A Craft Practice of Programming Language Research

Alan F. Blackwell
Computer Laboratory
University of Cambridge
Alan.Blackwell@cl.cam.ac.uk

Abstract
It has become increasingly common to consider programming as a craft practice, driven largely by tools that are sufficiently responsive for reflective conversation with material, agile design processes, and live coded performance. This paper considers some of the epistemological questions that arise in programming as a critical technical practice, and especially when programming language research itself is taken seriously to be a craft.

1. Introduction
As part of a 2010 investigation of software development practices among digital arts professionals, Woolford et al (2010) interviewed Rosy Greenlees, the director of the UK Crafts Council, in order to identify ways in which critical understanding of quality in artistic software development might be informed by craft practice traditions. Now is a good time to reflect on that investigation, in a year when the annual Psychology of Programming Interest Group meeting is hosted by the Art Workers' Guild.

We can continue to draw on Philip Agre's concept of a *critical technical practice*, originally introduced in his notes on attempting to reform AI (Agre 1997). Agre was concerned with the implications of research that is necessarily both philosophical and practical – involving technical construction alongside a discourse that analyses and describes the thing being built. True AI research always involves both elements (it is possible to talk about future AI systems without building them, but that approach is more commonly associated with science fiction than with science).

The study by Woolford et al was primarily concerned with criticality in artistic practice – how is the intellectual discourse of the arts shaped in relation to philosophical, social, political or other concerns? However the software developers interviewed in the course of that project, despite working in a professional arts context, emphasised the practical engineering disciplines inherent in their work as being primary. Unlike Agre's AI researchers, the principal outcome of their work is an artefact. Engineering is essential in their business because, above all else, the show must go on.

When meeting at the Art Workers' Guild, how might programming language researchers situate the technical and philosophical aspects of their investigations? Is a programming language primarily an idea, to be instantiated in mundane technical implementation, or is it an artefact whose construction leads us to understand what it is – in other words, a craft? Programming languages are closely associated both with AI and with digital arts, so is programming language research an art or a science?

Edsger Dijkstra (1977) famously rejected the notion that programming was a craft – or rather, he acknowledged that it was practiced as a craft, but argued that it should not be, that it must become a science. Nevertheless, contemporaries such as John C. Reynolds, in his formally-motivated textbook The Craft of Programming (1981) continued to appeal to the notion of craft as embodied skill for the expert practitioner. Antranig Basman (2016) playfully reflected on these distinct notions in his essay exploring the virtues of craft practice, turning the title of his PPIG paper into a footnoted lament that Building Software is Not $^{*[Yet]}$ a Craft

In my own experiments in programming language design, I have explicitly considered whether a programming language can itself be produced through a craft practice (Blackwell 2013, Blackwell & Aaron 2015), and have created an experimental language through this process (Blackwell 2014). There are, of course, many different practices that can be brought to software projects (Bergstom & Blackwell 2016), and my own experimental production of the Palimpsest language is only one of these. Nevertheless, as a distinctive methodological approach, it is particularly appropriate to reflect on its implications at this Art Workers' Guild meeting.

Notions of craft, extending beyond skilled expertise (the titular sense of Reynolds' proof-based programming textbook), to the embodied knowledge of a tool held in the hand, have become increasingly current in discussion of programming as tools become increasingly live (Tanimoto 2013). Facilities that accelerate programming through code completion, text prediction, refactoring support and so on mean that the programmer's intentions flow through the typing hands even more quickly than read-eval-print loops or interpreted environments such as Smalltalk. The agility of working in (and fluently modifying) the Smalltalk environment has already had profound effects on the software industry, and indeed on digital experience universally, not only through the interaction modes of the GUI (Blackwell 2006a), but in the live collaborative editing practices of the wiki (Leuf & Cunningham 2001), the design understanding of the pattern language (Blackwell & Fincher 2010) and the managerial philosophies of SCRUM and other agile development practices (Beck et al 2001), all of which emerged from the Smalltalk community.

But the question asked in this paper is whether these live and embodied craft practices can be turned back, not simply in the professional environment of agile software engineering, or the creative context of live coding performance, but as an element of programming language research itself. Can the commonplace analogy of the apprentice making her own tools be appropriately applied to the craft-making of a programming language as a tool?

