7 What Does Live Coding Know?

Live coding can be considered a specifically technical practice, arising from, engaged with, and resulting in technological tools, infrastructure, and ways of thinking. In these terms, live coding is fundamentally reliant on a certain level of technical knowledge. However, live coding is not only a technical practice—as a creative endeavor it both draws on and expands the fields of knowledge and ways of thinking within craft, design, and artistic processes, practices, and research. In asking "What does live coding know?," we invoke different possibilities of the meaning of *technē*—referring to technology, to the art of making or doing, to productive knowledge, and even to rhetorical strategy.[1] Accordingly, this chapter explores the engineering contexts in which the necessary technical knowledge has been generated and applied and the contexts of creativity in which know-how from several perspectives crosses the boundaries of computer science, craft practices, and artistic research. The intention is to explore beyond the knowledge needed to practice live coding, extending to the knowledge that is acquired or that emerges in and through that practice.

The interdisciplinary nature of live coding—or rather its transdisciplinary (even *un*disciplinary) character, emerging as it does between the lines of different scientific and artistic disciplines—requires that the question of what it *knows* must be approached from more than one epistemological perspective. As with the discussion on the liveness of live coding in chapter 5 and on the temporal, temporalizing, and time-critical dimension of live coding in chapter 6, it becomes clear that the question of what live coding knows is a highly complex and layered issue that does not sit comfortably within any singular theoretical framework or school of thought. Live coding forces different epistemological registers into proximity, where they become negotiated in and through its specific practice. This chapter attempts to create conversations between the different ways of thinking and knowing operative within live coding to ask: What forms of knowledge are prerequisites for live coding? What forms of knowledge are produced—or even cultivated—in and through live coding (including technical knowledge as well

as the human capacities of agile thinking and resourceful tact)? What does live coding know; moreover, how can this be shared? What is the epistemological contribution of live coding within a wider interdisciplinary discourse, whether in relation to computer engineering, musical composition, or artistic performance? Demonstrating a multimodal model of sensemaking emerging between the lines of different disciplinary demarcations, how might live coding offer insight into the commonalities, complementarity, and differences between ways of knowing within the sciences and the arts? How might it register uncertainty in this respect, *not knowing*—along with a desire to know?[2] So, across this chapter, we move from engineering and craft perspectives to the epistemic insights of artistic research, opening up discussion on uncertainty and in turn the indeterminacy that defines the limits of computation and computational aesthetics.[3]

In asking "What does live coding know?," the intention is also to expand beyond an anthropocentric understanding of knowledge. Here, what live coding knows is not synonymous with what the live coder knows but rather refers to the epistemological potential of the practice itself, including the knowledges arising in and through the collaboration between human and machine sensemaking. The technologies worked with in the practice of live coding are not only a result or outcome of human technical knowledge but also enable and mediate epistemological advances and are constitutive of knowledge and meaning.[4] As such, and as with previous chapters, the question of knowledge within live coding necessarily engages with the complexity of human and more-than-human entanglement.[5]

Critical Technical Practice

Live coding comprises artistic and technical as well as philosophical inquiry—for not only does it draw on and from specific fields of knowledge but as a practice also asks specific epistemological questions. Here, we draw again on Agre's notion of *critical technical practice* to demonstrate how critical and cultural theories can intervene in engineering practices.[6] Agre was concerned specifically with the conduct of artificial intelligence (AI) research and with the relationship between AI as a critical pursuit and its engineering methods. In his notes on trying to reform AI, he observes that it is necessary to be competent in both technological and critical discourses in order to understand the new kinds of knowledge being produced since these span the conventional separation of engineering and philosophy.[7]

Live coding also leads us to liminal regions in the conventional separation of engineering and philosophy, as well as art. Might we consider speculative AI experiments as

a kind of art-music within the software industry? The Turing test, the singularity, and epic human-machine contests in chess and Go are performances of virtuosic imagination only tenuously related to the mundane statistical operation of search engines and data mining. At the same time, performance notations, most familiarly music notation, should be understood as a kind of software that can be instantiated (performed) in multiple contexts and involves a theory that the maker brings to bear—exploratory questions, concepts of how the hearer will receive it, and a kind of autonomy or agency that results from its own internal logic.

Engineering Know-How

To address the question of *what* (and perhaps even *how*) live coding knows, it is useful to identify some historical and contextual coordinates, examining how live coding both extends (and also deviates) from earlier practices and conceptions of coding and programming. In one sense, this mapping of the historical coordinates that enable the emergence of live coding can be seen in relation to chapter 2. However, while the emphasis of that chapter is on the specific living narrative of live coding's many histories (the evolution of its specific practices, networks, and communities), the focus here is more on a wider sense of technical prerequisites, conditions, and epistemological shifts that have enabled creative, experimental practices such as live coding to emerge.

The scientific-industrial origins of coding since the 1940s seem far away from the notion of coding as craft,[8] as a creative or even artistic, performance-based practice. For the first half century of the computer industry, most coding was embedded within highly structured business and organizational environments. Computers and computer time were expensive, as was the labor of experienced programmers. Far from being "live," much effort was devoted to constraining and controlling the work that programmers did in order to ensure that their time was productively applied and that the results justified the investment. As a consequence, most practices of software engineering, in both technical and commercial applications, became formal processes of bureaucratic translation through which contractual specifications were systematically converted into program code. Analogies to industrial processes and mass production resulted in project management structures such as the structured systems analysis and design method (SSADM), in which business analysts or system designers rather than programmers were responsible for the creative work of invention, to be embodied in plans and specification documents.[9]

Within these industrial and bureaucratic environments, the conversion of business process inventions into mundane code eventually became the job of relatively

low-skilled labor, minimizing the time commitments from technical specialists to an extent that the role of programmers could be recast by analogy to construction site or factory workers expected faithfully to follow the instructions of software "architects" or "designers."[10] Within such environments, the suggestion that software development could be a craft process, with autonomy for the programmers, was vigorously opposed. In constructing an expensive and apparently fragile advanced technology, the expertise of the software project manager rested in understanding, predicting, and repeatedly controlling as many aspects of the process as possible.[11] In an industrial-bureaucratic context, the contingency, uncertainty, and variability of craft activity (qualities that are often positively associated with live coding) were associated with amateurism and inefficient preindustrial modes of organizing production and thus were to be eradicated wherever possible rather than nurtured.

Two critical factors have come together to radically change this situation in recent decades, opening up possibilities for coding as an experimental practice. The first was the steadily falling cost of hobbyist and educational computer hardware, to an extent that the capital resources necessary for software development now appear to be approaching zero. In contrast to the small number of wealthy geeks who could afford to build or buy their own home computers in the 1970s, almost all adults in developed nations now carry computers (mobile phones) in their pockets, and children and hobbyists can acquire low-cost (practically disposable) computers, such as the Raspberry Pi Zero, that are perfectly adequate for creative experimentation.[12] Access to technical hardware—that had been available only within certain industries or for a privileged few—has now become widespread. The tools and technologies needed for coding practices are now more democratically obtainable.

