The Experience and Meaning of Programming

Alan F. Blackwell

Draft contribution to Oxford Handbook of the Philosophy of Computer Science (chapter 51)

Abstract

Programming is an exercise of (mechanical) control and (personal) agency. The meaning and experience of programming relates to the immaterial abstract worlds of information, notation and representation through which control and agency are achieved. However, programming can be understood by analogy to human experience of control and agency in the real world. The everyday experience of controlling one's digital life has been described as "end-user programming", by contrast to the professional engineering apparatus of assembling digital infrastructure. This chapter interrogates those contrasts and dynamics, in relation to the distinctive information-theoretic infrastructure of the attention economy, where finite human capacity for the construction of meaning through experience becomes a measurable commodity.

Keywords

Programming, End-Users, Direct Manipulation, Attention Investment, Attention Economy

Body text

What is programming? At its simplest, this is the business of telling a machine what to do. A child can understand it. In the classic animated film "The Wrong Trousers", inventor Wallace explains his new techno-trousers while pushing the control buttons: "I think you'll find this present a valuable addition to our modern lifestyle! They're Techno-Trousers. Ex-NASA. Fantastic for walkies! All you do is attach the lead on here... then *program in*. Walkies, 10 minutes, 20 minutes. Oops!" In order to understand the nature of programming, we must also consider why anyone tells a machine what to do, instead of doing the job themselves. In most cases[1], the reason is labour-saving. Wallace wants to save himself the effort of taking his dog Gromit for a walk, so programs the techno-trousers to do this on his behalf. Of course, it all goes wrong.

These are the topics to be considered in this chapter - instructions, buttons (or other user interfaces), labour-saving, and the ways in which things can go wrong.

[1] There are, of course, other cases, which I will address later in this chapter – see "A digression on labour" below.

To clarify my scope, I am not going to focus primarily on the professional practices of software engineering. As with other kinds of professional engineering, the day to day work of the software engineer involves negotiation of requirements and specifications within practical constraints, the selection of standardised components to achieve aspects of the required functionality, testing within performance limits, scheduling of tasks among a team, and contract negotiation. Each of these aspects of engineering practice is realised, in the case of the software engineer, by applying particular kinds of digital "tools," many of which may include aspects of programming. Nevertheless, software engineering as a whole, sometimes called "programming in the large," for the most part involves conventional engineering duties that need not even be described as programming (Curtis & Walz 1990).

As a contrast to this kind of software engineering work, which can be best understood by reference to other engineering fields, my main concern in this chapter is with people who might still program computers, but are not software engineers. My introductory example of the Techno-Trousers can be considered as an exemplar. When referring to the kinds of people who might program things, but are not professional software engineers, a widely-used term is the category of "end-user programmer" (Ko et al 2011). End-user programmers are often regarded in computer science as a special case, having peculiar requirements that are different to the majority of conventional programmers. However, it seems important to acknowledge that the number of people in the world who are *not* software engineers is many times larger than the number who *are* software engineers. This means that end-user programmers, rather than a minority to be treated as a special case by computer scientists, are in fact the overwhelming majority. The remainder of this chapter therefore focuses on their needs and experiences, rather than the comparatively small (and already perhaps over-served) constituency of professional software developers. My reference to the "meaning" of programming therefore refers to what programming might mean, for a person who is not obliged by their professional identity to be a programmer, but is in a position to make choices that are meaningful to them.

To consider a more precise definition, Ko et al offer the formulation: 'We then define end-user programming as "programming to achieve the result of a program primarily for personal, rather public use." The important distinction here is that program itself is not primarily intended for use by a large number of users with varying needs.' By 'personal', the intention is that the end-user programmer is addressing their own needs or goals, and writing a program that will advance those goals, rather than acting on the instructions of others. Importantly, the end-user programmer often has a degree of choice in relation to the task they are doing – including, often, the choice not to write a program at all, but to complete the task in some other way. Despite the rather light-hearted introduction to this chapter, it is important to note that end-user programmers are not amateurs, or students, or unserious about their work. For example, a cardiac surgeon may be using a programmable tool to definie a

process that helps assess the risk of post-operative bleeding (Robinson et al 2022). This is clearly a serious person, doing professional work, who just happens to be using computational tools as an element of that work.