2. Experiencing Software Craft

Code, as observed by Daniel Cardoso-Llach (2015), carries metaphorical associations of weightlessness. Codes are disembodied languages rather than physical machinery. Yet codes are also rule-systems and orderings, specifying regularities over the material world of substance and phenomena. For example, performance coding as an artistic practice imposes order through sound and light fields, just as the sculptor's chisel extracts form from stone, or the painter's brush arranges it from pigment. Of course, although software is conceptually abstract, it has always also been wholly embodied. The computers in which these codes are constructed, executed and observed are complex and costly assemblages including rare minerals and sweatshop labour. Their operation depends on a material infrastructure of communications, power networks, server farms and processor clusters spanning the globe.

The tension between the materiality and immateriality of code constantly challenges the ways in which we understand it. 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, replacing fallible or undisciplined human hands with the precisely replicable actions of machines. Computer-aided design and manufacturing, 3D printing, and software itself, are engineering intermediaries between the industrial body and the governmental soul.

However, when live-coders turn code into a medium for artistic exploration, we overturn much of this conventional understanding. Rather than an instrument of control, through which engineers impose order on a chaotic world, code itself becomes a material within which the craft-coder develops and displays a creative practice. Code is not the chisel, but the wood. 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 practice.

This change is 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 (Wiberg 2013). Where software was once separate from the materiality of everyday life, sustaining a kind of technological mind-body dualism, we have now become thoroughly entangled with computers that are embedded in our clothing, our cars, our chests, our pets, or attached to our wrists and on our faces. After losing their screens, embedded computers become Tangible User Interfaces, joining the Internet of Things.

Interaction designers for this tangible, embodied and embedded world of computation are thus re-engaging with materials and craft practices in order to build their interactive metal, wooden or cloth prototypes. Rather than working with the "pure digital" (Hanse et al 2014), they have again become craft makers. And as reflective practitioners, this creative design research work draws their attention to the resultant conversation with materials that is familiar in the design theory of Donald Schön (1983). 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 (Gross et al 2013).

Unsurprisingly, this research engagement with new-found material practices has also led to a concern with the materiality of code itself. 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 after turning from the workbench back to their laptop keyboard, they are still having a conversation with a material (Schön 1983), in which it resists their intentions, disrupting their pure theoretical conceptualisations via the mangle of practice (Pickering 1995).

Coding is indeed hard, and code often seems to be resistant to the intentions and desires of the coder - experienced in much the same way as when physical materials resist craft labour. But if code is a material, it would appear to be a surprisingly immaterial one. The simultaneous immateriality of code means that it is equally resistant to this alternative characterisation by design theorists. Surely code cannot be material in the same sense as a plank of wood, or a ball of clay, which require (as Sennett describes) a dialogue between the head and the hand? Yet interaction design theorists persist in the argument that software is material, and that where there is a material, there must be a craft. Programming is described as a craft skill, with practitioners writing manifestos for "software carpentry" or "software craftsmanship". Even Sennett describes open source software development as "public craft".

Once again, this presents a challenge in drawing the appropriate analogies between our traditional understanding of craft and materials, and the experiences of making software. McCullogh's celebration of the "practiced digital hand" (1998) describes a master user of computer-aided design tools as engaged with coaxing reluctant or recalcitrant digital materials. Gross et al (2013) draw on Cohen's theory of artistic media 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.

The reader may be thinking that the matter-mind dualism implicit in these distinctions and discussion is either unwarranted or unsophisticated. Perhaps this problem results in part from the need for a new and more subtle conceptualisation of computation as extended cognition. Just as the 'pure' code of theoretical computer science is actually embedded in large material infrastructure, so craft practices are physically embodied in the craftsperson, and socially embedded in communities of practice.

One such long-standing community of practice is the demo-scene, an antecedent of the live-coding community that shares many common concerns with live coders. Demo-scene participants create virtuoso technical artworks, which they present to their peers in competition and performance events. Hansen et al (2014) undertook an ethnographic study of the demo-scene community, from which they developed a theory of craft practice, as observed among these code-artists. They see a relationship between the rhythmic elements of the artworks, and the rhythmic practice of tweaking and refining code. In their analysis this material practice results in a craft skill, moulding the practitioner at the same time as the material, developing technique as the basis for creative expression.

But should we expect the audience experience of a product to be the same thing as the experience of making it? We may admire the determination of the demo-scene perfectionist, tweaking his assembly code until every aspect of the sound and imagery are synchronised, but this repetition is surely not the same thing as the execution rhythms of the product itself.