The second factor in opening up the potential of coding as an experimental, creative practice is a transformed understanding of the practice of programming, which has developed alongside these democratized technical resources. The context for this transformation was a vision of enhancing human experience, set within a characteristically West Coast 1960s counterculture,[13] that led to systems in which the mode of operation was understood to be dynamic experimentation, rather than rigidly controlled and defined behaviors. Two projects in California were in the vanguard of changing attitudes toward programming—the Augmentation Research Center, founded by electrical engineer Douglas Engelbart at Stanford in the 1960s for developing and experimenting with new tools and techniques for collaboration and information processing, and computer scientist Alan Kay's KiddiComp concept (1968) and DynaBook proposal (1972), in conjunction with the agenda of Xerox PARC. Kay and his colleagues sought to produce a "personal computer for children of all ages."[14]

Kay's Smalltalk language for the Dynabook—the software component of his research proposal—in particular set out to offer capabilities through which children could create and modify their own artistic tools. The Smalltalk programmer goes about their work by literally *changing* the language, editors, and operating system as they work, inspecting and modifying parts of the running code, and experimenting with new features or alternatives by replacing them within the running system. Of course, Kay's ambition to deliver these technical capabilities to children was particularly challenging, and Smalltalk became far more influential on adults than children.[15] Nevertheless, the desire to make such capabilities accessible to relatively untrained individuals resulted in many of the graphical user interface (GUI) innovations that we take for granted in software tools today, including dialog windows and the earliest uses of the icon.[16] As mentioned in chapter 4, Kay's more far-reaching aims continue to inspire those looking to meet his vision, such as the global ambitions of the One Laptop Per Child Foundation, or the more recent DynamicLand Foundation, which aims to "enable universal literacy in a humane computational medium."[17]

At the time that these early personal computers were being developed, the distinction between *programming* a computer and *using* a computer was not quite as clear-cut as it is today—indeed, the GUI was interpreted by at least one influential researcher at the time as being a "step beyond" programming.[18] But the surface appearance of the Macintosh and other GUI products, relinquishing the command line that had been familiar to small business and home users until then, became associated with restricted technical capabilities and exclusion from the power of code.[19] Meanwhile, although Smalltalk had been created specifically with an agenda of creative empowerment, Xerox did not itself recognize the market opportunity that might arise from live programming in their machines.[20] On the contrary, Xerox was the archetypal enabler of bureaucracy, supplying the photocopiers by which all those other documents—memos, contracts, specifications, and forms—would be duplicated to support formal and professionalized software development. It would be some time (almost three decades) before live coding practices emerged in the early twenty-first century to truly seize the opportunities enabled by the increased accessibility and availability of hardware, alongside the experimental potential for creating and modifying programming languages.[21] Indeed, it is interesting that live coding eschews the restricted technical capabilities and exclusion from code (associated with the development of Macintosh and other GUI products) by often explicitly retaining rather than relinquishing the command line.

Companies such as Xerox could not be expected to recognize or support the ambitions of live coding. Nevertheless, the broader community associated with the Smalltalk language and its successors has continued to develop practices and concerns for

software development that are distinct from previously established software-engineering or business practices. These practices include the alternative characterization of design processes in the form of *agile* programming; the approach to documenting software in the wiki, and the characterization of ways of programming as a pattern language. Each of these reflects forms of liveness in software engineering and provides a further context from which to consider live coding.

Agile programming is a relatively live approach to professional software development work that treats the construction of the technical artifact as a performance of craft and is potentially undertaken in the presence of software users (so that they can direct or comment on the work in progress).[22] In the agile practice of *pair programming*, this performance becomes a duet, or perhaps pas de deux, in which a driver and navigator both sit at the same computer, jointly improvising the construction of a shared artifact. In agile software development, a full specification is not written in advance. Instead, the system evolves in a more emergent manner, with system users contributing stories about the way they would like the system to work and programmers responding in sprints to create new code to create the new capabilities that will enable those stories.

Wiki-editing software is another live practice that has arisen from the Smalltalk community.[23] Now best known as the basis for Wikipedia, the first wiki (or WikiWikiWeb) was another outcome of the agile development community.[24] In the Smalltalk language, everything is there to be changed. However, if other programmers or users are expected to use a constantly changing system, the documents describing the software must also be changed. In the agile development context, it is not only the software that is performatively created and shaped in front of the customer and users but also the software documentation, texts, or illustrations that accompany software or are embedded in the source code. This has to happen quickly—*wiki*, in the Hawaiian phrase that named the system.

Finally, the experience of live modification of software in the Smalltalk environment drew attention to the ways that different aspects or parts of a software system are more or less modifiable (or amenable to understanding by others after modifications have been made). The research community inventing these new practices drew on the work of architect Christopher Alexander, whose pattern language theory described the built environment in terms structured by the experiences of those who lived there.[25] The live development practices of the Smalltalk programmer inspired an analogy by which the Smalltalk environment became the place in which they lived—a virtual environment that might have pattern languages of its own. The resulting development of *design patterns* has become a fundamental of software-engineering practice and education, despite the fact that its original philosophical intentions have now become rather obscured.[26]

In addition to Smalltalk, its tools, and its descendents, practitioners of live programming also draw on the traditions of a far older language and associated tools—the Lisp environment. Although most computer programming tools through the 1960s and 1970s were shaped and constrained by the high costs of computer time, a small number of programmers worked in relatively wealthy AI research laboratories, such as that at MIT, where the Lisp language had been developed to exploit the luxury of direct access to powerful machines. As with Smalltalk (and Forth, as discussed in chapter 2), Lisp programmers are able to change aspects of the language itself, modifying the system through an interactive interpreter. The core of the language is the famous read-eval-print loop, often called REPL, in which the primary task of the system is to continually *read* the command the programmer has just typed, *evaluate* that command to change the system state, *print* out the result, and then simply (perhaps recursively) *loop* through those four steps indefinitely. As with Smalltalk, commands presented to the REPL can be used to define or redefine new functionality or even modify parts of the Lisp system (including the read, eval, or print statements themselves).

