The Craft of Code: Practice and Knowledge in the Production of Software

Pierre Depaz

January 2021

1 Introduction

Software development as a practice has been developing for the past sixty years, emerging as a corollary from the field of computer science. In the 21st century, the importance of code in everyday lives has been highlighted from general-audience journalism (Paul Ford, What is code) to government initiatives (Obama, Code). However, the distinction between programming and computer science is a blurry one. While a computer science degree does provides the appropriate formation for a programmer, successful programmers do not necessarily need a computer science background in order to be competent at their jobs. Indeed, while programming has historically emerged as an occupation which responded to the *ad hoc* requirements of computing as a developing field (Chun, Software Wants to Be Overlooked), developing further into a codified and identified practice (Djikstra), programming has nonetheless often been approached in media- and software-studies with a focus on the phenomenon of computation (notably, Galanter, Katherine Hayles, McKenzie), rather than on the reality of programming.

Starting from this observation, this article proposes to investigate a different tradition than that of the sciences to highlight the specificities of programming as a practice—the tradition of craftsmanship. Indeed, code isn't

just code, but rather a myriad of socio-technical *assemblages* composed of programming languages (e.g. Ruby, C, Julia, JavaScript), operating systems (e.g. Linux, BSD, OSX, Windows) and tools (e.g. IDEs, debuggers, compilers and processors). Those assemblages are in turn used within a cultural context made up of stories, sayings and texts, both from academic and folklore origins. This approach relies on a shift from a conceptual perspective of code, ton one in which the word *code* encapsulates varieties of activities (Brian Hayes, Cultures of Code), in which the variety of practices, self-identifications and narratives from programmers themselves is put at the forefront.

While links between craftsmanship and programming have existed as self-proclaimed ones by programmers themselves, as well as incidentally by academics (Sennett, Chandra), they have not yet been elucidated under specific angles. Indeed, craftsmanship as such is an ever-fleeting phenomenon, a practice rather than a theory, in the vein of Michel De Certeau's *tactics*, bottom-up actions designed and implemented by the users of a situation, product or technology as opposed to *strategies*, in which ways of doing are prescribed in a top-down fashion. It is this practical approach that this article chooses, the informal manners and standards of working, in order to provide an additional, cultural studies perspective, to its media and software studies counterparts.

A comparative approach of a broad mode of economic and cultural activity (craftsmanship) with a narrower technical know-how (software development) asks us first to verify to what extent such an approach is even valid. How, then, is the designation by software developers of their own practice as craftsmanship relevant? Where is the comparison a productive one, and where does it show its limits? How can such comparison enrich both our understanding of code as a practice, as well as craftsmanship practices within the highly networked environment that has become the backdrop of the 21^{st} century? Particularly, how does it re-present processes of knowledge acquisition and aesthetic judgment?

The article proceeds in a comparative fashion, mobilizing texts about craftsmanship as well as sources from the field of programming, describing programmers' self-identification with craftsmanship. From those, I analyze how those references by programmers enter into a productive dialogue with our historical and cultural conception of craftsmanship. To do so, I approach these questions through three different, contiguous topics. First, the focus is set on the historical unfoldings of craftsmanship and software development. After inquiring into the modes of organization and the economical development of both craftsmanship and software development, this section focuses on a particular comparison: that of programming with building, and in particular its relationship with architecture. Second, we shift our focus from a broad view to the specific practices of knowledge acquisition and production. There, we highlight the similarities in terms of tacit and personal knowledge, as well as the means of education and information available both to traditional craftspeople and software developers. This, in turn allows us to discuss the differences of learning environments by taking into account the networks of environment that have developed exponentially with the Internet. Finally, the third section turns to aesthetic judgment of the crafted product. Building upon discussions of code as art and code as literature, the focus here is on the standards which allow practicioners to ascribe beauty to a piece of software; particularly, this section discusses the materiality of code not in terms of bits, bytes and languages, but rather as a material that is worked with and worked through.

These incursions in the practices of software development via the lens of craftsmanship as a cultural practice concludes on the productive differences and similarities in terms of knowledge circulation and materialities when programming code; ultimately resulting in new understandings of both craftsmanship and software development as mutually influencing practices.

2 Social and historical developments

This section starts by providing an overview of the various perspectives and realities on craftsmanship, starting from the Late Middle-Ages until the 20^{th} century. This allows us to highlight some initial important features of craftsmanship: social organization, the nature of work and the transmission of knowledge. Jumping to the history of software development, starting in the mid-20^{th} century, it looks at the claims that programmers make about themselves in relation with the term and concept of craftsmanship. By examining formal and informal texts, we focus on the fact that software developers ground their practice in passion, know-how and myths. Finally, we inspect the place of architecture, both in historical craftsmanship and contemporary software development, in order to qualify further the relationship between design and implementation in those two fields.

