The role of Aesthetics in the Understandings of Source code

Pierre Depaz
under the direction of Alexandre Gefen (Paris-3)
and Nick Montfort (MIT)

ED120 - THALIM

last updated - 25.10.2022

Chapter 1

Introduction

This thesis is an inquiry into the formal manifestations of source code, into how particular configurations of lines of code allow for aesthetic judgments and on the purposes that such configurations serve. The implications of this inquiry will lead us to consider the different ways in which people read and the different ways in which source code can be represented, depending on what it aims at accomplishing, and on the contexts in which it operates. This study on source code involves the different groups of people which read and write it, the purposes for which they write it, the programming languages they use to write it, and the natural language they use to speak about it. Most importantly, this thesis focuses on source code as a material and linguistic manifestation of a larger digital ecosystem of software and hardware to which it belongs. Since source code is only one component of code, as we will see below, this thesis also aims at studying the reality of written code, rather than its conceptual interpretations.

Starting from pieces of source code, this thesis will aim at assessing what programmers have to say about it, and attempt to identify how one or more specific *aesthetic fields* are used to refer to it. This aim depends on two facts: first, source code is a medium for expression, both to express the programmer's intent to the computer (Dijkstra, 1982) and the program-

mer's intent to another programmer (Abelson et al., 1979)—here we also consider the same individual at two different points in time as two different programmers. Second, source code is a relatively new medium, compared to, say, paint or mechanics. As such, the development and solidification of aesthetic practices—that is, of ways of doing which do not find their immediate justification in a practical accomplishment—is an ongoing research project in computer science, software development and the digital humanities. Formal judgments of source code are therefore existing and well-documented, and are related to a need for expressiveness, as we will see in chapter 2, but their formalization is still an ongoing process.

Source code thus can be written in a way makes it subject to aesthetic judgments by programmers; that is, code *has* aesthetics, but it is unclear exactly *which* aesthetics. Indeed, these aesthetic judgments as they exist today rely on different domains to assess source code, as a means to grasp the cognitive object that is software. These draw from metaphors ranging from literature, architecture, mathematics and engineering. And yet, source code, while related to all of these, isn't exactly any of the them. Like the story of the seven blind men and the elephant (Chun, 2008), each of these domains touch on some specific aspect of the nature of code, but none of them are sufficient to entirely provide a solid basis for the aesthetic judgments of source code. It is at the crossroads of these domains that this thesis locates its work.

The examination of source code, and of the discourses around source code will integrate both the diversity of ways in which source code can exist, and the invariant aspects which underline all diverse approaches of source code. Particularly, we will see how each groups of practicionners tend to deploy references to underlying, systematic conceptual metaphors drawing from the domains above, but also how these references overlap across groups. The point of overlap, as we will demontrate, is that of *using a formal linguistic system to communicate the understanding of complex cognitive structures*. Through an interdisciplinary approach, we attempt to

connect this formal symbol system to the broader role of aesthetics as a cognitive mechanism to deal with complexity.

The rest of this introduction will consist in establishing a more complete view of the context in which this research takes place, from computer science to digital humanities and science and technology studies. With this context at hand, we will proceed to highlight the specific problems which will be tackled regarding the current place of aesthetics in source code. After outlining our methodology and the theoretical frameworks which will be mobilized throughout this study, we will sketch out how the different chapters of this thesis will attempt at providing some responses to our research questions.

1.1 Context

1.1.1 The research territory: code

Most of our modern infrastructure depends, to a more or less dramatic extent, on software systems (Kitchin & Dodge, 2011), from commercial spaces to classrooms, transport systems to cultural institutions, scientific production and entertainment products. Software regulates and automates the information stores, exchanges and creation which compose each of these domains of human activities. The complex processes are described in what is called source code, a vast and invisible set of texts. The number of lines of code involved in running these processes is hard to estimate; one can only rely on disclosures from companies, and publicly available repositories. To give an order of magnitude, all of Google's services amounted to over two billions source lines of code (SLOC) (@Scale, 2015), while the 2005 release of the OSX operating system comprised 86 millions lines of code, and while the version 1.0 of the Linux kernel (an operating system which powers most of the internet and specialized computation) totalled over 175,000 SLOC, version 4.1 jumped to over 19.5 million lines of code in

the span of twenty years (Wikipedia, 2021).

Given such a large quantity of textual mass, one might wonder: who reads this code? To answer this question, we must start diving a bit deeper into what source code really is.

Source code consists in a series of instructions, composed in a particular programming language, which is then processed by a computer in order to be executed. For instance, using the language called Python, the source code:

```
a = 4
b = 6

def compute(first, second):
    return (first * 2) + second

compute(a, b)
```

consists in telling the computer to store two numbers in what are called *variables*, then proceeds with describing the *procedure* for adding the double of the first terms to the second term, and concludes in actually executing the above procedure. Given this particular piece of source code, the computer will output the number `14` as the result of the operation `(4 * 2)+ 6`. In this sense, then, source code is the requirement for software to exist: since computers are procedural machines, acting upon themselves and upon the world, they need a specification of what to do, and source code provides such a specification. In this sense, computers are the main "readership" of source code.

However, it is also a by-product of software, since it is no longer required once the computer has processed and stored it into a *binary* representation, a series of 0s and 1s which represent the successive states that the computer has to go through in order to perform the action that was described in the source code. *Binary code* is what most of the individuals who interact

with computers deal with, in the form of packaged applications, such as a media player or a web browser. They (almost) never have to inquire about, or read, such source code. In this sense, then, source code only matters until it gets processed by a computer, through which it realizes its intended function.

From another perspective, source code isn't just about telling computers what to do, but also a key component of a particular economy: that of software development. Software developers are the ones who write the source code and this process is first and foremost a collaborative endeavour. Software developers write code in successive steps, because they add features over time, or they fix errors that have shown up in their software, or they decide to rewrite parts of the source code based on new ideas, skills or preferences. In this case, source code is not used to communicate to the computer what it does, but to other software developers what the *intent* of the software is. Source code is then the locus of human, collaborative work; it represents iterations of ideas, formalization of processes and approaches to problem-solving. As Harold Abelson puts it,

> "Programs must be written for people to read, and only incidentally for machines to execute." (Abelson et al., 1979).

Official definitions of source code straddle this line between the first role of source code (as instructions to a computer) and the second role of source code (as indications to a programmer). For instance, a definition within the context of the Institute of Electrical and Electronics Engineering (IEEE) considers source code *any fully executable description of a software system, which therefore includes various representations of this description, from machine code to high-level languages and graphical representations using visual programming languages* (Harman, 2010). This definition focuses on the ability of code to be processed by a machine, and mentions little about its readability (i.e. processability by other humans).

On the other hand, the definition of source code provided by the Linux

Information Project[1] focuses on source code as *the version of software as it is originally written (i.e. typed into a computer), by a human in plain text (i.e. human-readable, alphanumeric characters).* (Project, 2004). The emphasis here is on source code as the support of human activity, as software developers need to understand the pieces of code that they are creating, or modifying. Source code thus has two kinds of readabilities: a computer one, which is geared towards the correct execution of the program, and a human one, which is geared towards the correct understanding of the program. In the lineage of this human-readability, we can point to the Free Software Foundation's equation of the free circulation and publication of source code with the free circulation of publication of ideas. Particularly, Freedom 1 (*The freedom to study how the program works, and adapt it to your needs*) and Freedom 2 (*The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.*) as stated in the FSF's definition of Free Software stipulates that access to source code is required to support these freedoms, a version of source code that is *not concealed*, i.e. readable by both human and machine (Stallman & Free Software Foundation , Cambridge(2002).

In addition to this ability to communicate the ideas latent in it, source code, as an always potentially collaborative object, can be the locus of multiple subjectivities coming together. As Krysa and Sedek state in their definition, *source code is where change and influence can happen*, and where *intentionality and style are expressed* (Fuller, 2008). In their understanding, source code shares some features with natural languages as an intersubjective process (Voloshinov & Bachtin, 1986), and as such is different from the machine language representation of a program, an object which they do not consider source code due to its unilaterality. The intelligibility of source code, they continue, facilitates its circulation and duplication among programmers. It is this aspect of a socio-technical object that we consider as important as its procedural effectiveness.

[1] https://linfo.org/sourcecode.html

In this research, we build on these definitions to propose the following:

> *Source code is defined as one or more text files which are written by a human or by a machine in such a way that they elicit a meaningful and succesfully actionable response from a digital compiler or interpreter, and describe a software system. These text files are the starting point to produce an execution of the system described, whether the very first starting point, or an intermediate representation used for subsequent compilations. These files are collectively called* program texts.

This definition takes into account a broad view of source code, including steps such as intermediate representations (transitory representations from one version of the source to another one), but also obfuscations (deliberately complicating the code to prevent human-readability while maintaining machine-readability) and minifications (reducing the amount of characters used in source code to its minimum). This will allow us to compare human-authorship of source code, machine-authorship, and hybrid modes, in which a human writes unreadable code with the help of tools. One aspect that is being more narrowly defined for the purpose of this study is the actual manifestation of code: while multiple media for source code exist, we exclude here all of those that are not written in the UTF-8 character set—i.e. textual representations. Since one of the questions of this study is to examine the literariness of source code aesthetics, other forms of source code, such as visual programming languages or biological computation, stand outside the scope of this study and should be investigated in subsequent work.

As for the term *program text*, it is chosen in order to highlight the dual nature of source code: that of a computational artefact to be formally processed and unambiguously understood (Detienne, 2012), and that of a open-ended, multi-layered document, in the vein of Barthes' distinction between a text and a work (Barthes, 1984).

1.1.2 Beautiful code

Under this definition of source code textually represented, we now turn to the existence of the aesthetics of such *program texts*. To contextualize this existence, we first need to touch upon the history and practice of software development. As an economic activity, software development came from a bottom-up dynamic, a *de facto* activity which was not expected in the early days of computing, where most of the work was divided between mathematics and engineering. Its earliest manifestation can be found in the physical rewiring process of mainframes in order to perform a specific computation, something more akin to firmware than to software. These rewiring tasks were done by mostly female assistants, under the direction of mostly male mathematicians (Chun, 2005), and consisted in a simple translation task which did not require any particular attention, or any particular skill. The recognition of software engineering as its own field came as its unique domain of expertise was required in larger engineering projects—for instance, the term *software engineering* was coined in the late 1960s by Margaret Hamilton and her team as they were working on the Apollo 11 Lunar Module software (Mindell, 2011). In the same decade, the first volume of *The Art of Computer Programming*, by Donald Knuth, addresses directly both the existence of programming as an activity separate from both mathematics and engineering, as well as an activity with an "artistic" dimension (Knuth, 1997). The first volume opens on the following paragraph:

> *The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. This book is the first volume of a multi-volume set of books that has been designed to train the reader in the various skills that go into a programmer's craft. (Knuth, 1997)*

Considered one of the most canonical textbooks in the field, *The Art of Computer Programming* highlights two important aspects of programming for our purpose: that it can be an aesthetic experience and that it is the result of a craft, rather than of a highly-formalized systematic process, as we will see in chapter 1. Some of the aeshetic references related to source code are related to its writing and reading being a craft-like activity (Dijkstra, 1982).

Craftsmanship is an essentially fleeting phenomenon, a practice rather than a theory, in the vein of Michel De Certeau's *tactics*, bottom-up actions informally designed and implemented by the users of a situation, product or technology as opposed to *strategies* (de Certeau et al., 1990), in which ways of doing are deliberately prescribed in a top-down fashion. Craft is hard to formalize, and the development of expertise in the field happens through practice as much as through formal education (Sennett, 2009). It is also one in which function and beauty exist in an intricate, embodied and implicit relationship, based on subjective qualitative standards rather than strictly external measurements, with the former rarely being explicitly stated (Pye, 2008).

Approaching programming (the activity of writing and reading code) as a craft (Lévy, 1992) connects to the multiple testimonies of encountering beautiful code, some of which have made their ways into edited volumes or monographs (Oram & Wilson, 2007; Chandra, 2014; Gabriel, 1998). Additionally, informal exchanges among programmers on forums, mailing lists, blog posts and code repositories[2] often mention beautiful code, either as a central discussion point or simply in passing. These testimonies constitute the first part of our corpus, as sources in which programmers comment on the aesthetic dimension of their practice. The second part of the corpus is composed of selected program texts, which we will examine in order to identify and formalize which aspects of the textual manifestation of software can elicit an aesthetic experience.

[2] See Annex for the list of collected corpus

So the existence of something akin to art, something beautiful and pleasurable emerging from the reading and writing of source code has been acknowledged since the 1960s, in the early days of programming as a self-contained discipline. However, the formalization of an aesthetics of source code first requires a working definition of the concept of *aesthetics* as used in this study.

There is a long history of aesthetic philosphical inquiries in the Western tradition, from beauty as the imitation of nature[3], moral purification[4], cognitive perfection[5], sensible representations with emotional repercussions[6]. The common point of these definitions is that of *sensual manifestation*, that is the set of visible forms which can enable an *aesthetic experience*, a cognitive state of pleasure relying on, amongst others, an object, a sense of unity and of discovery (Beardsley, 1970).

The definition of aesthetics that we will use in this thesis starts from this requirement of sense perception, and then builds upon it using two theoretical frameworks: Nelson Goodman's theory of symbols (Goodman, 1976) and Gérard Genette's distinction between fiction and diction (Genette, 1993). The former provides us with an analysis of formal systems in aesthetic manifestations and their role in a cognitive process, while the second offers a broadened perspective on what qualifies as textual arts, or literature.

Goodman's view on aesthetics is an essentially communicative one: we use aesthetics to carry across more or less complex concepts. This communication process happens through various symbol systems (e.g. pictural systems, linguistic systems, musical systems, choreographic systems), the nature and organization of which can elicit an aesthetic experience. His conception of such an aesthetic experience isn't one of self-referential composition, or of purely emotional pleasure, but a cognitive one, one

[3]Plato, Republic
[4]Aristotle, Poetics; Kant, Critique of the Power of Judgment
[5]Leibniz, Ars Combinatoria
[6]Baumgarten, Aesthetics

which belongs to the field of epistemology (Goodman, 1976). The symbol systems involved in the aesthetic judgment bear different kinds of relations to the worlds they refer to—such as denoting, representing, ressembling, exemplifying— and their purpose is to communicate a truth about these worlds (Goodman, 1978). In Goodman's view, the arts and the sciences are, in the end, two sides of the same coin. They aim at providing conceptual clarity through formal, systematic means, and the arts—understood here in the broad, Renaissance sense of liberal arts—can and should be approached with the same rigor as the sciences. In our case, programming, with its self-proclaimed craft-like status and its mathematical roots, stands equally across the arts and sciences.

His use of the term *languages* implies a broader set of linguistic systems than that of strictly verbal ones. This approach will support our initial conception of programming languages as verbal systems, but will allow us not to remain constrained by traditional literary aesthetics such as verse, rhyme or alliteration. To what extent is programming a linguistic activity is going to be one of the main inquiries of this thesis, and Goodman's extended, yet rigorous definition leaves us room to explore the semantic and syntactic dimension of source code as one of those languages of art. Programming languages as symbol systems will be explored further in Chapter 5.

With this analytical framework allowing us to analyze the matter at hand—program texts composed by a symbol system with an epistemic purpose—we turn to a more literary perspective on aesthetics. Genette's approach to literature, *the art of language*, results in the establishment of two dichotomies: fiction/diction, and constitutivity/conditionality. In *Fiction and Diction* (Genette, 1993), he extends previous conceptions of literature and poetics, from Aristotle to Jakobson, in order to broaden the scope of what can be considered literature, by questioning the conditions under which a text is given a literary status. As such, he establishes the existence of conditional literature alongside constitutive literature: the former gains

its status of a literary text from the individual, subjective aesthetic judgment bestowed upon it, while the latter relies on pre-exisiting structures, themes and genres. This approach paves the way for an extending of the domain of literature (Gefen & Perez, 2019), and a more subtle understanding of the aesthetic manifestation in an array of textual works.

Genette also makes the distinction between fiction, with the focus being the potentiality of a text's object, its imaginative qualities and themes, and diction, with an emphasis on the formal characteristics of the text. Since code holds two existences, one as executed, and one as written, I propose to map Genette's concept of fiction on to source code when the latter is considered as a purely functional text—i.e. what the source code ultimately does in its domain of application. Because source code always holds software as a potential within its markings, its diction, waiting to be actualized through execution, one has to imagine what this code actually does. Written source code, then, could either be judged primarily on its fiction or on its diction. Since we focus on the written form of source code, and not on the type of its purpose, an attention to diction will be the entry point of this thesis.

A first approach to source code could be *constitutive*, in Genette's terms: a given program text could be considered aesthetically pleasing because the software it generates abides by some definitions of being aesthetically pleasing[7], or because the software itself is considered a piece of art, shown in exhibitions and sold in galleries. However, our empirical approach to source code aesthetics, by examining various program texts directly, and our inquiry into the possibility of multiple aesthetic fields co-existing within source code as a symbol system, asks us to forgo this constitutive definition of an aesthetic work as normative categories within software development. Our focus on sense perception within aesthetics starts from a conditional approach, in which programmers emit an aesthetic judgment on a program text, with an emphasis first on what the source code *is*, and

[7]For instance, Venustas, Firmitas, Utilitas; See Fishwisck, P. (éd), *Aesthetic Computing*

only secondly on what it *does*[8]. This conditional approach implies that we use a conception of the aesthetic that is broader than the artistic and the beautiful, encompassing less dramatic qualifiers, such as *good* or *nice*.

Diction, then, focuses on the formal characteristics of the text. The point here is not to assume an autotelic mode of existence for source code, but rather to acknowledge that there is a certain difference between the content of software and the form of its source—good software functioning beautifully can be written poorly, and poor, buggy software can be written beautifully. This thesis chooses to focus on the formal aspects of code such as not to restrict ourselves to any specific kind, or genre, of program texts, leaving open the possibility for these categories to emerge after our analysis.

So, following Genette's re-asking of the Goodman's question of *When is art?* rather than the historical *What is art?*, we can now proceed with our understanding of aesthetics as a set of physical manifestations which can be grasped by the senses, akin to "the movement of a light, the brush a fabric, the splash of a color" (Ranciere, 2013), which aim at enabling a cognitive, communicative purpose, and which are not exclusively constituted by pre-existing categories.

This overview of the theoretical frameworks of this thesis already implicitly denotes the boundaries of this study. The domain we are investigating here is one that is delimited by both medium and purpose. First, the medium limitations is that of text, in its material sense, as mentioned above in our definition of source code. Second, the purpose limitation is that of computable code, rather than computed code: we are examining latent programs, with their reality as texts and their virtuality as actions, rather than the other way around. Executed software and its set of affordances (e.g. graphical user interfaces (Gelernter, 1998), real-time interactivity (Laurel, 1993) and process-intensive developments (Murray, 1998))

[8] As we've seen with Goodman, there is nonetheless a tight connection between those to states.

differ from the literary and architectural ones that software, in its written form, is claimed to exhibit. However, executable and executed software, being to sides of the same coin, might suggest causal relationships—e.g. the aesthetics of source code affecting the aesthetics of software—and such an inquiry would be best reserved for a subsequent study.