Just as the alternative styles of programming enabled by the Smalltalk environment led to distinctive craft and social practices, the earlier Lisp language was also associated with critical developments that ultimately set the conditions for the open-source software revolution. Lisp programs were conventionally distributed as source code, which could be loaded directly for evaluation to recover a previous system state—sometimes as sophisticated macro packages that transformed the nature and capabilities of the language itself. The Lisp philosophy formed the basis of the Emacs editor—an interactive programming environment largely written in Lisp and indeed often used for writing Lisp programs—whether for other research purposes or just for enhancing Emacs to become an email client, system command shell, directory utility, newsreader, or anything else that might occur to the user. This fundamental extensibility resembled the way Smalltalk programmers could modify the features of the editor itself, changing its behavior while using it. Because the REPL philosophy of Lisp assumed that the program continued to run as it was being modified, the programmer could work in the editor to change the editor—an everyday live performance in which every keystroke or Command key that the user might type was immediately translated to interpreted code, immediately useful, and also modifiable on the fly.

The author of the most popular version of Emacs, Richard Stallman, extended these live customization practices into a political philosophy, advocating free software that could always be modified by its users, with the intention that no software user need have constraints placed on them by another. The longer-term goal of Stallman and the Free Software Foundation was to produce a complete operating system based on free

software principles,[27] although stalled by slow progress on their equivalent to the *kernel* at the very core of Unix-style operating systems. This set the scene for Linus Torvalds to develop his Linux kernel and build a complete operating system using tools from Stallman's GNU (GNU's Not Unix!) project. While few GNU/Linux tools offer the degree of live extensibility that Emacs allowed, the principle of operation of the Unix shell, flexibly combining small tools via the Command line, itself amounts to an advanced and highly capable live coding environment.[28] The resulting free/open-source system software underpins much of the public infrastructure of the internet today, despite the fact that, as one might expect, few businesses or other institutions embrace the idea of the free and flexible modification of such crucial software to the extent that would have been typical when Stallman started his work as a researcher in the MIT AI Lab.

In the cases of both Lisp and Smalltalk, the flexible practices of research environments encouraged tools that are mutable and able to be reconfigured by creative engineers who engage in craft practices of modifying the tools they use for their own work. In both cases these languages have resulted in sets of social practices that reconfigure business and media contexts where they are applied—democratizing and decentralizing. Open-source commentator Eric Raymond has drawn a distinction between the traditional structures of software production and these live and open practices, calling them the "Cathedral and the Bazaar."[29] The software engineering research community itself, while traditionally aligned with the systematic and predictable processes of large-scale software production, has responded to changing industry practices of agile programming, design patterns, informal documentation, and open software by developing its own specialist series of LIVE workshops,[30] distinct from the art and media practices that are the main focus of this book but developed with an element of dialogue and overlap between the software-engineering and live coding communities.

Craft and Making

As stated at the outset of this chapter, live coding is a technical practice, a specific type of craft and approach to artistic process, practice, and research. Yet in what ways can live coding be considered a craft, a material practice? Indeed, the tension between the materiality and immateriality of code constantly challenges the ways in which we understand it. Code, as observed by architect and design scholar Daniel Cardoso-Llach,[31] often carries metaphorical associations of weightlessness. It appears mathematical, platonic, abstract, insubstantial—the stuff that dreams are made of. Codes encode information—they are ciphers, disembodied languages, rather than physical machinery. Code is pure knowledge, pure calculation, and pure control.

Yet code impinges on the material world—programs are also rule systems and orderings, specifying regularities over the everyday world of substance and phenomena. In this respect, live coding exists within a wider context of the software industry and indeed the whole enterprise of technology, whose purpose, it might be argued, is to bring order to the world. As an abstract mechanism of control, code has traditionally been an instrument of government, relieving the military-industrial complex from the uncertainty of human workers and replacing fallible or undisciplined human hands with the precisely replicable actions of machines. Moreover, although software is conceptually abstract, the computers in which these codes are constructed, executed, and observed are complex and costly assemblages often incorporating rare minerals and/or the products of sweatshop labor.[32] Their operation depends on a material infrastructure of communications, power networks, server farms, and processor clusters spanning the globe.

In chapter 8, we explore some of the tensions between, on one hand, the ways in which contemporary life is increasingly determined and constrained by the algorithmic rules of bureaucratic systems, commerce, and government, and, on the other, the *unruly* potential of live coding. Live coding cannot be dislocated from the wider enterprise of technology and the systems of control and exploitation therein. However, for live coders, code is not an instrument of control for imposing order on a chaotic world. It is quite the opposite: an activity that generates chaos in a highly technical world. Indeed, the practice of live coding could not be further from the military-industrial use of code for control of human life. Its use of code is creative rather than regulative.

In one sense, live coding as an artistic practice could be seen as another way of creating order from matter—perhaps inviting an analogy in terms of the way that the sculptor's chisel extracts form from stone or wood or the painter's brush arranges form from pigment. Yet this analogy soon collapses under the pressure of examination: Should one conceive of code as the chisel (tool) or as the wood (material)? Such analogous comparisons are complicated by the practice of live coding—no neat equivalent can be applied. Here, our own analysis elaborates on problematics that were already apparent to the authors of the TOPLAP live coding manifesto in 2004. The manifesto statement that "programs are instruments that can change themselves" highlights the problematically material mutability of the tool within live coding.[33] It goes on to claim that "live coding is not about tools. Algorithms are thoughts. Chainsaws are tools. That's why algorithms are sometimes harder to notice than chainsaws."[34] It is clear that live coding complicates the distinction between tool and material. Yet this complication does not invite resolution through any attempt to achieve new or definitive reinterpretations of *tool*, *material*, *form*, *practice*, or *craft* that could adequately capture the challenges posed by live coding to any of these terms. Live coding provokes reflection

on the potential of a new ontology of code by refusing to be contained by existing categories and definitions.

There is much to learn from this new ontology of code—both in relation to the nature of technology and in relation to the nature of material practice. This change is also occurring as those parts of the software industry concerned with user experience and interaction design have had their attention drawn away from the traditional office workstation, with its visual display screen and typewriter keyboard, to the many forms and contexts in which digital processors are now found.[35] Where software was once separate from the materiality of everyday life, sustaining a kind of technological mind-body dualism through the separation of the (middle-class) home and (white-collar) office or the traditionally remote and white-coated abstraction of the mainframe server room, we have now become thoroughly entangled with computers that are in our pockets, cars, and homes; embedded in our clothing, chest cavities, and skulls;[36] in our pets;[37] or attached to our wrists and faces. Interaction designers for this tangible, embodied, and embedded world of computation are thus reengaging with materials and craft practices in order to embody their interactive metal, wooden, or cloth prototypes. Once they lose their screens and keyboards, embedded computers become tangible user interfaces,[38] joining the rematerialized Internet of Things.