A digression on labour

The theoretical grounding of this chapter, from one perspective, lies in behavioural economics, where the relative value of an automation technology rests in the potential for saving labour. I referred to this, in passing, in my opening example of a fictional domestic automation device. Such an analysis invites a Marxist interpretation, as for example in the work of Peter Damerow, in his history of the material tools of cognitive labour (Damerow 1995). Computers, whether programmed or not, certainly fall into this class. However, there are alternative frames that can be applied to programming, when considered in the context of creative self-expression, the economy of social media producer/consumers, and other discretionary uses of technology that are distinct from the office automation environment where commercial computation first emerged. While much of the scholarship of informatics and computer science developed within those business, military and industrial contexts, where "software engineering" was valorised as an enabling professional skill, there are other ways of doing programming that have little resemblance to the approved professional practice. Bergstrom and Blackwell, in their survey of the diverse "practices of programming" (Bergstrom & Blackwell 2016) catalogue provocative alternatives such as code-bending and live coding, being rather un-economic ways in which an end-user might find themselves creating code. In such cases, it seems likely that the experience is the meaning, rather than (as in professional software development) a means to an end.

The programming perspectives of machines and of users

The majority of programming languages are designed, not for the global majority, but for use by software engineers and computer science graduates – a privileged sub-category within the already privileged group of WEIRD (Western, Educated, Industrialized, Rich, and Democratic) people (Henrich et al 2010) who have historically been the focus of most psychological research into programming and related topics (Sarkar 2024). A relatively small number of languages have also been created by computer science professors for pedagogic purposes, to help their students acquire basic understanding through simple class exercises (i.e. "programming in the small") without being distracted by the features required to manage larger more complex projects. Occasionally, the simplicity and ease of learning prioritised by these educational languages has also made them popular among end-users, as occurred with educational languages BASIC and Pascal. However, a great deal of end-user programming is done using tools that do not resemble conventional programming

languages, for example the Excel spreadsheet, which is estimated to be used by more people than all other programming languages put together (Scaffidi et al 2005).

When considered from the broad perspective of all possible end-users, and considering visual notations ranging from spreadsheets to jigsaw puzzle assembly (as in the popular teaching environment Scratch (Maloney et al 2010)), or even the 'programming' done with control keys on a music keyboard, a sewing machine, a heating thermostat timer, an automated cat feeder, or perhaps even a fictional dog walker, what analysis could cover such a range of activities (Blackwell et al 2003)? Is the use of the word 'programming' for such tasks purely historical or metaphorical, perhaps not related to 'proper' programming in Python or Javascript?

If we step back from the specifics of syntax, screens and keys, there are three broad approaches to the specification of program behaviour (and hence philosophies of programming language design) – two of them familiar from mainstream computer science, and one more often associated with experimental approaches to end-user programming.

1. Imperative programming consists of a sequence of instructions that specifies what actions the machine is to carry out.

2. Declarative programming specifies the required outcome of the program execution rather than the steps that should be taken.

3. Demonstrational programming observes the actions taken by the user, in order to repeat some variant of these in future.

The historical precedent to these different perspectives is considerable. The world's first general-purpose stored program computer, designed to be used in scientific applications for end-user researchers in Cambridge, was introduced in a short user guide to programming EDSAC as follows: 'By the "program" for a calculation is meant the schedule of operating instructions which must be applied to the machine to enable it to carry out the calculation." (Wilkes, Wheeler & Gill 1951). This corresponds to the "diagram" used by Ada Lovelace to specify a sequence of operation cards for Babbage's Analytical Engine (Lovelace 1835/1989). Successive generations of programming technologies have aspired to make programming more accessible to users who are not familiar with the operation of the machine, and who might therefore struggle to articulate what specific sequence of operations instructions should be invoked (Arawjo 2020). Instead, the repeated ambition has been that

calculation should be described in terms of the result that the programmer expects – classic historical examples have been FORTRAN, the formula translator, which was intended to allow mathematicians to specify their intentions in a more familiar way, while automating the unfamiliar sequencing of the machine operations that might be familiar to computer engineers but not to mathematicians.

In these generations of programming language evolution over the course of decades, in examples such as the Common Business-Oriented Language COBOL (Sammet 1978), or the logic programming language PROLOG (Körner et al 2022), the ambition has been to achieve more declarative natural specification of the desired outcomes that might be accessible to new classes of user, rather than attending to mechanical details of the machine operation familiar to specialist programmers of the previous generation. Inevitably, since it is a machine that is being addressed, the practical elements of declaration become polluted with mundane considerations of execution order, as in the notorious case of the PROLOG "cut", which had no necessary semantic relation to the logical formalisms being expressed, but was critical in making the inference algorithm controllable. To become a proficient PROLOG programmer, it was necessary to master the cut statement, although this was seldom referred to in the popular advocacy of declarative programming as the ideal specification of AI software by logic alone.