Tim Ingold (2010) offers us an alternative conception of craft knowledge, in which there is no repetition (only machines repeat mindlessly), but rather one step after another, along a journeying path. The craftsman's tool seeks and responds to the grain of a material, in a process of accommodation and understanding rather than imposing form on inert substance. Material should be considered as 'matter-flow', in flux rather than stable, and the craftsman follows the material, in a manner that Deleuze and Guattari have described as itineration, rather than iteration. The craftsman is thus an itinerant wayfarer, whose practice is one of journeying with the material.

Ingold's work provides one of the most productive perspectives in contemporary discussion of materiality, and offers an ideal analytic perspective for the live coding situation. He applies Alfred Gell in identifying a kind of mistaken belief, in which an object is taken to be the starting point for an enquiry that traces backwards *from* the object to find the conditions and creative agent that caused it to exist. 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 familiar to the contemporary artist, for example as expressed in Paul Klee's classic evocation of drawing as 'taking a line for a walk.' It resonates equally well with the experience of the live coder, who is engaged in a process of programming, but with no intention to create a software product.

Ingold himself observes how different these material craft practices are from the 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 substance. If this is true, then surely code, as a rational rule system beyond all others, must be preeminently technological, and certainly not a craft material?

There are still computer scientists who resist the suggestion that computing might be a craft (Lindell 2014). The tension is so long-standing that even Babbage engaged in long-running dispute with Clement, the engineer building his Difference Engine, who claimed that he, rather than Babbage, should be recognised as its inventor (Cardoso-Llach 2015). Computer science is the domain of the gentleman academic rather than the rudely mechanical engineer, and its highest aspiration is to prove the correctness of its products in the manner of a mathematical theorem. Dijkstra's regret of the tendency for software development to be treated as a craft, rather than an automated and repeatable scientific discipline, follows in this line.

Nevertheless, 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 plan. Their practice, as with Suchman's situated cognition (1987), 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.

So while theorists of materiality in interaction design might argue that software is a design material like their other materials, and that where there is a material there must be a craft (Lindell 2014), an understanding of live-coding takes us in the reverse direction. Following the analyses of Ingold and Sennett, software construction is a craft - and given this craft, it seems that code must be its material. Its materiality arises from its fluidity. Through code, it seems that we have made language into a material, even though this material is insubstantial. Perhaps this is completely appropriate in an information economy and media society, whose products and commodities have also become insubstantial.

Furthermore, the 'conversation with materials,' that has been observed in craft and design practice by theorists such as Sennett and Schön, now becomes a more literal conversation 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 (Blackwell 2015). We can also appreciate a diversity of craft practices, extending beyond live coding to other communities of practice and other practices of programming (Bergström & Blackwell 2016).

3. Manipulate/Automate/Compose

My own research in creating the Palimpsest language deliberately followed an unconventional design process, as a strategy intended to generate novel approaches to existing problems. In particular, it was motivated by the creative design practice of "letting the material take the lead" (Michalik 2011), as elaborated in the previous section. Although the analogy between software and other "material" was always going to be problematic, there were two respects in which that strategy appeared feasible at the time.

The first was that reliance on the craft tradition embodied in specific professional tools allows the craft designer to carry out technically competent work without conscious critical intervention that might otherwise be an obstacle to innovation. In the context of Java development, this tool-embodied knowledge was obtained through the IntelliJ IDEA environment. IntelliJ has many productivity-assistance features that are clearly based on expert programmer practice, but integrated directly into the editor rather than via dependency analysis and structuring tools. Syntax-directed completion of identifier names, based on combined use of compiler symbol table, library data, and use of an English dictionary to recognize semantic structure in camel case identifiers, meant that most code was both rapid to enter, and correct at the time of entry. Large-scale refactoring tools, mainly providing syntactically-aware renaming of identifiers spanning filenames, class declarations and instance identifiers, made it possible to work fluidly with an evolving conceptual model of the system architecture and functionality. These development features have been introduced to professional tools through the demand for agile development practices, but this project suggests that they have also become an academic resource within a design research context.

The second point of comparison came from classic literature exploring "conversation with the material" from a cognitive perspective (e.g. Schön 1983, Schön & Wiggins 1992), through which design concepts would be refined by observing the development of a sketch or model. A standard account from design ideation research is that sketches are intentionally produced in a manner that allows ambiguous readings (Goldschmidt 1999), so that the designer may perceive new arrangements of elements that would not be recognized in verbal descriptions or internal mental imagery (Chambers & Reisberg 1985). In the case of the Palimpsest development, this "conversation" was achieved through using the system in development, not by systematic functional testing (although detection of bugs was an added benefit), but exploration of the artistic potential of the system. Since Palimpsest itself offered an alternative conception of programming, these experiences often provided insight into the system architecture and development process, not simply the user functionality.