Rather than working with the "pure digital,"[39] these software and interaction designers can be conceived as craft makers, "reflective practitioners" (in design theorist Donald Schön's terms) engaged in creative design research work that draws attention to the resultant conversation with materials.[40] The user experience research literature delights in this turn to a new materiality because of the way that it offers insights from more established branches of design research.[41] Unsurprisingly, this research engagement with newfound material practices has also led to a concern with the materiality of code itself for those researchers who see code as a site of human interaction rather than simply a mathematical abstraction.[42] Not all writers take the analogy this far, but many interaction designers perceive their experiences with the software of their prototypes as having a great deal in common with their rediscovered experiences of hardware. They feel that, even when they turn from their workbench back to their laptop keyboard, they are still having a conversation with a material in which the code resists their intentions,[43] disrupting their pure theoretical conceptualizations via the "mangle of practice," to use sociologist and philosopher Andrew Pickering's phrase.[44]

Certainly, code often seems to be resistant to the intentions and desires of the coder—experienced, some argue, in much the same way as when physical materials resist craft labor. Engineering scientist Malcolm McCullogh's celebration of the "practiced digital

hand" describes a "master" user of computer-aided design tools engaged with coaxing reluctant or recalcitrant digital materials.[45] Gross, Bardzell, and Bardzell draw on painter and early computer artist Harold Cohen's theory of artistic media, from the perspective of interaction design,[46] to explain why this very recalcitrance becomes a media resource for performance and exhibition wherein audiences appreciate the virtuosity that has been exhibited in the struggle with a "viscous" medium. But if code is a material, it would appear to be a surprisingly immaterial one.[47] Interaction design theorists persist in the argument that software is material and that where there is a material, there must be a craft. Even social critic Richard Sennett describes open-source software development as a "public craft."[48] Programming is persistently described by programmers themselves as a craft skill, with practitioners writing manifestos for "software carpentry" or "software craftsmanship."[49] Yet the use of exclusive terms such as *craftsman* when describing craft expertise raises further challenges for the live coding community since both craft and coding traditions have privileged a particular gendered sense of skillfulness and agency, along with the hierarchical and gendered notion of *mastery*, which many live coders seek to resist and reject.

Just as the pure algorithmic code of theoretical computer science is actually embodied and embedded in the large material infrastructure of computer hardware and networks, so, too, are craft practices physically embodied in the craftsperson and socially embedded in communities of practice. One such long-standing community of practice is the demoscene, an antecedent of the live coding community that shares many common concerns with live coders.[50] Demoscene participants create virtuoso technical artworks, which they present to their peers in competition and performance events. Hansen, Nørgård, and Halskov undertook an ethnographic study of the demoscene community, from which they developed a theory of craft practice as observed among these code artists.[51] They see a relationship between the rhythmic elements of the realized artworks and the rhythmic practice of the artist tweaking and refining code. In their analysis this material practice results in a distinctive craft skill, molding the practitioner at the same time as the material.

An alternative conception of craft knowledge is offered by anthropologist Tim Ingold,[52] in which there is no repetition (only machines repeat mindlessly) but rather one step after another, along a journeying path. The craftsperson's tool seeks and responds to the grain of a material in a process of accommodation and understanding rather than imposing form on an inert substance. Ingold foregrounds the importance of a "conversation with materials" or even of the "correspondence" (to use his term) between sentient practitioner and active material.[53] He emphasizes an "art of inquiry"

in which the way of the craftsperson is "to allow knowledge to grow from the crucible of our practical and observational engagements with the beings and things around us."[54] Here, Ingold states:

> Every work is an experiment: not in the natural scientific sense of testing a preconceived hypothesis, or of engineering and confrontation between ideas "in the head" and facts "on the ground," but in the sense of prising an opening and following where it leads. You try things out and see what happens.[55]

Material should be considered as "matter-flow," in flux rather than stable, and the craftsperson *follows* the material, in a manner (Ingold argues) that Gilles Deleuze and Félix Guattari have described as *itineration*, rather than iteration.[56] The craftsperson is thus an itinerant wayfarer whose practice is one of journeying with the material. Ingold's work on what he calls textility provides one of the most productive perspectives in the contemporary discussion of materiality and offers an ideal analytic perspective for the live coding situation.[57] He applies social anthropologist Alfred Gell's work on *Art and Agency* in identifying a kind of mistaken belief in which an object is taken to be the starting point for an inquiry that traces backward from the object to find the conditions and creative agent that caused it to exist.[58] The object becomes a static index of a prior causal chain, rather than a thing unfolding through the interaction of a maker via the flows and forces of material.

The alternative process-oriented perspective of flow and unfolding is unfamiliar to many technologists but more than familiar to the contemporary artist, as expressed in artist Paul Klee's classic evocation of drawing as "taking a line for a walk,"[59] or indeed in La Monte Young's aforementioned "Draw a straight line and follow it."[60] The process-oriented sense of "taking a line for a walk" resonates equally well with the experience of the live coder, who is engaged in a process of programming with no intention to create a finalized software product. Ingold himself observes how different these material craft practices are from the conventional world of technology. He describes technology itself as being an ontological claim. The claim of technology is that things come into being through the application of rules and rational processes and that objects are thus formed out of inert and undifferentiated substances. Indeed, there are many computer scientists who resist the suggestion that computing might be a craft rather than an objectively mathematical science.[61] The tension is so long-standing that the mathematician Charles Babbage engaged in a long-running dispute with Joseph Clement, the engineer building his Difference Engine who claimed that he, rather than Babbage, should be recognized as its inventor.[62]

In subsequent generations, true computer science has often been advocated as a more appropriate domain for the refined academic rather than the mechanical engineer,

where the highest aspiration of theoretical computer science is to prove the correctness of its products in the manner of a mathematical theorem.[63] Indeed, foundational theoreticians of the discipline, such as Edsger Dijkstra, have publicly regretted the tendency for software development to be treated as a craft rather than an automated and repeatable scientific discipline.[64] There are many challenges in drawing the appropriate analogies between our traditional understanding of craft and materials and the experiences of making software, and these alternative and competing conceptions are acknowledged in the conflicting implications of the statements made in the TOPLAP manifesto.

Whatever the theoretical and idealized ambitions of computer scientists, the everyday professional practices of agile software development, like the creative practices of the live coder, seem far more fluid than a desire for rigorous formality might suggest. Agile developers respond to events rather than simply following a specification plan. Their practice—just as Lucy Suchman's research on situated cognition reflects[65]—demonstrates the contingency of rational action, in which the rational agent improvises and adapts to the world rather than imposing order on it. In practice, code seldom attains the mathematical standards that theoretical computer scientists aspire to. The practice of live coding, in which code is a process to be experienced rather than an intermediate specification accounting for an indexical product, is indeed a craft. Metallurgist and research physicist Ursula Franklin offers additional support to this view in recognizing that the world of technology includes such craft activities, as opposed to activities of mass production.[66]