Given the historical aspirations of these categories, the ways that the underlying ambitions have often been frustrated, and the lack of clarity in advocacy of different approaches, it is unsurprising that there is considerable technical overlap between their application in practice. For example, a user programming by demonstration is unlikely to repeat precisely the same action. After having demonstrated 2 + 2 (for example), few people need to calculate 2 + 2 a second time, but perhaps 2 + 3, then 2 + 4. Inferring sequential variants from an initial demonstration requires judgment of which aspects should remain constant, and which are to be generalised. This kind of inferred generalisation is often called programming by example, rather than programming by demonstration. Generalisation can also be controlled declaratively, as when an Excel user places a "$" in front of a column letter or row number, to restrict the general pattern of inference that otherwise occurs when copying formulae between cells. (For readers unfamiliar with Excel, a typical example would be that, if the cell in row 1, column C is calculated as the sum of the previous cells in that row 'C1=A1+B1', when that formula is posted to the next row, the pasted formula gets changed by default to C2=A2+B2, then C3=A3+B3 and so on. The user can override this, by making the original formula 'C1=A1+B$1'. This has the effect of adding the same value B1 in every row the formula is pasted: C2=A2+B1, C3=A3+B1 …)

The aspiration for programming to be purely demonstrational, so that the user would not have to think about the mechanics of programming, is not always achieved. As a result, demonstrational programming turns out to be somewhat declarative, as the user has to assist the system with

generalisation (as in the Excel case just described). Similarly, the line between declarative and imperative programming may be fuzzy, since most ways of declaratively specifying the execution outcome are only effective with some knowledge and anticipation of what steps the algorithm might take in order to achieve them, as in the case of the PROLOG cut. Contrarily, imperative languages may offer single actions whose technical realisation is in reality very complex (for example, the iterative evaluation of an optimisation function), such that the programmer is not really aware of the individual steps that are really being executed.

In the early decades of commercial computing, all those who interacted directly with computers were described as engaged in programming (e.g. Weinberg 1971). It was not necessarily understood that programming implied managerial responsibility for the specification of abstract business processes, rather than mundane machine operation (Hicks 2017). In relation to the concerns of this chapter, the key distinction between the two is in how the actions of a user relate to generalisations of those actions (demonstration versus inference), and how the specification of actions might involve notated description of the behaviour, rather than notated description of the outcome (imperative versus declarative). The dramatic business success of the spreadsheet, in which the introduction of VISICALC drove the large scale business adoption of the Apple II computer that was needed to run the early releases, demonstrates the extent to which programmability, by business people without specialist training, has been a critical driver of personal computing. Indeed, the spreadsheet, integrating a tabular notation with a computational model in a way that by default hides both the ('formula') code and the details of execution, marked a critical transformation in the understanding of end-user programming, as documented at the time by organisational ethnographer Bonnie Nardi (Nardi 1993).

The contrast of direct manipulation

A particularly significant historical development is the now-familiar experience of "direct manipulation". This term was originally proposed by Shneiderman (1983) as "a step beyond programming languages," and became widespread around the time that graphical user interfaces started to be marketed for use by end users in the early 1980s. In direct manipulation, there is no generalisation at all. The state of the system changes only in response to the actions that the user demonstrates – for example, the direct action of dragging an icon to a waste bin results only in the single deletion that has been demonstrated, with no other effects. This was a significant contrast to the previously dominant file management paradigm of the command line, where syntax such as DEL *.* (which in the MS-DOS system, might permanently delete all files on a floppy disk in seconds) encoded simplified versions programming conventions such as functions, arguments and regular expressions. Shneiderman's advocacy of incremental and reversable actions on directly visible object

state was a relief to many nervous users who saw the potential for programmability as a source of risk and danger (Blackwell et al 2003). Atomic actions provide no further opportunity for abstract specification or generalised interpretation of the single desired effect, thus eliminating much of the risk, but also much of the power, of the code-like command line.

