Crafting Interaction: The epistemology of modern programming

Rikard Lindell

Mälardalen University
Box 883 SE-721 23 Västerås Sweden

rikard.lindell@mdh.se

Abstract

There is a long tradition in design of discussing materials and the craft of making artefacts. 'Smart' and interactive materials affected what constitutes a material. Interaction design is a design activity that creates the appearance and behaviour of information technology, challenged by the illusiveness of interactive materials. With the increased design space of ubiquitous devices, designers can no longer rely on a design process based on known interaction idioms, especially for innovative highly interactive designs. This impedes the design process, because non-interactive materials, by which designers create sketches, storyboards, and mock-up prototypes, do not provide the essential talkbacks needed to make reliable assessments of the design characteristics. Without a well-defined design the engineering process of artefacts has unclear ends, which are not encompassed in the rational epistemology of engineering. To value the experiential qualities of these artefacts the prototypes need to be interactive and crafted in code. This paper investigates the materiality of information technology, specifically programming language code from which interactive artefacts are made. A study of users of programming languages investigates how they describe programming language code as a material. If you have a material it is reasonable, because of the tradition in the material and craft fields, to say you have a craft. Thus, considering code a design material allows the metaphor of craft to be used for the activity of programming.

Keywords

Material, Materiality, Design, Interaction Design, Craft, Engineering, Software Engineering, Programming, Epistemology

Introduction

The creation of useful artefacts with rich experiential qualities required quality-driven interaction designers and programmers with the ability to do simultaneous problem-setting and problem-solving. People use interactive software, websites and mobile applications in different contexts for different purposes. It may be mandatory use (for example administrative systems) or devoted use of social

media, games, or artistic creation. Boehm shows a focus shift in software engineering to acknowledge usability and also that requirements of interactive artefacts cannot be defined a priori [1]. Users and other stakeholders cannot articulate their needs so they can be transformed into a well-defined specification. Nonetheless, models and methods in software engineering focus on solving problems and thus require commitments to well-defined requirements [1,2]. Interaction design has indulged itself in being a design practice that defines the appearance and function of digital artefacts [3]. Sketches, storyboards, videomatics, and interactive prototypes depict the appearance and functionality of digital artefacts, and at best convey requirements to software engineers [4-6]. The result from a design process is rich in clues to the finished product. However, the material in the design process is different from the code that implements the design into a working artefact [7].

The interplay between interaction design and software engineering is problematic [8]. These two activities have different epistemology; interaction design is a design practice [3], whereas software engineering is struggling to describe itself as engineering and science [1]. People who are active in these fields have different ways of thinking about how they work [9]. Designers are trained to see a plethora of future designs for a situation, in which they abide to a rigour of design practices [10,11]. Engineers, however, are trained to solve well-defined specific problems [9,12]. Löwgren [12] shows the epistemological differences between engineering and design. Engineering focuses on convergent processes to determine one solution to one problem in a sequential refining order in an objective manner whereas design explores problems through a parallel and divergent creative process before committing to a design. The design also bears the personal presence of the designer.

Buxton [9] answered software engineers who wanted "guidelines for maximizing user experience" in an open letter published in *Businessweek*. He was frustrated with the inability of engineers to recognise and respect the practice of experience design. Buxton requested at least *Design Awareness* from everyone in an organisation - especially from software engineers. Design awareness can and should be something that every employee of a company does their best to acquire, and the same applies to technology awareness. Buxton delineated three levels of design knowledge above design awareness; *Design Literacy*, *Design Thinking*,

Design Practice. All employees can acquire design literacy with more effort. Design thinking can be acquired by anyone who is committed to investing time to practise their skills, whereas design practice – according to Buxton – is not is accessible to all.

The context in which designers want to bring a new or changed artefact is explored through design. Design explains the phenomena of the context. It is about framing the problem space of the context, cut into a search tree of plentiful design proposition to reach the right user experience design of a future artefact [5,13]. Design is the exploratory use of malleable tactile materials and provides suggestions for possible future solutions [3,5,13]. The goal of the design process is to frame, as much as possible, the problem for an engineering process to solve. In the ideal case, every problem is well defined and known.

Sketches, storyboards, and paper prototypes work in design situations where the designer experiments with known interaction idioms. Users, design colleagues, and programmers fill the gaps and imagine the user experience for the finished artefact based on their experience with these idioms. However, this approach does not work for innovative forms of interaction and user experience. To get talkback from the interaction design it is necessary to create interactive prototype programs. Memmel [8] shows that the gap between designing digital artefacts and implementing them is not easy to bridge. The designer depicts the function and appearance in a different material from what the programmer uses to construct an artefact. Materials have inherent characteristics that affect and provide the preconditions for what can be created with it. Code provides other types of talkbacks than paper prototypes, and the design process does not stop when the programming starts; on the contrary, programming is a part of the design process. One way of approaching this issue is to view programming language code as a design material. If code can be considered a design material, then programming can be explained through the metaphor of a design craft. Thus, the epistemology of craft is applicable to programming. For this reasoning to be true we need to investigate how users of programming language express code as a design material.

Background

Schön introduces the concept of *Technical Rationality* and offers an explanation of the engineer's epistemology [14]. Schön discusses how faith in rational,

scientific, and technological solutions became dominant because of how these approaches were successfully applied during World War II, where the solution to a problem was to supply more resources [14]. The point he makes is that engineering is close to science. “They began to see laws of nature not as facts inherent in nature but as constructs created to explain observed phenomena, and science became for them a hypothetico-deductive system. In order to account for his observations, the scientist constructed hypotheses, abstract models of an unseen world which could be tested only indirectly through deductions susceptible to confirmation or disconfirmation by experiments. The heart of scientific inquiry consisted in the use of crucial experiments to choose among competing theories of explanation.” [14] (page 33). This quotation describes the belief in deductive reasoning that disconnects the explanation of the world from the material to be explained. A scientific approach allows the engineer to deduce, analyse, and define problems in a rational way: the positivist epistemology of science [14]. Technical rationality is part of the historical heritage in software engineering, where the metaphor of engineering is used to describe programming and the activity of developing software artefacts. Bennington [15] describes an engineering development model for software at the Symposium on Advanced Programming Methods for Digital Computers on 29 June 1956. According to Bennington, technical rationality was a success factor for their project: “It is easy for me to single out the one factor that I think led to our relative success: we were all engineers and had been trained to organize our efforts along engineering lines. We had a need to rationalize the job; […] In other words, as engineers, anything other than structured programming or a top-down approach would have been foreign to us.” This quote shows how the development of software engineering was organised. This instrumental and strictly top-down approach was named "The Waterfall Model" during the 70's [1]. This development method was an attempt to respond to demands of a technical, rational, and clearly defined problem. Focus and resources were allocated to solve the problem. The Waterfall Model is still important in the development of large projects [1], and the model is still the basis for education and literature in software engineering.
Boehm [1] describes in his exposé of over a half-century of software engineering how the field evolved with a major increase of focus on usability and “value”. Software engineering has realised the problems with the waterfall model and has