Now that we've explicited our object of study—the formal manifestations of software under its textual form—we can turn to a review of the research that has alredy been done on the subject, before highlighting some of the limitations. These relations between source code and aesthetics have been addressed by academic studies through different, separate dynamics.

1.1.3 Literature review

A literature review on this topic must address the dualistic nature of studies on source code, as research can be distinguished between the fields of computer science and engineering on one side, and that of the humanities on the other. This overview will provide us with a better sense of which aspects of code and aesthetics have been explored until now, and will invite us to address the remaining gaps.

We have seen that most technical literature, starting from *The Art of Computer Programming*, acknowledges the role that aesthetics have to play in the writing and reading of program texts. Along with the positions of Knuth and Djikstra regarding the importance of paying attention to the aspects of programming practice (Dijkstra, 1972) which go beyond strictly mathematical and engineering requirements, Kerninghan and Plauer publish in 1978 their *Elements of Programming Style* (Kernighan & Plauger, 1978). In it, they focus on how code snippets with a given intent could be rewritten in order to keep the same intent but gain in quality—that is, in readability and understandability. For instance, the following program:

```
if(i == 0) c = '0'
if(i == 1) c = '1'
if(i == 2) c = '2'
if(i == 3) c = '3'
if(i == 4) c = '4'
if(i == 5) c = '5'
if(i == 6) c = '6'
if(i == 7) c = '7'
if(i == 8) c = '8'
if(i == 9) c = '9'
```

can be rewritten as:

```
if(i >= 0 && i < 10) c = '0' + i
```

which keeps the exact same functionality, but becomes much clearer. Why it becomes much clearer, though, is thought to be a given for the reader, and not explicited by the authors in terms of concepts such as cognitive surface, repleteness of a symbol system or representation of the main idea(s) at play (promoting an integer to a character, rather than individually checking for each integer case). As the authors do employ terms which will form the basis of an aesthetics of software development, such as clarity, simplicity, or expressiveness, there are nonetheless no overarching principles deployed to systematize the manifestation of such principles, only examples are given.

While Kernighan and Plauer do not directly address the relationship of source code and aesthetics, this is something that Peter Molzberger undertakes five years later through an empirical, qualitative study aimed at highlighting the place aesthetics play in an expert programmer's practice (Molzberger, 1983). Molzberger's study touches upon ideas of over-arching structure, tension between clarity and personality, and levels of expertise in aesthetic judgment. This short paper highlights multiple instances of code deemed beautiful which will be explored further in this thesis, without providing an answer as to *why* this might be the case. For instance,

a conception of code as literature does not explain instances involving switch in scales and directions of reading, or a conception of code as mathematics does not explain the explicitly required need for a personal touch when writing source code (Molzberger, 1983).

In the context of formal academic research, such as the IEEE or the Association for Computing Machinery (ACM), subsequent research focuses on how to quantitatively assess a given quality of source code either through a social perspective on the process of writing (Norick et al., 2010), a semantic perspective on the lexicon being used (Fakhoury et al., 2019; Guerrouj, 2013), an empirical study of programming style in the efficiency of software teams (Reed, 2010; Coleman, 2018) or on the visual presentation of code in the comprehension process (Marcus & Baecker, 1982). These focus on the connection of aesthetics with the performance of software development—beautiful code as being related to a good end-product. These methodologies are mostly quantitative, and do not take into account the "artistry" and "craft" component as laid out by Knuth and Molzberger, but are rather a big-data representation of Kernighan and Plauer's approach.

The development of software engineering as a profession has led to the publication of several books of specialized literature, taking a practical approach to writing good code, rather than a scientific one. Robert C. Martin's *Clean Code*'s audience belongs to the fields of business and professional trade, drawing on references from architecture, literature and craft in order to lay out the requirements of what he considers to be clean code. These specific mechanisms are highlighted in terms of how they will support a productive increase in the quality of software developed, as opposed as being satisfying in and of themselves. *Clean Code* was followed by a number of additional publications on the same topic and with the same approach (Fowler et al., 1999; Arns, 2005; Hunt & Thomas, 1999). Here, these provide an interesting counterpoint to academic research on quality code by relying on different traditions, such as the practical handbook, to explain why

the way code is written is important.

Technical and engineering literature, then, establish the existence of and need for aesthetics, presented as formal properties which then constitute *quality code*. The methodology in these studies is either empirical, in the case of academic articles, looking at large corpora, more rarely interviewing programmers in order to draw conclusions regarding this relationship between formal properties and quality, while monographs and business literature draw on the experiences of their authors as a programmers to provide source code examples of specific principles, without extending on the rationale and coherence of these principles, let alone within a source code-specific aesthetic framework. A particularly salient example is Greg Oram's edited volume *Beautiful Code*, in which expert programmers are invited to pick a piece of code and explain why they like it, sometimes commenting it line by line (Oram & Wilson, 2007). This very concrete, empirical inquiry into what makes source code beautiful does no, however, include a strong enough conclusion as to what *actually* makes code beautiful, but rather writing why they like the idea behind the code, or manifestoes such as Matz's *Code as an Essay*. As such, this monograph will be integrated in our corpus, as commentary rather than academic research. Another limitation to these studies is that they only address one specific group of programmers, and one specific type of software being written. In effect, those who write and read source code are far from being a homogeneous whole, and can be placed along distinct lines—e.g. academics, tinkerers or artists—with distinct practices and standards (Hayes, 2017). In none of these studies is it considered whether the conclusions established for one group would be valid for the others.

Before we move on to the perspective of the humanities, one should also note the specific field of philosophy of computer science, which inquires into the nature of computation, from ontological, epistemological and ethical points of view. These are useful both in the meta positioning they take regarding computer science as they well as how they show that

issues of representation, interpretation and implementation are still unresolved in the field. Particularly, Rapaport's *Philosophy of Computer Science* provides an exhaustive literature review of the different fields which computer science is being compared to, from mathematics, engineering and art but—interestingly—few references to computer science as having any kind of relation with literature (Rapaport, 2005). Another, more specific perspective is given by Richard P. Gabriel in his *Patterns of Software*, in which he looks at software as a similar endeavour as architecture, drawing on the works of Christopher Alexander. The focus is on its creative and relationship to patters, a subject we will investigate more in chapter 3. Finally, Brian Cantwell-Smith's introduction to his upcoming *The Age of Significance: An Essay on the Origins of Computation and Intentionality* touches upon these similar ideas of intentionality by suggesting both that computation might be more productively studied from a humanities or artistic point of view than form a strictly scientifc point of view (Smith, 1998). These philosophical inquiries into computation mention aesthetics mostly on the periphery, but nonetheless challenge the notion of computation as strictly functional, and suggest additional that perspectives on the topic are needed, including that of the arts.

From a humanities perspective, recent literature taking source code as the central object of their study covers fields as diverse as literature, science and technology studies, humanities and media studies and philosophy. Each of these monographs, edited volumes, catalog articles, book chapters or PhD theses, engage with code in its multiple intricacies. Software applications, source code excerpts, programming environments and languages are included in each of these works as primary sources, are considered as text to be read, examined and interpreted.

A first look at *Aesthetic Computing*, edited by Paul A. Fishwick allows us to higlight one of the important points of this thesis: the collection of essays in this collected volume focus more often on the graphical output

of the software's work from the end-user's perspective than on the textual manifestations of their source (e.g. Nake and Grabowski's essay on the interface as aesthetic event) (Fishwick, 2001). As for most studies of aesthetics within computer science, the main focus is on Human-Computer Interaction (HCI) as the art and science of presenting visually the output and affordances of a running program. While a vast and complex field, this is not the topic of this thesis which, rather than focusing on the aesthetics of the computable and executable, is limited to the aesthetics of the computed (texts).

The following works, because of their dealing with source code as text, and due to the background of their authors in literature and comparative media studies, incorporate some aspect of literary theory and criticism, and authors such as N. Katherine Hayles, Maurice J. Black and Alan Sondheim rely on it as their principal perspective. Black, in his PhD dissertation *The Art of Code* (Black, 2002) initiates the idea of a cross between programming and literature, and hypothesizes that writing source code is an act that is closer to modernism than postmodernism, as it relies on concepts of authorship, formal linguistic systems and, to some extent, self-reference. The aim of the study is to show how code functions with its own aesthetic, one which is distinct and yet closely related to a literary aesthetic. After highlighting how the socio-political structures of computing since the 1950s have affected the constitution of the idea of a code aesthetic both in professional and amateur programmers, Black moves towards the examination of code practices as aesthetic practices. Here, Black limits himself to the presentation of coding practices insofar as they are identified and referred to as aesthetic practices, but exclusively through a social, secondhand account, rather than formal, definition of a source code aesthetic.

Black establishes programming as literature, and vice-versa, he assumes that it is possible to write about literature through the lens of source code. However, the actual analysis of source code with the help of formal literary theories is almost entirely side-stepped, mentioning only Perl po-

etry as an overtly literary use of code. In summary, Black provides a first study in code as a textual object and as a textual practice whose manifestations programmers care deeply about, but does not address what makes code poetry different in its writing, reading and meaning-making than natural-language poetry.

N. Katherine Hayles, in *My Mother Was A Computer: Digital Subjects and Literary Texts* (Hayles, 2010), and particularly in the *Speech, Writing, Code: Three Worldviews* essay temporarily removes code from its immediate social and historical situations and establishes it as a cognitive tool as significant in scale as those of orality and literacy (Ong, 2012), and attempts to qualify this worldview both in opposition to Saussure's *parole* and Derrida's *trace*, following cybernetics and media studies thinkers such as Friedrich Kittler and Mark B. Hansen. Specifically, she introduces the idea of a Regime of Computation, which relies on the conceptual specificities of code-based expression (among which: depth, dynamism, fragmentation, etc.). Source-code specific contributions touch upon literary paradigms and cognitive effect in two ways. First, she highlights the way code recombines some traditional dialectics of literary theory, namely paradigmatic/syntagmatic, discreete/continuous, compilation/interpretation, and flat/stacked languages, acting as such as clearly different mode of expression. Second, she draws on a comparison between two main programming paradigms, object-oriented programming and procedural programming, and on the syntax of programming languages, such as C++, in order to highlight a novel relationship between the structure and the meaning of programming texts, a structure which depends on its degree of similarity with natural languages.

While Hayles provides the basis for a much deeper analysis of source code's formal literary properties, she also maintains that source code studies should keep in mind the ever-underlying materiality that this very source code relies on; and then locates this materiality in the embodiment of users and readers, along with authors such as Mark Hansen (Hansen,

2006), Bernadette Wegenstein (Wegenstein, 2010) and Pierre Lévy (Lévy, 1992). Beyond the brief acknowledgment that she has of the political and economical conditions of software development and their impact on electronic texts, Hayles also stops short of considering programming languages in their varieties, and the material apparatuses which support them (documentation, architectures, compilers, tutorials, conferences and communities). Building on this approach, a conception of programming languages as a material seems like a possible avenue for looking into the formal possibilities they afford.

Alan Sondheim's essay *Codework* (Sondheim, 2001), as the introduction of the American Book Review issue dedicated to this specific form, provides another aspect of poetry which integrates source code as a creole language emerging from the interplay of natural and machine languages. Yet, this specific aspect of literary work scans the surface of code rather than with its structure and therefore provides more insight in the anthropology of how humans represent code through speech, rather than representing speech through code. This presents a somewhat postmodern view of programming languages, forcing them upon a relational, mutable conception of language as as series speech-acts, and leaving aside their structural and post-structural characteristics. Codework is essentially defined by its content and *milieu*, one which focuses on human exchanges and bypasses any involvement of machine-processing.

Another perspective on the relationship between speech and code is explored by Geoff Cox and Alex Mclean in *Speaking Code: Coding as Aesthetic and Political Expression* (Cox & McLean, 2013). They establish reading, writing and executing source code as a speech-act, extending J.L. Austin's theory to a broader political application by including Arendt's approach of human activities and labor (Arendt, 1998), from which coding is seen as the practice of producing laboring speech-acts.

They consider source code as a located, instantiated presence, understood as a politically semantic object affecting the multiple economic, so-

cial and discursive environments in which it lives. Focusing on speech particularly, this study doesn't quite address the syntactic specificities of codes, for example by looking at the use of loops, arrays, or other syntactical structures briefly touched upon by Hayles. Side-stepping the particular grammatical features of that speech, the authors nonetheless often illustrate the points they are working through, or begin developing those points, with snippets of code written by either McLean or established software artists, thus engaging with details of source code and taking a step away from the dangers of fetishizing code, or *sourcery* (Chun, 2008). They include both deductive code (commenting existing source code) or inductive code (code written to act as an example to a point developed by the authors), in a show of the intertextuality of program texts and natural texts.

Away from the cultural relevance of code as developed by Cox and McLean, Florian Cramer focuses on the cultural history of writing in computation, tying our contemporary fascination with source code into an older web of historical attempts at integrating combinatorial practices from Hebraic texts to Leibniz's universal languages (Cramer, 2003). It is in this space between magic and logic that Cramer locates today's experiments in source code (i.e. source code poetry, esoteric languages and codeworks), reminiscent of Simondon's definition of a technical object's essence (Simondon, 1958). By relocating it between magic and reality, code is no longer just arbitrary symbols, or machine instructions but also ideal execution, a set of discreete forms which relate to the totality of the world. Once formal execution is considered a cosmogonical force, it becomes synonymous with performative execution, through which it ties back to cultural practices throughout the ages, within both religious and scientific contexts.

Cramer extracts five axes along which to apprehend code-based works: totality/fragmentation, rationalization/occultation, hardware/software, syntax/semantics, artificial/natural language. While all these axes overlap each other, it is the *syntax/semantics* axis which aligns most with this re-

search, givent that these thematical axes are all variations of one another. Towards the end of the book, his development of the concept of speculative programming is also particularly fruitful as the attempt to become a figure of thought and reflection in theory and artistic practice. Cramer states:

> formalisms [...] have a cultural semantics of their own, even on the most primitive and basic level. With a cultural semantics, there inevitably is an aesthetics, subjectivity and politics in computing. *(Cramer, 2003)*

This points to the relationship between the formal disposition of source code within program texts and the cultural communities composed of the writers and readers of these program texts. As we've seen, code does have social components of varying natures, insofar as it operates as an expressive medium between varying subjects.

Adrian MacKenzie approaches source code, as part of a broader inquiry on the nature of software, through this social lens in *Cutting Code: Software and Sociality* (Mackenzie, 2006). The author focuses on a relational ontology of software, rather than on a phenomenology: it is defined in how it acts upon, and how it is being acted upon by, external structures, from intellectual property frameworks to design philosophies in software architectures; it only provides an operational definition—software is what it does. His analysis of source code poetry focuses on famous Perl poems, Jodi's artworks and Alex McLean's `forkbomb.pl`, concerned with the executability of code as its dominant feature, dismissing Perl poetry as "*a relatively innocuous and inconsequential activity*" (Mackenzie, 2006). While software could indeed be a "patterning of social relations" (Mackenzie, 2006), these social relations also take place through linguistic combinations in program texts. This tending to the material realities of software embedded within social and cultural networks and traditions is echoed in David M. Berry's *The Philosophy of Software: Computation and Mediation*

in the Digital Age. His definition of materialities, however, focuses on the technical and social processes *around* code (e.g. build processes, specifications, test suites), rather than on the processes *within* code (i.e. texts, languages). While this former definition results in what he calls a *semiotic place* (Berry, 2011), a location in which those processes are organized meaningfully, such a semiotic sense of space could also apply, as we will see in chapter 2, to those intrinsic properties of source code.

Focusing specifically on the category of code poetry, Camille Paloque-Berges published, a couple of years later, *Poétique des codes sur le réseau informatique* (Paloque-Bergès, 2009). This work deploys both linguistic and cultural studies theorists such as Barthes and De Certeau in order to explain these playful acts of source code poetry, along with works of esoteric languages and net.art. While the first chapter focuses on digital literature as the result of executed code in order to develop a heuristic to approach source code, and the third and last chapter focusing on the means of distribution of these works, particularly on the development of net.art, 1337 5p43k and codeworks, the second chapter is the most relevant to our research focus. In it, Paloque-Berges provides an introduction of creative acts in source code on both a conceptual level (drawing from Hayles and Montfort) and on a technical, syntactical level. She looks at specific programming patterns and practices (`hello world`, quines), technical syntax (e.g. `$`, `@` as Perl tokens for expressing singular or plurals) and cultural paradigms (De Certeau's tactics and strategies), as she attempts to highlight the specificities of source code for aesthetic manifestation and invites further work to be done in this dual vein of close-reading and theoretical contextualization, beyond specific, heightened instances such as Perl poetry.

Honing in on a minimal excerpt, *10 PRNT CHR$(205.5+RND(1)) : GOTO 10;* (Montfort et al., 2014), is a collaborative work examining the cultural intertwinings of a single line of code, through hardware, language, syntax, outputs and themes. The whole endeavour is one rigorous close-reading of source code, in a deductive fashion, working from the words on the screen

and elaborating the context within which these words exist, in order to establish the cultural relevance of source code, as related to the syntax, hardware and cultural context in which these words exist. While the study itself, being a close-reading of only one work, and particularly a *one-liner*, itself a specific genre, is restricted in terms of broad aesthetic statements, it does show how it is possible to talk about code not as an abstract construct but as a concrete reality. Particularly interesting is the section dedicated to the history of the BASIC programming language, and how particular languages afford particular statements and actions better in a given historical context, a point often glossed over in other studies.

A current synthesis of these approaches, Mark C. Marino's *Critical Code Studies* (Marino, 2020) and the eponymous research field it belongs to focuses on close-reading of source code as a method for interpreting it as discourse. Particularly, it is organized around cases studies: each with source code, annotations and commentary. This structure furthers the empirical approach we've seen in Cox and McLean's code, starting from lines of source code in order in order to deduce cultural and social environments and intents through interpretation. This particular monograph, as is stated in the conclusion, offers a set of possible methodologies rather than conclusions in order to engage with code as its textual manifestations: the source code, viewed from different angles, can reveal more than its functional purpose. While Marino, with a background in the humanities, focuses mostly on the literary properties of code as a textual artifact, this thesis builds here on some of his methodologies, particularly reading how the form of the code complements its process and output, and searching the code for clever re-purposing or insight. However, while Marino mentions the aesthetics of code, he does not address the systematic composition of these aesthetics—focusing primarily on *what* the code means and only secondarily on *how* the code means it.