2.1 Craftsmanship throughout history

Craftsmanship in our contemporary discourse seems most tied to a retrospective approach: it is often qualified as that which was *before* manufacture, and the mechanical automation of production. So while the practice of developing a skill in order to build something with a functional design has been considered at its apex of craftsmanship in Western late Middle-Ages, it should be noted here that non-Western craftsmanship are as equally rich and unique as their Western counterparts, for instance in China (https://www.atlantis-press.com/proceedings/ichssr-15/25840496) and Japan (The Unknown Craftsman: A Japanese Insight Into Beauty); however, these lie beyond the immediate scope of this article. Following Sennett, we'll define craftsmanship as *hand-held, tool-based work which produces functional artefacts, and in the process of which is held the possibility for unique mistakes* (Sennett).

Late Middle-Ages craftsmanship stands out as such for a couple of rea-

sons: their socio-economic organization, and their relationship to knowledge. First and foremost, craftspeople were indissociable from the guilds they belonged to (Guilds and Civil Society in European Political Thought from the Twelfth Century to the Present by Anthony Black). As tightly-knit communities, they exhibited strong cohesion: vertically, between a master and their apprentices, and horizontally, between equal practicioners, enforcing a uniform quality control assurance and price management (Managerial Techniques by Wolek). This cohesion, in turn, has limited the amount of individual fame and glory that craftspeople could accumulate, as compared to fine-artists (The Study of Medieval Craftsmanship, by Daniel V. Thompson Jr.).

Another aspect of the craftsmanship of this time is the relationship that they maintained with explicit, formalized standards. While various crafts did include specific lexical fields to describe the details of their trade (The Craftsman Revealed: Adriaen de Vries), usually compiled into glossaries, the standards for quality were less explicit. Cennino Cennini, in his *Libro dell'arte*, one of the first codexes to map out technical know-how necessary to a painter in the early Renaissance, lays out both practical advice on specific painting techniques, but does not explicitly lay out how to make something *good*. The assumption here is that such a quality work is both implied and obvious: a good craftsman knows a quality work when they see it (Sennett). Further work, at the eve of the Industrial Revolution, continued on this intent to formalize the practice of craftsmanship (Diderot, the Mechanical Arts, and the Encyclopédie: In Search of the Heritage of Technology Education).

A particularity of craftspeople's work then, is its implicit component, highlighted by the increasing amount of effort dedicated by others to laying out the nuts and bolts of the practice. Another one is the alleged incompatibility of craftsmanship (doing mainly by hand) with manufacture (doing mainly by machine) (Ruskin, The Wheelwright's shop). However, studies have shown that the craftsmanship, rather than standing at the strict op-

posite of the industrial (Calculating Machines), has been integrated into the process of modern industrialization (Gordon, David McGee From Craftsmanship to Draftsmanship). The practice of the craftsman, then, integrates into the design and operation of machines, and informs ways of making in our contemporary world (Optics).

These characteristics of tight and rigid communities, implicit knowledge, and ambiguous separation with design, framing the foundation of a desire for good work are particularly highlighted in the field of the built environment, and later in the development of architecture. Before examining how such a field has a connection to software development, we take a look at programming's emergence as a field and its proclaimed similarities with craftsmanship.

2.2 Software developers as craftsmen

Computer programming as an activity came to be as an offshoot of computer science, perhaps best illustrated by the collaboration of Charles Babbage and Ada Lovelace on The Analytical Engine, the prototype of the modern computer. With Babbage acting as the overall designer, Lovelace was key in practically implementing some of the mathematical formulas which The Analytical Engine was built to solve. What we see here is a dyad of work, distinguished between design and conception on one side, and implementation and practice on the other side. These two approaches are echoed throughout the early days of programming (1950s-1970s), with programmers becoming distinct from computer scientists by their approach to the problem (they'd rather write code on a terminal than write algorithms on a piece of paper) and by their background (trained as scientists but more comfortable with tinkering). In particular, the group of computer enthusiasts described as hackers developed organizational features similar to their historical counterparts: work was being done on distinct topics and fields in different geographic locations (Stanford, MIT, Bell Labs) (A Brief

History of Hackerdom, Eric S. Raymond), knowledge was acquired through practice by joining a laboratory, inquiring into peers' work (Hackers: Heroes of the Computer Revoution) and later formalized into bottom-up archives (Jargon File). Additionally, little accountability was required when it came to design explicitness. As examples, both the UNIX operating system and the TCP/IP protocol were originally realized without overarching supervision and without extensive ongoing documentation (Coders At Work, Cathedral And Bazaar).