While theorists of materiality in interaction design have argued that software is a design material like their other materials and that where there is a material there must be a craft,[67] a perspective from live coding follows those assertions in the reverse direction. Following the analyses of Ingold, Sennett, and Franklin, live coding *is* a craft—and given this craft, it seems that code must be its material despite the claims to immateriality made in its manifesto. Through code, it seems that we have made a linguistic tool into a material even though this material remains insubstantial.[68] Furthermore, the "conversation with materials" already observed in craft and design practice by theorists such as Sennett and Schön now becomes a more literal conversation through the medium of a programming language and thus composed of *linguistic* (or at least notational) exchanges. The regularities and explicit observability of code notations mean that we can more readily understand the patterns of experience inherent in such craft, reflecting on those experiences in the form of pattern language.[69] We can also appreciate a diversity of craft practices, extending beyond live coding to other communities of practice and other practices of programming.[70]

Creative Know-How and No-How

Within live coding, advances in technical and engineering knowledge meet with a sense of "knowing in practice" or even "practice of knowing,"[71] a practical form of "intelligent doing" or know-how perhaps more commonly (though certainly not exclusively) associated with craft-based practices and artistic practices.[72] The question of what live coding knows, or even how live coding thinks, draws the practice of live coding into closer proximity with the burgeoning field of artistic research. Indeed, artistic research offers a further framework to draw attention to the way that the practice of live coding is not fixed or foreclosed by a particular knowledge domain (such as computer science, electronic music or computer music, or even artistic practice). For some advocates of artistic research, its modes of thinking and knowing can be differentiated through emphasis on the nonpropositional or nonconceptual dimension of its sensemaking practices.[73] In parallel, attempts have been made to address the commonalities between artistic research and scientific research, drawing attention to reflexive practitioners operating at the boundaries between the arts, science, and technology, thereby unsettling some of the established assumptions of and about art and scientific research.[74]

For research scholar Robin Nelson, the multimodal epistemological model for research in and through artistic practice (or even arts praxis) involves a triangulation of *know-how* (comprising "insider close-up knowing"; experiential, haptic knowing; performative knowing, tacit knowledge, and embodied knowledge) and *know-what* (tacit knowledge made explicit through critical reflection), combined with the *know-that* of cognitive propositional knowledge.[75] His framing of know-how draws on Schön's (previously mentioned) work on the "reflective practitioner" and "knowing-in-action,"[76] philosopher and polymath Michael Polanyi's writing on the tacit dimension of knowledge,[77] and enactivist accounts of embodied knowledge (for example, the work of philosophers Francisco Varela and Alva Noë[78]), which for Nelson demonstrate that "cognition is not the representation of a pre-given world by a pre-given mind but it is rather the enactment of a world and a mind."[79]

Of specific interest for this publication is the way in which live coding operates at the threshold of different species of knowledge, troubling easy classification. Indeed, as scholars Dorothy Leonard and Sylvia Sensiper state:

> Knowledge exists on a spectrum. At one extreme, it is almost completely tacit, that is semi-conscious and unconscious knowledge held in people's heads and bodies. At the other end of the spectrum, knowledge is almost completely explicit or codified, structured and accessible to people other than individuals originating it. Most knowledge of course exists between these extremes.[80]

Live coding not only exists *between* these extremes but also demonstrates the coexistence, cooperation, and even complementarity between seemingly divergent knowledge paradigms.

Live coding is a distinctly hybrid practice, operating at a critical interstice between different disciplines and oscillating between a problem-solving approach (often associated with design-based practices) and a problem-finding, questioning,[81] even obstacle-generating tendency (often part of the research impetus for artistic inquiry).[82] Live coding brings computational knowledge into dialogue with those (alternative) forms of knowledge—tacit knowledge, sensuous knowledge modeled on experienced continuity of process rather than discontinuous abstraction; not knowing, the value of trial and error and of "feeling one's way"; and technē, conceived as a practical or even tactical knowledge underpinned by kairotic and mêtic intelligence as much as a craft[83]—that have been habitually eclipsed or even marginalized within a knowledge hierarchy that favors a form of abstract, rational logic.

Live coding pressures the *if-then* thinking of computational logic toward the *what-if* of speculative experimentation. Within live coding, neither the programmer nor program makes finite choices or decisions as such; rather, each becomes mutually part of an open-ended and contingent process of both problem generation and problem-solving. Live coding is conceived as a live practice for testing the possibilities of *this* or *this* or *this* or *this* for exploring the potential of what if. For philosophers Erin Manning and Brian Massumi, "Potential is not of the if-then. Potential is allied to what-if."[84] They assert that performing in the key of what-if "is never a question of formally working something out in advance" but rather "a movement precise with training but still open to regeneration."[85] This "see what happens" or "what-if" approach appears as a central modus operandi for many live coders.

Although many live coders acknowledge some formal training in computing, music, or artistic methods, the knowledge of the process required for live coding emerges often through experimentation, through the accumulation of trial and error, and through innumerable versions and iterations, tests, and attempts. The etymology of the word *experiment* refers to a species of practical knowledge based on experience, with origins in *experiri*, "to try, test"; from *ex-*, "out of," and from *peritus*, "experienced, tested." Experiential knowledge is gained through trial and error, through the process of doing and undoing, and through the repeated labor of trying something out again and again. However, theorist Dieter Mersch differentiates between the role of the experiment within art and science: "The scientific system is the medium *through* which experiments are first constituted. . . . In the arts, this relation is turned around; it is the *experiri* of

the *experimentum* that is the medium *through* which artistic research takes place."[86] He identifies the characteristics of "experimentality" within aesthetic research thus:

> First, it follows the original meaning of *empeiría*, "to open to risks" and "make visible" (*exponere*) that which binds and allows for ex-perience. . . . Second, self-referentiality or self-reflexivity is the main access to as well as pull of the "esoteric" of such a wealth of experience, so that it is at the same time an experimental experience and an experiencing (itself) in the experimental.[87]

Live coding is seemingly underpinned by these same principles: an openness to risk (the potential for the unpredictable, for chance and error related both to live coding's liveness as well as the loss of authorial agency through working with algorithmic processes) and to "making visible" the process of its own procedural unfolding, often involving self-reflexivity and self-criticality.

Live coding performances actively disclose to an audience their moments of not knowing, of trial and error, and of testing something out (through endless subtractive and additive procedures, the testing of constants and variables); moreover, the promise of not knowing (and the risk, uncertainty, and sometimes even messiness therein) is arguably part of live coding's improvisational performativity. The *showing of the screen* within many live coding performances makes visible not only the unfolding code—the liveness of the *working out*—but also has a capacity for showing a level of self-reflexivity through the inclusion of live annotation, commentary, notes, and marginalia (reflecting during the live performance) embedded within the textual frame of the code. The making visible of thinking within live coding practice sheds light on the nature of knowledge production and the mode of intelligence operative therein, generating insights into this habitually unseen aspect of creative endeavor. However, the exposition of process (within live coding) is not always concerned with explication; it also has the capacity for adding layers of complexity, further enriching the work itself. Indeed, within live coding (and artistic inquiry more broadly) the process of making visible one's reasoning is not always about creating transparency but sometimes about furthering opacity, desirable confusion, and bewilderment (or even sometimes a humorous effect).