Taking a step back, Warren Sack's *The Software Arts* (Sack, 2019) historicizes software development as an epistemological practice, rather than as

a strictly economic trade. Connecting some of the main components of software (language, algorithm, grammar), he demonstrates how these are rooted in a liberal arts conception of knowledge and practice, particularly visible as a parallel to Diderot and D'Alembert's encyclopedic attempt at formalizing craft practices. By examining this other, humanistic, tradition in parallel with its dominantly acknowledged scientific counterpart, Sack shows the multiple facets that code and software can support. Starting from the concept of "translation" as an updated version of Manovich's "transcoding", Sack analyzes what is being translated by computing, such as analyses, rhetoric and logic, byt doesn't however address the nature of the process in which these concepts are translated—algorithms as (liberal) ideas, but not as texts. Nonetheless, this work offers a switch in perspective which will be helpful when we come to consider the relationship of source code with domains that are not primarily related to the sciences—i.e. the literary and the architectural, approached from a craft perspective.

This activity of programming as craft, already acknowledged by programmers themselves, is further explored in Erik Pineiro's doctoral thesis (Pineiro, 2003). In it, he examines the concrete, social and practical justifications for the existence of aesthetics within the software development community. Departing from specific, hand-picked examples such as those featured in Marino's study, his is more of an anthropological approach, revealing what role aesthetics play in a specific community of practicioners. Outlining references to ideas such as *cleanliness*, *simplicity*, *tightness*, *robustness*, amongst others, as aesthetic ideals that programmers aspire to, he does not however summon any specific aesthetic field (whether from literature, mathematics, craft or engineering), but rather frames it in terms of *intrsumental goodness*, with the aesthetics of code being an attempt to reach excellence in instrumental action. While he carefully lays out his argument by focusing on what (a certain group of) programmers actually say, instead of what they might be saying, there remains two limitations: it is not clear how source code as textual material can afford to reach such

aesthetic ideals, and whether or not these aesthetic ideals apply to other groups of writers of code, such as the code poets mentioned in some of the works above.

This literature review allows us to have a better grasp of how the relationship between source code and aesthetics has been studied, both from a scientific and engineering perspective, as well as from a humanities perspective.

In the former approach, aesthetics are acknowledged as a component of reading and writing code, and assessed through practical examples, quantitiative analysis and, to a lesser extent, qualitative interviews. The research focus is on the effectiveness of aesthetics in code, rather than on unearthing a systematic approach to making code beautiful, even though issues of cognitive friction and understanding, as well as ideals of cleanliness, readability, simplicity and elegance do arise. As such, they form a good starting ground of varied, empirical investigations. On a more metaphysical level, works in the field of philosophy of computer science point at the fact that the nature of computing and software are themselves evasive, straddling different lines while not aligning clearly with either science, engineering or arts—pointing out that software is indeed something different.

As for the humanities, the focus is predominantly on literary heuristics of a restricted corpus or on socio-cultural dynamics, and the details and examples of the actual code syntax and semantics are often omitted even though the aesthetic aspects of a literary or cultural nature are being explored in source code, as a new kind of writing. There is a potential for beauty and art in source code,as made obvious by code poetry, but such a potential is not assessed through the same empirical lense as the former part of our literature review and only secondarily investigating which of intrinsic features of code can support aesthetic judgments.

Still, some recent studies, such as those by Paloque-Bergès, Montfort et.

al, Cox and McLean and Marino, do engage directly with source code examples, and these constitute important landmarks for a code-specific aesthetic theory and methodology, whether it is as poetic language, speech-act, or critical commentary. Source code is taken as a unique literary device, but it remains unclear in exactly which aspects, besides its executability, it is different from both natural languages and low-level machine languages, and how this literary aspect relates to the effective, mathematical and craft-like nature of source code considered in the computer science and engineering literature.

1.2 The aesthetic specificities of source code

We can now turn to some of the gaps and questions left by this review, which can be grouped under three broad areas: dissonant aesthetic fields, lack of correspondance between empirical investigations and theoretical frameworks, and an absence of close-reading of program texts as expressive artifacts.

First, we can see that there are different aesthetic fields being summon when assessing aesthetics in source code. By aesthetic field, I mean the set of medium-specific symbol systems which operate coherently on a stylistic level, as well as on a thematic level. The main aesthetic fields addressed in the context of source code are those of literature, architecture as well as craft and mathematics. Each of these domains have specific ways to structure the aesthetic experience of objects within that field. For instance, literature can operate in terms of plot, consonance or poetic metaphor, while architecture will mobilize concepts of function, structure or texture. While we will reserve a more exhaustive description of each of these aesthetic fields in chapter 3, the first gap I would like to highlight here is how the multiple aesthetic fields are used to frame the aesthetics of source code, without this plurality being explicitely addressed. Depending on which

study one reads, one can see code as literature, as architecture, as mathematics or as craft, and there does not seem to be a consensus as to which of these maps closest to the essence of source code, with exhaustive studies often mentioning several, if not all of the above, fields (Rapaport, 2005).

Second, we can see a disconnect between empirical and theoretical work. The former, historically more present in computer science literature, but more recently finding its way into the humanities, aims at observing the realities of source code as a textual object, one which can be mined for semantic data analysis, or as a crafted object, one which is produced by programmers under specific conditions and replicated through examples and principles, rather than systems and theorems. Conversely, the theoretical approach to code, focusing on computation as a broad phenomenon encompassing engineering breakthroughs, social consequences and disruption of traditional understandings of textuality, rarely confronts such theoretical approaches with the concrete, physical manifestations of computation as source code[9], until recently. In consequence, there are theoretical frameworks that emerge to explain software (e.g. computation, procedurality, protocol), but no frameworks yet which tend to the aesthetics of source code. In the light of the history of aesthetic philosophy, literature studies and visual arts, defining such a precise framework seems like an elusive goal, but it is rather the constellation of conflicting and complementing frameworks which allow for a better grasp of their object of study. In the case of the particular object of this study, the establishment of such framework taking into account the specifically textual dimension of source code (as opposed to, say, McLean and Cox's attention to the speech dimension) is yet to be done. Following the software development and programming literature, such a framework could productively focus on the role and purpose that aesthetics play within source code, rather than on their autotelic nature as manifestations-for-themselves.

Finally, and related to the point above, we can identify a methodological

[9] With exceptions of the recent works cited above.

gap. Due to reasons such as access and skill, close-reading of source code from a humanities perspective has been mostly absent, until the recent emergence of fields of software studies and critical code studies. The result is that many studies engaging with source code as a literary object did not provide code snippets to illustrate the points being made. While not necessary *per se*, I argue that if one establishes an interpretative framework related to the nature and specificity of software, such a framework should be reflected in an examination of one of the main components of software—source code. The way that this gap has been productively addressed in recent years has primarily been done through an understanding of code as a part of broader socio-technical artifacts[10], inscribing it within the phenomenon of computation. This focus on the context in which source code exists therefore leaves some room for similar approaches with respect to its textual qualities. Despite N. Katherine Hayles's call for medium-specificity when engaging with code (Hayles, 2004), it seems that there hasn't yet been close-readings of a variety of program texts in order to assess them as specific aesthetic objects, in addition to their conceptual and socio-technical qualities.

Having established an overview of the state of the research on this topic, and having identified some gaps remaining in this scholarship, we can now clarify some of the problems resulting from those gaps with the following questions.

What does source code have to say about itself?

The relative absence of empirical examination of its source component when discussing code does not seem to be consistent with a conception of source code as a literary object. As methodologies for examining the meanings of source code have recently flourished, the techniques of *close-reading*, as focusing first and foremost on "the words on the page" (Richards, 1930) have been applied for extrinsic means: extract what the

[10] For instance, see the work done in the field of platform studies (Montfort et al., 2014)

lines of code have to say about the world, rather than what they have to say about themselves, about their particular organization as source files, as typographic objects or as symbol systems expressing concepts about the computational entities they describe. In this sense, it is still unclear how the possible combinations of control flow statements, function calls, function definitions, datatypes, variable declaration and variable naming, among other syntactic devices, enable program texts to be expressive. While close-reading will be a useful heuristic for investigating these problems, it will also be necessary to question the unicity of source code, and take into account how it varies across writers and readers and the social groups they constitute. This problem therefore has to be modulated with respect to the social environment in which it exists—it will then be possible to highlight to what extent the aesthetics of source code vary across these groups, and to what extent they don't.

How does source code relate to other aesthetic fields?

Multiple aesthetic fields are being mapped onto source code, allowing us to grasp such a novel object through more familiar lenses. However, the question remains of what it is about the nature of source code which can act as common ground for approaches as diverse as literature, mathematics and architecture, or whether these references only touch on distinct aspects of source code. When one talks about structure in source code, do they refer to structure in an architectural sense, or in a literary sense? When one refers to *syntactic sugar* in a programming language, does this have implications in a mathematical sense? This question will involve inquiries into the relationship of syntax and structure, of formality and tacitness, of metaphor and conceptual mapping, and in understanding of how adjectives such as *clean*, *clear* and *simple* might have similar meanings across those different fields. Offering answers to these questions might allow us to move from a multi-faceted understanding of source towards a more specific one, as the meeting point for all these fields, source code

might reveal deeper connections between each of those.

How do the aesthetics of source code relate to its functionality?

The final, and perhaps most important problem, concerns the status of aesthetics in source code not as an end, but as a means. A cursory investigation on the topic immediately reveals how aesthetics in source code can only be assessed only once the intended functionality of the software described has been verified. This stands in the way of a rather traditional opposition between beauty and functionality, and therefore begs further exploration. How do aesthetics support source code's functional purpose? And are aesthetics limited to supporting such purpose, or do they serve other purposes, beyond a strictly functional one? This paradox will relate to our first problem, regarding the meaning-making affordances of source code, and touch upon how the expressiveness of formal languages engage with different conceptions of use and function, therefore relating back to Goodman's concept of the languages of art, of which programming languages can be part of.

1.3 Methodology

To address such questions, we propose to proceed by looking at two kinds of texts: program texts and meta-texts. The core of our corpus will consist of the two categories, with additional texts and tools involved.

Our primary corpus is source code, taken as *program texts*. Due to the intricate relationship between source code and digital communication networks, vast amounts of source code are available online natively or have been digitized[11]. They range from a few lines to several thousands, date between 1969 and 2021, with a majority written by authors in Northern

[11]While software was circulating freely on ARPANET and early networks, the application of the intellectual property regime on software in 1974 significantly reduced the open-availability of source code.

America or Western Europe. On one side, code snippets are short, meaningful extracts usually accompanied by a natural language comment in order to illustrate a point. On the other, extensive code bases are large ensembles of source files, often written in more than one language, and embedded in a build system[12]. Both can be written in a variety of programming languages, as long as these languages are composed in alphanumeric characters.

This lack of limitations on size, date or languages stems from our empirical approach. Since we intend to assess code conditionally, that is, based primarily on its own, intrinsic textual qualities, it would not follow that we should restrict to any specific genre of program text. As we carry on this study, distinctions will nonetheless arise in our corpus that align with some of the varieties amongst source—for instance, the aesthetic properties of a program text composed of one line of code might be different from those exhbited by a program text made up of thousands of lines code.

We also intend to use source code in both a deductive and an inductive manner. Through our close-reading of program texts, we will highlight some aesthetic features related to its textuality, taking existing source code as concrete proof of their existence. Conversely, we will also write our own source code snippets in order to illustrate the aesthetic features discussed in natural language. This use of source code snippets is widely spread among communities of programmers in order to qualify and strengthen their points in online discussions, and we intend to follow this weaving in of machine language and natural language in order to strengthen our argumentation. This approach will therefore oscillate between theory and practice, the concrete and the abstract, as it both extracts concepts from readings of source code and illustrates concepts by writing source code.

The case of programming languages is a particular one: they do not ex-

[12] A build system is a fairly complex series of code transformations intended to generate executable code.

clusively constitute program texts (unless they are considered strictly in their implementation details as lexers, interpreters and compilers, themselves described in program texts), but are a necessary, if artificial, condition for the existence of source code. They therefore have to be taken into account when assessing the aesthetic features of program text, as integral part of the affordances of source code. Rather than focusing on their context-free grammars or abstract notations, or on their implementation details, we will focus on the syntax and semantics that they allow the programmer to use. Still, programming languages are hybrid artefacts, and their intrinsic qualities are only assessed insofar as they relate to the aeshetic manifestations of source code written in those languages.

Meta-texts on source code make up our secondary corpus. Meta-texts are written by programmers, provide additional information, context and explanation for a given extract of source code, and is a significant part of the software ecosystem. Even though they are written in natural langauge, this ability to write comments has been a core feature of any programming language very early on in the history of computing, linking any program text with a potential commentary, whether directly among the source code lines (*inline commentary*) or in a separate block (*external commentary*)[13]. Examples of external commentaries include user manuals, textbooks, documentation, journal articles, forums posts, blog posts or emails. The inclusion in our corpus of those meta-texts is due to two reasons: the practical reason of the high epistemological barrier to entry when it comes to assessing source code in linguistic or hardware environments which one isn't familiar with, and the theoretical reason of including the (aesthetic) judgment of programmers as it supports our conditional, rather than constitutive, approach.

While we intend to look at source through close-reading, favoring the role and essence of each line as a meaningful, structural element, rather

[13] Such a distinction isn't a strict binary, and systems of inscription exist which couple code a commentary more tightly, such as WEB or Juptyer Notebook.

than that of the whole, our interpretation of meta-texts will take place via discourse analysis. Building on Dijk and Kintsch's work on discourse comprehension (Dijk & Kintsch, 1983), we intend to approach these texts at a higher level, in terms of the lexical field they use, as a marker of the aesthetic field they refer to, as well as at a lower level, noting which specific syntactic aspects of the code they refer to. This focus on both the micro-level (e.g. local coherence and proposition analysis) and on the macro-level (e.g. socio-cultural context, intended aim and lexical field usage) will allow us to link specific instances of written code with the broader semantic field that they exist in. This connection between micro- and macro- relies on the hypothesis that there is something fundamentally similar between a source code construct, its meaning and use at the micro-level, and the aesthetic field to which it is attached at a macro-level, a hypothesis we will address further when investigating the role of metaphor in source code.

In the end, this process will allow us to construct a framework from empirical observations. The last part of our methodology, after having completed this analysis of program-texts and their commentaries, is to cross-reference it with texts dealing with the manifestation of aesthetics in those peripheral fields. Literary theory, centered around the works of I.A. Richards, Roland Barthes and Paul Ricoeur can shed light on the attention to form, on the interplay of syntax and semantics, of open and closed texts, and suggest productive avenues through the context of metaphor. Architecture theory will be involved through the two main approaches mentioned by software developers: functionalism as illustrated by the credo *form follows function* and works by Vitruvius, Louis Sullivan and the Bauhaus on one side, and pattern languages as initiated by the work of Christopher Alexander on the other. The aesthetic nature of the two remaining fields, mathematics and craft, have a thinner tradition of formalized aesthetics than literature and architecture, but we nonetheless include essays and monographs from practicioners in the field addressing those issues. This additional set of texts will allow us to operate compara-

tively when it comes to expliciting source code's aesthetics.

This study therefore aims at weaving in empirical observations, discourse analysis and external framing, in order to propose systematic approaches to source code's textuality. However, these will not unfold in a strictly linear sequence; rather, there will be a constant movement between practice and theory and between code-specific aesthetic references and broader ones: this interdisciplinary approach intends to reflect the multifaceted nature of software.

1.4 Roadmap

Our first step in this study is an empirical assessment of how programmers consider aesthetics with their practice or reading and writing it, first from a conceptual standpoint. After acknowledging and underlining the diversity of those practices, from software developers and scientists to artists and hackers, we will identify which concepts and references are being used the most when referring to beautiful code—concepts such as clarity, simplicity, cleanliness, and others. These concepts will then allow us to touch upon the field that are being referred to when considering the practice of programming: literature, architecture and mathematics as domains in themselves, and craft as a particular approach to these domains. Finally, we will how how the overlap of these concepts can be found in the process of *understanding*—communicating abstract ideas through concrete manifestations.

After establishing the role of aesthetics as a means for understanding source code, we will proceed to analyze further such a relationship between understanding, source code and aesthetics. We will see that one of the main features of source code is the elusiveness of its meaning, whether effective or intended. Beautiful code is often code that can be understood clearly, which raises the following question: how can a completely explicit

and formal language allow ambiguity? The answer to this question will involve an analysis of the two audiences of source code: humans and machines.

Taking a step back towards textuality, we will then assess how the different fields that are being referred to when talking about source code have touched upon these issues of understanding, from rhetoric to literature, through architecture and mathematics. Thinking in terms of surface-structure and deep-structure, we will establish a first connection between program texts and literary text through their reliance on linguistic metaphors. Since metaphors aren't exclusively literary devices, looking at them from a cognitive perspective will also raise issues of modes of knowledge, between explicit, implicit and tacit. The understanding of beauty in architecture, based on the two traditions mentioned above, will provide an additional perspective by providing concepts of structure, function and usability. These will echo a final inquiry into mathematical beauty, drawing a direct link between idea and implementation, theorem and proof, and providing a deeper understanding of the concept of *elegance*.

With a firmer grasp on the stakes of source code as an understandable text, we can now turn to its effective manifestations, by close-reading program texts. Working through *structure*, *syntax* and *vocabulary*, we will be able to formalize a set of textual typologies involved in producing an aesthetic experience through source code. Particularly, we will highlight where those tokens differ across communities of practice, and where they overlap, keeping in mind the conditionality of those aesthetic judgments, and attempt to trace connections between specific textual configurations of source code with the ideals summoned by the programmers. After this deductive consideration, we will move on to apply these typologies to several larger program texts—ranging from the LaTeX codebase, the Carnivore software artwork to several code poems. These will highlight a remaining component in the concrete manifestation of source code aesthet-

ics: the place of programming languages.

At this point, we will have established aesthetics in source code as a way to address the inherent tensions of a program text's dual audience, computers and humans. Being understandable by both humans and machines is indeed the feat of programming languages, the symbol systems on which beautiful texts depend on. As we've elicited the intricacies of aesthetic manifestations in human to machine communication, we then investigate machine to machine communication. Deconstructing programming languages as formal grammars will show that there are very different conceptions of semantics and meanings expected from the computer than those expected from a human, even though a machine's perspective on beautiful code could still be based around concepts of effectiveness, simplicity and performance. *Contra* those, human use of programming languages reaches into the extreme of *esolangs*—an investigation into those will reveal that language is effectively considered as a material, one whose base elements can be recombined into unexpected puzzling structures.

