The Practices of Programming

Ilias Bergström
MobileLife group,
KTH Royal Institute of Technology
Stockholm, Sweden
iliasb@kth.se

Alan F. Blackwell
Computer Laboratory
University of Cambridge
Cambridge, UK
afb21@cam.ac.uk

Abstract— How diverse are the ways that programming is done? While a variety of accounts exist, each appears in isolation, neither framed in terms of a distinct *practice,* nor as one of many such practices. In this work we explore accounts spanning *software engineering, bricolage/tinkering, sketching, live coding, code-bending, and hacking*. These *practices of programming* are analyzed in relation to ongoing research, and in particular HCI's 'practice turn', offering connections to accounts of practice in other contexts than programming. The conceptualization of practice helps to interpret recent interest in program code as craft material, and also offers potential to inform programming education, tools and work as well as future research.

Keywords— Practices of Programming; Digital Craft; Material; Practice Turn; Material Turn; Parameter Mapping; Mutable Mapping; Creative Coding; Programming as Art; Tinkering; Bricolage; Sketching; Code-Bending; Hacking; Live Coding; Software Engineering; Education.

I. Introduction

A key achievement of the VL/HCC conference series has been the recognition that there are different kinds of programmers: people write programs for different reasons, and do so in different ways. The essence of a 'human-centric' approach is to better understand people's needs and behavior, hopefully allowing us to make tools better fitted to those needs. Past successes from this human-centered strategy have included better understanding of novice programmers (students who benefit from languages more specifically designed to help them learn), and of end-user programmers who are not specialists, but do programming tasks in the context of carrying out their regular jobs (e.g. a teacher making a spreadsheet grade book).

The goal of this paper is to look forward and ask how we might recognize and understand further human-centric models of programming. On the basis of past research achievements, we can anticipate what the interesting outcomes might be of such an investigation - we are interested in learning more about different ways in which people approach programming, leading to new strategies, new preferences, and new design trade-off choices for software tool developers. In order to gain a new perspective, we draw here on a current trend in HCI research more broadly described as "the practice turn" [1]–[3].

II. Background

Kuutti & Bannon [3], coin the term *"interaction paradigm"*, to refer to HCI studies focusing on momentary, ahistorical situations, disconnected from a particular time and space. Such research traditionally draws from the methodology of psychological sciences, using controlled, short-term laboratory studies. This contrasts to the *"practice paradigm"*: studying longer-term actions, situated in time and space, and richly dependent on their material and cultural environment. Such research methodologically derives from design and social sciences, involving qualitative, observational modes of knowledge production. The focus is widened, to *"studying an overall activity, involving people, artifacts, organizational routines and daily practices"* [3]. The shift from the interactional to the practice paradigm, then, is what constitutes HCI's practice turn.

The practice turn is influenced by theories of tool-use, the nature of knowledge, the structuring of society into professions, and other dynamics in philosophy and social science. In this paper, we will draw on these theoretical developments, but will also pay close attention to the existing literature and communities in which people have already described the practices of programming. We acknowledge and celebrate the success of VL/HCC research into novice and end-user programming, but in this paper, we wish to look elsewhere, understanding other descriptions of what the practices of programming might be.

We believe that the practice turn will allow us to explore some critical recent questions. At VL/HCC 2015, Aghaee et al opened up the very general question of *why* people engage in programming, through broad analysis of motivations and personality, challenging and extending beyond typical conceptions of why programmers program [4]. In this paper we might be said to consider *how* people do programming, once again extending beyond typical conceptions - or perhaps more provocatively (because one might think we know the answer already), *what* people do when they are doing programming. This echoes challenges to foundational questions such as 'What is Programming' [5]? However, where earlier research of that kind has considered programming as an individual and cognitive activity, our analysis in this paper makes no such assumptions.

The practice turn in HCI derives from work such as that of Lave, who observed that practice consists of activities situated in a social and material world [6]. Rather than explicitly articulated 'knowledge', professional practice is a 'craftwork' in which knowledge is constructed and transformed in use, and always complexly problematic. Lave warns that things assumed to be natural categories, such as "bodies of knowledge," "learners," or "cultural transmission," require reconceptualization as cultural, social products. In order to escape previous conceptions, we therefore adopt new research approaches, complementary to existing concerns of VL/HCC.

978-1-5090-0252-8/16/$31.00 ©2016 IEEE

III. Methods

Our method in the current piece of research is analytic, rather than empirical. The world of practice is one of embedded and lived experience, not always directly accessible to observation and measurement. As a result, the turn to practice in HCI has been associated with giving priority to the reflective writing of practitioners, and more recently with the traditional analytic method of the 'essay,' itself situated within an existing body of humanistic literature [7]–[9]. Our evidence in this work is therefore based in accounts of practice, from as broad a perspective as we could find, with the intention of discovering new insights beyond the well-established achievements in VL/HCC and cognate human-centric fields (such as empirical studies of software engineering, and psychology of programming).

We aimed not to be distracted by definitional questions, taking the inclusive approach to definitions of programming that has been advocated in previous research such as [5]. For the purposes of HCI research, Kuutti and Bannon describe practices as "*(…) routines consisting of interconnected and inseparable elements: physical and mental activities of human bodies, the material environment, artifacts and their use, contexts, human capabilities, affinities and motivation. Practices are wholes, whose existence is dependent on the temporal interconnection of all these elements, and cannot be reduced to, or explained by, any one single element*" [3].

As source material for our analysis, we therefore collected accounts of how programming is done, using an inclusive interpretation of the term, but noting ways in which each practice described is explicitly delineated and named by the authors. We considered books, journals and conferences in specialist fields extending to software engineering, human computer interaction, interaction design and digital humanities, as well as emergent interdisciplinary ventures such as Creative Coding [10] and Art-Science [11].