As computer science solidified as a distinct field in the 1960s (HE DEVELOPMENTOF COMPUTER SCIENCE: A SOCIOCULTURAL PERSPECTIVE, MATTI TEDRE), there was a process of formalizing the hitherto *ad hoc* techniques of programming computers. As a response to the myth of carefully hand-made code (The Story of Mel, The Real Programmer) and unconstrained approaches to writing code (GOTO Statement Considered Harmful, Djikstra) came the structured programming approach, initally proposed by E. W. Djisktra (Djikstra). With the operating system and the personal computer revolutions, access to tools became widespread, and transformed tightly communities into a global network of exchange, first via Usenet (source), then through the Web (source). Inquiries into the relationship of craftsmanship with programming started to take place in the mid-1970s from an educational perspective (Craftsman or Scientist?), from an organizational perspective (Mythical Man Month, No Silver Bullet) and an inter-personal perspective (The Psychology of Programming), and culminated with the publication of several trade books (Clean Code, Software Craftsmanship), explicitly connecting the craft of programming with previous craft activities, and emphasizing the need for intrinsic motivation and the aim of a job well-done (Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman, Code Craft: The Practice of Writing Excellent Code).

short conclusion here

2.3 Architecture between craftsmanship and programming

The field of architecture helps us tie these two traditions together a little more explicitly. Architecture as a field and the architect as a role have been solidified during the Renaissance (The Term 'Architect' in the Middle Ages), consecrating a separation of abstract design and concrete work, in which the craftsman is relegated to the role of executioner, until the arrival of civil engineering and blueprints overwhelmingly formalized the discipline.

The classical architect, here, serves as the counterpart to the computer scientist, except in an inverse relation: the architect emerged from centuries of hands-on work, while the computer scientist (formerly known as mathematician) was first to a whole field of practicioners as programmers, followed by a need to regulate and structure those practices. Different sequences of events, perhaps, but nonetheless mirroring each other. On one side, construction work without an explicit architect, under the supervision of bishops and clerks, did indeed result in significant results (Notre Dame de Paris, Basilica of Sienna). On the other side, letting go of structured and restricted modes of working characterizing computer programming up to the 1980s resulted in a comparison described in the aptly-named *The Cathedral and the Bazaar*. This essay described the Linux project, the open-source philosophy it propelled into the limelight, and how the quantity of workers without rigid working structures (which is not to say without clear designs) can result in better quality work than if made by a few, select, highly-skilled individuals.

What we see, then, is a similar result: individuals can cooperate on a long-term basis out of intrinsic motivation, and without clear, individual ownership of the result; a parallel seen in the similar concepts of *collective craftsmanship* in the Middle-Ages and the *egoless programming* of today (Mythical Man Month). The further sections will investigate how such a phenomena of building complex structures through horizontal networks is

possible, from both epistemological and aesthetic perspectives.

3 Knowledge acquisition and production

3.1 Bus factor and implicit knowledge

The problem of knowledge in software development can be examplified by the ”bus factor” (https://en.wikipedia.org/wiki/Bus_factor). This practical term describes the risk of crucial information being lost due to the disappearance or incapacity of one of the programmers of the project, and it provides us with a hint with the problem of *essential complexity* (No Silver Bullet). Given the inherent complexity of programming as a task, along with the compulsive behaviours sometimes exhibited by programmers as a by-product of intrinsic motivation (Weizenbaum), the gap between design and implementation—the domain of the craftsman—relies on tacit knowledge (Collins).

Explicit knowledge, in programming, is carried through books, academic programs and, more recently, web-based content either structured (MOOCs, Codeacademy, Khan Academy) or unstructured (blog posts, forums), but has proven to be insufficient to reach an expert level (Models and theories of programming strategy). As demonstrated by a popular comic (https://xkcd.com/844/), the road to good code is unclear, particularly when communicated in such a highly-formal language as diagramming. Given the fact that an individual can become a programmer through non-formal training—as opposed to, say, an engineer or a scientist—, self-learning has to deal with an acquisition of implicit knowledge.

3.2 Knowledge in traditional craft

The acquisition of such implicit knowledge in craftsmanship takes place in two different ways: the apprentice-master relationship, and the act of

copying. First comes the apprentice-master relationship, in which a learner starts by exactly copying the way of working of the master (Sennett), resulting in a *teaching by showing*, where important aspects of the craft are being demonstrated to the apprentice by a more experienced practicioner, rather than formalized and learned *a priori* of the practice. Relationship to a master, when not implemented explicitly through practices such as pair programming (Pair Programming Illuminated), is re-interpretated by programmers through fictional accounts designed to impart wisdom on the readers, and taking inspiration from Taoism and Zen (The Tao of Programming, Some AI Koans http://catb.org/ esr/jargon/html/koans.html). From higher-level programming advice weaving in the figuration of leading programmers such as Marvin Minsky and Donald Knuth, this sort of informal teaching by showing has been implemented in various languages as a practical learning experience (http://rubykoans.com/). Without the presence of an actual master, the programming apprentice nonetheless takes the program writer as their master to achieve each of the tasks assigned to them. The experience historically assigned to the master craftsman is delegated into the code itself, containing both the problem, the solution to the problem and hints to solve it, straddling the line between formal exercises and interactive practice.