introduced iterative models such as the spiral model [16] or the Rational Unified Process (RUP) [2]. These models deal with changes in problem-setting in iterations. In RUP, the focus is on use-cases, architecture and well-defined goals for each iteration [2]. However, each iteration of the spiral model and RUP is a waterfall model. Therefore the situation is only marginally improved in these models. The foundation is still the technical rationality epistemology.

Figure 1. Buxton's image of the organisation for the engineering-driven product development.

Buxton describes an engineering-driven organisation [5], see Figure 1. The first step is to conduct research and development, step two is a comprehensive engineering, and in the last step a marketing department tries to adapt, sell, and spread the products. This type of organisation requires an agreement on ends. The process is not particularly suited to encompass external influences and change. Dissonance and physiological entropy arise in the organisation when changes and difficulties in clearly defining the problem occur because "Technical Rationality depends on agreement about ends." "When ends are fixed and clear, the decisions to act can present themselves as an instrumental problem. But when ends are confused and conflicting, there is as yet no *problem* to solve. A conflict of ends cannot be resolved by the use of techniques derived from applied research; it is rather through the non-technical process of framing the problematic situation that we may organise and clarify both the ends to be achieved and the possible means of achieving them" [14]. In the citation Schön delineates how technical rationality does not address situations where the result is uncertain and where there is no ready-defined problem to solve. In the quote, Schön also provides a clue of how to deal with difficult situations.

The Agile Manifesto

The Manifesto for Agile Software Development (2001) was written as a critique of the rigid approach to requirements specification, analysis, construction and documentation. It focuses the creation of useful artefacts with rich user experience. The manifesto reads: Individuals and interactions over processes and

tools, Working software over comprehensive documentation, Customer collaboration over contract negotiation, and Responding to change over following a plan [17]. The manifesto reflects programmers' frustration at spending most of their time documenting and managing projects instead of writing code.

Two popular agile software development methods, as they are known, are Extreme programing (XP) and Scrum [18]. Both XP and Scrum were created years before the Agile Manifesto. They are currently regarded as the methods that best live up to the agile Manifesto dogma. The methods are based on informal user stories that describe the features from a user perspective [8]. User stories are implemented in iterations and are evaluated and revised after each iteration. In Scrum one iteration is called a *sprint* that lasts between a couple of weeks or up to a month [19]. After each iteration the developed features must be demonstrable, and the result of a *sprint* is revised in a *sprint review meeting*. Despite the emphasis on usable features, Lárusdóttir et al. [20] have shown that scrum teams often fail to attend user experience values of a design. Scrum is designed to handle chaos and change [21], yet programmers still spend up to a month in problem-solving committed to ends before they can re-evaluate the problem-setting.

Agile Development with XP and Scrum in particular is a big step for software engineering in the direction of focusing on service qualities and user experience as opposed to non-agile development models, such as RUP, the spiral model, and the waterfall model. However, despite the Agile Manifesto, XP, Scrum, and other iterative development models have still a clear plan-implement-evaluate cycle that extends over a longer period, at least weeks, but in practice longer. A common feature of these methods is agreement about ends. Scrum, however, is different from the others by being designed to accommodate change, but the method does this by a technical rational approach: "One of the key pillars of Scrum is that once the Scrum Team makes its commitment, the Product Owner cannot add new requests during the course of the Sprint. This means that even if halfway through the Sprint the Product Owner decides that they want to add something new, they cannot make changes until the start of the next Sprint." [22]. This quotation shows that the method prevents continuous problem-setting and problem-solving to handle difficult situations. Longer plan-implement-evaluate cycles impede agile development and lock the scrum team to goal commitments.

In recent years, the Kanban development model has attracted attention by providing freedom for adaptation [23]. “Scrum is less prescriptive than XP, since it doesn’t prescribe any specific engineering practices. Scrum is more prescriptive than Kanban though, since it prescribes things such as iterations and cross-functional teams. ... Kanban leaves almost everything open. The only constraints are Visualize Your Workflow and Limit Your WIP [work in progress]. Just inches from Do Whatever, but still surprisingly powerful.” [23]. This quotation shows how Kanban can support an agile development process in constant change. The model allows the goal of a work in progress (WIP) change during the process. This means that a WIP can have an open end. Thus, Kanban is a radically different approach than the earlier development models. Kanban has become popular in game development. This is no coincidence; game development is focused on highly interactive experience and *game play*. Game development is also a guild instead of a professional discipline [24]. Game developers have to work their way up from being level designers, and are promoted as they demonstrate their skills to get gradually closer to developing the game engine.