IV. Analytic Context

During this process, we organized our observations according to some broad categories informed by the historical development of distinctive programming practices. In our overview of the practices in the next section, we will group these under the terms *Software Engineering*, *Bricolage/Tinkering*, *Sketching*, *Live Coding*, *Hacking*, and *Code-Bending*. We will also consider the relationship of these practices to particular kinds of tool, through consideration of the *affordances* of those tools. However, before reporting those findings, we provide a brief overview of the historical and theoretical contexts for our analysis.

A. Software Engineering and Cognitive Ergonomics

The term *Software Engineering* was coined in response to the 1960s 'software crisis', and advocated that the increasing difficulty of delivering software within budget should be addressed by more disciplined, systematic software development and maintenance [12], [13]. The acknowledgement that software development was a human problem soon led to a concern with understanding human performance. This branch of human factors research moved beyond previous research into physical ergonomics, accuracy and reaction times, to address 'cognitive ergonomics' of the machines. Interdisciplinary investigations into the Psychology of Computer Programming [14] developed from the understanding *"(…) that programming tools and technologies should not be evaluated based on their computational power only, but also on their usability from the human point of view, that is, based on their cognitive effects"* [15].

Concern with improving the efficiency of human performance was associated with emphasis on methods for creating and reusing standardized engineering components, extending beyond program code, to processes and models [12]. Standardized *processes* proposed ways to organize the formulation of requirements, design, implementation, testing and maintenance of each new software system [16]. Different models of component interaction were associated with alternative *paradigms* for expressing problem structure such as Structured, Declarative, Object Oriented, Event-Driven, or Dataflow. Specific processes and paradigms were often encapsulated in software development *tools* extending beyond the programming language to support libraries, debuggers, analyzers, modelers, unit test tools, Integrated Development Environments (IDEs), Rapid Application Development (RAD) tools, etc.

B. Beyond Software Engineering

The earliest development of software engineering was associated with an assumption that direct interaction with computers would remain a specialist technical task. Books such as Weinberg's *Software Psychology* [14] made little distinction between user and programmer. However, the increasing ubiquity of computers that came with decreasing size and cost led to an understanding that computers would be used for purposes beyond technical ones, and to a research concern with the extent to which computer users of all ages might be able to become programming-literate, as in Alan Kay's Dynabook proposal [17, p. 393].

The Smalltalk system created by Kay and Goldberg did indirectly lead to greatly increased computer usage through its introduction of the WIMP paradigm (Windows, Icons, Menus, Pointer), but the resulting direct manipulation interfaces ironically became increasingly distanced from programming. The efforts towards reintroducing automation and abstract functionality into this more graphical environment, have become associated with End-User Programming (EUP), and a full-circle return to End-User Software Engineering that offers more systematic processes and tools to non-experts [18]. For our purpose, the key question is whether such tools are associated with novel *practices* distinguishable from software engineering.

The Smalltalk project placed an early emphasis on use of the computer as a creative tool, rather than simply engineering and business. The painting, publishing and music applications that Kay and Goldberg envisaged are now ubiquitous. The use of programming in professional arts contexts has also increased in popularity, and has been described as *creative coding* [10]. This development naturally follows the many algorithmic artforms predating the computer, including textile weaving, Islamic and Celtic decorative patterns, or musical change-ringing. In modern times, designers and artists have followed

in the footsteps of pioneers Ben Laposky and John Whitney [19], creating procedural art using computers.

Finally, the study *of* programming (as opposed to its application) has also in recent years extended beyond the technical concerns of software engineering. The broad area of Digital Humanities concerns itself with all research where the fields of computing and the humanities intersect, often also involving the study of programming and program code, as in the fields of Software Studies, Computational Culture or the code aesthetics of M.J. Black [20]. *Aesthetic computing*, or "the application of the theory and practice of art to the field of computing" [21], widens the scope of aesthetics in computing, emphasizing how artistic aesthetics may inform all computing practice.

C. Affordances in Programming

These historical shifts in audiences and objectives are reflected in the programming paradigms, languages and tools that have developed to support them. Each of these ensembles is characterized by a distinctive set of properties. We describe these joint technical properties of paradigms, languages and tools as *affordances*, used here in the original sense introduced by Gibson [22]: a relation between an object or environment and an organism, that affords the opportunity for that organism to perform an action. McCullough [23] discusses affordances as "*the workable capacities in a medium",* relating them to the notion of a medium's *constraints*. Together these form a medium's structure of expression, establishing what that medium can and cannot be used for.

A familiar example illustrating the constraints and affordances of different software environments is the trade-off between those that afford high computational efficiency on one hand and those that prioritize rapid development and experimentation on the other. The dataflow paradigm affords fluid manipulation and reconfiguration of signal flow, but limited opportunity to work with the kinds of dynamic data-structure traditionally handled with recursive algorithms [24] Programmers might attempt to circumvent such limitations by implementing some components in a language using a different paradigm, but system design spanning multiple paradigms quickly becomes cumbersome.

Two particularly interesting affordances are Directness [25] and Liveness [26] *"Directness means a user interface designer can initiate the process of examining or changing the attributes, structure, and behavior of user interface components by pointing at their graphical representations directly, as opposed to navigating through an alternate representation. Liveness means the user interface is always active and reactive - objects respond to user actions, animations run, layout happens, and information displays are updated continuously. Directness and liveness are properties of the physical world: to examine and change a physical object, you manipulate it directly while the laws of physics continue to operate"* [27]. The implications of these affordances for programming more generally have been explored by Tanimoto [28], and have currency through increasing interest in environments that allow changes to be made and observed in a program already executing [29]. These affordances, which have long suggested that the boundary between interaction and programming needs to be redefined, are gaining further attention in HCI [30].