The limitations inherent in Shneiderman's so-called "step beyond programming" were apparent at the time, with widespread protest among expert PC-users, as the abstract specification capabilities of the MS-DOS command line were increasingly replaced by the restrictive and repetitive dragging around of icons on the Apple Macintosh screen. Nevertheless, the direct manipulation style, and the graphical user interface (GUI), eventually became the most popular approach to interactive "programming" despite complete lack of any abstraction. Those conventions are so familiar today that few computer scientists would even consider the operation of the GUI to be programming, despite the original 1980s formulation of this approach as an advanced "step beyond".

The reason for the popularity of the GUI is that it is so easy to control – indeed this is the very advantage that Shneiderman was drawing attention to. Because no other effect results from each action, the user need not spend any time analysing what might occur as a result of their choices (referred to as "the frame problem" in AI planning (Shoham 1987)). Direct manipulation interfaces do *exactly* what the user specifies, and *only* what the user specifies. The loss of abstract expressive power represents a gain in ease of use. Of course, as with the earlier observations about the aspirations of declarative or demonstrational programming paradigms, the reality of many products and systems is that this desired simplicity often comes with subtle exceptions, such as the cut in PROLOG, or the dollar sign in Excel, pragmatically introducing features that may not be consistent with the designer's primary aspirations. As a result, strong statement of aspirations (such as the words "exactly" and "only" above) can often be undermined by close inspection of any actual product.

In an era when direct manipulation has become the accepted standard of usability for all end-user computing, the programming capabilities that were lost in direct manipulation – abstract specification and generalisation – are now recognised as the distinctive essence of programming. In effect, the experience of programming might be defined in contrast to direct manipulation, resulting in a naive assumption that programming is not direct manipulation, and vice versa. The key programming attributes of abstract specification and generalisation, by analogy to the AI frame problem, require greater cognitive effort than direct manipulation, because the user must think about ways in which their abstract instructions (whether explicitly specified or implicitly generalised) will be interpreted in other contexts, possibly in the future when they are not present to observe, restrain, or correct any unforeseen consequences (Blackwell 2002). A classic example is 'programming' the timer of a central heating boiler, which is routinely described with that word, despite a user interface that does not

resemble any conventional programming language. Although the buttons on the controller may appear physically direct, they do not turn the boiler on and off, but are the editor interface for a program notation – the times and setting displayed – whose execution will continue in the user's absence.

Direct manipulation has become such a familiar standard of usability, that an obligation to engage in 'code' programming could be considered *unusable* by definition. Design guidelines and established practice, from smartphones to websites, assume that every action should have direct, predictable and observable effects. Inferential AI "mixed initiative" systems (Allen et al 1999), that offer labour saving options by observing repetitive patterns and suggesting automated programmatic shortcuts, are often seen as intrusive and untrustworthy, most notoriously in the case of the Microsoft Office Assistant Clippy. Useful abstract specification tools such as command lines, scripts and markup languages are also viewed with suspicion as an obscure "code." There might be an element of expert mystique to "coders" and "coding", but to many students, they are no more welcome than Latin grammar or school algebra.

Vibe coding and Agentic AI as end-user programming

Recent advances in the use of transformer-based Large Language Models (LLMs) for code generation are resulting in significant reconfiguration of many of these dynamics. It is a challenge to write a reference handbook at a time of rapid change, when much of the popular discourse is driven by the demands of product marketing following a period of unusually intensive investment (Haigh 2025). Currently popular marketing terms (which may not stand the test of time) are "vibe coding" and "agentic AI". Both interaction styles are absolutely central to the argument of this chapter, building on much the same experiences and meanings as previous generations of end-user programming tools.

Whether or not these specific fashionable terms have any longevity, they are a useful guide to the aspirations of meaning and experience that their advocates expect to achieve. For example, celebrity AI developer Andrej Karpathy introduced the term `Vibe coding` to describe his personal experience[2], at the time a provocative challenge to the industry standard routine of careful manual supervision and code inspections (Karpathy 2025). His use of the word "vibe" clearly nods toward the dynamics identified by Bergstrom & Blackwell, an activity that is pleasurable rather than onerous. In current use at the time of writing, it refers to a style of LLM use, in which prompts are written in natural language (generally English) to control a system that is generating and modifying programming language source code. Although LLM enhancements had already been made to most mainstream software

[2] The huge public attention that Karpathy's short post attracted led to one dictionary nominating vibe coding as its word of the year.

engineering tools one or two years previously, typical integration involved the programmer focusing mainly on the source code in the centre of their screen, while a sidebar-based agent would act as a programming assistant ("CoPilot" in the term introduced Github/OpenAI, then extended by Microsoft to refer to all the AI integrations in their suite of Office applications). In vibe coding, by contrast, the programmer never looks at the generated source code (Karpathy says 'forget the code even exists'), but simply issues instructions via the LLM, trusting that the resulting code will correspond to what they have requested.