Reflection-in-action and interaction design

Technical rationality and focus on ends have a different epistemological dimension than Reflection-in-action - Schön's term for the reflective practitioner way of thinking and acting. The reflective practitioners have practical knowledge (knowledge-in-practice); they can be aware or unaware of this knowledge regardless of guild. Reflective practitioners deal with problem-setting, and unique and complex situations, mainly through reflection-in-action (reflection-in-action). Schön depicts reflection-in-action as the following scenario:
“When good jazz musicians improvise together, they also manifest a “feel for” their material and they make on-the-spot adjustments to the sound they hear. Listening to one another and to themselves, they feel where the music is going and adjust the playing accordingly. They can do this, first of all, because their collective effort at musical invention makes use of a scheme – a metric, melodic, and harmonic schema familiar to all the participants – which gives a predictable order to the piece. In addition, each of the musicians has at the ready a repertoire of musical figures which he can deliver at appropriate moments. Improvisation consists of varying, combining, and recombining a set of figures within the

schema which bounds and gives coherence to the performance. As the musicians feel the direction of the music that is developing out of their interwoven contributions, they make new sense of it and adjust their performance to the new sense they have made. They are reflecting-in-action on the music they are collectively making and on their individual contributions to it, thinking what they are doing and, in the process, evolving their way of doing it. Of course, we need not suppose that they reflect-in-action in the medium of words. More likely, they reflect through a "feel of music"..." [14].

This quote illustrates that reflection-in-action happens on the fly and is a thought process happening while the practitioners perform their activities. This can be done consciously, but it is more likely that it is a subconscious thought process (*thinking on your feet*), for example, when a musician improvises while communicating through the music with the rest of the ensemble and the audience. Reflection-in-action can be summarised in three phases that are repeated: (1) Frame the problem, assess the situation, and understand the working material. (2) Perform moves over the situation. These moves are parts of the practitioner's repertoire. They are small experiments with the intentional result, but often with unintended effects (both positive and negative). (3) Reflect and evaluate the consequences of action in conversation with the situation. Practitioners take in and reflect on how the situation responds (talkbacks). The conversation happens in what Schön calls the medium's language [25]. Then the process restarts.

Design problems are often vague, complex, and contradictory [11]. In the problem-setting phase interaction designers name the phenomena that they will pay attention to and work with. They create concepts through various design techniques to better understand and frame the problem. Concept designs are evaluated and refined through introspection, criticism, and user studies, such as the Wizard of Oz method [5]. Interaction designers also increase their understanding of the situation and context through sketching interfaces and designing mock-up prototypes [26].

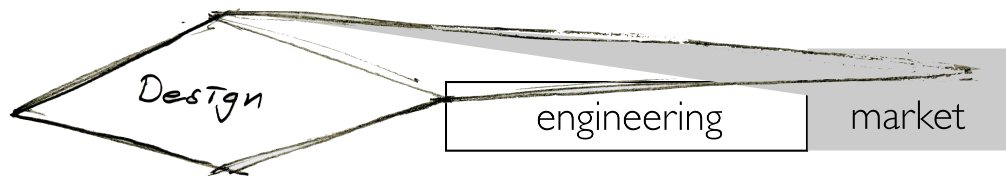

Figure 2. Modified figure of Buxton's model of a design-driven organisation. Design is first diverging then converging to a design that can be handed over to engineering.

Buxton [5] describes how a design-driven organisation can manage resources for the research, development, and construction of new artefacts. He believes that a successful design process faster answers issues that would otherwise be answered in the research and development. Figure 2 also shows how the design team follows the design through the entire process and that marketing meets up. Design and marketing focus on users; the first to create value, and the other to communicate value so that users are willing to pay.

Interaction designers have a repertoire of interaction styles that they can apply to different problems [27]. Digital artistry and creation of aesthetically pleasing looks - especially for highly interactive graphical user interfaces - are futile unless there is a whole and situation adaptation. To be really able to design great interfaces, interaction designers should master a programming language. It is part of being conscious of the design material [6,9]. While interaction designers can implement a design by composing software, they must not be seduced by technologies for technology's sake.

Interaction designers create architecture for interactive artefacts and their spatial and temporal properties. They design the artefact's topology, the artefact's appearance on the screen or in the room and how the artefact changes over time because of interaction. Interaction designers understand the consequences of different designs and have a feel for how a design can be realised. This feeling can be obtained by transforming design into technology. This is similar to the model building of architects. They build models of future buildings to understand the consequences of what they have designed and drawn. Similarly, interaction designers build interactive prototypes for technical substantiation and in full understand what they have designed. We are talking about material consciousness [28]. The difference between the architect and interaction designer is that the latter

builds a model in full scale, albeit quickly, and at times chaotic, but it is a model and not a product.

Craft

The profession, the knowledge, and ability to design and create interactive artefacts constitute a creative craft. McCullough [29] discusses the craft related to interactive technology use and how an artisan approach can enrich interaction design. Golsteijn et al. [30] found that the craft of making digital content is an important aspect in cherishing digital objects. Furthermore, traditional digital media (image, audio, and video files) can be manipulated with a craftsmanship approach by establishing a close coupling to tangible physical materials [31]. The traditional creative artisan work can benefit from information technology, and thus bring the craftsmanship approach to technology appropriation to extend the users' repertoire [32].

According to McCullough, there is a wide gap between the design of digital artefacts, and computer science and software engineering. The rejection of the craft in today's computer science and software engineering is similar to how engineers in the emerging Industrial Revolution saw the craft, illustrated in the following quote from Diderot's Encyclopédie 1751-80 [29]

“CRAFT. The name is given to any profession that requires the use of the hand, and is limited to a certain number of mechanical operations to produce the same piece of work, made over and over again. I do not know why people have a low opinion of what this word implies; for we depend on the crafts for all the necessary things of life. [...] The poet, the philosopher, the minister, the warrior, the hero would all be nude, and lack bread without this craftsman, the object of their cruel scorn.”

That fact that craftsmanship has not been highly regarded is not new. Within software engineering, the concept of craftsmanship is sometimes used derogatorily to describe careless programming. Boehm [1], for instance, uses the notion of craftsmanship as an analogy for the 1960s careless "cowboy programming" and lack of professional discipline. However, negligence has nothing to do with craft and craftsmanship. On the contrary, the craftsman is described by Sennett [28] as quality-driven, bordering onto the manic, busy perfecting his/her work. Wallace and Press [33] found that this quality-drivenness

has become of major importance for crafting aesthetically pleasing digital artifacts. But this goes deeper than just creating appearance. For instance, the core animation application programming interface (API) of the iOS provides App-creators with functionality that gives animated cues to the behaviour of onscreen objects and makes the interface feel more luxurious.