V. The Practices of Programming

We now describe the broad categories of distinctive programming practices that we have identified in the literature spanning these historical and theoretical contexts.

A. Established Software Engineering Practice

By this term, we refer to the practice discussed in the previous section, where, in some form, a specification has been formulated, and is implemented, evaluated, and refined, through defined phases, at varying levels of granularity depending on which of the many existing software engineering processes is followed. In the 'waterfall model' this cycle is supposed to be repeated once, in the 'spiral model' [31] several times, while in 'agile development' [32] it may be repeated with intervals of weeks or even days. Nevertheless, the notion of phase and progression, and of distinction between specification and implementation, is always present. This established practice is well-documented, so we include it for completeness of analysis rather than for new insight.

B. Bricolage & Tinkering

The earliest suggestions that an alternative approach to programming might be desirable, different to that of professional software engineering, appeared in the context of teaching programming to the general public, and particularly to children. What has later been named Bricolage and Tinkering approaches were advocated by the creators of the first programming languages intended for teaching, e.g. LOGO [33], Smalltalk [34], and their many descendants.

Turkle and Papert [35] introduced the notion of Bricolage programming, adopting the term from anthropologist Lévi-Strauss [36]. Turkle and Papert provide the following definition: *"The bricoleur resembles the painter who stands back between brushstrokes, looks at the canvas, and only after this contemplation, decides what to do next. For planners, mistakes are missteps; for bricoleurs they are the essence of a navigation by mid-course corrections. For planners, a program is an instrument for premeditated control; bricoleurs have goals, but set out to realize them in the spirit of a collaborative venture with the machine. For planners, getting a program to work is like "saying one's piece"; for bricoleurs it is more like a conversation than a monologue."*

Note that the loop between distinct phases of designing and then implementing is largely done away with. There may well be no plan at all in bricolage, or the plan may be made up at the very instant that the program is entered. The process is as much happy accidents, trying things out and seeing what happens, as it is deliberate action with a particular outcome in mind. The practice has also been related to the word *tinkering* [37], [38], a casual kind of mechanical play or dabbling that is associated with amateurs and hobbyists, but is also characteristic of curiosity and invention. The tinkerer is not an engineer, and may be unsure of how material will react. She learns through action, trying different manipulations and tools, and responds as her material takes different forms.

C. Sketching with Code

In art and design, sketching is central to how ideation is carried out [39]. Through freely and quickly trying out a large number of variations of a loosely defined idea, a more concrete conception of what is desired takes form. The sketches can either then be discarded, having served their purpose, or chosen ones kept, if they sufficiently represent their author's communicative intent. Note the crucial difference to the role of prototypes in software engineering, where in processes such as Boehm's spiral model [31], or agile development [32] a prototype is developed as a single, well-thought out design that is to be evaluated for its soundness, with its re-evaluation occurring through a clear cycle, and where each iteration of the prototype serves as the concrete representation of the continuously refined design under evaluation.

An early advocate of sketching as an approach to programming was Miller Puckette, originator of the visual languages Max/MSP and Pure Data that are now widely used for music and other creative applications [40]. Puckette states that he emphasized the sketching analogy by presenting users with a blank canvas, on which the dataflow program can be incrementally built up as a directed graph (Figure 1). Sketching also features prominently in the Processing language and environment [41], which expands on ideas in John Maeda's *Design by Numbers* language, created for teaching the *"idea of computation to designers and artists"* [42]. Others have since created alternative programming environments with sketching specifically in mind, e.g. Baader and Bødker [43] and Blackwell [44]. In all of these environments, the goal has been to facilitate creative practices in programming, by analogy to artists and designers working in traditional media.

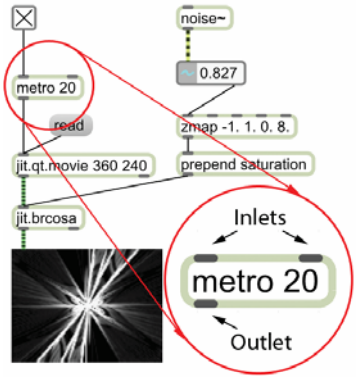

Figure 1 – The visual language of Max/MSP 'patches' were described by Miller Puckette as supporting sketch (this example created by IB).

In the words of the Processing authors [45]: *"It is necessary to sketch in a medium related to the final medium so that the sketch can approximate the finished product. Painters may construct elaborate drawings and sketches before executing the final work. Architects traditionally work first in cardboard and wood to better understand their forms in space. Musicians often work with a piano before scoring a more complex composition. To sketch electronic media, it's important to work with electronic materials. Just as each programming language is a distinct material, some are better for sketching than others, and artists working in software need environments for working through their ideas before writing final code. Processing is built to act as a software sketchbook, making it easy to explore and refine many ideas within a short period of time".*

D. Live Coding

The implications of liveness are taken to an extreme in the practice of live-coding, where artists write code as a means of performance, with an audience experiencing the output of the program at the same time as viewing a large-screen projection of the continuously modified program code [46]. Predominantly, live-coding is used in musical performance, but visual or audiovisual performances are not uncommon (Figure 2), and there is nothing to keep the practice from being applied to any other context in which the output of generative algorithms is presented to an audience (e.g. dance, textiles) [47].

Live Coding is closely reliant on the availability of a programming language that affords liveness. Otherwise, it is impossible to program and immediately perceive the result, without interrupting output and re-executing the program. So, live-coding currently requires the use of specialized programming environments capable of interpreting the code on the fly as it is entered by the performer, without restarting or recompiling the whole program. While several environments already had this capability (including Max/MSP and Pure Data), many have been created specifically with live-coding in mind (for example ChucK, Impromptu, Fluxus and the JITlib facility for SuperCollider [48] as well as a large number of more recent examples).