While Karpathy's original coinage made a nod to hacker cultures such as those of Bergstrom's playful code-benders, subsequent advances in the performance of LLM-based tools, especially (at the time of writing) Claude Code from Anthropic, means that this approach is no longer a playful counter-culture among tech enthusiasts, but fast becoming the new standard approach to code generation, likely to be comparable in its business impact to previous high-level languages such as FORTRAN and COBOL, whose users experienced them in a way comparable to vibe coding – generating assembler code that programmers would seldom inspect. In this sense, a professedly non-professional, creative, playful 'vibe' has already become a new generation of professional tool, in the same way that the more accessible graphical user interface, initially disparaged as unsuitable for professional users, was gradually adopted as the preferred interaction style even for computing professional.

Another buzzword being applied in business, technical and policy discourse at the time of writing is 'agentic AI'. Unlike vibe coding, this term cannot be attributed to a single individual. Companies seldom provide a strict definition of marketing buzzwords, preferring to maintain flexibility of interpretation in case fashions change (Haigh 2025). Similarly, academic commentators may find it expedient to adopt terms from commercial practice when arguing for the potential impact of their work, rather than insisting on careful philosophical grounding. Nevertheless, a reasonable characterisation is that 'agentic' technologies are intended to act on behalf of the user, saving effort and attention in the same way as the fictional persona of company acting as a legal agent (Watson 2021). In the discourse of ethical and responsible AI, the term 'agent' also alludes to the potential ethical 'agency' of an autonomous and conscious AI worker, robot, vehicle and so on.

When considering agency in the light of this chapter's title, as an experience, we might consider the loss of human agency when machines take control, in contrast to the feeling of agency that results from personal effort and achievement. Human-computer interaction researcher David Coyle and his colleagues used methods from clinical psychology to demonstrate how the feelings of agency for a computer user are diminished when an intelligent feature takes the initiative to act on the user's behalf (Coyle et al 2012). The speculative consciousness of an autonomous AI agent should therefore be

considered alongside the potential for psychological harm to the actual conscious human who interacts with the ‘agent’.

Whether or not the code-generating ‘agentic’ LLM is promoted by marketers as a potentially conscious autonomous agent, we can certainly understand the automation opportunity in the light of end-user programming. All end-user programming has the intention to help the user automate some action, as introduced at the start of this chapter. At a crude level, every ‘automation’ involves the creation of a mechanical ‘agent’, constructed to carry out some task that the developer would otherwise have to perform themselves (or delegate – perhaps to a human ‘agent’). So in this sense, agentic AI is nothing more or less than a repetition of the promise of end-user programming. Once again, a person who needs to get something done can achieve this by instructing a computer – by programming. The programming language certainly looks very different (in the vibe coding era), involving natural language statements rather than the algebraic controlled vocabulary of FORTRAN or Python. But as already explained, the desire for usable automation has already led to many different interaction styles, not only new and more abstract declarative programming languages, but also the novel representational formalisms of the spreadsheet or the graphical user interface. The prompts that might be used to specify task automation in an ‘agentic AI’ product are the latest technical variation on this theme.

The attention economy of end-user programming

The experience and meaning of programming arises from, and is entangled with, all these dynamics. Standards of usability such as the NASA Task Load Index (TLX) (Cao et al 2009) discourage the mental effort of abstraction, and of abstract notation. The spreadsheet is a fascinating case study, designed such that concrete objects of interest can be directly manipulated, while more abstract elements such as cell formulae and macros are hidden from view. Yet programming, whether the abstractions are declarative, imperative, or demonstrated by example, inevitably requires effortful abstract thought to anticipate the consequences of abstract decisions (Blackwell 2002). If programming were simply an exercise in optimising the attentional labour economy, one might ask, why should a computer user ever willingly engage in effortful thought? The fields of usability engineering and cognitive ergonomics, since the origin of direct manipulation, have for decades been dedicated to minimising effort and complexity, not advocating to increase them.