The craftsman must be patient and not be tempted to do quick fixes. External factors - social and economic conditions, poor tools, or bad work environment - can be obstacles to the craftsman's good work. However, the craftsman's commitment is to perform good craftsmanship for its own sake [28].

Sennett describes the craftsman's ability to simultaneously identify problems and solve them. This is consistent with Schön's ideas about reflection-in-action, discussing problems qualifying in difficult situations. Sennett says that problem-setting and problem-solving have a rhythm that relates to subconscious and conscious reflection-in-action.

“Every good craftsman conducts a dialogue between hand and head. Every good craftsman conducts a dialogue between concrete practices and thinking; this dialogue evolves into sustaining habits, and these habits establish a rhythm between problem-solving and problem-finding. The relationship between hand and head appears in domains seemingly as different as bricklaying, cooking, designing a playground, or playing the cello...” [28].

The craftsman is thus characterised by an ability to see and solve problems simultaneously in the dialogue between hand and mind. Schön [25] calls this dialogue a conversation with the situation. The conversation is enabled by the craftsman’s material consciousness and mastery earned through at least ten thousand hours of practice.

Material

McCullogh discusses the concept and use of interactive technologies, especially artistically creative users, and portrays this practice as a craft:

“Virtual craft still seems like an oxymoron; any fool can tell you that a craftsperson needs to touch his or her work. This touch can be indirect – indeed no glassblower lays a hand on molten material – but it must be physical or continual, and it must provide control of whole processes. Although more abstract endeavours such as conducting an orchestra or composing elegant software have

often been referred to as crafts, this has always been in a more distant sense of the word. Relative to these notational crafts, our nascent digital practices seem more akin to traditional handicrafts, where a master continuously coaxes a material. This new work is increasingly continuous, visual, and productive of singular form; yet it has no material." [29].

McCullogh believes that information technology is not a material because it has no physical properties. But, says McCullogh, the craftsmanship and handwork is still important, whether you use a drawing tablet to draw or navigate using automated and habitual manipulations. However, Dourish and Mazmanian [34] have found that there is a materiality of digital representations, and that digital technologies need to be studied on their own materiality and on their particular forms of practice. Bertelsen et al. [35] found that the concept of *materiality* described the engaging talkbacks and resistance from creative music patch programming with the MSP/Max graphical programming language. In their view, the materiality of the software encompasses the inherited historicity of music –the domain knowledge – and the potential of computer technology – the ability to use and abuse technology in new ways. Information technology, according to Löwgren and Stolterman [27], is a material which has no recognisable features. This view combines interaction with "traditional" design trades and crafts.

The similarity between the industrial designer and architect on the one hand and the interaction designer on the other lies in creating technology. However, the industrial designer and architect's material is traditionally concrete as opposed to interaction designer material that is intangible. Robles and Wiberg argue that interaction designers should attend to the designs materiality and learn from more traditional design disciplines [36]. With new adaptive, dynamic and context-aware material Bergström et al. suggest that they become material over time, which implies that industrial designers and architects should rely on methods from interaction design [37]. The material they are working on has traditionally distinguished disciplines based on what they create, but they have similar practices, methods, and approaches to design. Information technology is on the surface visual, auditory, or haptic, but this is an illusion created by calculations and represented in ones and zeros and described with programming languages. The media and language for interaction design are sketches of the appearance of

the interface, creating paper prototypes, and to write working prototype computer programs that embody digital artefacts' behaviour.

Empirical Study

The background shows that volatile digital materials have materiality in application areas where artisanal approach is appropriate [29-32]. There are indications in the literature that there is materiality to be considered in programming too [28,34,35].

In my experience material can be a metaphor of programming language code. The creative work of implementing interactive multimedia artefacts and products using a mixture of languages has made me reflect upon the materiality of code, if it is pliable, malleable, solid, or brittle. But as much as this is my personal experience, I need to know the view and attitude in the community of practitioners, both interaction designers and developers. The idea was to start the inquiry by sending out an open and informal question to users of programming languages; colleagues, friends, and alumni students: "I would like you to write a sentence or two describing what you think and feel about your favourite programming language." However, the investigation resulted in 33 responses, providing ten pages of text. Each answer offered a varying amount of data, ranging from an entire page to just a few sentences. *Grounded Theory* [38] was considered the most pragmatic approach to analyse the data. The method works with any kind of data [39] and describes a workflow that drives the analysis [40,41]. The method needs only the amount of data required to achieve saturation, and is thus independent of large quantities of data [38].

The data were considered enough to fill a gap of programming language code as material in the background literature. The advantage of the grounded theory data analysis method is that you can begin with what you have. Later on if you discover that you do not have enough data to delineate the categories and the core category you can collect more data, in a second phase of theoretic selection. You collect data and continue the analysis until your categories are saturated [38].

A grounded theory grows in three or four phases [40,41], and a grounded theory analysis in each of these phases is done in four activities: *theoretic selection*, *theoretic coding*, *comparison*, and *conceptualising* [41]. Here, the theoretical selection is the community of users of programming languages. During coding,

sentences or words are marked or labelled as *indicators* that contribute to the growing theory. Pre-conceptualised ideas to theories are written as memos. Then the indicators are compared, sorted and commented to be woven into a theory during conceptualising. For each phase the theory gets more general and saturated. In a second phase of analysis of the analysis, I have also included related theories and literature for the creation and saturation of the categories. The method allows you to treat these findings as data, which is one of its advantages [39].
Descriptive categories and concepts were induced from the collected data. These were: *material* (core category), *rational*, *pragmatic*, *mastery*, *learning*, and *explorative*. Figure 3 shows the relationship of the categories to one another and their dimensional relationship to Sennett's concepts of craftsmanship and quality-driven [28]. The diagram depicts the categories' relationship to the core category *material.* The size of the circles – except for the material category – describes the category's saturation.

Material

The core category in the collected data is *material*. *Material* concerns the code's materiality, and sets the conditions for the situation and use of programming languages. This is closely related to the materiality concept of Bertelsen et al. [35], but here it is transferred to a broader domain of programming. Material provides talkbacks for those trying to understand and describe the world through programming. Material also provides talkbacks for those who do sketches or explore a design through programming. For example, hardware programming of embedded systems is a different material from robot programming, although they are closely related, whereas large data volumes are different in their materiality compared with embedded systems. The material properties are different for this: it may involve explorative programming and exploration of the nature of the data set, or to make runs over the material as a commercial service. In the collected data, statements on flexibility and simplicity recur. The material's internal malleability is important to the informants. That is, language and data can be processed and transformed according to desire and need.