Perhaps uniquely to live-coding, this programming practice is not carried out with the ultimate goal of realizing some design outcome, but is instead a continuous performance, with the journey itself being the principal intended outcome. To stress this point, early live-coding performers often ended their acts by purposefully breaking what they had created: inserting faults into their code, which crash, disrupt or delete their program, ideally producing interesting visual and audio glitch effects as it dies.

Figure 2 – Benoit and the Mandelbrots, from the Supercollider Symposium 2012 "Livecode Evening": *"Codefaced people hacking music live in front of your eyes. Live coding is a new direction in electronic music and video: live coders expose and rewire the innards of software while it generates improvised music and/or visuals"* (Image copyright Steve Welburn, CC BY-SA).

E. Hacking

The word *hacking* has often been used as a casual reference to an informal style of programming. However, it has been

appropriated for many other purposes, extending to criminal activity, political ethos, technocratic subculture, critical theory and others [49]. We are concerned specifically with the *practice* of hacking, and not these many other senses. Even then, the term may vary greatly in definition depending on context. Erickson [50] writes: *"Hacking is the art of creative problem solving, whether that means finding an unconventional solution to a difficult problem or exploiting holes in sloppy programming. Many people call themselves hackers, but few have the strong technical foundation needed to really push the envelope"*.

Informal practices of creative problem solving, while an important aspect that is consistent with the historical origins and self-identity of 'hacker' communities, overlap with the other categories of practice we have identified. It is therefore useful to focus specifically on the second sense identified in Erickson's definition: modifying or otherwise interfering with a pre-existing piece of software, in order to make it perform differently from what its original designers intended. This specific practice can be seen as an originating concept for the many other interpretations of the word 'hacker': hacking requires deep understanding of computers and software; it may involve machine-level programming (to intervene when source code is protected or hidden); this might be legitimate reverse engineering [51], but in many cases will be illegal, violating license agreements or copyright legislation. These illicit connotations of hacking practices give the word a romantic 'underground' flavor, providing counter-cultural and anti-authoritarian anarchist credentials even for mainstream and publicly-funded artists (see for example the 2011 Netherlands Media Art Institute exhibition "the art of hacking" [52]).

F. Code-Bending

This practice is named by analogy to *Circuit-Bending*, a term coined by Reed Ghazala [53] to refer to creative experimental modification of an electronic device. One well-known circuit-bending target is Mattel's *Speak & Spell*, which can be changed into a musical instrument producing otherworldly vocal sounds (Figure 3). It is not assumed that circuit benders understand how the circuitry at hand works: although such knowledge is undeniably beneficial, lack of training is not a barrier to entry. Ghazala refers to his technique as anti-theoretical: not in the sense of rejecting theoretically informed practice, but providing a complementary alternative to it.

Where circuit-bending opens up a plastic case to access electrical connections, in code-bending [54], the internal API of open-source software is re-purposed, so that instead of fulfilling its originally intended internal function, it is used for external communication. While internal interfaces are not usually accessible in closed-source programs, with open-source software, one can figuratively "lift the lid" to experiment with these interfaces, even if the APIs are not fully documented. Existing software can thus, in a comparatively rapid, playful manner, be repurposed, encouraging an explorative approach to implementation.

Figure 3 – A Circuit-Bent Speak & Spell, as exhibited at the London Science Museum. Switches, buttons and potentiometers have been added to the device, controlling the modifications made to its circuits (Image copyright Loz Pycock, CC BY-SA).

Code-bending may involve *mapping* of variables across separate parameter spaces [55], or *mutable mappings* [56] that are gradually altered, created and destroyed. Code-bending is conducted in two phases: first exposing the previously inaccessible contact points in open-source programs, for example using the *Open Sound Control* (OSC) protocol, and then executing these programs, so that while they are running, one can experiment with altering mappings and adding data sources; either in search of a new mapping to subsequently finalize, or continuously, as a form of performance.

Code-bending has points of similarity to *Opportunistic Software Development* [57] through integration and re-use of existing systems; and to *Mashup* programming [58], an approach to end-user web-development by combining data and/or functionality from different, originally unrelated online data sources or services.

VI. Application

We do not expect that the above survey of practices will be definitive. However, it does provide sufficient range to assess the potential value that might be realized from describing programming as practice. In particular, whether or not the categories we have identified represent a complete set, we have confirmed that a diversity of practices does exist. These offer the potential for new insights, with regard to alternative practices that might be borrowed or adapted by other kinds of user – for example, even expert software engineers will on occasion need to engage in creative exploration, or tinker with an unfamiliar tool or undocumented API. The following observations consider ways in which these findings might therefore be applied, and what new insights might be gained.

A. Practices are Best Practiced in Combination

These practices need not be mutually exclusive. A software engineer might identify parts of a project where she cannot write a specification, so instead sketches out alternative ideas. During live coding performance, the second (mapping) stage of code-bending may take place as part of the act. Code-bending might also be used as a sketching technique, or as a design evaluation strategy within a software engineering process.

Finally, a programmer might combine tinkering and hacking: she tinkers with a piece of closed source software to familiarize herself with the material at hand, having the goal of then hacking, exposing functionality originally intended to remain inaccessible. Further combinations are certainly possible.

B. Social Relations Around Practice

While software engineering might in principle be practiced by a single person working in isolation, much of its literature is concerned with how groups can successfully coordinate their efforts [59]. In contrast, some of the practices identified within creative fields are developed and refined by individual practitioners pursuing personal objectives, with individual creative attribution of the process and outcomes.

There is also wide variation in the audience for the programming effort, with the intended audience influencing the practices that are chosen. In software engineering, the 'audience' is often a paying client, and then perhaps the end-users to whom the client intends to deliver the product. The resulting expectations in this set of commercial relations (ease of use, quality, value for money, etc), are worlds apart from the requirements and expectations of visitors at a temporary interactive installation, a permanent museum exhibit, or in the audience of a live-coded 'algorave'.