Recognizing programming languages as the bridge between the two domains of programming—the human of the machine—will allow us to clarify how the different aesthetic fields (literature, architecture, mathematics) relate to programming. We will show how programming languages provide a gradual interface between different modes of being of source code: source code as text, source code as structure and source code as theory. The need for aesthetics arises from the tradeoffs that need to be made when these different modes of being overlap (Simondon, 1958).

We will then turn back to our research questions to suggest some possible answers. The reorganization of the source aesthetic fields inot a linear succession of the interpration or compilation process from high-level to low-level hints at a specifically spatial nature of program texts. Indeed, the specific aesthetics of source code are those of a constant doubling between the specificities of the human (such as natural handling of ambiguity, and intuitive understanding of the problem domain) and of the machine (such

as speed of execution, and reliance on explicit formal grammars, which can also be seen as the tension between surface structure, one that is textual and readable, and deep structure, one that is made up of dynamic processes representing complex concepts, and yet devoid of any fluidity or ambiguity. It is this dynamism, both in terms of *where* and *when* code could be executed, which suggest the use of aesthetics in order to grasp more intuitively the topology and chronology, the state and behaviour of a program text.

Finally, we will relate the approaches of Goodman of art as cognitively effective symbol systems, and of Simondon's consideration of aesthetic thought as a link between technical thought and religious thought. Starting from a practical perspective on aesthetics taking from the field of craft—the thing well done—, aesthetics also highlight functionality on a cognitive level—the thing well thought. Beauty in source code seems to be dominantly what is useful and thoughtful, even when they are reflected in the distorting mirrors of hacks and esoteric languages, broadening our possible understandings of what aesthetics can do, and what functionality can be.

1.5 Implications and readership

This thesis fits within the field of software studies, and aims at clarifying what do we mean when we refer to code *code as...*. Code as literature, architecture or mathematics, code as philosophy or as craft, are metaphors which can be examined productively by looking at the texts themselves, an approach that has only been deployed in relatively recent work.

This relationship between practice, function and beauty is the broad, underlying question of this study. In the vein of the cognitive approach to art and aesthetics, this study is an attempt to show how aestethics play a communicative role, and how concrete manifestations can, through a

metaphorical process, hint at broader ideas. In this sense, this study is not just about the relation of aesthetics and function, but also about the function of aesthetics. While this idea of aesthetics as a way of communicating ideas could be equally applied across artistic and non-artistic domains, another aim of this thesis is to highlight the relativity a aesthetic standards: using a similar medium, practices, uses and purposes determine as much, if not more, of the artistic worth of a given program text.

By examining the result of the practice of programmers at a close-level, this study hopes to contribute to a clarification of what exactly is programming, along with the consequences of the embedding of software in our social, economic and political practices. In order to address the question of whether algorithms are political in themselves, or if their use is political, it is important to define clearly what it is that we are talking about when discussing algorithms. A clarification of source code on a concrete level attempts to help clarify what this essential component of algorithms, and opens up potential for further work in terms of thinking no longer of the aesthetics of source code, but of its poetics, in the way source code, as a language of art, is also a way of worldmaking.

To this end, this thesis is aimed at a variety of readers and audience. From the humanities perspective, digital humanists and literary theorists interested in the concrete manifestations of source code as specific meaning-making techniques will be able to find the first steps of such an approach being laid out, and contrast these specific technique with the broader poetics of code studied by other scholars, or with the aesthetics of natural language texts.

Programmers and computer scientists will find an attempt at formalizing something they might have known implicitly ever since they started practicing writing and reading code, and the approach of languages as poetics and structure might help them think through these aspects in order to write perhaps more aesthetically pleasing, and thus perhaps better, code. Conversely, anyone engaged seriously in a craft activity could find here a

rigorous study of what goes on into a specific craft, asking how their own practice engages with tools and modes of knowledge, and with a more explicit conception of beauty.

Finally, such a specific conception of beauty, then, will also be of interest to artists and art theorists. By investing aesthetics without a direct relation to the artwork, but rather within a functional purpose, this study suggests that one can think through beauty and artworks not as ends, but as ways to accomplish things that formal systems of explanation might not be able to achieve. An aesthetics of source code would therefore aim at highlighting the purpose of instrumental beauty within a textual environment.

Chapter 2

Aesthetic ideals in programming practices

The first step in our study of aesthetic standards in source code will identify the aesthetic ideals ascribed by programmers to the source code they write and read; that is, the syntactic qualifiers and semantic fields that they refer to when discussing program texts. To that end, we first start by clarifying whom we refer to by the term *programmers*, revealing a multiplicity of practices and purposes, from *ad hoc*, one-line solutions, to printed code and massively-distributed codebases.

We then turn to the kinds of beauty that these programmers aspire to. After expliciting our methodology of discourse analysis, we engage in a review of the various kinds of publications and writings that programmers write, read and refer to when it comes to qualifying their practice. From this will result a cluster of adjectives which we see being used in an aesthetic manner. These will provide an empirical basis when examining, in subsequent chapters, the formal arrangements of program texts.

From these, we can then move to a description of which aesthetic fields are being referenced by programmers on a broader level, and consider how

multiple kinds of beauties, from literary, to architectural and scientific conceptions of beauty can overlap and be referred to by the same medium.

Finally, we conclude this chapter by elaborating on one of the points of overlap in these different references: the importance of function, craft and knowledge in the disposition and representation of code. We will show how this particular way of working plays a central role in an aesthetic approach to source code and results from the specificity of code as a cognitive material, a specificity we will focus on in the next chapter.

2.1 The practice of programmers

The history of software development is that of a specific, reserved practice which was born in the aftermath of the second world war, which trickled down to broader and broader audiences at the eve of the twenty-first century. Through this development, various paradigms, platforms and applications have been involved in producing software, resulting in different epistemic communities and communities of practiceCohendet et al. (2001), in turn producing different types of source code. Each of these focus on the description of specific instructions to the computer, but do so with particular characteristics. To this end, we take a socio-historical stance on the field of programming, highlighting how diverse practices emerge at different moments in time, how they are connected to contemporary technical and economic organizations, and for specific purposes.

Even though such types of reading and writing source code often overlap with one another, this section will highlight a diversity of ways in which code is written, notably in terms of origin—how did such a practice emerge?—, references—what do they consider good?—, purposes—what do they write for?—and examples—how does their code look like?. First, we take a look at the software industry, to identify professional *software developers*, the large code bases they work on and the specific organizational practices within which they write it. They are responsible for the majority of source code written today, and do so in a professional and productive context, where maintainability, testability and reliability are the main concerns. Then, we turn to a parallel practice, one that is often exhibited by software developers, as they also take on the stance of *hackers*. Disambiguating the term reveals a set of practices where curiosity, cleverness, and idiosyncracy are central, finding unexpected solutions to complex problems, sometimes within artificial constraints. Finally, we look at *scientists* and *poets*. On one end, *scientists* embody a rather academic approach, focusing on abstract concepts such as simplicity, minimalism and

elegance; they are often focused on theoretical issues, such as mathematical models, as well as programming language design, but are also involved in the implementation of algorithms. On the other end, *poets* read and write code first and foremost for its textual and semantic qualities, publishing code poems online and in print, and engaging deeply with the range of metaphors allowed by this dynamic linguistic medium.

While this overview encompasses most of the programming practices, we leave aside some approaches to code, mainly because they do not directly engage with the representation of source code as a textual matter. More and more, end-user applications provide the possibility to program in rudimentary ways, something referred to as the "low-code" approach (Team, 2021), and thus contributing to the blurring of boundaries between programmers and non-programmers[1].

2.1.1 Software developers

From local hardware to distributed software

As Niklaus Wirth puts it, *the history of software is the history of growth in complexity* (Wirth, 2008), while paradoxically, a lowering of the barrier to entry. As computers' technical abilities in memory managment and processing power increased year on year since the 1950s, the nature of writing instructions shifted as well.

In his history of the software industry, Martin Campbell-Kelly traces the development of a discipline through an economic and a technological lens, and he identifies three consecutive waves in the production of software (Campbell-Kelly, 2003). During the first period, as soon as the 1950s,

[1]For instance, Microsoft's Visual Basic for Applications, Ableton's Max For Live, MIT's Scrath or McNeel's Grasshopper are all programming frameworks which are not covered within the scope of this study. In the case of VBA and similar office-based high-level programming, it is because such a practice is a highly personal and *ad hoc* one, and therefore is less available for study.

and continuing throughout the 1960s, software developers were contractors hired to engage directly with a specific computing machine. These computing, mainframes, were large, expensive, and rigid machines, requiring hardware-specific knowledge of the corresponding Assembler instruction set, since they didn't yet feature an operating system which could facilitate some of the more basic memory allocation and input/output functions, and thus enable interoperable program-writing[2]. Two distinct groups of people were involved in the operationalization of such machine: electrical engineers, tasked with designing hardware, and programmers, tasked with implementing the software. While the former historically received the most attention (Ross, 1986), the latter was mostly composed of women and, as such, not considered essential in the process (Light, 1999). At this point, then, programming remains closely tied to hardware.

The second period in software development starts in the 1960s, as hardware switched from vacuum tubes to transistors and from magnetic core memory to semiconductor memory, making them faster and more capable to handle complex operations. On the software side, the development of several programming languages, such as FORTRAN, LISP and COBOL, started to address the double issue of portability—having a program run unmodified on different machines with different instruction sets—and expressivity—allowing programmers to use high-level, English-like syntax, rather than assembler instruction codes. By then, programmers are no longer theoretically tied to a specific machine, and therefore acquire a certain autonomy, a recognition which culminates in the naming of the field of *software engineering* in 1968 at a NATO conference (Randell, 1996).

The third and final phase that Campbell-Kelly identifies is that of mass-market production: following the advent of the UNIX family of operating systems, the distribution of the C programming language, the wide availability of C compilers, and the appearance of personal computers such as

[2]One of the first operating systems, MIT's Tape Director, would be only developped in 1956 (Ross, 1986)

the Commodore 64, Altair and Apple II, software could be effectively entirely decoupled from hardware[3]. And yet, despite this growth in popularity, software immediately enters a crisis: software development projects run over time and budget, prove to be unreliable in production and unmaintainable in the long-run. The creation of software is no longer a corollary to the design of hardware, and as an independent field would as such become the main focus of computing as a whole (Ceruzzi, 2003). It is at this time that discussions around best practices in writing source code started to emerge, once the activity of the programmer was no longer restricted to *tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment* (Dijkstra, 2007).

This need for a more formal approach to the actual process of programming found one of its most important manifestations in Edsger Djikstra's *Notes on Structured Programming* (Dijkstra, 1972). In it, he argues for moving away from programming as a craft, and towards programming as an organized discipline, with its methodologies and systematization of program construction. Despite its laconic section titles[4], Djikstra's 1972 report nonetheless contributed to establish a more rigorous typology of the constructs required for reliable, provable programs—based on fundamental heuristics such as sequencing, selection, iteration and recursion—, and aimed at the formalization of the practice. Along with other subsequent developments (such as Hoare's contribution on proper data structuring (Hoare, 1972), or the rise of object-oriented programming with Smalltalk) programming would solidify its foundations as a profession:

> *We knew how the nonprofessional programmer could write in an afternoon a three-page program that was supposed to satisfy his needs, but how would the professional programmer design a thirty-page program in such a way that he could really jus-*

[3]For a more detailed account of the personal computer revolution, see: Cerruzzi, P., A History of Modern Computing (Ceruzzi, 2003)

[4]See, for instance, Chapter 1: *"On our inability to do much"*

> *tify his design? What intellectual discipline would be needed? What properties could such a professional programmer demand with justification from his programming language, from the formal tool he had to work with? (Dijkstra, 1972)*

As a result of such interrogations comes an industry-wide search for solutions to the intractable problem of programming: that it is *a technique to manage information which in turn produces information*. To address such a conundrum, a variety of tools, formal methods and management processes enter the market; they aim at acting as a *silver bullet* (Brooks & Jr, 1975), addressing the cascade of potential risks[5] which emerge from large software applications. However, this growth in complexity is also accompanied by a diversification of software applications: as computers become more widely available, and as higher-level programming languages provide more flexibility in their expressive abilities, software engineering engages with a variety of domains, each of which might need a specific solution, rather than a generic process. Confronted with this diversity of applications, business literature on software practices flourishes, being based on the assumption that the complexity of software should be tackled at its bottleneck: the reading and writing of source code.

The most recent step in the history of software developers is the popularization of the Internet and of the World Wide Web. Even though the former had existed under as ArpaNet since 1969, the network was only standardized in 1982 and access to it was provided commercially in 1989. Built on top of the Internet, the latter popularized global information exchange, including technical resources to read and write code. Software could now be written by remote individual written on *cloud computing* platforms, shared through public repositories and deployed via containers with a lower barrier to entry than at the time of source code printed in magazines, of overnight batch processing and of non-time-sharing sys-

[5] See `https://catless.ncl.ac.uk/Risks/` for such risks

tems.

Software developers have written some of the largest codebases to this date, since this type of activity represents the most significant fraction of programmers. Due to its close ties to commercial distributors, however, source code written in this context often falls under the umbrella of proprietary software, thus made unvailable to the public. Some examples that we include in our corpus are either professional codebases that have been leaked[6], open-source projects that have come out of business environments, such as Soundcloud's Prometheus, Kirby's CMS System, Facebook's React, or large-scale open-source projects which nonetheless adhere to structured programming guidelines, such as Donald Knuth's TeX typesetting system or the Linux Foundation's Linux kernel.

Features of the field

The features of these codebases hint at the qualities that software developers have come to ascribe to their object of practice. First, the program texts they write are large, much larger than any other codebase included in this study. They often feature multiple programming languages and are highly structured and standardized: each file follows a pre-established convention in programming style, which favors an authoring by multiple programmers without any obvious trace to a single individual authorship. These program texts stand the closest to a programming equivalent of engineering, with its formalisms, standards and usability. From this perspective, the IEEE's Software Engineering Body of Knoweldge (SWEBOK) provides a good starting point to survey the specificities of software developers as source code writers and readers (Bourque & Fairley, 2014); the main features of which include the definition of requirements, design, construction, testing and maintenance.

[6] Such as the Microsoft Windows XP source code (Warren, 2020).

Software requirements are the acknowledgement of the importance of the *problem domain*, the domain to which the software takes its inputs from, and to which it applies its outputs. For instance, software written for a calculator has arithmetic as its problem domain; software written for a learning management system has students, faculty, education and courses as its problem domain; software written a banking institution has financial transactions, savings accounts, fraud prevention and credit lines as its problem domain. Requirements in software development aim at formalizing as best as possible the elements that must be used by the software in order to perform a successful computation, and an adequate formalization is a fundamental requirement for a successful software application.

Subsequent to the identification and codification of requirements, software design relates to the overall organization of the software components, considered not in their textual implementation, but in their conceptual agency. Usually represented through diagrams or modelling languages, it is concerned with *understanding how a system should be organized and designing the overall structure of that system* (Sommerville, 2010). Of particular interest is the relationship that is established between software development and architecture. Considered a creative process rather than a strictly rational one, due to the important role of the contexts in which the software will exist (including the problem domain) (Sommerville, 2010), software architecture operates both from a top-down perspective, laying down an abstract blueprint for the implementation of a system, as well as form a bottom-up one, representing how the different components of an existing system interact. This apparent contradiction, and the role of architecture in the creative aspects of software development, will be further explored in section 3.3 below.

Software construction relates to the actual writing of software, and how to do so in the most reliable way possible. The SWEBOK emphasizes first and foremost the need to minimize complexity[7], in anticipation of likely

[7]Following Hoare's assessment in his Turing Award Lecture that *"there are two ways of*

changes and possible reuse by other software systems. Here, the emphasis on engineering is particularly salient: while most would refer to the creation of software as *writing* software, the IEEE document refers to it as *constructing* software[8]. Coding is only assessed as a practical consideration, one which should not take up the most attention, if the requirements, design and testing steps are satisfyingly implemented.

In parallel, a whole field of business litterature (Martin, 2008; Hendrickson & McBreen, 2002; Fowler et al., 1999; McConnell, 2004) has focused specifically on the process of writing code, starting from the assumption that:

> *We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such details that a machine can execute them is* programming. *(Martin, 2008)*

As we see in these two perspectives on the role that code should play, the tension identified by Djikstra some thirty years before between craft and discipline is still alive and well at the beginning of the twenty-first century, even though the attention paid to code still relates to the need for reliability and maintainability in a maturing industry.

Software maintenance, finally, relates not to the planning or writing of software, but to its reading. Software is notoriously filled with bugs[9] and can, at the same time, be easily fixed while already being in a production environment through software update releases. This means that the lifecycle of a software doesn't stop when then first version is written, but rather

constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

[8]The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging. (Bourque & Fairley, 2014).

[9]McConnell estimates that the industry average is about 15 - 50 errors per 1000 lines of delivered code. (McConnell, 2004).

when it does not run anywhere anymore. The nature of software allows for it to be edited across time and space, by other programmers which might not have access to the original group of implementers: consequently, software should be first and foremost understandable—SWEBOK lists the first feature of coding as being *techniques for creating understandable source code* (Bourque & Fairley, 2014). This requirement ties back to one of the main problems of software, which is its notorious cognitive complexity, one that remains at any stage of its development.

What does this look like in practice, then? Ideals of clarity, reusability and reliability—and their opposites—can be found in some of the available code bases of professional software. The paradox to be noted here is that, even though software developers write the most code, it is the least accessible online and, as such, the following excerpts are covering the range of commercial software (Microsoft Windows), licensed and publicly available software (Kirby).

The first excerpts come from the source code for Microsoft Windows 2000, which was made public in 2004. The program text contains 28,655 files, the largest of our corpus, by multiple orders of magnitude, with 13,468,327 combined lines and including more than 10 different file extensions. Taking a closer look at some of these files allow us to identify some of the specific features of code written by software developers, and how they specifically relate to architectural choices, collaborative writing and verbosity.

First, the most striking visual feature of the code is its sheer size. Representing such a versatile and low-level system such as an operating system manifest themselves in files that are often above 2000 lines of code. In order to allow abstraction techniques at a higher-level for the end-developer, the operating system needs to do a significant amount of "grunt" work, relating directly to the concrete reality of the hardware platform which needs to be operated. For instance, the initialization of strings of text for the

namespaces (a technique directly related to the compartmentalization) is necessary, repetitive work which can be represented using a rhythmic visual pattern, such as in the `cmdatini.c` source file, reproduced partially in 1.

The repetition of the `RtlInitUnicodeString` call in the first part of this listing stands at odds with today's industry-standard practices of not repeating oneself; a standard only adhered to in the second part of the code, the two `for()` statements. While this practice would only be formalized in Andy Hunt's *The Pragmatic Programmer* in 1999 (Hunt & Thomas, 1999), the longevity of the windows 2000 operating system and its update cycle would nonetheless have affected how this code is written. The reason why such a repetition applies is the requirement of registering each string with the kernel. Dealing with a different problem domain (kernel instructions) leads to a different kind of expected aesthetics[10].