Code's ability to be copied and executed on many different machines provides an interesting counterpoint to the argument of software as craftsmanship. Traditionally, since craftsmanship has been understood as that which is done by hand, and since craftsmen were working with unique artefacts (no artefact can be perfectly copied), copying one's solution proved to be physically inconceivable. The realm of software, on the opposite, relies heavily on the technical affordace of code to be duplicated, uploaded, downloaded and executed on multiple platforms through source code files. The first immediate consequence of this is the ability for all to inspect and use source code, both on an institutional level (as guaranteed by projects such as GNU (gnu.org)), and on a vernacular level (as enabled by Web 2.0

platforms such as StackOverflow). Even though the ability to perfectly copy anyone else's work became widely available to programmers, the difference between amateur and expert programmers lied in the extent to which they blindly copied external code, or write their own, inspired by the external code (https://softwareengineering.stackexchange.com/questions/36978).

Practices from Eastern craftsmanship further qualify these essentially different approaches to copying. *Moxie*, a Chinese term for copying and practice, is a key concept to understand how an apprentice can equal his master through thougtful replication (Influence of Global Aesthetics on Chinese Aesthetics: The Adaptation of Moxie and the Case of Dafen Cun), an approach equally present in Japanese crafts history (Copying the Master and Stealing His Secrets: Talent and Training in Japanese Painting). Here again, copying from established quality work to seize their elusive essence is an essential aspect to craftsmanship.

3.3 The problem with copying

If implicit knowledge can be acquired through a showing and copying of code, software development as a craft presents an additional dimension to this, which we call *piecemeal knowledge*. Best represented by Stack Overflow, a leading question and answer forum for programmers, on which code snippets are made available as part of the teaching by showing methodology, this piecemeal knowledge can both help programmers in solving issues as well as deter them in solving issues *properly* (Understanding Stack Overflow Code Fragments). Code as such is freely and easily accessible as piecemeals, but lack the proper context to improve knowledge in the user.

So while programmers are to acquire implicit knowledge through a process of learning by doing, we now need to investigate how much of it happens through observing. Implied in the apprentice-master relationship is that what is observed should be of *good quality*. Coming back to the relationship between architecture and software development, Christopher

Alexander asks, in the preface of Richard P. Gabriel's *Patterns of Software*,

> "For a programmer, what is a comparable goal? What is the Chartres of programming? What task is at a high enough level to inspire people writing programs, to reach for the stars?"

Indeed, if a craftsman learns their trade by comparing their work with work of a higher quality (either their master's, or publicly available works), the programmer is faced with a different problem: a lot of examples, but a few good ones. At this point, copyright stands in the way of pedagogical copying. With software become protectable under copyright laws in the 1980s (Computer Software as Copyrightable Subject Matter), great works of programming craft became unacessible to programmers, despite the value they would bring in knowledge acquisition. One of the most famous examples is *Lions' Commentary on UNIX 6th Edition, with Source Code* by John Lions, an annotated edition of the UNIX source code, was circulated illegaly in classrooms for twenty years before its official publication was authorized by the copyright owners.

With implicit knowledge being a key component in both disciplines, its manifestation through the copying of source code in software development is hampered either by decontextualized code snippets or by copyrighted protection on works. Still, if the key advice given to apprentice-programmers is to practice, this hints at another aspect: direct engagement with code.

4 Material aesthetics

at the heart of knowledge transmission and acquisition stands the *practice*, and inherent in the practice is the *good practice*, the beautiful one. this section investigates the aesthetics of code within the broader context of the aesthetics of craftsmanship, highlighting how code can act as a material.

4.1 The Aesthetics of Craftsmanship

- pye - sennett - harold osborne

4.2 The Aesthetics of Code

- functional beauty (also in osborne: the difference between fine arts and craftsmanship is selflessness, practical utility) - clear, simple, elegant - no ornament?

4.3 Code as material

- essential complexity - code smells, spaghetti, etc. - the role of tools (terminal)

5 Conclusion

there are some self-proclaimed similarities, but the most reliable/salient ones come from the parallel with architecture: what makes possible these works in software is a similar conception of knowledge (tacit, practical) and aesthetics (functional, elegant), making ultimately the case for code as material. (find a better definition of *material*)