I was concerned that to some extent, these craft practices simply resembled undisciplined programming – hacking, in the old terminology – as opposed to professional software design development. It is true that this project, as with much academic software development, was carried out using methods that differ from professional practice. However, as with academic research, there are also essential aspects of discipline that are a prerequisite of success. The first is an awareness of relevant theory and design precedents that inform an original research question (e.g. systems such as Sketchpad, Smalltalk, Garnet, Visicalc, Toontalk, Scratch and others), while the second is rigorous reflection on the work in progress. This need not be conducted in the manner of an engineering investigation – indeed, the most valuable findings were insights that occurred after periods of relatively unconscious reflection, and in informal journal entries (Blackwell & Aaron 2013).

However, the design intentions for the Palimpsest language were also strongly influenced by the earlier investigations that I had carried out into the cognitive demands of end-user programming (Blackwell 2006b, 2013). In particular, my theoretical approach to support of end-user programming was motivated by the Attention Investment model of abstraction use (Blackwell 2002). Although this is a cognitive model oriented toward design analysis, it does not directly offer design recommendations. Previous work by Wilson, Burnett et al. (2003) had operationalized the attention investment model in a specific design strategy for end-user debugging that they describe as "Surprise, Explain Reward". One objective of the Palimpsest project was to identify further concrete design guidelines of this kind.

Much of the Palimpsest design was motivated by the need for smooth transitions between direct manipulation and the definition of abstract behavior. This supported exploratory artistic practices, and also avoided the negative consequences of the Cognitive Dimension of *abstraction hunger*, as exhibited by many programming languages and tools. Reflection on development and use of

Palimpsest made it clear that there were in fact two transitions in the level of abstraction provided during system exploration. The first was between direct manipulation of a control on a Palimpsest image layer, and indirect manipulation of the same parameter via a value layer. The second transition is the composition of the behaviors created using value layers and references, by collapsing into collections, by copying, or by modifying the references with indirection layers.

This chain of attention investment transitions from direct manipulation to more complex automated functions and then to scripts had previously been explored in the context of the tangible programming system Media Cubes (Blackwell & Hague 2001), where domestic remote control buttons were used as tangible representations of the actions they controlled. The design intention, as in the Piagetian approaches to teaching mathematics or programming, was that users could build confidence in the physical elements of the representation through familiar concrete operations, using them as simple remote controls. Once those unit operations had become sufficiently familiar (perhaps over a period of months or years), the physical objects would naturally start to be treated as symbolic surrogates for those direct actions, and then used as a reference when automating the action (for example, setting a timer to invoke the relevant action). Once the use of references had become equally familiar, the user might even choose to compose sequences of reference invocations, or other more sophisticated abstract combinations.

In homage to the Surprise, Explain, Reward design strategy developed by Burnett's group, this approach to the transition between direct manipulation and programming functions can be described as Manipulate, Automate, Compose. The user is able to achieve useful results, and also become familiar with the operation of the system, through direct *Manipulation* that provides results of value. The notational devices by which the direct manipulation is expressed can then be used as a mechanism to *Automate* them, where the machine carries out actions on the user's behalf. Finally, all of the functions that the user interacts with in these ways can be *Composed* into more abstract combinations, potentially integrated with other powerful and/or complex computational functions.

In the case of Palimpsest, it was visual representation rather than tangible representation that was used as the familiar concrete element. Palimpsest was aimed at users who, although they may prefer not to use text in their work (cf Church et al 2012), are comfortable and expert in the use of pictorial representations. By applying minimal constraint on the content of those pictures, the system supports engagement of the same kind provided in drawing and painting tools. The facilities to automate and compose direct manipulations of the image are not imposed on the user at the outset (the Cognitive Dimension of *abstraction hunger*), but are available for use at any time (*abstraction tolerance*).

This design approach could, in principle, be applied to any interactive system that offered scripting or extension capabilities. In practice, most software products do not offer users the second two steps in this skill-development process - GUI users are expected to remain at the "Manipulate" phase, and are given little encouragement to move on to Automating and Composing - precisely the points at which computers offer real labour-saving potential. Programming by Demonstration systems (Cypher 1993) aim to facilitate transition from Manipulate to Automate, while Programming by Example systems (Lieberman 2001) uses additional inference methods to Compose automated functions over a range of invocation contexts. All of these transitions can in principle be systematized and characterized in terms of Tanimoto's 5th and 6th levels of liveness (2013).