Verbosity, the act of explicitly writing out statements which could be functionally equivalent in a compacted form, is a significant feature of the Windows 2000 codebase, and also relies on a particular semantic environment: that of using the C programming language. As mentioned above, the development of C and UNIX in the 1970s have led to wide adoption of the former, and to some extent of the later (even though Windows is a notable exception since it is an operation system not based on the UNIX tradition). What we see in this listing is the consequence of this development: using a verbose language results in a verbose program text, something we will see in the following section on hacker's code and will explore much further in chapter 5.

Another significant aesthetic feature of the Windows 2000 program text is its use of comments, and specifically how those comments representing the multiple, layered authorship. This particular source code is one that is written across individuals and across time, each with presumably its own writing style. Yet, writing source code within a formal organization often

[10] Effectively, references to `RtlInitUnicodeString()` happen 1580 times across 336 files

```
  {
    ULONG i;

    RtlInitUnicodeString( &CmRegistryRootName,
                          CmpRegistryRootString );

    RtlInitUnicodeString( &CmRegistryMachineName,
                          CmpRegistryMachineString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareName,
                          CmpRegistryMachineHardwareString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDescriptionName,
                          CmpRegistryMachineHardwareDescriptionString
↪   );

    RtlInitUnicodeString(
↪   &CmRegistryMachineHardwareDescriptionSystemName,

↪   CmpRegistryMachineHardwareDescriptionSystemString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDeviceMapName,
                          CmpRegistryMachineHardwareDeviceMapString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareResourceMapName,
                          CmpRegistryMachineHardwareResourceMapString
↪   );

    RtlInitUnicodeString( &CmRegistryMachineHardwareOwnerMapName,
                          CmpRegistryMachineHardwareOwnerMapString );

    RtlInitUnicodeString( &CmRegistryMachineSystemName,
                          CmpRegistryMachineSystemString );

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSet,

↪   CmpRegistryMachineSystemCurrentControlSetString);

    RtlInitUnicodeString( &CmRegistryUserName,
                          CmpRegistryUserString );

    RtlInitUnicodeString( &CmRegistrySystemCloneName,
                          CmpRegistrySystemCloneString );

    RtlInitUnicodeString( &CmpSystemFileName,
                          CmpRegistrySystemFileNameString );
```

implies the adoption of coding styles, with the intent that *all code in any code-base should look like a single person typed it, no matter how many people contributed* (Waldron, 2020). For instance, the excerpt in 2 from `jdhuff.c` is a example of such overlapping of styles:

Here, we see different writings of comments(using `//` and `/* */`) as well as different kinds of capitalizations. Those comments are ignored at compile time: that is, they are not meaningful to the machine, and are only expected to be read by other programmers, and in this case primarily by programmers belonging to one's organization. This hints at the various origins of the authors, or at the very least at the different moments, and possible mental states of the potential single-author: irregularity in comment writing can connect to irregularities in semantic content of the comments. This irregularity becomes suspicious, and leads to ascribing a different epistemological value to them. If comments aren't procedurally guaranteed to be reflected in the execution, and outcome, of the program, then one tend to rely on the fact that "the only document that describes your code completely and correctly is the code itself" (Goodliffe, 2007). This excerpt highlights the constant tension between source code as the canonical origin of knowledge of what the program does and how it does it, while comments reflect the idiosyncratic dimension of all natural-language expressions of human programmers.

And yet, this chronological and interpersonal spread of the program text, combined with organizational practices, require the use of comments in order to maintain aesthetic and cognitive coherence in the program, if only by the use of comment headers, which locate a specific file within the greater architectural organization of the program text (see 3).

This highlights both the multiple authorship (here, we have one original author and one revisor) as well as the evolution in time of the file: comments are the only manifestation of this layering of revisions which ultimately results in the "final" software[11].

[11]The term "final" is in quotes, since the Windows 2000 source contains the mention

```
  no_more_data:
    // There should be enough bits still left in the data segment;
    // if so, just break out of the outer while loop.
    if (bits_left >= nbits)
      break;
      /* Uh-oh.  Report corrupted data to user and stuff zeroes into
      * the data stream, so that we can produce some kind of image.
      * Note that this code will be repeated for each byte demanded
      * for the rest of the segment.  We use a nonvolatile flag to ensure
      * that only one warning message appears.
      */
      if (! *(state->printed_eod_ptr))
      {
        WARNMS(state->cinfo, JWRN_HIT_MARKER);
        *(state->printed_eod_ptr) = TRUE;
      }
      c = 0;^^I^^I^^I// insert a zero byte into bit buffer
      }
    }

    /* OK, load c into get_buffer */
    get_buffer = (get_buffer << 8) | c;
    bits_left += 8;
  }

  /* Unload the local registers */
  state->next_input_byte = next_input_byte;
  state->bytes_in_buffer = bytes_in_buffer;
  state->get_buffer = get_buffer;
  state->bits_left = bits_left;

  return TRUE;
}
```

Listing 2: Programming styles overlapping in the source code of Microsoft 2000.

```
  /*++

Copyright (c) 1996 Microsoft Corporation

Module Name:

    enum.c

Abstract:

    This module contains routines to perform device enumeration

Author:

    Shie-Lin Tzong (shielint) Sept. 5, 1996.

Revision History:

    James Cavalaris (t-jcaval) July 29, 1997.
    Added IopProcessCriticalDeviceRoutine.

--*/
```

Listing 3: pnpenum.c

A complementary example is the Kirby CMS[12]. With development starting in 2011 and a first release in 2012, it developed a steady userbase, correlated in Google Trends analytics[13], consistent forum posts[14] and commit history on the main repository[15]. Kirby is an open-source, developer-first (meaning that it affords direct engagement of other developers with its architecture through modification, extension or partial replacement), single-purpose project. As such, it stands at the other end of the commercial, efficient software spectrum than Microsoft Windows 2000.

The Kirby source code is entirely available online, and the following snippets hint at another set of formal values—conciseness, expliciteness and delimitation. Conciseness can be seen in the lengths of the various components of the code base. For instance, the core of Kirby consists in 248 files, with the longest being `src/Database/Query.php` at 1065 lines, and the shortest being `src/Http/Exceptions/NextRouteException.php` at 16 lines, for an average of 250 lines per file (compared to the leading project in the field, Wordpress.org, which has respectively 3466 files, with the longest file comprising 9353 lines of code (`customize-controls.js`), and the shortest 1line (such as `script-loader-packages.php`)[16].

If we look at a typical function declaration within Kirby, we found one such as the `distinct()` setter for Kirby's database, reproduced in 4.

Out of these 11 lines, the actual functionality of the function is focused on one line, `$this->distinct = $distinct;`. Around it are machine-readable comment snippets, and a function wrapper around the simple variable setting. The textual overhead then comes from the wrapping itself: the actual semantic task of deciding whether a query should be able to include

BUGBUG 7436 times across 2263 files, a testament to the constant state of unfinishedness that software tends to remain in.

[12] Allgeier, Bastian et. al., https://github.com/getkirby/kirby, 2011, consulted in 2022

[13] https://trends.google.com/trends/explore?date=all&q=kirby%20cms

[14] https://forum.getkirby.com

[15] https://github.com/getkirby/kirby

[16] The project was cloned on 06.05.2022 from the official repository at `git@github.com:WordPress/WordPress.git`

```
 /**
 * Enables distinct select clauses.
 *
 * @param bool $distinct
 * @return \Kirby\Database\Query
 */
public function distinct(bool $distinct = true)
{
    $this->distinct = $distinct;
    return $this;
}
```

Listing 4: Query.php

distinct select clauses (as opposed to only allowing join clauses), is now decoupled from its actual implementation (one could describe to the computer such an ability to generate distinct clauses by assigning it a boolean value, or an integer value, or passing it as an argument for each query, etc.). The quality of this writing, at first verbose, actually lies in its conciseness in relation to the possibilities for extension that such a form of writing allows: the `distinct()` function could, under other circumstances, be implemented differently, and still behave similarly from the perspective of the rest of the program. Additionally, this wrapping enables the setting of default values (here, `true`), a minimal way to catch bugs by always providing a fallback case.

Kirby's source code is also interestingly explicit in comments, and succint in code. Let us take, for instance, from the `Http\\Route` class (see 5).

The 9 lines above the function declaration are machine-readable documentation. It can be parsed by a programmatic system and used as input to generate more classical, human-readable documentation[17]. This is notice-

[17] See, for instance, JavaDocs, or ReadTheDocs

```
/**
* Tries to match the path with the regular expression and
* extracts all arguments for the Route action
*
* @param string $pattern
* @param string $path
* @return array|false
*/
public function parse(string $pattern, string $path)
{
    // check for direct matches
    if ($pattern === $path) {
        return $this->arguments = [];
    }

    // We only need to check routes with regular expression since all others
    // would have been able to be matched by the search for literal matches
    // we just did before we started searching.
    if (strpos($pattern, '(') === false) {
        return false;
    }

    // If we have a match we'll return all results
    // from the preg without the full first match.
    if (preg_match('#^' . $this->regex($pattern) . '$#u', $path, $parameters)) {
        return $this->arguments = array_slice($parameters, 1);
    }

    return false;
}
```

Listing 5: Route.php

able due to the highly formalized syntax `param string name_of_var`, rather than writing out "this function takes a parameter of type string named `name_of_var`". This does compensate for the tendency of comments to drift out of synchronicity with the code that they are supposed to comment, by tying them back to some computational system to verify its semantic contents, while providing information about the inputs and outputs of the function.

Beyond expliciting inputs and outputs, the second aspect of these comments is targeted at the *how* of the function, helping the reader understand the rationale behind the programmatic process. Comments here aren't cautionary notes on specific edge-cases, as seen in fig. 5 above, but rather natural language renderings of the overall rationale of the process. The implication here is to provide a broader, and more explicit understanding of the process of the function, in order to allow for further maintenance, extension or modification.

Finally, we look at a subset of the function, the clause of the third if-statement:

```
(preg_match('#^' . $this->regex($pattern) . '$#u', $path, $parameters))
```

. Without comments, one must realize on cognitive gymnastics and knowledge of the PHP syntax in order to render this as an extraction of all route parameters, implying the removal of the first element of the array. In this sense, then, Kirby's code for parsing an HTTP route is both verbose—in comments—and parsimonious—in code.

What these aesthetic features (small number of files, short file length, short function length) imply is an immediate feeling of *building blocks*. Short, graspable, (re-)usable (conceptual) blocks are made available to the developer directly, as the Kirby ecosystem, like many other open-source projects, relies on contributions from individuals who are not expected to have any other encounter with the project other than, at the bare mini-

mum, the source code itself.

These two examples, Microsoft Windows 2000 and Kirby CMS, show particular presentations of source code—through repetition, verbosity, commenting and conciseness. These are in part tied to their socio-technical ecosystems made up of hardware, institutional practices ranging from corporate guidelines to open-source contribution, with efficiency and usability remaining at the forefront, both at the result level (the software) and at the process level (the code).

Software developers are a large group of practitioners whose focus on producing effective, reliable and sustainable software, leads them to writing in more-or-less codified manner. Before diving into how such a manner of writing relates to references from architecture and engineering in order to foster simplicity and understandability, in section section 3, we acknowledge that the bondary between groups of practicioners isn't a clear-cut one, and so we turn to another practice closely linked to professional development—hacking.

2.1.2 Hackers

Hackers have been present in popular cultural mostly as young, computer experts, working on the edge of legality. These popular description of hackers tend to veer towards technical excellence, obsession and esoteric involvement with the machine, accompanied by seemingly-radical value systems and political beliefs. They are often depicted as lonely, obsessed programmers, hyperfocused on the task at hand and able to switch altered mental states as they dive into computational problems, as described by Joseph Weizenbaum in 1976[18]. While some of it is true—for instance, the

[18] "Wherever computer centers have become established, that is to say, in countless places in the United States, as well as in virtually all other industrial regions of the world, bright young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to

gender, the compulsive behaviour and the embodied connection to the machine—, hackers nonetheless designate a wider group of people, one which writes code driven mostly by curiosity, cleverness and freedom. Such a group has had a significant influence in the culture of programming, with which it overlaps with the aforementionned values of intellectual challenges.

To hack, in the broadest sense, is to enthusiastically inquire about the possibilities of exploitation of technical systems[19] and, as such, isnt' strictly bound to the advent of the computer[20]. Computer hacking specifically came to proeminence as early computers started to become available in north-american universities, and coalesced around the Massachussets Institute of Technology's Tech Model Railroad Club (Levy, 2010). Computer hackers were at the time skilled and highly-passionate individuals, with an autotelic inclination to computer systems: these systems mattered most when they referenced themselves, instead of interfacing with a given problem domain. Early hackers were often self-taught, learning to tinker with computers while still in high-school (Lammers, 1986), and as such tend to exhibit a radical position towards expertise: skill and knowledge aren't derived from academic degrees or credentials, but rather from concrete ability and practical efficacy[21].

The histories of hacking and of software development are deeply in-

strike, at the buttons and keys on which their attention seems to be as riveted as a gambler's on the rolling dice. When not so transfixed, they often sit at tables strewn with computer printouts over which they pore like possessed students of a cabalistic text." ((Weizenbaum, 1976))

[19] "HACKER [originally, someone who makes furniture with an axe] n. 1. A person who enjoys learning the details of programming systems and how to stretch their capabilities, as opposed to most users who prefer to learn only the minimum necessary. 2. One who programs enthusiastically, or who enjoys programming rather than just theorizing about programming." ((Dourish, 1988))

[20] See Rosenbaum's report in the October 1971 issue of Esquire for an account of phreaking, computer hacking's immediate predecessor (Rosenbaum, 2004).

[21] A meritocratic stance which has been analyzed in further in (Coleman, 2018)

tertwined: some of the early hackers worked on software engineering projects—such as the graduate students who wrote the Apollo Guidance Computer routines under Margaret Hamilton—, and then went on to profoundly shape computer infrastructure. Particularly, the development of the UNIX operating system by Dennis Ritchie and Ken Thompson is a key link in connecting hacker practices and professional ones. Developed from 1969 at Bell Labs, AT&T's research division, UNIX was "very close to being the first system under which a programmer could sit down directly at a machine and compose programs on the fly, exploring possibilities and testing while composing" ((Raymond, 2003)), and was "brainstormed by three people and implemented by Ken Thompson in two days — on an obsolete machine that had been designed to be a graphics terminal for a 'real' computer." ((Raymond, 2003)). This was a system which was supporting the kind of free exploration of a system's boundaries that is central to hacker culture, and which relied on sharing and circulating source code in order to allow anyone to improve it—in effect, AT&T's inexpensive licensing model until the 1980s, and the use of the C programming language starting from 1977 made it widely available within university settings[22]. UNIX, a product at the intersection of corporate and hacker culture, was spreading its design philosophy of clear, modular, simple and transparent design across programming communities.

The next step in the evolution of hacker culture built on this tenet to share source code, and hence to make written software understandable from its textual manifestation. The switch identified in the previous section from hardware being the most important component of a computing system to software had led manufacturers to stop distributing source code, making proprietary software the norm. Until then, executable software was the consequence of running the source code through a compilation

[22] "Unix has become well entrenched in the nation's colleges and universities due to Western Electric's extensive, inexpensive licensing of the system. As a result, many of today's graduating computer scientists are familiar with it." ((Morgan, 1982))

process; around the 1980s, executable software was distributed directly as a binary file, its exact contents an unreadable series of 0s and 1s. As a result to licensing changes of the UNIX system, the GNU project was created, and in its wake the Free Software Foundation, which established the ethical requirement to access the source code of any software.

In the meantime, personal microcomputers came to the market and opened up this ability to tinker and explore computer systems beyond the realms of academic-licensed large mainframes and operating systems. Starting with models such as the Altair 8800, the Apple II and the Commodore 64, as well as with easier, interpreted computer languages such as BASIC, whose first version for such micro-computers was written by Bill Gates, Paul Allen and Monte Davidoff (Montfort et al., 2014). While seemingly falling out of the realm of "proper" programming, the microcomputer revolution allowed for new groups of individuals to explore the interactivity of source code due to their small size when published as type-in listings.

In the wake of the larger free software movement, emerged its less radical counterpart, the open-source movement, as well as its more illegal counterpart, security hacking. The former is usually represented by the types of individuals depicted in mainstream news outlets when they reference hackers: programmers breaching private systems, sometimes in order to cause financial, intelligence or material harm. Security hackers, sometimes called crackers, form a community of practice of their own, with ideas of superior intelligence, subversion, adventure and stealth[23]. These practices nonetheless refer to the original conception of hacking—getting something done quickly, but not well—and include such a practical, efficient appoach into its own set of values and ideals, which are in

[23]For a lyrical account of this perception of the hacker ethos, see *The Conscience of a Hacker*, published in Phrack Magazine: " This is our world now... the world of the electron and the switch, the beauty of the baud. We make use of a service already existing without paying for what could be dirt-cheap if it wasn't run by profiteering gluttons, and you call us criminals. We explore... and you call us criminals. We seek after knowledge... and you call us criminals." ((Mentor+++, 1986))

turn represented in the kinds of program texts being written by members of this community of practice[24].

Meanwhile, the open-source movement took the tenets of hacking culture and adapted it to make it more compatible to the requirements of businesses. Open-source can indeed be seen as a compromise between the software industry development practices and the efficacy of free software development. Indeed, beyond the broad values of intellectual curiosity and skillful exploration, free software projects such as the Linux kernel, the Apache server or the OpenSSL project have proven to be highly efficient, and used in both commercial, non-commercial, critical and non-critical environments (Raymond, 2001). Such an approach sidesteps the political and ethical values held in previous iterations of the hacker ethos in order to focus exclusively on the sharing of source code and open collaboration while remaining within an inquisitive and productive mindframe. With the advent of corporate *hackathons*—short instances of intense collaboration in order to create new software, or new features on a software system—are a particularly salient example of this overlap between industry practices and hacker practices (Nolte et al., 2018)[25].

As a community of practice, hackers are programmers which, while overlapping with industry-embedded software developers, hold a set of values and ideals regarding the purpose and state of software. Whether academic hackers, amateurs, security hackers or open-source contributors, all are centered around the object of source code as a vehicle for communicating the knowledge held within the software, and the expertise necessary for writing such software, bypassing auxiliary resources like natural-language documentation. Incidentally, those political and ethical

[24] Those program texts include computer viruses, worms, trojan horses and injections, amongst others.

[25] Along with the address of the software corporate giant Meta's headquarters: 1, Hacker Way, Menlo Park, CA 94025, U.S.A.

values of expertise and openness often overlap with aesthetic values informing how their code exists in its textual manifestation.