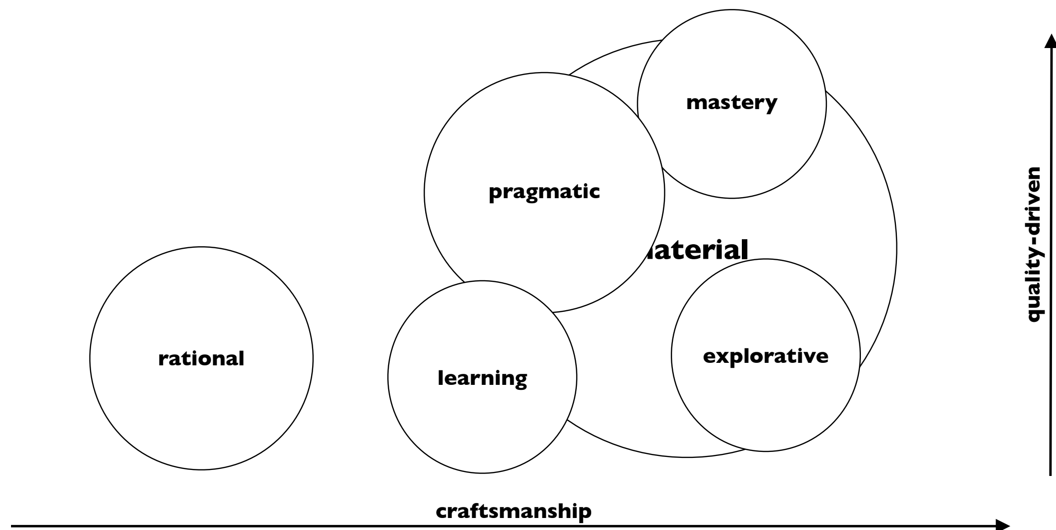

Figure 3. Six categories of how programmers feel and think about their favourite programming language. The horizontal dimension depicts, ascending rightwards, to what extent each category relates to the concept of *craftsmanship*. The vertical dimension, ascending upwards, shows the concept *quality-driven* in craftsmanship-related categories. (The concept quality-driven is not applicable to the *rational* category). The diagram also shows the influence of the core category *material* on each category, the closer the more influence. The size of the circles – except for the *material* category – describes the saturation of each category.

Talkbacks from the Material relate to Bjurwill's four interoperation frames [42]. A situation is understood by making thought experiments and by conversation with the material through four interpretation frames: practice, theory, value, and context. The practice frame deals with the practitioner's skill, "The media, languages, and repertoires that practitioners use to describe reality and conduct experiments" [43], for instance subconsciously knowing that design pattern should be applied to a specific problem. The theory frame is also relevant to the core category of material. It is through the theory frame that the programmer applies a theoretical model to explain the situation and phenomena. Practitioners choose the explanation that best fits the situation through thought experiments, obtaining information, skills and creativity.

Rational

For the rational approach the informants indicate the programming language's particular theoretical-technical features, such as polymorphism, abstraction levels, and performance. The language paradigm is important for this category. There are

also those describing multi-paradigm language, which is a rather pragmatic choice (see below), but where the focus is on delivering specific properties.
Here is an illustrative quote: “... the language supports multiple levels of abstraction. Depending on the application you can choose either to write code at a high level of abstraction, with object orientation, encapsulation, inheritance, dynamic binding and so on, or, on a more hardware related level, with standard C functions, simple data types and structs, and so on.”
The rational category is close to a scientific approach to programming derived from academic studies in computer science. The rational category is not about what the language is used for, and there are no emotional reasons as to why a specific language is one’s favourite language.

Learning

The *learning* category refers to learning through programming. This partly refers to the inexperienced learning from layman to master, but also the master’s learning and discovery of the world through programming. The informants who provided data here were discussing the discovering of the world of programming. "Suddenly there was a language based on logical rules and, as a computer game geek, I was very motivated to learn."
The previous quote is an example of the layman's contact with a programming language. The following quotation describes when mastery of the programming art is taken to a higher level: "... when I worked a lot with prologue, the declarative approach felt amazing, to describe the world as it was, and what you can do with it instead of writing cookie recipes felt right!"
The following quote shows how the programming language describes a world: “Once you have crossed the threshold into one programming language, you have learnt to think in algorithms. It is another way of seeing the world, a wholly different set of problem-solving tools”.
These three quotations also link to the category of material. In the first two, how code describes the world, and, in the last case, how the code’s talkbacks tell the programmer how the world wants to be designed. Also, there is an aspiration towards a rational approach because the quotes are about programming paradigms and their properties.

Mastery

This category describes the pursuit for full control over the artefact and its inner bits and bytes. The following quote illustrates this: "It is a new and enjoyable challenge to try to optimise an ordinary code loop to massive parallelisation where you have to think about how much [performance] it costs to read and write to memory." This specific quote discusses CUDA, a language that takes advantage of graphics processors' massive parallelism. Another example of mastery and control is the following quote: "My favourite is C. When I start the programming environment, it feels like I have total control, no stress, no funny business."
The pursuit of mastery may also lie in the choice to use a more obscure language. Programming for the informants in this category has an almost irrational and therapeutic function.
This category is related to the core category of *material* in that the mastery is about controlling the machine's behaviour in detail. The processor's silicon die design, and the computer hardware architecture, feature an influence on the situation for this category. Furthermore, this category is also connected to Sennett's idea of quality-driven manic craftsmanship [28]. The focus on quality can also be found in Martin et al.'s book *Clean Code: A Handbook of Agile Software Craftsmanship* [44] which instructs the reader how to gain full control over the code. Another empirical finding that saturates this category is the Manifesto for Software Craftsmanship – Raising the bar (2009). The manifesto berates pragmatic "working software" and praises well-crafted software.