It is interesting to note that theoretical approaches to programming language design generally proceed in the opposite order - the mathematical principles of language design as presented in programming language design textbooks (e.g. Finkel 1996) are fundamentally concerned with composition. The syntactic features of the language are recognized as having implications for usability, even though they may be "syntactic sugar" from a mathematical perspective. Once the mathematical and syntactic properties of the language are established, interface libraries provide facilities to support external functionality such as disk, network or graphics output. Finally, a programming environment is defined, in which the user is able to manipulate the syntactic notation to controls these elements. After a language has been in use for a while, live debugging environments might even provide the ability to directly manipulate objects of interest from the user domain, within the context of program development. The Manipulate/Automate/Compose strategy of Palimpsest is thus one starting point for a craft-oriented and user-centred, rather than computationally-centred, programming language design.

4. Build/Reflect/Notate - a Process for Crafting Languages

Despite the formal and abstract presentation of many programming language design textbooks, most programming language designers are motivated by a need to change user experience of programming – usually based from introspection on their own experience, or perhaps (if academics) from observation of conceptual problems faced by their students. As a result, the great majority of programming languages are designed for people who are already programmers. Research in end-user programming attempted to address this deficiency through the application and adaptation of user-centered HCI research methods for the programming domain – including development of user models, techniques for analytic evaluation and critique, and a mix of contextual and controlled user study methods.

The Palimpsest development explored an alternative approach, derived from the methods of practice-led design research. It relied on the availability of a new generation of "craft" tools for agile software development, enabling conceptual advance to be made in the context of prototype construction. Almost every aspect of the Palimpsest design, including its conceptual foundations, developed through reflection on the experience of building the system. The research outcomes can be attributed in part to insight and serendipity, in a manner that while probably recognizable to many research scientists, is not normally specified either in prescriptions of scientific method, or of user-centred software development methodology.

Apart from the craft tradition and tools that enabled discovery through construction, the central focus of the project was to understand a class of potential user (the digital artist) through doing (as a researcher) the same things that they do. In one sense, user-centred design has always valued the deep understanding of ethnographic participant observation. Indeed, the long-term research into artistic practices that informed the Palimpsest project had always been conducted in collaboration with academic anthropologists – initially as invited observers, then as project leaders and arts process researchers in their own right (Leach 2006, Barry et al 2008, Leach 2011).

However, this work stepped away from social science as a form of "requirements capture", back to the traditions of craft professions in which tools are made and adapted by those who use them, learning with the hands and from the materials. The Palimpsest project intentionally blurred the boundary of software development and creative exploration, and deliberately avoided many of the conventional practices in each field. Rather than interdisciplinary collaboration and consultation, the central period of the research was conducted in isolation, living and working in a remote forest location in New Zealand, to provide a self-imposed focus on the personal experience of design.

As a meta-strategy for programming language research, this could be described as a process of "build, reflect, notate", echoing the suggested end-user programming experience of "manipulate, automate compose" – but with the end result at this meta-level being a new notational convention for expressing and manipulating abstract computation. It is possible that this strategy may be particularly (or solely) suited to creating new programmable tools for use in artistic contexts. However the results of this project also suggest potential benefit of similar craft approaches in the design of end-user programmable and customisable products more generally.

5. References

Agre, P. E. (1997). Towards a critical technical practice: lessons learned in trying to reform AI. In Bowker, G. Star, S. L & Turner, W. (Eds) *Social Science, Technical Systems and Cooperative Work*, Lawrence Erlbaum, Mahwah, NJ, pp. 131-157.

Barry, A., Born, G., and G. Weszkalnys (2008). Logics of interdisciplinarity. *Economy and Society*, **37**(1): 20-49.

Basman, A. (2016). Building software is not (yet) a craft. In Proceedings of PPIG 2016.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Kern, J. (2001). Manifesto for agile software development.

Bergstrom, I. and Blackwell, A.F. (2016). The practices of programming. In *Proceedings of IEEE Visual Languages and Human-Centric Computing (VL/HCC)* 2016.

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. (2006a). The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4), 490-530.

Blackwell, A.F. (2006b). Psychological issues in end-user programming. In H. Lieberman, F. Paterno and V. Wulf (Eds.), *End User Development*. Dordrecht: Springer, pp. 9-30

Blackwell, A.F. (2013). The craft of design conversation. In A. Van Der Hoek and M. Petre, (Eds), *Software Designers in Action: A Human-Centric Look at Design Work*. Abingdon: Chapman and Hall/CRC, pp. 313-318.