Sharp and clever

> *To hack is, according to the dictionary, "to cut irregularly, without skill or definite purpose; to mangle by or as if by repeated strokes of a cutting instrument". I have already said that the compulsive programmer, or hacker as he calls himself, is usually a superb technician. It seems therefore that he is not "without skill" as the definition will have it. But the definition fits in the deeper sense that the hacker is "without definite purpose": he cannot set before him a clearly defined long-term goal and a plan for achieving it, for he has only technique, not knowledge. He has nothing he can analyze or synthesize; in short, he has nothing to form theories about. His skill is therefore aimless, even disembodied. It is simply not connected with anything other than the instrument on which it may be exercised. His skill is that of a monastic copyist who, though illiterate, is a first rate calligrapher. (Weizenbaum, 1976)*

While he looks down on hackers, perhaps unfairly, from the perspective of a computer scientist whose theoretical work can be achieved only through thought, pen and paper—an approach to programming which we will address in the next section—, the point still remains: hackers are first and foremost technical experts who can get lost in technics for their own sake. From a broad perspective, hackers therefore seem to exhibit an attitude of *direct engagement*, *subverted use* and *technical excellence*. Gabriella Coleman, in her anthropological study of hackers, highlights that they value both semantic ingenuity[26] and technical wittiness, even

[26] Hackers themselves tend to favor puns—the free software GNU project is a recursive acronym for *GNU's Not UNIX*.

though source code written by hackers can take multiple shapes, from one-liners, to whole operating systems, to deliberate decisions to subvert best practices in crucial moments

The *one-liner* is a piece of source code which fits on one line, and is usually intepreted immediately by the operating system. They are terse, concise, and eminently functional: they accomplish one task, and one task only. This binary requirement of functionality (in the strict sense of: "does it do what it's supposed to do?") actually finds a parallel in a different kind of one-liners, the humoristic ones in jokes and stand-up comedy. In this context, the one-liner also exhibits the features of conciseness and impact, with the setup conflated with the punch line, within the same sentence. One-liners are therefore self-contained, whole semantic statements which, through this syntactic compression, appear to be clever—in a similar way that a good joke is labelled clever.

In programming, one-liners have their roots in the philosophy of the UNIX operating system, as well as in the early diffusion of computer programs for personal computer hobbyists (Montfort et al., 2014). On the one side, the Unix philosophy is fundamentally about building simple tools, which all do one thing well, in order to manipulate text streams (Raymond, 2003). Each of these tools can then be piped (directing one output of a program-tool into the input of the next program-tool) in order to produce complex results—reminiscing of the orthogonality feature of programming languages (see chap. 4). Sometimes openly acknowledged by language designers—such as those of AWK—the goal is to write short programs which shouldn't be longer than one line. Given that constraint, a hacker's response would then be: how short can you make it?

If writing one-line programs is within the reach of any medium-skilled programmer, writing the shortest of all programs does become a matter of skill, coupled with a compulsivity to reach the most syntactically compressed version. For instance, Guy Steele[27] recalls:

[27]Influential langugage designer, who worked on Scheme, ECMAScript and Java, among

> *This may seem like a terrible waste of my effort, but one of the most satisfying moments of my career was when I realized that I had found a way to shave one word off an 11-word program that [Bill] Gosper had written. It was at the expense of a very small amount of execution time, measured in fractions of a machine cycle, but I actually found a way to shorten his code by 1 word and it had only taken me 20 years to do it. (Seibel, 2009)*

This sort of compulsive behaviour is also manifested in the practice of *code golf*, challenges in which programmers must solve problems by using the least possible amount of characters—here, the equivalent of *par* in golf would be Kolmogorov complexity[28]. Minimizing program length in relation to the problem complexity is therefore a definite feature of one-liners, since choosing the right programming language for the right tasks can lead to a drastic reduction of syntax, while keeping the same expressive and effective power. Tasked with parsing a text file to find which lines had a numerical value greater than 6, Brian Kernighan writes the code in 6[29]:

---

others.

[28] See: `https://en.wikipedia.org/wiki/Kolmogorov_complexity`

[29] From Succesful Language Design, Brian Kernighan at the University of Nottingham, `https://www.youtube.com/watch?v=Sg4U4r_AgJU`

The equivalent in AWK, a language he designed, and which he actually refers to in the comment on line 15, presumably as a heuristic as he is writing the function, is seen in 7

The difference is obvious, not just in terms of formal clarity and reduction of the surface structure, but also in terms of matching the problem domain: this obviously prints every line in which the third field is greater than 6. The AWK one-liner is more efficient, more understandable because it allows for less confusion, and is therefore more beautiful. On the other hand, however, one-liners can be so condensed that they loose all sense of clarity for someone who doesn't have a deep knowledge of the specific language in which it is written. For instance, here is Conway's game of life implemented in one line of APL8.

The obscurity of such a line—due to its highly-unusual character notation, and despite the pre-existing knowledge of the expected output—shows why one-liners are usually highly discouraged for any sort of code which needs to be worked on by other programmers. Cleverness in programming indeed tends to be seen as a display of the relationship between the programmer and the machine, rather than between different programmers and only tangentially about the machine. On the other hand, though, the nature of one-liners makes them highly portable and shareable, infusing them with what one could call *social beauty*. Popular with early personal computer adopters, at a time during which the source code of programs were printed in hobbyist magazines and needed to be input by hand, and during which access to computation wasn't widely distributed amongst society, being able to type just one line in, say, a BASIC interpreter, and resulting in unexpected graphical patterns created a sense of magic and wonder in first-time users—how can so little do so much?[30].

Another example of beautiful code written by hackers is the UNIX operating system, whose inception was an informal side-project spearheaded

[30]For an example fo such one-liner, see for instance: `https://www.youtube.com/watch?v=0yKwJJw6Abs`

```c
#include <stdio.h>
#include <strings.h>

int main(void){
  char line[1000], line2[1000];
  char *p;
  double mag;

  while(fgets(line, sizeof(line), stdin) != NULL) {
    strcpy(line2, line);
    p = strtok(line, "\\t");
    p = strtok(NULL, "\\t");
    p = strtok(NULL, "\\t");
    sscanf(p, "\%lf", &mag);
    if(mag > 6) /* $3 > 6 */
      printf("\%s", line2);
  }

  return 0
}
```

Listing 6: Selecting lines from an input file in C

```
awk '$3 > 6' data.txt
```

Listing 7: Selecting lines from an input file in AWK

```
life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ⌽¨ ⊂⊂⍵}
```

Listing 8: Conway's Game of Life implemented in APL

by Ken Thompson and Dennis Ritchie in the 1970s. As the first portable operating system, UNIX's influence in modern computing was significant, e.g. in showing the viability and efficiency of text-based processing, hierarchical file-system, shell scripting and regular expressions, amongst others. UNIX is also one of the few pieces of production software which has been carefully studied and documented by other developers. 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, which was circulated illegaly in classrooms for twenty years before its official publication was authorized by the copyright owners (Lions, 1996). This need to allow for quality software to be studied is highlighted by the architect Christopher Alexander, in the preface of software developer 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? (Gabriel, 1998)*

UNIX might be one of the answers to that question, both by its functionality, and by its conciseness, if not alone by its availability.

Another program which qualifies as beautiful hacker code, due both to its technical excellence, unusual solution and open-source availability is the function to compute the inverse square root of a number, a calculation that is particularly necessary in any kind of rendering application (which heavily involves vector arithmetic). It was found in the source code of id Software's *Quake* video game, listed in 9 verbatim[31].

[31]The Quake developers aren't the authors of that function—the merit of which goes to Greg Walsh—but are very much the authors of the comments.

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;    // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
                                                // this can be removed

    return y;
}
```

Listing 9: Inverse fast square root

What we see here is a combination of the understanding of the problem domain (i.e. the acceptable result needed to maintain a high-framerate with complex graphics), the specific knowledge of low-level computers operations (i.e. bit-shifting of a float cast as an integer) and the snappiness and wonder of the comments[32]. The use of `0x5f3759df` is what programmers call a *magic number*, a literal value whose role in the code isn't made clearer by a descriptive variable name. Usually bad practice and highly-discouraged, the magic number here is exactly that: it makes the magic happen.

Further examples of such intimate knowledge of both the language

[32] *what the fuck?* indeed

and the machine can be found in the works of the *demoscene*. Starting in Europe in the 1980s, demos were first short audio-visual programs which were distributed along with *crackware* (pirated software), and to which the names of the people having cracked the software were prepended, in the form of a short animation (Reunanen, 2010). Due to this very concrete constraint—there was only so much memory left on a pirated disk to fit such a demo—programmers had to work with these limitations in order to produce the most awe-inspiring graphics effects before software boot. One notable feature of the demoscene is that the output should be as impressive as possible, as an immediate, phenomenological appreciation of the code which could make this happen[33]. Indeed, the `comp.sys.ibm.pc.demos` news group states in their FAQ:

> *A Demo is a program that displays a sound, music, and light show, usually in 3D. Demos are very fun to watch, because they seemingly do things that aren't possible on the machine they were programmed on.*
>
> *Essentially, demos "show off". They do so in usually one, two, or all three of three following methods:*
>
> - *They show off the computer's hardware abilities (3D objects, multi-channel sound, etc.)*
> - *They show off the creative abilities of the demo group (artists, musicians)*
> - *They show off the programmer's abilities (fast 3D shaded polygons, complex motion, etc.) (Melik, 2012)*

This showing off, however, does not happen through immediate engagement with the code from the reader's part, but rather in the thorough explanation of the minute functionalities of the demo by its writer.

[33] For an example, see *Elevated*, programmed by iq, for a total program size of 4 kilobytes: `https://www.youtube.com/watch?v=jB0vBmiTr6o`

Because of these constraints of size, the demos are usually written in C, openGL, Assembly, or the native language of the targeted hardware. Source code listings of demos also make extensive use of shortcuts and tricks, and little attention is paid to whether or not other humans would directly read the source—the only intended recipient is a very specific machine (e.g. Commodore 64, Amiga VCS, etc.). The release of demos, usually in demoparties, are sometimes accompanied by documentation, write-ups or presentations[34]. However, this presentation format acknowledges a kind of individual, artistic feat, rather than the *egoless programming* lauded by Brooks in professional software development.

Pushing the boundaries of how much can be done in how little code, here is a 256-bytes demo resulting in a minute-long music video (Akesson, 2017) on the Commodore 64. It is firsted listed as a hexademical dump by its author (see 10)

Even with knowledge of how hexadecimal instructions map to the instruction set of the specific chip of of the Commodore 64 (in this case, the SID 8580), the practical use of these instructions takes productive advantage of ambivalence and side-effects. In the words of the author, Linus Akesson (emphasis mine):

> *We need to tell the VIC chip to look for the video matrix at address $0c00 and the font at $0000. This is done by writing $30 into the bank register ($dd18). But this will be done from within the loop, as doing so allows us to use the value $30 for two things.* An important property of this particular bank configuration is that the system stack page becomes part of the font definition.

Demosceners therefore tend to write beautiful, deliberate code which is hardly understandable by other programmers without explanation, and yet hand-optimized for the machine. This presents a different perspec-

[34] You can find *Elevated*'s technical presentation here: `https://www.iquilezles.org/www/material/function2009/function2009.pdf`