Pragmatic

The most widespread approach to programming is category pragmatism. The core category of material has great influence on the category of pragmatic because the focus is on using tools for specific purposes in specific situations. Partly, the informants describe problem-setting activities, but the main emphasis is on problem-solving. A typical quote for this category is: "Today, I write mostly in Perl, Scheme and Java. Scheme is the more elegant, Perl the most efficient, and Java the most durable."
Here we also find those who use more emotionally charged words about what they do. A positive tone is exemplified in the following quote: "I love the fact that you can do so much with scripting languages in Linux ... in no time. It is

extremely powerful because it combines the various scripting languages together, and also combines it with Linux commands. It is amazing.” However, there are those who are more sceptical to programming: “I do not have a favourite language - they are all evil and should be met with healthy doses of scepticism.”
Here is a quote that shows the proximity between the pragmatic and the rational attitude: “Ruby is a language that simplifies the path to better code quality with unit testing while it makes for productivity with its elegant dynamic duck typing. You get things done while a readable structure can be maintained. [...] Java will probably come number two on the podium, but then you don’t get as much done. Moreover, it is not as much fun!” The focus is still here on the pragmatic (how much can be done) and on the emotional (if it is fun).
Among the pragmatic programmers are also those who give the impression of not being quality-driven. For this group features that make their job easier are important, such as type checking and garbage collection. Focus in the data is on what you can get away with, which stands in stark contrast to *mastery*.

Explorative

This category mainly describes a conversation with the material. It is a continuing problem-setting and problem-solving with move-making-experiments and exploring of mini hypotheses through reflection-in-action. The programmer can strive here to design aesthetic expressions, functions and interactions. They explore the problem in conversation with the material to achieve a result. Programming in this category is not stable or even correct. Here programming languages are tools to incrementally explore and understand a problem. “What I cherish the most is that [Processing] is incremental so that I can test my way forward. Sometimes it feels like sketching, in the truest/best sense, when I can try my way to a new idea into an interactive behaviour. Sometimes.”
Here are two quotes that demonstrate the exploration and problem-setting approach: “Both in the case of Flash and Processing you get to see directly and graphically the results of your coding, a kind of feedback that really enhances your comprehension of programming concepts, such as: "Oh, that's what happens if I loop it!", and "Hold it right there till someone presses a button!". ”
“The awesome [thing] with Lingo was/is that I as a creator with lots of ideas could just sit down and do instead of planning. I could start at the wrong end and

still get it into something I could use." This quote also shows that the exploratory approach is all about problem- setting and not about writing robust code.

The exploratory approach with clear talkbacks from the material also applies to those programmers who have a more computer science approach: "What I like about Haskell is how natural it feels to divide a program into functions (once you have taken in that concept ...) and how clean and well organised the code becomes." This quote suggests the materiality talkbacks. Explorative programming explores designs; designers do both problem-setting and problem-solving. Dan Ingalls says that explorative programming may take you where your original goal does not matter [45]. Users of programming languages in this category obtain a material consciousness of digital artefacts.

If code is a material then programming is a craft

This is of course not a deductive logical rule stated in the heading of this section. However, there is a tradition of discussing material and craft and how they relate to each other [46]. This discussion has a long tradition which goes back to nineteenth century handicraft[1] education [47]. Thus, if you have one it is reasonable to assume you have the other.

The core category that emerged in the theoretical phase of the study above is called *material*. Data on the materiality of computer programming languages are provided for all categories except the *rational*. The categories of *rational* and *mastery* are in a dimension with one endpoint in a scientific approach to software engineering as described by Boehm [1], and the other end close to Sennett's [28] quality-driven craftsmanship approach. The category of *mastery* often refers to the materiality of the silicon-based artefact's design. In the *pragmatic* category statements about the type of data to be processed or which problem to be solved recur, and the materiality of the code here is indicated to be dynamic and malleable. The materiality of programming is explicitly mentioned in the category *explorative* - where most of the submitted data come from interaction designers and digital artists. The categories *explorative* and *learning* recur in stories that indicate talkbacks from the material similar to Schön's description [25]. The

---

[1] Salomon is using the Swedish word *Slöjd* that means the work skill in craft applied without an economic purpose.

learning category is about learning through programming to obtain a practice that is either pragmatic or rational.

Programmers who have an artisanal approach have a good chance also to acquire design thinking or even design practice, and this is indicated by the *explorative* category. Several pioneers interviewed in William Moggridge's book Designing Interactions have a background in computer science [48]. Bill Buxton and Jonas Löwgren are both examples of prominent interaction designers with an undergraduate degree in Computer Science.

The interaction designer and the programmer are standing on common ground, since the reflective practitioners' reflection-in-action, dealing with messy situations, and continuous problem-setting and problem-solving are pillars of the programmers' work. As indicated by category material in the study above, it is reasonable to say that programming follows craftsmanship epistemology. Hence the interaction design process does not have to turn into an engineering process as portrayed in Figure 2. The ongoing design process turns into a software craftsmanship process, see Figure 4.

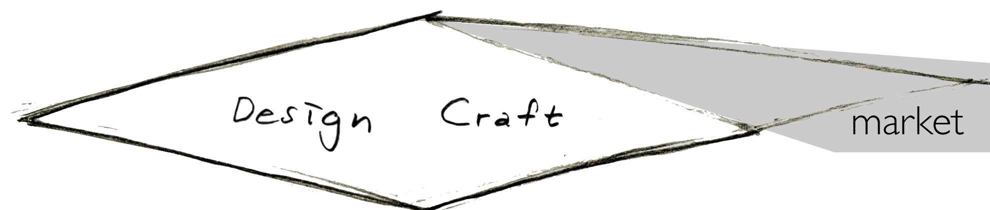

Figure 4. Design and programming as a craft facilitate the transition in the design work's change of materials from paper to pixels, and from sketches to code.