C. The Boundaries are Unclear: is Debugging a Practice?

There are cases where distinctions are less clear-cut: is *debugging* a programming practice in itself, or a sub-practice of software engineering, or perhaps a superset of practices? The act of finding a bug is very different to writing the code. But then, debugging can be carried out in many ways – does this make it a superset of distinct alternative practices [60]? On the other hand, a mistake in bricolage programming, sketching, or live coding, may never need to be debugged. If it gives rise to interesting results, the programmer might want to understand what happened, and to harness it - but would this still be debugging? And if a live coder makes a mistake, she needs to carry on with the performance. She has no time for debugging, but must instead improvise, incorporating the mistake into the performance. All of the above are equally valid, thus making debugging difficult to singularly characterize across all contexts.

D. Away from the Process Cycle: Programming as Craft

It is essential to software engineering processes that a specification is formulated, describing what the programming effort should achieve. In some approaches the specification is expected to be more refined than in others, and the frequency at which this goal is reviewed and revised throughout the process varies wildly between the waterfall model at one end, and extreme programming at the other. But *specification always precedes the act of programming a solution.* Not defining a specification for the programming effort is considered very bad practice, referred to with derogatory terms such as 'Cowboy Coding' [12]. The dichotomy has even been formulated, between the 'correct' way of doing things, and its antithesis, the incorrect approach, pointedly referred to as *craft*. Dijkstra declared that programming is a *discipline and not a craft* [61], precisely to stress how programming *shouldn't* be done [12], [13].

In contrast to this justifiable engineering philosophy, the emergence of the Interaction Design (IxD) field has encouraged an explicit shift in focus, toward programming as the *craft* [62] that is associated with a new design practice [63], responding to the pragmatic engineering challenge that requirements of interactive artefacts cannot always be defined *a priori* [12].

E. Links to Craft Practices in HCI and IxD

The themes of this paper have emerged from consideration of the practice turn as a matter of current concern in HCI. It is often the case that application of HCI principles to programming draws attention to specific opportunities that can provoke further innovation in HCI tools themselves. For this reason, we believe that there is useful potential for comparison of this work to recent thinking on craft practices in IxD, such as Vallgårda and Fernaeus' [64] discussion of bricolage, Buxton's account of sketching [65], and understanding of specific materials, such as Bdeir's account of sketching with electronics [66]. There will certainly be overlap with these accounts, but we should not expect that the practices for programming will translate to IxD, or to its various composing disciplines and materials, without further research.

F. Practices and Materiality

One specific theoretical concern in HCI, to which this account of programming practices does contribute, relates to the materiality of program code [67]. The perspective in which IxD, as a design discipline, incorporates a craft element, intersects with the observations that we reported from Lave at the start of this paper, situating practice in terms of craft knowledge that is embedded in a material context.

The interest in materiality for HCI more broadly is derived from ubiquitous, tangible and embodied computing. Work in these domains draws increased attention to craft aspects in IxD, as part of a *material move* [68], [69]. Craft practitioners interact with, learn about and reflect on their materials. In this context, questions are raised about the materiality of program code, with much discussion, and divergent views. Vallgårda & Redström [70], posit that program code has to be *"(…) combined with other materials to come to expression as material"*. Löwgren and Stolterman, describe information technology as a *"material without properties"* [71]. A contrasting conception of material is advanced by Lindell [67], and by Dourish and Mazmanian [72]: material as an abstract construct, grounded only in its usefulness to the practitioner for understanding her practice, and its usefulness to the perceiver/consumer in making sense of the result. Material is in other words not manifest in some physical reality, but is a purely mental construct. Blackwell and Aaron suggest that even this abstract materiality exhibits a resistance to the intentions of the programmer, generating new knowledge through the 'mangle of practice' [73]. These perspectives, as revealed in programming practices, can be inspected through the lens that Ingold [74] describes as the "two faces of materiality": *"On one side is the raw physicality of the world's 'material character'; on the other side is the socially and historically situated agency of human beings who, in appropriating this physicality for their purposes, are alleged to project upon it both design and meaning in the conversion of*

naturally given raw material into the finished forms of artefacts".

Adopting Ingold's terms, it is this second face of materiality that we find particularly relevant to program code: collecting around it a set of socially constructed traditions and connotations, properties towards its use, and the *practices* these afford. It is here that attention to practices makes a contribution, because the understanding of a material is incomplete without the practices in which it is employed. These constructs are useful for informing practitioners' work, reflection and discussion. They also provide observers with another lens, a way in, to understanding the work of the craftsman.

Treating software as a material brings the consequence that the granularity of material is not fixed, but depends entirely on the intent of the practitioner, and context at hand. It may be a composite material, comprising physical materials, electronics and code; it may be written in a number of programming languages; or it may simply be a single program. In the digital realm, it is rarely clear where the digital tools applied to the digital material end, and where the digital material begins [23]. But as observed by Ingold, what is most interesting is not what a material *is,* but what it can *do* in the hands of an artisan.

G. Embodied Programming Practices for Physical Domains

While the immaterial-materiality of program code is subtle and disputed, an equally neglected but undisputable consideration is the fact that the programmer does have a body. The embodiment of the programmer can be brought to the fore, when developing for and through various forms of physical performance during the development [75]. When creating a system for a domain which requires full body interaction, whether a golf-training simulator, or a software instrument for live musical performance, how do the practices from the application domain inform the choice of which programming practices to employ during development? Assuming the developer does acquire distinct practices from the application domain, she might then adopt different programming practices as lenses through which to better her understanding of the application domain. After reflection, these application domain practices may even inspire new programming practices, physical or otherwise.