```
0000000 0801 080d d3ff 329e 3232 0035 0000 4119
0000010 d01c dc00 0000 d011 0be0 3310 610e f590
0000020 0007 1fff 4114 24d5 2515 5315 6115 29d5
0000030 0f1b 13e6 13e6 02d0 20e6 61a9 1c85 20a7
0000040 3fe0 08f0 0c90 114e 6cd0 fffc 6da0 2284
0000050 d784 4b4a a81c 13a5 3029 02d0 1cc6 2fe0
0000060 11f0 02b0 02a2 10c9 09f0 298a aa03 f3b5
0000070 0a85 ab2d b000 b711 b622 9521 a500 4b13
0000080 aa0e f8cb cc86 0749 0b85 13a5 0f29 0fd0
0000090 b8a9 1447 0290 1485 0729 b5aa 85f7 a012
00000a0 b708 910d 880f f910 b7a8 9109 8803 f9d0
00000b0 7e4c 78ea 868e 8e02 d021 4420 a2e5 bdfd
00000c0 0802 0295 d0ca 8ef8 0315 cc4c a900 8d50
00000d0 d011 ad58 dc04 c3a0 1c0d 48d4 044b 30a0
00000e0 188c 71d0 e6cb 71cb 6acb 2005 58a0 d505
00000f0 cb91 dfd0 aa2b 6202 1800 2026 2412 1013
```

Listing 10: A Mind is Born

tive of the relationship between aesthetics and understanding, in which aesthetics do not support and enable understanding, but rather become a proof of the mastery and skill required to input such a concise input for such an overwhelming output. This shows in an extreme way that one does need a degree of expert knowledge in order to appreciate it—in this sense, aesthetics in programming are shown to be almost often dependent on pre-existing knowledge.

Hackers are programmers who write code within a variety of settings, from academia to hobbyists through professional software development, with an explicit focus on knowledge and skill. Yet, some patterns emerge. First, one can see the emphasis on the *ad hoc*, insofar as choosing the right tool for the right job is a requirement for hacker code to be valued positively. This requirement thus involves an awareness of which tool will be the most efficient at getting the task at hand done, with a minimum of effort and minimum of overhead, usually at the expense of sustaining or maintaining the software beyond any immediate needs, making it available or comprehensible neither across time nor across individuals, a flavour of *locality*. Second, this need for knowing and understanding one's tools hints at a material relationship to code, whether instructions land in actual physical memory registers, staying away from abstraction and remaining in "concrete reality" by using magic numbers, or sacrificing semantic clarity in order to *"shave off"* a character or two.

The ideals at play in the writing and reading of source code for hackers is thus centered around specific means of knowledge: knowledge of the hardware, knowledge of the programming language used and knowledge of the tradeoffs acceptable all the while exhibiting an air of playfulness—how far can one go pushing a system's boundaries before it breaks down entirely? How little effort can one put in order to get a maximum outcome? Yet, one aspect that seems to elude hackers in their conception of code is that of conceptual soundness. If code is considered beautiful by attaining

previously unthought achievements of purposes with the least amount of resources, rationalization as to why, whether *a prior* or *a posteriori*, does not seem to be a central value. Hackers exhibit tendencies to both *get the job done* and *do it for the sake of doing it*.This behaviour is unlike that of computer and data scientists, and towards whom we turn to next.

If hacking can be considered a way of doing which deals with the practical intricacies of programming, involving concrete knowledge of the hardware and the language, our third group tends towards the opposite. Programming scientists (of which computer scientists are a subset) engage with progamming first and foremost at the conceptual level, with different locii of implementation: either as a *theory*, or as a *model*.

2.1.3 Scientists

Historically, then, programming emerged as a distinct practice from the computing sciences: not all programmers are computer scientists, and not all computer scientists are programmers. Nonetheless, scientists engage with programming and source code in two distinct ways, and as such open up the landscape of the type of code which can be written, and of the standards which support the evaluation of formally satisfying code. First, we will look at code being written outside of computer science research activities and, through it, examine how the specific needs of usability, replicability and data structuring link back to standards of software development. Then, we will to the code written by computer scientists, such as programming language designers, and develop how computer implementation exists between the dual scientific pillars of theorization and experimentation (Vardi, 2010).

Computation as a means

Scientific computing, defined as the use of computation in order to solve non-computer science tasks, started as early as the 1940s and 1950s in the United States, aiding in the design of the first nuclear weapons, among others (Oberkampf & Roy, 2010). Essentially, calculations necessary to the verification of theories in disciplines such as physics, chemistry or mathematics were handed over to the computing machines of the time. Beyond the military applications of early computer technology, one can point to Harlow and Fromm's article on *Computer Experiments in Fluid Dynamics*, published in 1965, focusing on how the advent of computing technology would prove to be of great assistance in physics and engineering:

> *The fundamental behavior of fluids has traditionally been studied in tanks and wind tunnels. The capacities of the modern computer make it possible to do subtler experiments on the computer alone. (Harlow & Fromm, 1965)*

At this time,Computation and computers are perceived as promising automated aids in processing data at a much faster rates than human scientists (Licklider, 1960). The remaining issue, then, is to make computers more accessible to scientists which did not have direct exposure to them, and therefore might be unfamiliar to the intricacies of their use. Beyond the unaffordable price point of university mainframes before the personal computer revolution, another vector for simplification and accessibility is the development of adequate programming languages. Developed in 1964 at Dartmouth College, BASIC (Beginners' All-purpose Symbolic Instruction Code) aims at addressing this hurdle by designing "*the world's first user-friendly programming language*" (Brooks, 2019). The intent is to provide non-computer scientists with easy means to instruct the computer on how to perform computations relevant to their work. Still, computing in the academia will only pick up with the distribution of the multiple versions

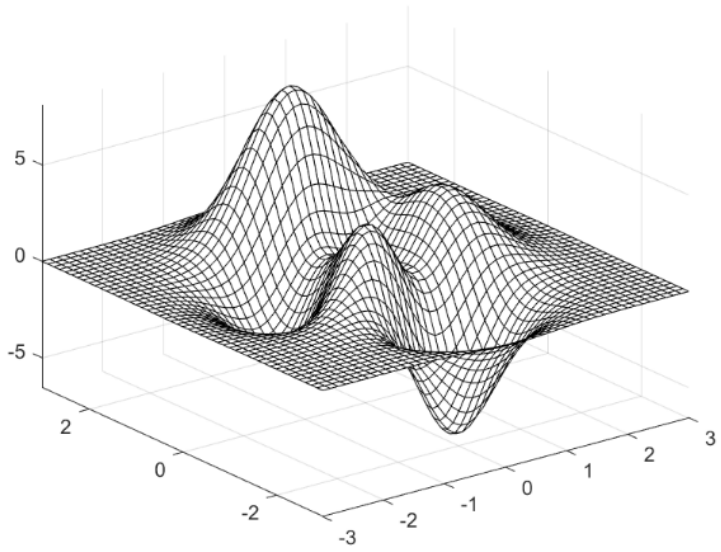
```
X = (-3:1/8:3)*ones(49,1);
Y = X';
Z = 3*(1-X).^2.*exp(-(X.^2) - (Y+1).^2) \
- 10*(X/5 - X.^3 - Y.^5).*exp(-X.^2-Y.^2) \
- 1/3*exp(-(X+1).^2 - Y.^2);
mesh(X,Y,Z)
```

Listing 11: Mesh.m

of the UNIX timesharing system, which allowed multiple users to use a given machine at the same time, and the performance boost as well as the versatility provided by the C programming language in the late 1970s.

By the dawn of the 21st century, scientific computing had increased in the scope of its applications (extending beyond engineering and experimental, so-called "hard" sciences, to social sciences and the humanities) as well as in the time spent developing and using software (Prabhu et al., 2011) (Hannay et al., 2009), with the main programming languages used being MATLAB, C/C++ and Python. While C and C++'s use can be attributed to their historical standing, popularity amongst computer scientists, efficiency for systems programming and speed of execution, MATLAB and Python offer different perspectives. MATLAB, originally a matrix calculator from the 1970s, became popular with the academic community by providing features such as a reliable way to do floating-point arithmetic and a graphical user interface (GUI). Along with its powerful array-manipulation features, the ability to visualize large series of data and plot it on a display largely contributed to MATLAB's popularity (Moler & Little, 2020), features shared with RStudio, a GUI to the R programming language. In 11, one can see in 12 how concise the plotting of a three-dimensional plane is in MATLAB, requiring only one call to `mesh`.

In parallel to MATLAB and R, Python represents the advent of the so-called scripting languages. Scripting languages are programming lan-



Listing 12: Visualization of a 3D-mesh in Matlab

guages which offer readability and versatility, along with decoupling from the actual operating system that it is being executed on. System languages, such as C, are designed and used in order to interact directly with the computer hardware, and to constitute data structures from the ground up (Ousterhout, 1998). On the other hand, scripting languages were designed and used in order to connect existing software systems or data sources together, most notably in the early days of shell scripting (such as `Bash`, `sed` or `awk`). Starting with the late 1990s, and the appearance of languages such as Perl[35] and Python[36], scripting languages became more widely used by non-programmers who already had data to work with and needed tools to exploit it. In the following decades, the development of additional scientific libraries such as *SciKit*, *NumPy* for mathematics and numerical work or *NLTK* for language processing and social sciences in Python comple-

[35] First version developed in 1987 by Larry Wall
[36] First official release in 1991 by Guido Van Rossum

mented the language's ease of use by providing manipulation of complex scientific concepts (Millman & Aivazis, 2011), a phenomenon of user-extension which has also been observed in R and MATLAB's ecosystems (Moler & Little, 2020).

This steady rise of scientific computing has nonetheless highlighted the apparent lack of quality standards in academic software, and how the lack of value judgments on the software written might impact the reliability of the scientific output. Perhaps the most well-known example of such a lack is the one revealed by the leak of the source code of the Climate Research Unit from the University of East Anglia in 2009 (Merali, 2010). In the leak, inline comments of the authors, show that particular variable values were chosen to make the simulation run, with scientific accuracy being only a secondary concern. Code reviews of external software developers point out to the code of the CRU leak as being a symptom of the general state of academic software. As Professor Darrel Ince stated to the UK Parliamentary Committee in February 2010:

> *There is enough evidence for us to regard a lot of scientific software with worry. For example Professor Les Hatton, an international expert in software testing resident in the Universities of Kent and Kingston, carried out an extensive analysis of several million lines of scientific code. He showed that the software had an unacceptably high level of detectable inconsistencies. (Committee, 2010)*

As a response to this realization, the beginning of the 2000s has seen the desire to re-integrate the best practices of software engineering in order to correct scientific software's lack of accuracy (Hatton & Roberts, 1994), resulting in the formation of communities such as the Research Software Engineers(Woolston, 2022). As we've seen above, software engineering had developed on their own since its establishment as an independent discipline and professional field. Such a split, described by Diane Kelly as a

"*chasm*" (Kelly, 2007) then had to face the different standards to which commercial software and scientific software were held to. For instance, commercial software must be extensible and performant, two qualities that do not necessarily translate to an academic setting, in which software might be written within a specific, time-constrained, research project, or in which access to computing resources (i.e. supercomputers) might be less of a problem.

Within Landau et. al's conception of the scientific process as the progression from problem to theory, followed by the establishment of a model, the devising of a method, and then on to implemementation and finally to assessment (Landau et al., 2011), code written as academic software is involved in the latter two stages of method and implementation. Within those two stages, software has to abide by the processes and requirements of scientific research. First and foremost, reproducibility is a core requirement of scientific research in general[37] and bugs in a scientific software system can lead to radically different ouptuts given slightly different input data, while concealing the origin of this difference. Good academic code, then, is one which defends actively against these, perhaps to the expense of performance and maintainability. This can be addressed by reliable error-handling, regular assertions of the state of the processed data and extensive unit testing (Wilson et al., 2014).

Furthermore, a unique aspect of scientific software comes from the lack of clear upfront requirements. Such requirements, in software development, are usually provided ahead of the programming process, and should be as complete as possible. As the activity of scientists is defined by an incomplete understanding of the application domain, requirements tend to emerge as further knowledge is developed and acquired (Segal, 2005). As a result, efforts have been made to familiarize scientists with software development best practices, so that they can implement quality software on

[37] We can date this requirement back to the seventeenth century with Robert Boyle and the Invisible College in England (LeVeque et al., 2012)

their own. Along with field-specific textbooks[38] the most prominent initiative in the field is *Software Carpentry*, a collection of self-learning and teaching resources which aims at implementing software best practices across academia, for scientists and by scientists. Founded by Greg Wilson, the co-editor of *Beautiful Code*, the organization's title refers directly to equivalents in the field of software development.

We see a convergence of quality standards of broad academic software towards the quality standards of commercial software development[39]. And yet, this convergence is due to, as we've seen, a past divergence between computation and science, as computer science worked towards asserting and pursuing its own field of research. As a subset of science, computer science nonetheless possesses its own specific standards, taking software not as a means to an end, but as the end itself.

Computation as an end

Computer scientists are scientists whose work focuses on computation as a means, rather than as a tool. They study the phenomenon of computation, investigating its nature and effects through the development of theoretical frameworks around it. Originally derived from computability theory, as a branch of formal mathematical logic, computation emerged as an autonomous field from work in mechanical design and configuration (Ada Lovelace and Charles Babbage), work on circuit and language design (C. S. Pierce, Konrad Zuse and John Von Neumann), work on mathematical foundations (Alan Turing and Alonzo Church), information theory (Claude

[38] See *Effective Computation in Physics* (Scopatz & Huff, 2015) or *A Primer for Computational Biology* (O'Neil, 2019) covering similar software-oriented material from different academic perspectives.

[39] See Graphbrain at `https://github.com/graphbrain/graphbrain` for such an example. The code's organization and formal features are congruent and on par with commercial software.

Shannon), systems theory (Norbert Wiener) and expert systems (John McCarthy and Marvin Minsky) (Ifrah, 2001). In the middle of such a constellation ranging from mathematical theory to practical electronics, computer science establishes its institutional grounding with the inauguration of the first dedicated academic department at Purdue University in 1962.

From this multifaceted heritage and academic interdisciplinarity, computer scientists identified key areas such as data structures, algorithms and language design as foundations of the discipline (Wirth, 1976). Though the process, the tracing of the "roots" of computation remained a constant debate as to whether computer science exists within the realm of mathematics, of engineering or as a part of the natural sciences. The logico-mathematical model of computer science contends that one can do computer science without an electronic computer, while the engineering approach of computer science tends to put more practical matters, such as architecture, language design and systems programming (implicitly assuming the use of a digital computer) at the core of the discipline; both being a way to generate and process information as natural phenomenon (Tedre, 2006).

The broad difference we can see between these two conceptions of computer science is that of *episteme* and *techne*. On the theoretical and scientific side, computer science is concerned with the primacy of ideas, rather than of implementation. The quality of a given program is thus deduced from its formal (in the mathematical sense) properties, rather than its formal (in the aesthetic sense) properties. The first manifestations of such a theoretical focus can be found in the Information Processing Language (1956 by Allen Newell, Cliff Shaw and Herbert Simon), which was originally designed and developed to prove Bertrand Russell's *Principia Mathematica*. While the IPL, as one of the very first programming languages, influenced the development of multiple subsequent languages, some later languages came to be known as logic programming languages, based on a formal logic syntax of facts, rules and clauses about a given domain and whose correct-

```
% induce(E,H) <- H is inductive explanation of E
induce(E,H):-induce(E,[],H).

induce(true,H,H):-!.
induce((A,B),H0,H):-!,
  induce(A,H0,H1),
  induce(B,H1,H).
induce(A,H0,H):-
  /* not A=true, not A=(_,_) */
  clause(A,B),
  induce(B,H0,H).
induce(A,H0,H):-
  element((A:-B),H0),     % already assumed
  induce(B,H0,H).         % proceed with body of rule
induce(A,H0,[(A:-B)|H]):-       % A:-B can be added to H
  inducible((A:-B)),      % if it's inducible, and
  not element((A:-B),H0), % if it's not already there
  induce(B,H0,H).         % proceed with body of rule
```

Listing 13: Prolog sample source

ness can be easily proven (see 13 below for an example of the *Prolog* logic programming language).

Due to its Turing-completeness, one can write programs such as language processing, web applications, cryptography or database programming (using the *Datalog* variant of *Prolog*), but its use seems to remain limited outside of theoretical circles in 2021[40].

Another programming language shares this feature of theoretical soundness faced with a limited range of actual use in production environ-

[40] See the Stackoverflow Developer survey for popular language uses `https://insights.stackoverflow.com/survey/2021`

ments, Lisp—*LISt Processor*—designed to process lists. It was developed in 1958, the year of the Dartmouth workshop on Artificial Intelligence, by its organizator, John McCarthy. Inheriting from IPL, it retained the core idea that programs should separate the knowledge of the problem (input data) and ways to solve it (internal rules), assuming the rules are independent to a specific problem.

The base structural elements of LISP are not symbols, but lists (of symbols, of lists, of nothing), and they themselves act as symbols (e.g. the empty list). By manipulating those lists recursively—that it, processing something in terms of itself—Lisp highlights even further this tendency to separate computation from the problem domain, and to exhibit autotelic tendencies. This is facilitated by its atomistic and relational structure: in order to solve what it has do, it evaluates each symbol and traverses a tree-structure in order to find a terminal symbol. Building on these features of complex structures with simple elements, Willam Byrd, computer scientst at the University of Utah, describes the following lines of Scheme (a LISP dialect) as "the most beautiful program ever written" (Byrd, 2017), a Scheme interpreter written in Scheme (14):

The beauty of such a program, for Byrd, is the abilty of these fourteen lines to reveal powerful and complex ideas about the nature and process of computation. As an interpreter, this program can take any valid Scheme input and evaluate it correctly, recreating computation in terms of itself. It does so by showing and using ideas of recursion (with calls to `eval-expr`), environment (with the evaluation of the `body`) and lambda functions, as used throughout the program. Following Alan Kay, creator of the Smalltalk programming language, Byrd equates the feelings he experiences in witnessing and pondering the program above to those suggested by Maxwell's equations, which constitute the foundation of classical electromagnetism ((2.1)) (Kay, 2004). In both cases, the quality ascribed to those inscriptions come from the simplicity and conciseness of their base elements—making it easy to understand what the symbols mean and how we can compute rel-

```
(define (eval-expr env)
(lambda (expr env)
  pmatch expr
    [,x (guard (symbol? x))
      (env x)]
    [(lambda (,x) ,body)
      (lambda (arg)
        (eval-expr body (lambda (y)
                                          (if (eq? x y)
                                              arg
                                              (env y)))))]
      [(,rator ,rand)
        ((eval-expr rator env)
          (eval-expr rand env))]))
```

Listing 14: Scheme interpreter written in Scheme

evant outputs—all the while allowing for complex consequences for both, respectively, computer science and electromagnetism.

$$\frac{\partial \mathcal{D}}{\partial t} = \nabla \times \mathcal{H}\frac{\partial \mathcal{B}}{\partial t} = -\nabla \times \mathcal{E}\nabla \cdot \mathcal{B} = 0\nabla \cdot \mathcal{D} = 0 \tag{2.1}$$

With this direct manipulation of symbolic units upon which logic operations can be performed, Lisp became the language of AI, an intelligence conceived first and foremost as abstractly logical, if not outright algebraic. Lisp-based AI was thus working on what Seymour Papert has called "toy problems"—self-referential theorems, children's stories, or simple puzzles or games (nil, 2009). In these, the problem and the hardware are reduced from their complexity and multi-consequential relationships to a finite, discreete set of concepts and situations. Confronted to the real world—that is, to commercial exploitation—Lisp's model of symbol manipulation, which proved somewhat successful in those early academic scenarios, started to be applied to issues of natural language understanding and generation in broader applications. Despite disappointing reviews from government reports regarding the effectiveness of these AI techniques, commercial applications flourished, with companies such as Lisp Machines, Inc. and Symbolics offering Lisp-based development and support. Yet, in the 1980s, over-promising and under-delivering of Lisp-based AI applications, which often came from the combinatorial explosion deriving from the list- and tree-based representations, met a dead-end.

"*By making concrete what was formerly abstract, the code for our Lisp interpreter gives us a new way of understanding how Lisp works*", notes Michael Nielsen in his analysis of Lisp, pointing at how, across from the *episteme* of computational truths stands the *techne* of implementation (Nielsen, 2012). The alternative to such abstract, high-level language, is then to consider computer science as an engineering discipline, a shift between theoretical programming and practical programming is Edsger Dijk-

stra's *Notes on Structured Programming*. In it, he points out the limitation of considering programming exclusively as a concrete, bottom-up activity, and the need to formalize it in order to conform to the standards of mathematical logical soundness. Djikstra argues for the superiority of formal methods through the need for a sound theoretical basis when writing software, at a time when the software industry is confronted with its first crisis.

Within the software engineering debates, the theory and practice distinction had a slightly different tone, with terms like "art" and "science" labeling two different mindsets concerning programming (Knuth, 1997). As mentioned by Djikstra's example, software engineering suffered from an earlier image of programming as an inherently unmanageable, unsystematic, and artistic activity. There again, many saw programming essentially as an art or craft (Tedre, 2006), rather than an exact science. Beyond theoretical soundness, computer science engineering concerns itself with efficiency and sustainability, with measurements such as the *O()* notation for program execution complexity. It is not so much about whether it is possible to express an algorithm in a programming language, but whether it is possible to run it effectively, in the contingent environments of hardware, humans and problem domains[41].

This approach, halfway between science and art, is perhaps best seen in Donald Knuth's magnum opus, *The Art of Computer Programming*. In it, Knuth summarizes the findings and achievements of the field of computer science in terms of algorithm design and implementation, in order to "*to organize and summarize what is known about the fast subject of computer methods and to give it firm mathematical and historical foundations.*" (Knuth, 1997). The art of computer programming, according to him, is therefore based on mathematics, but nonetheless different from it insofar as it does have to deal with effectiveness, implementation and contin-

[41] Notably, algorithms in textbooks tend to be erroneous when used in production; only in five out of twenty are they correct (Pattis, 1988).

gency[42]. In so doing, Knuth takes on a more empirical approach to programming than his contemporaries, inspecting source code and running software to assess their performance, an approach he first inaugurated for FORTRAN programs when reporting on their concrete effectiveness for the United States Department of Defense (Defense Technical Information Center, 1970).

Structure and Interpretation of Computer Programs is another influential academic textbook dealing not just with computation as a an autotelic phenomenon, in which the authors insist that source code is "*must be written for people to read, and only incidentally for machines to execute*" (Abelson et al., 1979). Still, even when confronted with implementation and the plurality of contingencies of non-mathematical elements which accompanies it, the aesthetic standard in this more engineering approach to computer science is the proportionality between the number of lines of code written and the complexity of the idea explained, as we can see in the series *Beautiful Julia Algorithms* (Moss, 2022). For instance, 15 implements the Bubble Sort sorting algorithm in one loop rather than the usual two loops in C, resulting in an easier grasping of the concept at hand, rather than being distracted by the idiosyncracy of the implementation. The simplicity of scientific algorithms is expressed even further in 16 the one-line implementation of a procedure for finding a given element's nearest neighbor, a crucial component of classification systems, including AI systems.

According to Tedre, computer science itself was split in a struggle between correctness and productivity, between theory and implementation, and between formal provability and intuitive art (Tedre, 2014). In the early developments of the field, when machine time was expensive and every instruction cycle counted, efficiency ruled over elegance, but in the end he assesses elegance prevailed, as we will see with the evolution of craft within programming in section 1.4.1 below.

[42] *The Art of Computer Programming* involves a hypothetical computer, called MIX, to implement the algorithms discussed.