Craftsmen do problem-setting and problem-solving simultaneously. "Still the experimental rhythm of problem-solving and problem-finding makes the ancient potter and the moored programmer members of the same tribe." [28]. Thus, the design process is not over because the artisan's hands have a different repertoire. When we consider programmers as craftsmen of code, they have a repertoire of moves and skills, similar to designers. There is an ability to identify and understand the situations and know what tools are useful for the specific situation, to master specific techniques, APIs, frameworks, libraries, patterns, and programming languages. For example, the development of an artefact prototype can be implemented with dynamic languages by embedding a Lua or Javascript interpreter. This crafting of interaction can be done in the manner delineated in the

explorative category described above, the talkbacks from the code providing grounding for design decisions. As the design converges, the code starts to solidify, and the dynamic code can be reshaped into static code, written in for instance the C programming language. This is performed according to either the *pragmatic* or *mastery* category.

Programmers can develop a different feel for the code's properties. In the discussion of programming language features one programmer described the Ruby language as "cuddly faux fur", soft and comfortable but nothing one can use to build solid structures. The Python language was described as a "scaffold" that can quickly be moved and that you can build into anything. I myself use Lua which feels like play-dough for static modules in the C programming language and a way to explore the problem. C, however, feels like therapy and getting silicon under my fingernails.

The development of software - programming - is an activity with a wide range of intrinsic properties that are closer to (handi)craft than science or engineering. Sennett [28] describes the Linux programmer as the modern craftsman.

"People who participate in "open source" computer software, particularly in the Linux operating system, are craftsmen who embody some of the elements first celebrated in the hymn to Hephaestus, but not others. The Linux technicians also represent as a group Plato's worry, though in a modern form; rather than corned, this body of craftsmen seem an unusual, indeed marginal, sort of community. The Linux system is public craft."

Martin et al. [44] stress the importance of quality-driven and disciplined practice in the craft of programming. They focus on the code, to carefully write clean code based meticulous attention to the principles and guidelines for the scope of a function or method, of responsibility for a class, how test-driven development is pursued, how concurrency is best implemented, etc. Clean code is easy to read, easy to maintain and free from side effects and glitches. Above all, Martin et al. show that the problem cannot be solved at once but a problem can be explored by writing tests and constant iteration of possible improved solutions. This further connects the categories *explorative* and *mastery*.

Discussion

According to Buxton [5] problem-setting should be done without writing code. However, programming is a tool for a design that is difficult to portray on paper; for example, collaborative, pliable, or highly interactive features. Innovative interaction techniques require interactive prototypes. Interactive prototypes describe, demonstrate, and answer; they are specific, refined, and are used to performing tests [5]. However, exploratory programming allows various designs to be explored in order to set a problem [5,12,13], to validate the possible solutions [13,47], and in retrospect to transform the code into clean code [44]. One way to explore a design is to propose solutions by writing tests. By writing tests, the programmer explores and sets the problem while simultaneously solving the problem [44]. The ongoing tests are move-testing-experiments [25], in which the bugs and unwanted conditions are talkbacks from the material that drives the development process forward.

The empirical study shows that the 33 programmers or users of programming languages have different approaches to programming. A big group describes a rational approach to programming that is decoupled from the material, but the majority has a coupling to the materiality of code and talkbacks from it. Those with an exploratory approach in their programming practice are closest to the material, and they show most of the material continuousness and rhythm of problem-setting and problem-solving that Sennet describes [28]. Narratives in the *mastery* category reveal manic quality-driven craftsmanship. Material is an important part of the basis for craftsmanship in general, and implicitly here infers programming as a craft.

The literature and the study above suggest that it is meaningful to use material as a metaphor for code and crafts as a metaphor for programming. When code can be seen as a design material, I suggest that it may have implications in education and in the organisation of software development. In education, Kapor [50] has said that programming should be part of the interaction designer's repertoire, and that designers need to learn programming to get respect from software engineers. Both Kapor and Buxton point out that it is important that the development process is based on design, Figure 2. Code as a design material goes deeper than bridging engineering and design. Programming should be part of the designers' material consciousness, and thus be a part of an interaction design curriculum.

The creation of innovative highly interactive digital artefacts will benefit from not using the metaphor of engineering as described by Boehm [1] in the design and development process. Buxton [5] convincingly portrays the importance of the 1.0 version of software and that the original experience design remains stable through the artefact's entire lifecycle. Engineering can be a useful metaphor to describe the activities dealing with updates and non-functional issues beyond the original version,whereas a quality-driven artisanal ethos creates an environment to equip the world with well-designed tools and user experiences. We need to use methods of agile software development with a different approach to acknowledging experiential values. The development model Kanban contains characteristics that allow an artisanal approach [23]. It allows open-endedness, does not prescribe specific roles, and accommodates continual change. Programmers and designers can simultaneously be doing problem-setting and problem-solving. The model allows the concrete material from the design process – mood boards, sketches, storyboards, videomatics etc – to be used instead of user stories for features. The model affords explorative programming, and as the artefact takes shape, the development process can adopt a more pragmatic or mastery approach. However, to organise a project in an artisanal manner, the participants in the project need to have craftsmanship epistemology.

Acknowledgements

Thanks to Prof. Jonas Löwgren för providing valuable input, to Dr. Tomas Kumlin whose expertise in grounded theory was very valuable, to Egle Kristensen for proofreading from the field, and to my wife Eva Lindell who as a PhD student in Business Administration provided input.

References

[1] Boehm B (2006) A view of 20th and 21st century software engineering. In Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 12-29.

[2] Kroll P, von Krüchten P (2003) The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP. Addison-Wesley Professional

[3] Fällman D (2008) The Interaction Design Research Triangle of Design Practice, Design Studies, and Design Exploration. Design Issues MIT press 24.3:4-18

[4] Löwgren J, Stolterman E (2004) Design av informationsteknik - materialet utan egenskaper. Studentlitteratur

[5] Buxton B (2007) Sketching User Experiences - getting the design right and the right design. Morgan Kaufmann

[6] Lindell R (2009) "Jag älskar att allt ligger överst" – En designstudie av ytinteraktion för kollaborativa multimedia-framträdanden. Mälardalen University Press Dissertations: 72

[7] Vallgårda A, Sokoler T (2010) A Material Strategy: Exploring Material Properties of Computers. International Journal of Design vol 4 issue 3

[8] Memmel T, Gundelsweiler F, Reiterer H (2007) Agile human-centered software engineering. In Proceedings of the 21st British HCI Group Annual Conference on People and Computers vol 1:167-175

[9] Buxton B (2009) On Engineering and Design: An Open Letter. Businessweek April 29. http://www.businessweek.com/innovate/content/apr2009/id20090429_083139.htm. Accessed April 2012

[10] Wolf T.V, Rode J, Sussman J, Kellogg W.A (2006) Dispelling "design" as the black art of CHI. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06) 521-530

[11] Stolterman E (2008) The Nature of Design Practice and Implications for Interaction Design Research. in International Journal of Design 2(1):55-65

[12] Löwgren J (1995) Applying design methodology to software development. In Proceedings of Designing Interactive Systems 87-95.