The making visible of the operational thinking and working out within live coding thus seems predicated less on a model of *how to* (apply logical, algorithmic thinking within practice). It is not so much about direct explanation or instruction, an approach based on the pedagogical transmission of a specific method, or its technics and techniques. Rather, the revelation and live reworking of digital code through its performance is inherently advocatory, even political, involving showing and sharing the unfolding logic of a language so instrumental to contemporary life but in which still so few are fluent. The live coding performance thus has a constitutional function, helping augment and inaugurate a community of existing and future live coders (a community

of practice, to echo the earlier discussion in relation to the demoscene) through the revelation of a shared or common performance language.[88] Live coding makes visible its process of thinking so that others might then modify, build upon, and creatively develop this further still, reinforcing the importance of sharing code and *commoning* within this community, referred to earlier in relation to a free/open-source ethos as well as through comparison with the oral culture explored in chapter 4.

For technical know-how to be detoured—or even *détourned*, to borrow the situationist term for the "rerouting" or "hijacking" of an existing practice or artifact, especially the software practices and products of digital capitalism, toward a new subversive meaning[89]—in the direction of live coding's experimental performance, a prerequisite level of technical knowledge is undoubtedly needed. Indeed, many live coders report a commitment made over time to the preparatory practicing of programming for developing the high levels of familiarity, process fluency, and agility needed for live coding, both in terms of mental cognition and physical dexterity. However, live coding performance is not one of simply showing one's knowledge (as rehearsed and scripted in advance). It also seeks to create the germinal conditions wherein something unplanned for or unanticipated might arise.

Reflecting on the artistic journey into the "unforeseen" or "unforeseeable," writer-curator Sarat Maharaj differentiates between innovation (conceived as the "improvement and incremental adding to what is already there") and a species of creativity that is "about discontinuity, about rupture, about production and emergence, and the spasmic appearance of something entirely unexpected and new."[90] Indeed, receptivity to the experience of not knowing is as necessary for invention and intervention within artistic inquiry as it is within practice-based craft research. For curator Elizabeth Fisher and artist Rebecca Fortnum, "Artists often begin something without knowing how it will turn out. In practice, this translates as thinking through doing."[91] They argue that this mode of thinking through doing is often associated with a "largely negative lexicon" (the uncertain, invisible, incomprehensible). At other times, "not knowing is not only to be overcome, but sought, explored and savored; where failure, boredom, frustration and getting lost are constructively deployed."[92] The not knowing of live coding seems inseparable, then, from the question "What does live coding know?," and the notion of uncertainty is addressed further through the specific lens of indeterminacy later in this chapter.

In parallel, the on-the-fly improvisatory quality of live coding invites further reflection on the relation between improvisation and cognition, on the nature of improvisatory thinking, and on the kinds of knowledges generated in and through improvisation.[93] The improvisatory approach of live coding embraces a sense of play

within its process of experimentation. For sociologist Roger Caillois, play is an inherently "uncertain activity. Doubt must remain until the end. . . . An outcome known in advance, with no possibility of error or surprise, clearly leading to an inescapable result, is incompatible with the nature of play."[94] Live coders appropriate and redirect (hack) a specific form of technical knowledge and language, which is then played with. Understood as a form of artistic experimentation or perhaps even aesthetic play, live coding is inherently without function or purpose (beyond itself)—it is willfully *autotelic*. Indeed, play refuses to be coerced into the service of something else. As Caillois states, "A characteristic of play, in fact, is that it creates no wealth or goods. . . . At the end of the game, all can and must start over again at the same point. Nothing has been harvested or manufactured, no masterpiece has been created, no capital has been accrued."[95] A different way of looking at the playful excess of live coding is by returning to a point partly developed in chapter 4 (referring to Mark Fisher's take on *red plenty*), which argues that the pressures of capitalism are not about creating wealth but necessarily *restricting* wealth. Live coding then is not about producing *nothing* but producing overwhelming plenty.[96]

For philosopher and semiologist Paolo Virno, what characterizes the "work of the performing artist" is that their "actions have no extrinsic goal. They don't create a lasting product since they aim only at their own occurrence. They don't create new objects, but rather a contingent and singular event. . . . The purpose of their activity coincides entirely with its own execution."[97] As Mersch argues, "Art always begins anew. There is no finality in the art, no satisfying closure, state of peace, or generalizable result."[98] For Mersch, art does not "lead to cognitive gains and their supposed truths, but rather to a break in or destabilization of the reigning codes of knowledge."[99] Mersch foregrounds the question "How does art know?" by asking whether we can conceive of a specific mode of "aesthetic thinking" or of "art as theōria."[100] For him, this is a question of "what kind of thought artistic praxis generates and to what extent it can be conceived of as a particular, even singular form of thought, in contrast to a notion of thought linked to linguistics"—that is, thought "based on a 'propositional act.'"[101] Indeed, in these terms the question of live coding's knowledge is related not only to how technical computational knowledge is deployed therein but also to the specificity of live coding's "art *as* a thinking process."[102] Here, modifying the question of "What does live coding know?" to that of "How does live coding *think*?" mirrors a wider interest within other disciplines concerned with understanding the specificity of their thinking in and through the doing of practice.

For Mersch, the challenge is one of differentiating an artistic or even aesthetic mode of thought beyond both the vocabulary of linguistic discursivity and process methodology,

where the specificity—even alterity—of an aesthetic epistemology is made explicit. He argues that *discursiveness*—"making a statement or formulating an 'argument' in the form of sentences"[103]—and *methodology* based upon a "scientific, i.e. methodological, research process" have "advanced to become the main criteria for the production of episteme," neither of which, he claims, are "particularly suited to artistic practice."[104] Significantly, Mersch is keen to avoid "favouring tacit knowledge as is the trend in science studies and the history of science."[105] Instead, he asks what "thought in other media" might mean, where "thought is understood as a practice, as acting *with* materials, *in* materials, or *through* materials . . . or *with* media, *in* media or *through* media."[106] He argues that artistic thought "reveals itself in the form of those practices that 'work in the work,' the 'becoming' of the processes themselves."[107] According to Mersch, the process of artistic thought involves "the stimulation of effects or leaps rather than directional intentions or calculated efforts that follow a precise plan and aim for closure in a manner imagined at the work's inception."[108] Moreover, it is through the act of "showing" rather than "saying" that "aesthetic knowledges" communicate their distinctiveness. Following Mersch, then, the thinking of live coding is perhaps best expressed when it is shown at work *with*, *in*, and *through* the materiality and mediality of its performance through "surrender(ing) to the event and its experience."[109]