A different perspective might consider, as an alternative to this efficiency-based focus on the mechanics of programming, other kinds of experience and motivation (Aghaee et al 2015, Bergstrom et al 2016). This perspective could be considered a poetics of programming, in which the programmer experiences not mundane operation, but the exercise of will, the manifestation of consciousness as an

intentional agent, and creative acts of original construction, whether an artist on stage in the performance genre of live coding (Blackwell et al 2022), or a mathematician assembling the components of a sophisticated proof. Will, consciousness and creativity are not utilitarian values of the kind optimised by ergonomists, but fundamental to the experience of being human, aspects of the soul that makes a person human, rather than animal or machine.

In responding to Charles Babbage's economic proposal of a "manufactory for numbers" (Schaffer 1994), Karl Marx observed that humans would become no more than conscious linkages in the machine (Blackwell 2019). The economic drivers that have underpinned the recent LLM-based AI boom (Vertesi et al 2026) are those of cloud capitalism or techno-feudalism (Varoufakis 2024), in which large-scale regulatory failures encourage platform dynamics of surveillance (Zuboff 2019), data colonialism (Couldry & Mejias 2020) and "enshittification" (Doctorow 2025). As explained by Matteo Pasquinelli in his social history of AI, the extraction of value involves capturing conscious subjectivity and control from human authors, transferring credit to the AI machine (Pasquinelli 2023, 2). . The experience of programming, in which humans rather than machines are the conscious and creative agents, promises by contrast to be an emancipatory force in an era of feudal platform capitalism. Technically literate commentators such as Cory Doctorow (2025) draw attention to the significance of APIs (he emphasises that the P is for Programming) as the locus of competition and regulation.

Attention is fundamental, both to the experience of programming (Blackwell 2002), and to the dynamics of the knowledge economy (Williams 2018). Adam Smith observed that the purpose of capital is to be attended to (Williams 2018, 60), meaning that the liberating potential of programming within a technofeudal economy is precisely the extent to which we might regain creative control over our own attention. It was Gogol, not Google, who best described the feudal economy of the soul (Gogol 1842/2004). But as consciousness itself becomes a commodity, discussion of programming should be aware of the moral significance in how we invest and exchange the finite hours of each person's life (Blackwell 2026).

References

Aghaee, S., Blackwell, A.F., Kosinski, M. and Stillwell, D. (2015). Personality and Intrinsic Motivational Factors in End-User Programming. Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2015), pp. 29-36.

Allen, J. E., Guinn, C. I., & Horvtz, E. (1999). Mixed-initiative interaction. IEEE Intelligent Systems and their Applications, 14(5), 14-23.

Arawjo, I. (2020). To write code: The cultural fabrication of programming notation and practice. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (pp. 1-15).

Bergstrom, I. and Blackwell, A.F. (2016). The Practices of Programming. In Proceedings of IEEE Visual Languages and Human-Centric Computing (VL/HCC 2016), pp. 190-198.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 2-10.

Blackwell, A.F. (2017). End-user developers - what are they like? In F. Paternò and V. Wulf (Eds). New Perspectives in End-User Development. Springer, pp. 121-135.

Blackwell, A.F. (2019). Artificial intelligence and the abstraction of cognitive labour. In M. Davis (Ed.), Marx200: The significance of Marxism in the 21st century. London: Praxis Press, pp. 59-68.

Blackwell, A.F. (2026). Conversations with, among, and about machines. Workshop paper for Science in the Forest, Science in the Past V, 4-6 June 2026, Cambridge UK. Publication forthcoming with Berghahn.

Blackwell, A.F., Cocker, E., Cox, G., Magnusson, T. and McLean, A. (2022). Live Coding: A User's Manual. Cambridge, Mass.: MIT Press.

Blackwell, A.F., Hewson, R.L. and Green, T.R.G. (2003) Product design to support user abstractions. In E. Hollnagel (Ed.) Handbook of Cognitive Task Design. Lawrence Erlbaum Associates. ISBN 0-8058-4003-6, pp. 525-545.

Blackwell, A.F., Petre, M. and Church, L. (2019). Fifty years of the Psychology of Programming. International Journal of Human-Computer Studies 131, 52-63.

Cao, A., Chintamani, K. K., Pandya, A. K., & Ellis, R. D. (2009). NASA TLX: Software for assessing subjective mental workload. Behavior research methods, 41(1), 113-117.

Couldry, N and Mejias U.A. (2020) The costs of connection: How data is colonizing human life and appropriating it for capitalism. Redwood City, CA: Stanford University Press.