[13] Krippendorf K (2006) The Semantic Turn. CRC Press, Taylor & Francis Group

[14] Schön D.A (1983) From Technical Rationality to Reflection-in-Action. Chap 2 in The Reflective Practitioner - how professionals think in action, Basic Books

[15] Bennington H (1983) Production of Large Computer Programs. Annals of the History of Computing vol 5 issue 4

[16] Boehm B (1985) A Spiral Model of Software Development and Enhancement. In Proceedings of an International Workshop on Software Process and Software Environments

[17] Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R.C, Mellor S, Schwaber K, Sutherland J, Thomas D (2001) Agile Manifesto. http://agilemanifesto.org/. Accessed April 2012

[18] Lindvall M, Basili V, Boehm B, Costa P, Dangle K, Shull F, Tesoriero R, Williams L, Zelkowitz M (2002) Empirical Findings in Agile Methods. Extreme Programming and Agile Methods — XP/Agile Universe 2002 Lecture Notes in Computer Science, Volume 2418/2002:81-92

[19] Kniberg H (2007) Scrum and XP from the Trenches: How we do Scrum. C4Media inc

[20] Lárusdóttir M.K, Cajander Å, Gulliksen J (2012) The Big Picture of UX is Missing in Scrum Projects. In Proceedings of the 2nd International Workshop on the Interplay

between User Experience Evaluation and Software Development, In conjunction with the 7th Nordic Conference on Human-Computer Interaction

[21] Schwaber K (1995) SCRUM Development Process. in Workshop Report: Sutherland, Jeff. Business Object Design and Implementation of 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications Addendum to the Proceedings. 6:4:170-175

[22] Deemer P, Benefield G (2006) Scrum Primer, Yahoo, 2006. in The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process

[23] Kniberg H, Skarin H (2010) Kanban and Scrum - making the most of both. C4Media inc

[24] Annika Waern (2011). Personal communication.

[25] Schön D.A (1983). Design as a Reflective Conversation with the Situation. Chap 3 in The Reflective Practitioner - how professionals think in action, Basic Books

[26] Tversky B (2002) What do sketches say about thinking? In Proceedings of AAAI Spring Symposium on Sketch understanding 148-151

[27] Löwgren J, Stolterman E (2004) Design av informationsteknik - materialet utan egenskaper, Studentlitteratur

[28] Sennett R (2008) The Crafsman. Penguin Books

[29] McCullough M (1998) Abstracting Craft - the practiced digital hand, MIT Press

[30] Golsteijn C, van den Hoven E, Frohlich D, Sellen A (2012) Towards a More Cherishable Digital Object. in DIS 2012

[31] _ (2013) Hybrid Crafting: Towards an Integrated Practice of Crafting with Physical and Digital Components. In journal on Personal and Ubiquitous Computing Journal, special issue on Material Interactions - From Atoms & Bits to Entangled Practices.

[32] Rosner D.K, Ryokai K (2009) Reflections on craft: probing the creative process of everyday knitters. In Proceedings of the Seventh ACM Conference on Creativity and Cognition 195-204

[33] Wallace J, Press M (2004) All This Useless Beauty: The Case for Craft Practice in Design For a Digital Age. The Design Journal 7 (2):42-53

[34] Dourish P, Mazmanian M (2011) Media as Material: Information Representations as Material Foundations for Organizational Practice. In Proceedings of the Third International Symposium on Process Organization Studies

[35] Bertelsen O.W, Breinbjerg M, Pold S (2007) Instrumentness for creativity mediation, materiality & metonymy. In the 6th Conference on Creativity & Cognition 233-242

[36] Robles E, Wiberg M. (2011) From materials to materiality: thinking of computation from within an Icehotel. interactions 18:32-37

[37] Bergström J, Clark B, Frigo A, Mazé R, Redström J, Vallgårda A (2010) Becoming materials: material forms and forms of practice, Digital Creativity 21:3 155-172

[38] Glaser B, Strauss A (1967) Discovery of Grounded Theory. Strategies for Qualitative Research. Sociology Press

[39] Glaser B (1999) The Future of Grounded Theory. Qualitative Health Research, vol 9, issue. 6, pp836-845

[40] Hartman J (2001) Grundad Teori. Studentlitteratur
[41] Guvå G, Hylander I (2003) Grundad teori ett teorigenererande forskningsperspektiv. Liber
[42] Bjurwill C (1998) Reflektionens praktik. Studentlitteratur
[43] Schön D.A (1983) Patterns and Limits of Reflection-in-Action Across the Professions. Chap 9 in The Reflective Practitioner - how professionals think in action, Basic Books
[44] Martin R.C, Feathers M.C, Ottinger T.R 2010. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall
[45] Seibel P (2009) Coders at Work - Reflections on the craft of programming. Chapter 10, Dan Ingalls. 373-412. Apress
[46] Redström J (2005) On Technology as Material in Design. In Design Philosophy Papers: Collection Two: 31-42. Team D/E/S Publications
[47] Salomon O (1891) 'Introductory Remarks', from The Teachers' Handbook of Slöjd. Boston: Silver, Burett & Co excerpted in: Adamson G (ed) The craft reader. Berg, Oxford
[48] Moggridge W (2007) Designing Interactions. MIT Press
[49] Löwgren J (2007) Interaction design, research practices and design research on the digital materials. Published at webzone.k3.mah.se/k3jolo 2007-03-16
[50] Kapor M (1991) A Software Design Manifesto. Dr.Dobbs Journal