According to Manning and Massumi, "Every practice is a mode of thought, already in the act. To dance: a thinking in movement. To paint: a thinking through color. . . . the practice in question will be construed as a mode of thought *creatively* in the act."[110] They conceive this mode of thought variously as "thinkings-in-the-act" or as "thought-in-the act."[111] How then, is live coding's "thought in the act"? Connections have been made between the thought in the act of live coding and that of weaving, especially preindustrial loom weaving. Certainly, live coding's thinking-knowing resonates with the embodied thought-in-motion or loom-thinking activated while working on the loom:[112] both require heightened alertness to the live circumstances of their own production, a form of live or even kairotic thinking-in-action immanent to the process itself rather than conceived in advance.[113] The research project Weaving Codes, Coding Weaves (which we discussed in chapter 4 in relation to notation and in chapter 5 in relation to liveness) involved a radical recuperation of a largely ignored relation between ancient handweaving (as a mode of thought-in-motion) and the thinking that takes place within the practice of coding. The project attempted to retrieve a sense of lost or buried connections between coding and weaving by disrupting or dislodging the privileged position of the Jacquard loom in the historical conceptualization of these practices, arguing that the ancient warp-weighted loom prefigured the dyadic or Pythagorean arithmetic necessary for computational logic.[114] Rather than conceive the

connection between weaving and coding through the prism of machinic mass production and its privileged concepts of optimization, efficiency, productivity, and standardization, the research emphasis of this project was on technical processes that resist the standard template and that require the interweaving of multiple methods not possible to accommodate within standard mass production design: techniques involving the complex collaboration and cooperation between human and machine that, moreover, are predicated on the activation of embodied knowledge.

Weaving Codes, Coding Weaves focused on the points of resonance between two (temporally disconnected) practices to explore how an engagement with the past (the historical practice of ancient weaving) might open up new ways of thinking about the future (of live coding). These research concerns have since been expanded within the frame of the five-year project PENELOPE: A Study of Weaving as Technical Mode of Existence.[115] The research aim of the PENELOPE project is to integrate ancient weaving into the history of science and technology, especially digital technology, toward a better understanding of the textility of the processes of making and also thinking. The project encompasses an investigation of ancient sources as well as the practices and technological principles of ancient weaving, unfolding within a Penelopean laboratory for exploring the models and topologies of weaves and developing codes to make them virtually explorable. Both ancient weaving and live coding involve a live, embodied process of decision-making and knowledge activation that operates in excess of or *between the lines* of conventional notational systems. How might we account for the cognitive and bodily intelligences activated at the point where abstract algorithm meets with the lived experience of the weaver-coder?

Within live coding, the challenge seems less one of responding with learned behavior or an already rehearsed script than of how to harness the potential unique to every contingent situation. However, this is not about placing faith in a form of tacit knowledge if this describes an already embodied know-how. Instead, what is activated is a *known-not* knowledge closer perhaps to Maharaj's articulation of the flux of *no-how*, "distinct from the circuits of know-how that run on clearly spelled out methodological steel tracks. It is the rather unpredictable surge and ebb of potentialities and propensities. . . . No-how embodies indeterminacy, an 'any space whatever' that brews up, spreads, inspissates."[116] Live coding involves a mode of knowing that is activated in response to uncertainties of the situation and that alone is adequate to the task of responding to that situation.

Live coding offers a transdisciplinary challenge to some of the assumptions of what constitutes knowledge and how it might be articulated in ways in which the outcomes are less certain or prescribed. Our suggestion is that the nontraditional methods of

live coding, inclusive of the live interactions of programmer, program code, and the practice of coding, further expand the range of possibilities for knowledge production. To put this in the context of artistic research means to place live coding within a more general critique of epistemological paradigms and the possibilities for alternative forms to reshape what we know and how we know it and to redefine some of the limits of knowledge and what escapes its confines (in the realm of alternative knowledges or nonknowledge).[117] Likewise, a productive context for considering the alternative knowledges of live coding is the expanded perspective of postcognitivist approaches to thinking, including situated, embodied, enactivist, extended, and distributed modes of cognition that have developed in resistance to the computational theory of mind that is central to the development of cognitive science.[118] It becomes clear that formal epistemologies are inherently paradoxical and that alternative paradigms (such as live coding understood as artistic research or in relation to the broader context of postcognitivism) might help to reshape how and what we know about how knowledge is produced through real-time operations and recursive feedback loops.

Uncertainty

Live coding presents a further challenge to the conventions of research practices in its embrace of uncertainty and what live coding *could* know, including those methods that attempt to incorporate practice as a mode of research to destabilize expectations or goal-oriented approaches to research design. The embrace of uncertainty and not knowing within practice is often conceptualized (in relatively anthropocentric terms) from the perspective of the practitioner. However, the uncertainty referred to in this last section draws on a sense of indeterminacy that takes its cue from the decision problem that helps to define the limits of computation. As discussed in chapter 6, and according to Alan Turing's 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem (Decision Problem),"[119] some things are incapable of being computed, including problems that are well defined and understood. Computation contains its own inner paradoxes, and every Turing-complete language carries the potential for undecidable effects. It is not logically possible to write a computer program that can reliably distinguish between those programs that halt and those that loop forever. The decision problem unsettles certainties about what computers are capable of and what they can and cannot do. Yet, at the same time, decision-making processes become ever more automated and pervasive (especially in the era of post-truth), and it is possible to speak of a kind of "algorithmic decisionism" that values quick decisions more than correct ones.[120] Here we see the instrumental reasoning of the machine (especially the AI

machine) take form, and as such it seems important to question how to "think in terms of the means through which error, indeterminacy, randomness, and unknowns in general have become part of technoscientific knowledge and the reasoning of machines."[121] Live coding might be considered to operate in terms of decision-making in the way the uncertainty of outcomes is made evident through the introduction of further live and improvised actions outside the computer. All decisions are revealed to be contingent and subject to internal and external forces that render the performance itself undecidable and the broader apparatus an active participant. It becomes clear that research practices, methodologies, and infrastructures of all kinds are part of wider material-discursive apparatuses through which emergent forms of human and nonhuman knowledge are produced and circulated.