Blackwell, A.F. (2014). Palimpsest: A layered language for exploratory image processing. *Journal of Visual Languages and Computing* 25(5), pp. 545-571.

Blackwell, A.F. (2015). Patterns of user experience in performance programming. In *Proc. First International Conference on Live Coding.* Zenodo. http://doi.org/10.5281/zenodo.19315

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. & Fincher, S. (2010). PUX: Patterns of User Experience. *interactions* 17(2), 27-31.

Blackwell, A.F. and Aaron, S. (2015). Craft practices of live coding language design. In *Proc. First International Conference on Live Coding*. Zenodo. http://doi.org/10.5281/zenodo.19318

Blackwell, A.F. and Hague, R. (2001). AutoHAN: An architecture for programming the home. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 150-157.

Cardoso Llach, D. (2015). Software comes to matter: Toward a material history of computational design. *Design Issues* 31(3), 41-54.

Chambers, D. & Reisberg, D. (1985). Can mental images be ambiguous? *Journal of Experimental Psychology: Human Perception and Performance* 11:317-328.

Church, L., Rothwell, N., Downie, M., deLahunta, S. and Blackwell, A.F. (2012). Sketching by programming in the Choreographic Language Agent. In *Proceedings of the Psychology of Programming Interest Group Annual Conference* (PPIG 2012), pp. 163-174.

Cypher, A., ed. (1993) *Watch What I Do: Programming by Demonstration.* MIT Press.

Dijkstra, E.W. (1977) Programming: From craft to scientific discipline. *International Computing Symposium*.

Finkel, R.A. (1996). *Advanced Programming Language Design*. Addison Wesley.

Goldschmidt, G: 1999, The backtalk of self-generated sketches, *in* JS Gero & B Tversky (eds*), Visual and Spatial Reasoning in Design*, Cambridge, MA. Sydney, Australia: Key Centre of Design Computing and Cognition, University of Sydney, pp. 163-184.

Gross, S., Bardzell, J., and Bardzell, S. (2013). *Personal and Ubiquitous Computing*

Hansen, N.B., Nørgård, R.T. and Halskov, K. (2014). Crafting code at the demo-scene. *Proceedings of Designing Interactive Systems (DIS),* pp. 35-38.

Ingold, T. (2010). The textility of making. *Cambridge Journal of Econometrics*, 34, 91-102

Leach, J. (2006). Extending contexts, making possibilities: an introduction to evaluating the projects. *Leonardo* 39(5), 447-451.

Leach, J. (2011). The self of the scientist: material for the artist. emergent distinctions in interdisciplinary collaborations' 2011. *Social Analysis* 55(3): 143-163.

Leuf, B., & Cunningham, W. (2001). *The Wiki way: quick collaboration on the web*. Addison-Wesley

Lindell, R. (2014). Crafting interaction: The epistemology of modern programming. *Personal and Ubiquitous Computing* 18, 613-624

McCullough, M. (1998). *Abstracting craft: The practiced digital hand*. MIT Press.

Michalik, D. (2011) Cork: Letting the material take the lead. *Core 77*, entry dated 4 Oct 2011. http://www.core77.com/blog/materials/cork_letting_the_material_lead_20707.asp [last accessed 27 Aug 2012]]

Pickering, A. (1995). *The mangle of practice: time, agency and science.* University of Chicago Press.

Reynolds, J. C. (1981). *The craft of programming*. Prentice Hall

Schön, D.A. (1983). *The reflective practitioner - how professionals think in action.* New York, Basic Books.

Schön, D.A. 1983, *The Reflective Practitioner: How professionals think in action*, Basic Books, New York, NY.

Schön, D.A. and Wiggins, G: 1992, Kinds of seeing and their function in designing, *Design Studies*, **13**:135-156.

Sennett, R. (2008). *The Craftsman.* Yale University Press.

Suchman, L.A. *Plans and situated actions: The problem of human-machine communication.* Cambridge university press, 1987.

Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming* (pp. 31-34). IEEE Press.

Wiberg, M. (2013). Methodology for materiality: Interaction design research through a material lens. *Personal and Ubiquitous Computing*, 1-12

Wilson, A. Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., and Rothermel, G. (2003). Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (CHI '03). ACM, New York, NY, USA, 305-312.