Coyle, D., Moore, J., Kristensson, P.O., Fletcher, P. & Blackwell, A.F. (2012). I did that! Measuring users' experience of agency in their own actions. Proceedings of the 30th ACM Conference on Human Factors in Computing Systems (CHI 2012) ACM Press - Association for Computing Machinery, pp. 2025-2034.

Curtis, B., & Walz, D. (1990). The psychology of programming in the large: Team and organizational behaviour. In Psychology of programming (pp. 253-270). Academic Press.

Damerow, P. (1995). Abstraction and representation: Essays on the cultural evolution of thinking.

Doctorow, C. (2025). Enshittification: Why everything suddenly got worse and what to do about it. Verso Books.

Gogol, N., tr. R.A. Maguire (1842/2004). Dead Souls. Penguin Classics.

Haigh, T. (2025). Artificial Intelligence Then and Now. Communications of the ACM, 68(2), 24-29.

Henrich J., Heine S.J. and Norenzayan, A. (2010). The weirdest people in the world? Behavioral and Brain Sciences 33(2-3) 61-83.

Hicks, M. (2017). Programmed inequality: How Britain discarded women technologists and lost its edge in computing. MIT press.

Lovelace, A. (1835/1989) The Analytical Engines in A. Hyman (Ed), Science and Reform: Selected works of Charles Babbage. Cambridge University Press 242-311

Karpathy, Andrej [@karpathy] (2025). There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes

until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works. eX-Twitter, 2 February 2025.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M., Erwig, M., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.-B., Rothermel, G., Scaffidi, C., Shaw, M., and Wiedenbeck, S. (2011). The State of the Art in End-User Software Engineering. ACM Computing Surveys 43(3), Article 21.

Körner P, Leuschel M, Barbosa J, Costa VS, Dahl V, Hermenegildo MV, Morales JF, Wielemaker J, Diaz D, Abreu S, Ciatto G. (2022). Fifty years of Prolog and beyond. Theory and Practice of Logic Programming 22(6), 776-858.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. ACM Transactions on Computing Education (TOCE), 10(4), 1-15.

Nardi, B. A. (1993). A small matter of programming: perspectives on end user computing. MIT press.

Pasquinelli, M. (2023). The eye of the master: A social history of artificial intelligence. Verso Books.

Robinson D., Church, L., Blackwell, A.F., Vuylsteke, A., O'Hara, K. and Besser, M. (2022). Investigating uncertainty in postoperative bleeding management: Design principles for decision support. In Pro$^{c.}$ 35th International BCS Human-Computer Interaction Conference (HCI2022). DOI: 10.14236/ewic/HCI2022.25

Sammet, J. E. (1978). The early history of COBOL. In History of Programming Languages (pp. 199-243).

Sarkar, A. (2024). Diversity in study participants and a critique of the "representative sample" in human-computer interaction research. In 2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (pp. 394-399). IEEE.

Scaffidi, C., Shaw, M., and Myers, B. A. 2005. Estimating the numbers of end users and end user programmers. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. 207--214.

Schaffer, S. (1994). Babbage's intelligence: Calculating engines and the factory system. Critical inquiry, 21(1):203– 227.

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. Computer, 16(08), 57-69.

Shoham, Y. (1987). What is the frame problem?. In The frame problem in artificial intelligence (pp. 5-21). Morgan Kaufmann.

Varoufakis, Y. (2024). Techno feudalism: What killed capitalism. Penguin Random House.

Vertesi, J., Boyd, D., Taylor, A. S., & Shestakofsky, B. (2026, June). Reckoning with the Political Economy of AI: Avoiding Decoys in Pursuit of Accountability. In The 2026 ACM Conference on Fairness, Accountability, and Transparency (pp. 2186-2205).

Watson, S. (2021). Viewing artificial persons in the AI age through the lens of history. In Technology and Corporate Law (pp. 21-41). Edward Elgar Publishing.

Weinberg, G. M. (1971). The psychology of computer programming (Vol. 29). New York: Van Nostrand Reinhold.

Wilkes, Wheeler and Gill. (1951). A supplement to 'The Preparation of Programs for an Electronic Digital Computer. University Mathematial Laboratory, Cambridge.

Williams, J. (2018). Stand out of our light: Freedom and resistance in the attention economy. Cambridge University Press.

Zuboff, S. (2019). The age of surveillance capitalism: The fight for a human future at the new frontier of power. London: Profile books.