With reference to philosopher Foucault's *Archaeology of Knowledge*,[122] this contingent aspect is already well established, and thereby what constitutes knowledge is necessarily open to question and to alternative forms of knowledge or nonknowledge. To reiterate the point, it becomes necessary to conceptualize knowledge that is less human-centered to include ways of understanding and acting in the world that exceed what is rendered knowable through exclusively human sensing and sensemaking.[123] Live coding provides a good example of how the epistemic aspects of computational processes and technological infrastructures might be revealed through such means—for instance, in the sharing of code and screens and other sensemaking phenomena. Building on the discussions in the previous section around artistic research, we might then ask what live coding knows in itself.[124] The question helps to further establish how live coding is able to unsettle some of the certainties around how knowledge is produced. By referring to the notion of *onto-epistemology*, we invoke the feminist new materialism of Karen Barad,[125] who draws together epistemology (the theory of knowing) and ontology (the theory of being) into an ethical framework that she calls "apparatus" (after Foucault[126]). Her interest is in how apparatuses and subject/objects mutually create and define each other and as such are thoroughly active and "productive of (and part of) phenomena."[127]

If we apply this to live coding, we take our departure from an anthropocentric tendency to situate the programmer as the one who introduces uncertainty (as, for instance, through artistic intention or human error.[128]) Instead we suggest a more nuanced understanding to take into account the wider apparatuses in which human/nonhuman subjects/objects cocreate and establish outcomes together. In other words, the making, doing, and becoming of live code, coding, and coders are materialized through what Barad would call a complex intra-action of elements. As we hope is clear by now, our intention here is to demonstrate the potential of live coding to unsettle

some of the knowledge regimes through which it circulates—across various domains and networks of practice (performance, experimental computer music, computational practice, artistic, and so on). The potential to undermine expectations and introduce uncertainty is particularly important, we think, in the wider context of closed and opaque coded systems that do not provide access to any understanding of their inner operations, let alone encourage the manipulation of them at the level of writing or reading code or through the questioning of how live coding operates as part of a wider network or relations. One of the challenges then, we would argue, is to identify code as an integral part of coding practices such that it can be understood for what it is, how it is produced, and what it might become. This is arguably part of the criticality of live coding: to expose the conditions of possibility or, in other words, to remain attentive to the uncertainty of what constitutes knowledge in contested fields of practice and to demonstrate modes of uncertainty in what otherwise would seem to be determinate processes.

It is this condition of uncertainty that makes it important to us, related to the theory of probability, to stress that there is an indeterminism between human and nonhuman knowledge that comes close to the *uncertainty principle*.[129] Things are evidently known and unknown at the same time, and this is clearly the case in live coding. Events and processes are constantly renegotiated without recourse to any preexisting notion of space and time according to Barad, who is referring to both the uncertainty principle that asserts the trade-off between knowing or not knowing about position and momentum and physicist Niels Bohr's *complementarity principle* to understand how individual things have their own independent sets of determinate properties that exclude other properties.[130] This is not a consequence of the design of the experiment or the will of the scientist—or the composition of the live coding performance or intentionality of the programmer—but results from the material conditions in which the apparatus operates. According to Bohr's epistemological framework, an object is not independent of its scientific observation but is a "phenomena" and thus a condition for the way scientific knowledge is produced.[131]

Here Bohr's notion of apparatus comes close to Foucault's more complex formulation that extends the apparatus into a far wider relational network. Barad draws together these ideas—from Foucault and Bohr—to challenge the "epistemological and ontological inseparability of the apparatus from the objects and the subjects it helps to produce; and produces new understandings of materiality, discursivity, agency, causality, space, and time."[132] Her contention is that a more dynamic conception of materiality is required to take account of all bodies—human and nonhuman—that more fully describes the contemporary relation between power, knowledge, and bodies. As a result, there is not only the realization that uncertainties exist over space and time

but also the realization that apparatuses do not simply change in time but materialize through time. It is an open temporal (even time-critical) process that is not deterministic or straightforwardly causal in activating the movement from cause to effect. Rather, it can be argued that material-discursive apparatuses are constituted through constant intra-actions and that they are performative and necessarily open-ended in their production of bodies and meanings. The consequence is that spaces and times are more open to other possibilities of critical-political practice where indeterminacy and contingency coexist with causality and determinacy.

Machines, including computers, are already epistemological technologies if we follow these arguments and are not simply determined by human intentions or preexisting scripts. The live coding performance demonstrates the point really well that determinism and indeterminism coexist in their mutual and iterative performativity. This takes place across various spaces and temporalities and also at different scales that link together the various elements to provide connections and complementarities. Drawing on the new materialism of Barad in this way allows us to assert that it is simply not possible to generate knowledge outside of the material and ontological substrate through which it is mediated. Thus, the interrelation of epistemology and the ontological dimension of the materials, technologies, program codes, and bodies can be established as active and constitutive parts of knowledge. Live coding offers a useful paradigm in which to establish how the know-how of code is exposed in order to more fully understand how code subjects and objects mutually create and define each other.

Without this know-how, we would simply miss the detail on how various effects are produced and circulated, how human-nonhuman relations are coconstituted, and how codes are embedded in wider systems of control. We might further speculate on what it means for machines to know—especially in the context of machine learning, which produces its own distinctive form of know-how as "reasoning through and with uncertainty."[133] All this helps to establish the uncertainties of live coding along with the indeterminacy of computation. More expansive conceptions and diverse forms that emerge in the complementarity between certainties and uncertainties in coding and live performances become part of the critical potential of live coding: to expose the conditions of possibility and to speculate on the emergent forms of human and nonhuman knowledge that challenge humanist traditions of thinking. The need to approach these questions from a decolonial perspective is further explored in chapter 8, where we highlight the necessity of a more pluriversal approach that challenges Western epistemological frameworks, including many of those relied on in this chapter and within this book more broadly.

This is a section of doi:10.7551/mitpress/13770.001.0001

Live Coding

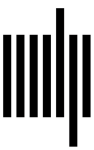
A User's Manual

By: Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, Thor Magnusson

Citation:

Live Coding: A User's Manual
By: Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, Thor Magnusson
DOI: 10.7551/mitpress/13770.001.0001
ISBN (electronic): 9780262372633
Publisher: The MIT Press
Published: 2022

The open access edition of this book was made possible by generous funding and support from the author



The MIT Press

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Stone Serif and Stone Sans by Westchester Publishing Services.

Library of Congress Cataloging-in-Publication Data

Names: Blackwell, Alan F., author. | Cocker, Emma, author. | Cox, Geoff, author. | McLean, Alex, 1975– author. | Magnusson, Thor, author.
Title: Live coding : a user's manual / Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson.
Description: Cambridge, Massachusetts : The MIT Press, [2022] | Series: Software studies | Includes bibliographical references and index.
Identifiers: LCCN 2022008717 (print) | LCCN 2022008718 (ebook) | ISBN 9780262544818 (paperback) | ISBN 9780262372626 (epub) | ISBN 9780262372633 (pdf)
Subjects: LCSH: Computer programming—Philosophy. | Agile software development. | Creation (Literary, artistic, etc.) | Algorithms—Psychological aspects.
Classification: LCC QA76.6 .B5794 2022 (print) | LCC QA76.6 (ebook) | DDC 005.1301—dc23/eng/20220527
LC record available at https://lccn.loc.gov/2022008717
LC ebook record available at https://lccn.loc.gov/2022008718