H. Practices of Programming in Education

Mathematics educators understand that there is a huge gulf between the subject as it is taught in the classroom, and the mathematical practices of the outside world [76]. This recognition, that a body of knowledge is associated with a diverse set of professional and life practices, is essential in making the transition from teaching a specialist elite subject to a broad population literacy.

Even for students who do intend to become professional specialists, whether software engineers, interaction designers, or artists, exposure to a range of different practices will enhance their development as reflective practitioners. They might start by tinkering during early stages of familiarization with a new language; sketch to understand different ways that a problem can be approached (possibly coding live, for especially fluid feedback); then apply software engineering methods to structure a large and complex development effort. Understanding these practices from early on will help dispel the misunderstanding that the only correct way of programming is software engineering – and instead convey that the choice of which practice to use is contingent on the goal adopted in a particular programming effort.

The first author has designed an introductory programming course that applies concepts of varying programming practices early on, with students' activities throughout the course structured around the practice most relevant to that stage of the learning activity. It is designed as a contextualized course [77], [78], with all activities grounded in an application domain that provides students with a meaningful context for their learning. It currently uses the Processing language, facilitating applications with interactive graphics and sound. The course benefits from existing teaching materials for Processing that emphasize contextualized learning, such as Daniel Shiffman's books *Learning Processing* [79] and *The Nature of Code* [80].

The intention is that explicit description of the practices of programming, discussing the contexts where they might be advantageous, will allow students to use these reflectively, both separately and in combination. Findings from this teaching experiment will be published in the future.

I. Contrasting and Comparing Practices

In the previous section, we discussed how programming practices can be used in combination. It is also interesting to reflect on how the practices differ, and over which dimensions: What differentiates tinkering from hacking? Tinkering from sketching? Hacking from code-bending? In this section, we draw some comparisons.

In tinkering, achieving a final product is usually a secondary objective, if it is a goal at all. With no intended target-state for the material to reach, people engage in tinkering to gain familiarity with material. In hacking on the other hand, there is a clear goal: exposing and taking advantage of "exploits" in closed source software.

While in tinkering the material is unfamiliar to the practitioner, in sketching, it is the end-result that is unfamiliar – a goal is acknowledged, but underspecified. So a programmer that sketches is familiar with her materials and tools, and rather than exploring them, is exploring a design space.

Hacking addresses a closed codebase, a program binary not meant to be altered. Code-bending on the other hand, uses code that is open to read and modify, and instead concerns itself with finding new ways in which the code can be used.

With the distinctions outlined above as a starting point, we find the following dimensions across which practices vary.

- *What the end-goal is*: perhaps to learn, whether about the material (tinkering) or about the design space (sketching); perhaps to create a final product (e.g. engineering); or perhaps to produce an experience, with no end product at all (e.g. live-coding).
- *The extent to which the end-goal is defined*: in engineering the goal specification may span thousands of pages; in sketching, it may be a brief, abstract description of a

design space; in live-coding, it may range from a carefully rehearsed performance, to free improvisation; and in tinkering, there need not even be an end goal at all.

- *The extent of programming effort, over time, and in the size of the resulting codebase*: an engineering effort can extend over decades, producing millions of lines of code; sketching can last weeks or days, producing thousands of lines; tinkering and live-coding may last only a few minutes, producing no codebase whatsoever.
- *The extent of familiarity with codebase and tools*: varying greatly from very low (tinkering), to very high (hacking, with respect to the tools but not codebase, and engineering, with respect to both tools and codebase).
- *The relation between intended and subsequent use of the program*: In engineering, the intended use is defined, and a program is made to satisfy this specification. A hacked program on the other hand, is definitely not used as originally intended. And, while a code-bent program starts from open-source code, the resulting use was not one this code was originally designed to cater for.

A dedicated examination of the dimensions across which programming practices vary is most certainly needed, but until that research is done, we let the above points stand as initial groundwork towards such an effort.

VII. Conclusions

In summary, what do we gain from explicitly talking about *Practices of Programming*, as an umbrella term under which to identify, name and gather information about the specific characteristics of distinctive practices?

The descriptions and analyses above seem to be usefully orthogonal to earlier considerations at the VL/HCC conference of different kinds of programmers (e.g. novices and end-users), and different motivations for doing programming (e.g. in terms of attention investment and personality types).

Consideration of programming practices has also shown us that, although the social and craft context of these practices can be identified with previously identified research concerns (such as data flow languages for sketching or level 4 liveness for live coding), specific practices are in no way constrained to a particular user community or tool set.

On the contrary, analysis of practices draws attention to new opportunities for reflective discussion about the act of programming, and informed choices of what practices to follow during the programmer's development as an artist, craftsman or engineer, and at different stages of a particular project.

The practices of programming are a useful lens, in other words, on the one hand for programmers and educators to consciously choose how to best move through the phases of a project and through their own career development, and on the other hand for reflective practice, towards expanding our understanding of what the activity of programming may entail.

Acknowledgments

IB's work has been supported by an EU Presenccia grant, Spanish INNPACTO Melomics project IPT-300000-2010-010, and a Swedish ICT TNG grant "Music in Somaesthetic Design". AB's work has been supported by a grant from the Boeing Corporation. The authors would like to thank IB's colleagues at KTH, for their comments on earlier versions of this article.

References

[1] Y. Fernaeus, J. Tholander, and M. Jonsson, "Towards a new set of ideals: consequences of the practice turn in tangible interaction," in *Proceedings of the 2nd international conference on Tangible and embedded interaction*, 2008, pp. 223–230.

[2] R. Miettinen, D. Samra-Fredericks, and D. Yanow, "Re-turn to practice: an introductory essay," *Organ. Stud.*, vol. 30, no. 12, pp. 1309–1327, 2009.

