constellation of other ways of creating with data and computation—including algorithmic expression that did not require a computer at all. Software is a creative material, and the tools you use to work with it matter immensely. Finding the right tools can open up your creative capacities.

I loved being a data journalist: In this role, I could combine my love of storytelling with my love of data to find insight and meaning. And I was constantly learning new coding and technology tools to help me do my work, which I found immensely satisfying. Yet I sensed something was missing: Our team spent all of our efforts on reporting and communicating stories, yet we spent almost no time understanding our audiences. What issues did they care about? Were they understanding the complex energy topics we tried to explain? So I left journalism to pursue a PhD

in Information Science so that I could learn how to understand people and how they interact with information and technology.

While earning my PhD, I fell in love with a particular kind of research called research through design. In the way I do this kind of research, this means that I want to understand how people interact with technologies that don't yet exist—so I create situations that help us explore these hypothetical technology futures. Sometimes, it means creating design fictions—images, sounds, pictures, stories—that help people imagine the consequences of technology. Sometimes, this means asking people to imagine future technology with me, through hands-on activities. Sometimes, this means developing new technological artifacts, often prototypes of varying levels of fidelity and functionality. In this kind of work, I treat coding as a means to express ideas in ways that are tangible for other people to experience. I am more concerned with what coding helps me achieve, than with what the code looks like.

Fast forward through many more half-finished projects, and thousands of lines of clunky code that is just good enough to get the job done, and I somehow now find myself a Computer Science professor. This is a strange identity for me, as I don't feel fully at home in a CS department. Part of this is because my field, human-computer interaction, is often viewed as "soft" CS, a fringe activity that isn't core to computing. I disagree vehemently, and instead think that without human-computer interaction, the rest of computing is meaningless.

In this role, I was tasked with teaching a Software Design class to juniors and seniors who have taken more formal CS classes than I have. I brought a lot of anxiety to teaching this class: I am not a great programmer! I haven't touched Java in 20 years! To develop this course, I leaned heavily on my parents and their combined 100-or-so years of software development experience. As I built the syllabus, planned the assignments, and crafted the lectures for each week, I spent countless hours on the phone and in person with my parents. I enjoyed learning from them, and I discovered software design as a discipline is really cool! I am blown away by their wisdom and expertise. And yes, I still got quite frustrated at times. But there were also times when I discovered I knew way more than I realized (like when trying an activity where I was refactoring "ugly" code). Yes, there are areas where my expertise is lacking (software design patterns), but I encountered other areas where I had deep expertise (identifying and analyzing requirements, product design).

I still feel like much of my software design knowledge is theoretical rather than procedural (that is, I can tell you about it, but I have trouble doing it), but I have reached a point where I am excited to learn more, and to teach a Software Design class again. I wish I could learn without the pressure of having to teach something that I don't feel confident about. But then again, without that pressure, would I have learned as much? Definitely not.

I still have complex feelings around software. I constantly feel like there are additional skills I need to learn to be able to truly call myself a programmer. (This will never end.) I have memories that cause me to doubt myself—like the times I was told that I wasn't "technical" enough, or the time a developer colleague reacted in shock when he saw that I used the default terminal settings. I wish I had embraced software development earlier in life, but better late than never. And I have come to value my outsider/beginner perspective, and treasure it while I still have it: It helps me be a better teacher.

References

[1] James Bridle. 2022. *Ways of being: Animals, plants, machines: The search for a planetary intelligence.* Penguin UK.

[2] Patrick J. Caudill and Allen Wirfs-Brock. 1986. A third generation Smalltalk-80 implementation. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '86)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/28697.28709

[3] Stephen Kell. 2017. Some were meant for C: the endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for

Computing Machinery, New York, NY, USA, 229–245. https://doi.org/10.1145/3133850.3133867

[4] Philippe Kruchten. 2008. The Biological Half-Life of Software Engineering Ideas. *IEEE Software* 25, 5 (2008), 10–11. https://doi.org/10.1109/MS.2008.127

[5] Fritz Machlup. 1962. *The Production and Distribution of Knowledge in the United States.* Number v. 10 in Princeton paperbacks. Princeton University Press. https://books.google.com/books?id=kp6vswpmpjoC

[6] Agustín Ostachuk. 2019. The organism and its Umwelt: a counterpoint between the theories of Uexküll, Goldstein, and Canguilhem. In *Jakob von Uexküll and Philosophy.* Routledge, 158–171.

[7] Tomas Petricek. [n. d.]. *Cultures of Programming.* Cambridge University Press. to appear.

[8] Donald A. Shön. 1983. *The Reflective Practitioner: How Professionals Think In Action.* Basic Books, New York.

[9] Rebecca Wirfs-Brock. 2023. Observations on Growing a Software Design Umwelt. In *Proceedings of the 29th Conference on Pattern Languages of Programs* (Virtual Event) *(PLoP '22).* The Hillside Group, USA, Article 13, 10 pages.

revised 2024-08-31