[3] K. Kuutti and L. J. Bannon, "The turn to practice in HCI: Towards a research agenda," in *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, 2014, pp. 3543–3552.

[4] S. Aghaee, A. F. Blackwell, D. Stillwell, and M. Kosinski, "Personality and intrinsic motivational factors in end-user programming," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, 2015, pp. 29–36.

[5] A. F. Blackwell, "What is Programming," in *14th workshop of the Psychology of Programming Interest Group*, 2002, pp. 204–218.

[6] J. Lave, "The practice of learning," *Contemp. Theor. Learn.*, pp. 200–208, 2009.

[7] K. Williams, "An Anxious Alliance," *Aarhus Ser. Hum. Centered Comput.*, vol. 1, no. 1, p. 11, 2015.

[8] J. Bardzell, "HCI and the Essay: Taking on 'Layers and Layers' of Meaning," presented at the CHI 2010 Workshop on Critical Dialogue, 2010.

[9] J. Bardzell, *Humanistic HCI*. Morgan & Claypool Publishers, 2015.

[10] J. Maeda, *Creative code*. Thames & Hudson London, 2004.

[11] J. Ox, "Art-Science Is a Conceptual Blend," *Leonardo*, vol. 47, no. 5, pp. 424–424, 2014.

[12] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 12–29.

[13] N. Wirth, "A Brief History of Software Engineering," *IEEE Ann. Hist. Comput.*, vol. 1, no. 3, pp. 32–39, 2008.

[14] G. M. Weinberg, *The psychology of computer programming*. Van Nostrand Reinhold New York, 1971.

[15] J. Sajaniemi, "Psychology of programming: Looking into programmers' heads," *Probl. Prof.*, p. 3, 2008.

[16] I. Sommerville, *Software Engineering*, 9th ed. Addison Wesley, 2010.

[17] N. Wardrip-Fruin and N. Montfort, *The NewMediaReader*, vol. 1. MIT Press, 2003.

[18] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The State of the Art in End-user Software Engineering," *ACM Comput Surv*, vol. 43, no. 3, pp. 21:1–21:44, Apr. 2011.

[19] "Digital Art Museum (DAM)." [Online]. Available: www.dam.org. [Accessed: 11-Oct-2012].

[20] M. J. Black, "The art of code," 2002.

[21] P. A. Fishwick, Ed., *Aesthetic Computing*. The MIT Press, 2008.

[22] J. J. Gibson, "The theory of affordances," *Hilldale USA*, 1977.

[23] M. McCullough, *Abstracting craft: The practiced digital hand*. MIT press, 1998.

[24] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput Surv*, vol. 36, no. 1, pp. 1–34, 2004.

[25] B. Shneiderman, "Direct manipulation: A step beyond programming languages," in *ACM SIGSOC Bulletin*, 1981, vol. 13, p. 143.

[26] S. L. Tanimoto, "VIVA: A visual language for image processing," *J. Vis. Lang. Comput.*, vol. 1, no. 2, pp. 127–139, 1990.

[27] J. H. Maloney and R. B. Smith, "Directness and liveness in the morphic user interface construction environment," in *Proceedings of the 8th annual ACM symposium on User interface and software technology*, 1995, pp. 21–28.

[28] S. L. Tanimoto, “A perspective on the evolution of live programming,” in *Live Programming (LIVE), 2013 1st International Workshop on*, 2013, pp. 31–34.

[29] D. Ungar and R. B. Smith, “The thing on the screen is supposed to be the actual thing,” in *Proceedings of LIVE 2013, Workshop on Live Programming*, San Francisco, CA, 2013.

[30] J. Hook, G. Schofield, R. Taylor, T. Bartindale, J. McCarthy, and P. Wright, “Exploring HCI’s Relationship with Liveness,” in *CHI ’12 Extended Abstracts on Human Factors in Computing Systems*, New York, NY, USA, 2012, pp. 2771–2774.

[31] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[32] M. Fowler and J. Highsmith, “The agile manifesto,” *Softw. Dev.*, vol. 9, no. 8, pp. 28–35, 2001.

[33] W. Feurzeig, S. Papert, M. Bloom, and R. Grant, “Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final Report on the First Fifteen Months of the LOGO Project.,” Nov. 1969.

[34] A. C. Kay, “The early history of Smalltalk,” in *History of programming languages—II*, 1996, pp. 511–598.

[35] S. Turkle and S. Papert, “Epistemological pluralism: Styles and voices within the computer culture,” *Signs*, pp. 128–157, 1990.

[36] C. Lévi-Strauss, *The savage mind*. University of Chicago Press, 1968.

[37] A. F. Blackwell, “Gender in domestic programming: From bricolage to séances d’essayage,” in *CHI’2006 workshop on End User Software Engineering*, 2006.

[38] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, “The Scratch Programming Language and Environment,” *Trans Comput Educ*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[39] C. Eckert, A. Blackwell, M. Stacey, C. Earl, and L. Church, “Sketching across design domains: Roles and formalities,” *Artif. Intell. Eng. Des. Anal. Manuf.*, vol. 26, no. 03, pp. 245–266, 2012.

[40] M. Puckette, “Pure Data: another integrated computer music environment,” *Proc. Second Intercollege Comput. Music Concerts*, pp. 37–41, 1996.

[41] C. Reas and B. Fry, *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2007.

[42] J. Maeda, *Design by numbers*. The MIT Press, 2001.

[43] S. Baader and S. Bødker, “SketchCode – An Extensible Code Editor for Crafting Software,” in *End-User Development*, P. Díaz, V. Pipek, C. Ardito, C. Jensen, I. Aedo, and A. Boden, Eds. Springer International Publishing, 2015, pp. 211–216.

[44] A. F. Blackwell, “Palimpsest: A layered language for exploratory image processing,” *J. Vis. Lang. Comput.*, vol. 25, no. 5, pp. 545–571, 2014.

[45] C. Reas and B. Fry, “Processing: Programming for Designers and Artists,” *Des. Manag. Rev.*, vol. 20, no. 1, pp. 52–58, Jan. 2009.

[46] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, “Live Coding in Laptop Performance,” *Organised Sound*, vol. 8, no. 03, pp. 321–330, 2003.

[47] E. Cocker, “Live Notation:–Reflections on a Kairotic Practice,” *Perform. Res.*, vol. 18, no. 5, pp. 69–76, 2013.

[48] J. Rohrhuber and A. de Campo, “Just in time programming,” *SuperCollider Book MIT Press Camb. Mass.*, 2011.

[49] M. Wark, *A Hacker Manifesto*. Cambridge: Harvard University Press, 2004.

[50] J. Erickson, *Hacking: the art of exploitation*. No Starch Press, 2008.

[51] E. J. Chikofsky, J. H. Cross, and others, “Reverse engineering and design recovery: A taxonomy,” *Softw. IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[52] “The Art of Hacking Exhibition.” [Online]. Available: www.nimk.nl/eng/the-art-of-hacking-exhibition. [Accessed: 11-Oct-2012].

[53] Q. R. Ghazala, “The Folk Music of Chance Electronics: Circuit-Bending the Modern Coconut,” *Leonardo Music J.*, pp. 97–104, 2004.

[54] I. Bergstrom and R. B. Lotto, “Code Bending: A new creative coding practice,” *Leonardo*, vol. 48, no. 1, pp. 25–31, 2015.

[55] E. R. Miranda and M. M. Wanderley, *New digital musical instruments: control and interaction beyond the keyboard*, vol. 21. AR Editions, Inc., 2006.

[56] I. Bergstrom, A. Steed, and B. Lotto, “Mutable Mapping: Gradual Re-routing of OSC Control Data as a Form of Artistic Performance,” in *Proceedings of the International Conference on Advances in Computer Entertainment Technology*, New York, NY, USA, 2009, pp. 290–293.

[57] C. Ncube, P. Oberndorf, and A. W. Kark, “Opportunistic Software Systems Development: Making Systems from What’s Available,” *IEEE Softw.*, vol. 25, no. 6, pp. 38–41, Nov. 2008.

[58] J. Wong and J. I. Hong, “Making mashups with marmite: towards end-user programming for the web,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 1435–1444.

[59] F. P. Brooks, *The mythical man-month*, vol. 1995. Addison-Wesley Reading, MA, 1975.

[60] E. Regelson and A. Anderson, “Debugging practices for complex legacy software systems,” in *Proceedings of International Conference on Software Maintenance*, 1994, pp. 137–143.

[61] E. W. Dijkstra, *Notes on structured programming*. Technological University Eindhoven Netherlands, 1970.

[62] R. Lindell, “The Craft of Programming Interaction,” presented at the 7th Nordic Conference on Human-Computer Interaction, 2012, pp. 26–30.

[63] D. Fallman, “The interaction design research triangle of design practice, design studies, and design exploration,” *Des. Issues*, vol. 24, no. 3, pp. 4–18, 2008.

[64] A. Vallgårda and Y. Fernaeus, “Interaction Design as a Bricolage Practice,” in *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction*, 2015, pp. 173–180.

[65] B. Buxton, *Sketching user experiences: getting the design right and the right design: getting the design right and the right design*. Morgan Kaufmann, 2010.

[66] A. Bdeir, “Electronics as Material: LittleBits,” in *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, New York, NY, USA, 2009, pp. 397–400.

[67] R. Lindell, “Crafting interaction: The epistemology of modern programming,” *Pers. Ubiquitous Comput.*, vol. 18, no. 3, pp. 613–624, 2014.

[68] Y. Fernaeus and P. Sundström, “The material move how materials matter in interaction design research,” in *Proceedings of the Designing Interactive Systems Conference*, 2012, pp. 486–495.

[69] M. Wiberg, “Methodology for materiality: interaction design research through a material lens,” *Pers. Ubiquitous Comput.*, vol. 18, no. 3, pp. 625–636, 2014.

[70] A. Vallgårda and J. Redström, “Computational composites,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 513–522.

[71] J. Löwgren and E. Stolterman, *Thoughtful interaction design: A design perspective on information technology*. Mit Press, 2004.

[72] P. Dourish and M. Mazmanian, “Media as material: Information representations as material foundations for organizational practice,” *Matter Matters Objects Artifacts Mater. Organ. Stud.*, vol. 3, p. 92, 2013.

[73] A. F. Blackwell and S. Aaron, “Craft Practices of Live Coding Language Design.”

[74] T. Ingold, *Making: Anthropology, archaeology, art and architecture*. Routledge, 2013.

[75] M. Jonsson, J. Tholander, and Y. Fernaeus, “Setting the stage–Embodied and spatial dimensions in emerging programming practices,” *Interact. Comput.*, vol. 21, no. 1–2, pp. 117–124, 2009.

[76] T. Nunes, A. D. Schliemann, and D. W. Carraher, *Street mathematics and school mathematics*. Cambridge University Press, 1993.

[77] O. Bälter and D. A. Bailey, “Enjoying Python, Processing, and Java in CS1,” *ACM Inroads*, vol. 1, no. 4, pp. 28–32, Dec. 2010.

[78] C. B. Price and W. R. Worcester, “Can the Fine Arts Inform Software Development,” in *JICC Conference, LMU*, 2007.

[79] D. Shiffman, *Learning Processing: A Beginner’s Guide to Programming Images, Animation, and Interaction*. Morgan Kaufmann, 2009.

[80] D. Shiffman, S. Fry, and Z. Marsh, *The nature of code*. D. Shiffman, 2012.