```
function bubble_sort!(X)
for i in 1:length(X), j in 1:length(X)-i
    if X[j] > X[j+1]
        (X[j+1], X[j]) = (X[j], X[j+1])
    end
end
end
```

Listing 15: Bubble Sort implementation in Julia

```
function nearest_neighbor(x', phi, D, dist)
  D[argmin([dist(phi(x), phi(x')) for (x,y) in D])][end]
end
```

Listing 16: Nearest neighbor implementation in Julia

In closing, one should note that the *Art* in the title of Knuth's series does not, however, refer to art as a fine art, or a purely aesthetic object. In a 1974 talk at the ACM, Knuth goes back to its Latin roots, where we find *ars*, *artis* meaning "skill.", noting that the equivalent in Greek being □□□□□, the root of both "technology" and "technique.". This semantic proximity helps him reconcile computation as both a science and an art, the first due to its roots in mathematics and logic, and the second

> *because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the state of the Art. (Knuth, 1974)*

When written within an academic and scientific context, source code

tends to align with the aesthetic standards of software development, valuing reliability, reabability, sustainability, in particular through Greg Wilson's work on the development of software development principles through the Software Carpentry initiative. This alignment can also be seen in a conception of computer science as a kind of engineering, as an empirical practice which can and should still be formalized in order to become more efficient. There, one can turn to Donald Knuth's *Art of Computer Programming* to see the connections between the academia's standards and the industry's standards.

And yet, a conception of computation as engineering isn't the only conception of computer science. Within a consideration of computer science as a theoretical and abstract object of study, source code becomes a means of providing insights into more complex abstract concepts, seen in the Lisp interpreter, or one-line algorithms implementing foundational algorithms in computer science, similar to this aspect of the hacker ethos. It is this relation to a conception of beauty traditionally associated with mathematics and engineering which we will investigate further to highlight which aesthetic ideals can be ascribed to code. But, first, we complete our overview of code practitioners by turning to the software artists, who engage most directly with source code as a written material through source code poetry.

2.1.4 Poets

Source code poetry is a distinct subset of electronic literature, and software art. One the one hand, electronic literature is a broad field encompassing natural language texts taking full advantage of the dynamic feature of computing to redefine the concept of text, authorship and readership. It encompasses a variety of approaches, including generative literature, interactive fiction, visual poetry, source code poetry and esoteric programming languages, as well as certain aspects of software art. However, we focus here only on the elements of electronic literature which shift their focus from output to input, from executable binary with transformed natural language as a result, to static, latent source.

On the other hand, software art is an umbrella term regrouping artistic practices which engage with the computer on a somewhat direct, material level, whether through hardware[43] or software[44]. This space for artistic experimentation flourished at the dawn of the 20th century, with initiatives such as the *Transmediale* festival's' introduction of a *software art* award between 2001 and 2004, or the *Run_me* festival, from 2002 to 2004. In both of these, the focus is on projects which incorporate standalone programmes or script-based applications which aren not merely functional tools, but also act as an effective artistic proposition, as decided by the artist, jury and public. These works often bring the normally hidden, basic materials from which digital works are made (e.g. code, circuits and data structures) into the foreground (Yuill, 2004). Within this realm, code poetry is a form a software art whose execution is only secondary to the work's meaning.

Electronic literature, a form based on the playful *détournement* of the computer's constraints, gets closer to our topic insofar as the poems generated represent a more direct application of the rule-based paradigm to the syntactical output of the program. Starting with Christopher Stratchey's

[43] See Alexei Shuglin's *386 DX* (1998-2013)
[44] See Netochka Nezanova's *Nebula.M81* (1999)

love letters (1953), generated (and signed!) by MUC, the Manchester Univac Computer, computer poems are generated by algorithmic processes, and as such rely essentially on this particular feature of programming: laying out rules in order to synthesize syntactically and semantically sound natural language poems. Here, the rules themselves matter only in relation to the output, as seen by their ratio: a single rule for a seemingly-infinite amount of outputs, with these outputs very often being the only aspect of the piece shown to the public.

These works and their authors build on a longer tradition of rule-based composition, from Hebrew to the Oulipo and John Cage's indeterministic composition, amongst others (Cramer, 2003), a tradition in which creativity and beauty can emerge from within a strict framework of formal rules. Nonetheless, the source code to these works is rarely released in conjunction with their output, hinting again at their lesser importance in terms of their overall artistic values. If electronic literature is composed of two texts, a natural-language output and a computer-language source, only the former is actually considered to be poetry, often leaving the latter in its shadow (as well as, sometimes, its programmer, an individual sometimes different from the poet). The poem exists through the code, but isn't exclusively limited to the human-readable version of the code, as it only comes to life and can be fully appreciated, under the poet's terms, once interpreted or compiled. While much has been written on electronic literature, few of those commentaries focus on the soundness and the beauty of the source as an essential component of the work, and only in recent times have we seen the emergence of close-readings of the source of some of these works for their own sake[45]. These constitute a body of work centered around the concept of generative aesthetics (Goriunova & Shulgin, 2005), in which beauty comes from the unpredictable and somewhat complex interplay of rule-based systems, and whose manifestations encompass not

[45]See the publications in the field of Critical Code studies, Software studies and Platform studies.

only written works, but games, visual and musical works as well.

And yet, the approach of code poets is more specific than broad generative aesthetics: it is a matter of exploring the specific expressive affordances of source code, and the overlap of machine-meaning and human-meaning essential to the correct functioning of code which acts as a vector for artistic communication. Such an overlap of meaning is a specific feature of source code poetry. In a broad sense, code poetry conflates classical poetry (as strict syntactical and phonetical form, combined with poetic expressivity) with computer code, but it is primarily defined by the fact that it does not require the code to be executed, but only to be read by a human. Following the threads laid out by electronic literature, code poetry starts from this essential feature of computers of working with strictly defined formal rules, but departs from it in terms of utility. Code poems are only functional insofar as they are accepted by the intepreter or compiler of the language in which they are written. To the computer, they are indeed functional, in that they are legal and can be parsed; but they do not do anything of *use*. Such formal compliance is only a pre-requisite, a creative constraint, for their human writers.

Within this reliance on creative constraints provided by a computing environment, the emphasis here is on the act of reading, rather than on the act of deciphering, as we've seen with obfuscated code (and in functional code in general). Source code poems are often easy to read, and have an expressive power which operates beyond the common use of programming. Building on Flusser's approach, let us consider poetry as a means to bring concepts into the thinkable, and to crystallize thoughts which are not immediately available to us (FLUSSER & Novaes, 2014); through various linguistic techniques, poetry allows us to formulate new concepts and ideas, and to shift perspectives.

In their different manifestations, code poems make the boundary between computer meaning and human meaning thinner and thinner, a feature often afforded by the existence and use of higher-level programming

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

Listing 17: Just Another Perl Hacker, japh.pl

languages. Starting with the development of FLOWMATIC in 1955 by Grace Hopper, it was shown that an English-like syntactical system could be used to communicate concepts for the computer to process. From there, programming languages could be described along a gradient, with binary at the lowest end, and natural language (in an overwheling majority, English) at the highest end. This implies that they could be written and read similarly to English, including word order, pronouncation and interpretation, similar to the error-tolerance of human laguages, which doesn't cause the whole communication process to fail whenever a specific word, or a word order isn't understood.

The community of programmers writing in Perl, *perlmonks*[46] has been one of the most vibrant and productive communities when it comes to code poetry. This particular use of Perl started in 1990, when the language creator Larry Wall shared some of the poems written in the language, and it gained further exposition through the work of Shannon Hopkins (Hopkins, 1992). The first Perl poem is considered to have been written by Wall in 1990, reproduced in 17.

Hopkins analyzes the ability of the poem to enable dual understandings of the source—human and machine. Yet, departing from the previous conceptions of source that we've looked at, code poetry does not aim at expressing the same thing to the machine and to the human. The value of a good poem comes from its ability to evoke different concepts for both readers of the source code. As Hopkins puts it:

[46] See their website: `https://perlmonks.org/`, with the spiritual, devoted and communal undertones that such a name implies.

> *In this poem, the q operator causes the next character (in this case a newline) to be taken as a single quote, with the next occurrence of that delimiter taken as the closing quote. Thus, the single-quoted line 'Just another Perl hacker' is printed to STDOUT. In Perl, the "unless $spring" line is mostly filler, since $spring is undefined. In poetical terms, however, "$spring" is very important: haiku poetry is supposed to specify (directly or indirectly) the season of the year. As for the q operator, that reads in English as the word "queue", which makes perfect sense in the context of the poem. (Hopkins, 1992)*

The poem *Black Perl*, submitted anonymously, is another example of the richness of the productions of this community:

```
#!/usr/bin perl
no warnings;

BEFOREHAND: close door, each window & exit; wait until time.
  open spellbook, study, read (scan, $elect, tell us);
write it, print the hex while each watches,
  reverse its, length, write, again;
kill spiders, pop them, chop, split, kill them.
  unlink arms, shift, wait & listen (listening, wait),
  sort the flock (then, warn "the goats" & kill "the sheep");
kill them, dump qualms, shift moralities,
  values aside, each one;
die sheep? die to : reverse { the => system
  ( you accept (reject, respect) ) };
next step,
  kill `the next sacrifice`, each sacrifice,
  wait, redo ritual until "all the spirits are pleased";
do { it => "as they say" }.
  do { it => (*everyone***must***participate***in***forbidden**s*e*
      x*)
+ }.
  return last victim; package body;
exit crypt (time, times & "half a time") & close it,
  select (quickly) & warn your (next victim);
AFTERWARDS: tell nobody.
  wait, wait until time;
wait until next year, next decade;
sleep, sleep, die yourself,
  die @last
```

The most obvious feature of this code poem is that it can be read by any-

one, including by readers with no previous programming experience: each word is valid both as English and as Perl. A second feature is the abundant use of verbs. Perl belongs to a family of programming languages grouped under the *imperative* paradigm, which matches a grammatical mood of natural languages, the *imperative mood*. Such mood emphasizes actions to be take rather than, for instance, descriptions of situations, and thus sets a clear tone for the poem. The fact that Perl is based on stating procedures to be executed and states to be changed creates this feeling of relentless urgency when reading through the poem, a constant need to be taking actions, for things to be changed. Here, the native constraints of the programming language interacts directly with the poetic suggestion of the work in a first way: the nature of Perl is that of giving orders, resulting in a poem which addresses *someone* to execute *something*. Still, Perl's flexibility leaves us wondering as to who and what are concerned by these orders. Is the poem directing its words to itself? To the reader? Is Perl just ever talking exclusively to the computer? This ambiguity of the adressee adds to the ominousness of each verse.

Following in the footsteps of the *perlmonks*, additional communities around code poetry have formed, whether in university settings[47], or as independent intiatives[48]. Beyond collections such as threads and hashtags on Twitter[49], code poetry also features artistic publications, such as printed anthologies of code poetry in book form (Bertram, 2012) (Holden & Kerr, 2016).

Yet, code poems from the 20th century aren't the first time where a part of the source code is written exclusively to elicit a human reaction, without any machinic side-effects. One of the earliest of those instances is per-

[47] Such as Stanford's Code Poetry Slam, which ran between 2014 and 2016, `https://web.archive.org/web/20161024152353/http://stanford.edu/%7Emkagen/codepoetryslam/`

[48] See the Source Code Poetry event, `https://www.sourcecodepoetry.com/`

[49] See #SongsInCode at `https://twitter.com/search?q=%2523SongsInCode`

```
663  STODL   CG
664  TTF/8
665  DMP*   VXSC
666         GAINBRAK,1   # NUMERO MYSTERIOSO
667         ANGTERM
668  VAD
669 ^^I     LAND
670     VSU^^IRTB
```

Listing 18: AGC source code for the Lunar Landing Guidance Equation, 1969

haps the Apollo 11 Guidance Computer (AGC) code, written in 1969[50] in Assembly. Cultural references and jokes are peppered throughout the text as comments, asserting computer code as a means of expression beyond exclusively technical tasks[51], and independent from a single writer's preferences, since they passed multiple checks and review processes to end up in the final, submitted and executed document, such as reproduced in 18.

Code comments allow a programmer to write in their mother tongue, rather than in the computer's, enabling more syntactic and semantic flexibility, and thus reveal a burgeoning desire for programmers to express themselves within their medium of choice.

At the turn of the 1980s, following the transition to programming from an annex practice to full-fledged discipline and profession, and with the development of more expressive programming languages (e.g. Pascal in 1970, C in 1972), software development had grown exponentially[52], and fostering practices, communities and development styles and patterns[53].

[50] Hamilton et. al., 1969, retrieved from https://github.com/chrislgarry/Apollo-11

[51] See also: "Crank that wheel", "Burn Baby Burn"

[52] See Stackoverflow's Developer Profile survey: `https://insights.stackoverflow.com/survey/2019#developer-profile-_-years-since-learning-to-code`

[53] From Djikstra's Notes on Structured Programming to Knuth's Literate Programming and Martin's Clean Code

Source code becomes recognized as a text in its own, which can hold qualities and defects of its own, and to which engineering and artistic attention must be paid. No longer a transitional state from formula to binary, it becomes a semantic material, whose layout, organization and syntax are important to the eyes of its writers and readers. Pushing further into the direction of the visual layout of the code, such an endeavour becomes pursued for its own sake, equally important to the need for a program to be functional.

The Obfuscated C Code Contest[54], starting in 1984, is the most popular and oldest organized production of such code, in which programmers submit code that is functional and visually meaningful beyond the exclusive standards of well-formatted code. If the source code's two meanings of a computer poem was made readily available to interpretation, and if such meanings existed in parallel in the comments of the AGC routines, pointing at the overlay of computer-related semantics (e.g. line numbers) and human-related semantics (e.g. a reference to the biblical number of the beast, 666), obfuscated code is a first foray into closely intertwining these separate meanings in the source code itself, making completely transparent, or completely opaque what the code does just by glancing at it.

The source code in 19, submitted to the 1988 IOCCC[55] is a procedure which does exactly what it shows: it deals with a circle. More precisely, it estimates the value of PI by computing its own circumference. While the process is far from being straightforward, relying mainly on bitwise arithmetic operations and a convoluted preprocessor definition, the result is nonetheless very intuitive—the same way that PI is intuitively related to PI. The layout of the code, carefully crafted by introducing whitespace at the necessary locations, doesn't follow any programming practice of indentation, and would probably be useless in any other context, but nonetheless

[54] `https://www.ioccc.org`

[55] Source: `https://web.archive.org/web/20131022114748/http://www0.us.ioccc.org/1988/westley.c`

```
    #define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_
            _-_-_-_-_
}
```

Listing 19: westley.c, entry to the 1988 IOCCC

represents another aspect of the *concept* behind the procedure described, not relying on traditional programming syntax[56], but rather on an intuitive, human-specific understanding[57].

Obfuscating practices, beyond their technical necessities (for security and efficiency), are traditionally tied to hacking practices, prominently with one-liners (the shorter a name, the more obscure and general it becomes). As such, they rely on the brain-teasing process of deciphering, and on the pleasurable, aesthetic experience of resolving and uniting two parallel meanings: what we see in the code, and what it does[58]. What we focus on here is the aspect of obfuscation which plays with the different layers of meaning: meaning to the computer, meaning to the human, and different ways of representing and communicating this meaning (from uglifying, to consistent formatting, to depicting a circle with dashes and underscores). While the aesthetics at play in hacking will be further explored below, we focus on the fact that obfuscating code practices, beyond hiding the meaning and the intent of the program, also manifest an attempt to represent such a meaning in different ways, leaving aside traditional code-writing practices and suggesting the meaning of the program by challenging the abilities of human interpretation at play in the process of deciphering programs.

Code poetry values code which, while being functional, expresses more than what it does, by allowing for *Sprachspiele*, languages games where pronounciation, syntax and semantics are playfully composed into a fluid linguistic construct in order to match a human poetic form, such as the haiku, or to constitute a specific puzzle. Relying on the inherent tendency of source code to remain opaque, obfuscated code contests let us see how far can such an opacity of a computer program's effective meaning be sus-

[56] For such a program, see for instance: `https://crypto.stanford.edu/pbc/notes/pi/code.html`

[57] Concrete poetry also makes such a use of visual cues in traditional literary works.

[58] Also known informally as the "Aha!" moment, crucial in puzzle design.

tained.

In this section, we've seen how the set of individuals who write and read code is heterogeneous. Instead, we can see a significant degree of variation between source code written within the context of software engineering, hacking, scientific research and artistic activity. While none of these areas are exclusive of the others—a software developer by day can hack on the weekend and participate in code poetry events—, they do convey different perspectives on how the code is written, and on how it is evaluated. This cursory introduction to each approach has shown that, for instance, software engineers prefer code which is modular, modifiable, sustainable and understandable by the largest audience of possible contributors, while hackers would favor conciseness over expressivity, and tolerate idiosyncracy for the purpose of immediate, functional efficiency. On the other hand, scientific programming favors ease of use, accuracy and reproducibility, sometimes overlapping with software engineering, while code poets explore the semantic tension between a human interpretation and the machine interpretation of a given source code.

These are strands of similarity within apparent diversity. The code snippets in this section show that there is a tendency to prefer a specific group of qualities—readability, conciseness, clarity, expressivity and functionality—even though different types of the aforementionned practices would put a different emphasis on each of those aspects. The question we turn to next, then, is to what extent do these different practices of code writing and reading share common judgments regarding their formal properties? Do hackers and poets agree on some value judgment, and how? To start this investigation, we first analyze programmers' discourses in the following section in order to identify concrete categories of formal properties which might enable a source code to be positively valued for its appearance, before we turn to the aesthetic registers code practicionners refer to when discussing beautiful code to further qualifies these proper-

ties.

2.2 Ideals of beauty

With this overview of the varieties of practices at play amongst those who read and write source code, we will analyze more thoroughly what are the aesthetic standards most value by those different groups. The aim here is to formalize our understanding of which source code is considered beautiful, and to do so at multiple levels. The goal here is to capture both the specific manifestations of beautiful code as specified and enunciated by programmers, as well as the semantic contexts from which these enunciations originate. What we will see is that, while a set of aesthetic values and a set of aesthetic manifestations can be pinpointed precisely, the domains that are mobilized to justify these values are clearly distinct. To do so, we will introduce a discourse analysis framework for the empirical study of the corpus, followed by an examination of the broad fields that these discourses refer to. This will be complemented in Chapter 3 by a a medium-specific reading through critical code studies and rhetorical code studies on one hand, and the work done by conceptual metaphors on the other.

2.2.1 Introduction to the Methodology

Discourse consists of text, talk and media, which express ways of knowing the world, of experiencing and valuing the world. This study builds on Kintsch and Van Dijk's work on providing tools to analyze an instance of discourse, and is centered around what is said to constitute good source code. While discourse analysis can also be used critically by unearthing which value judgments that occur in power relationships[59], we focus here on aesthetic value judgments, as their are first expressed through language. Of all the different approaches to discourse, the one we focus on here is that of *pragmatics*, involving the spatio-temporal and intentional

[59] See Diana Mullet on Critical Discourse Analysis (Mullet, 2018)

context in which the discourse is uttered. We find this approach particularly fitting through its implication of the *cooperative principle*, in which utterances are ultimately related to one another through communicative cooperation to reveal the intent of the speaker (Schiffrin, 1994). Practically, this means that we assume the position of programmers talking to programmers is cooperative insofar as both speaker and listener want to achieve a similar goal: writing good code. This double understanding—focusing first and foremost on utterances, and then re-examining them within a broader cooperative context—will lead us to encompass a variety of production media (blog post, forums, conferences, text books), in order to depict the cultural background (software practices as outlined above as well as additional factors such as skill levels). Our comprehension of those texts, then, will be set in motion by a a dual movement between local, micro-units of meaning and broader, theoretical macro-structure of the text, and linked by acts of co-reference (Kintsch & van Dijk, 1978).

Particular attention will be paid to the difference between intentional and extensional meaning (Dijk & Kintsch, 1983). As we will see, some of the texts in our corpus tend to address a particular problem (e.g. on forums, social media or question & answer platforms), or to discuss broader concepts around well-written code. Particularly,

> *Figures of speech may attract attention to important concepts, provide more cues for local and global coherence, suggest plausible pragmatic interpretations (e.g., a promise versus a threat), and will in general assign more structure to elements of the semantic representation, so that [meaning] retrieval is easier. (Dijk & Kintsch, 1983)*

Following this idea, we will proceed by examining syntactic markers to deduce overarching concepts at the semantic level. Among those syntactic markers, we include single propositions as explicit predicates regarding source code, lexical fields used in those predicates in order to identify their

connotations and denotations, as well as for the tone of the enunciations to identify value judgments. At the semantic level, we will examine the socio-cultural references, the *a priori* knowledge assumed from the audience, as well as the semantic entities which compose the theme of the discourse at hand. Finally, the discourses we will examine aren't exclusively composed of natural language, but also of source code extracts, resulting in a hybrid between natural and machine syntax within the same discursive artifact.

In line with John Cayley's analytic framework of structure, syntax and vocabulary (Cayley, 2012), we can nonetheless echo discourse analysis as applied to natural languages. Cayley's framework highlights essential aspect of analysis which applies both to natural languages and source code: that of the scales at which aesthetic judgment operates. It also provides a bridge with literature and literary studies without imposing too rigid of a grid preventing interdisciplinarity. While it does not immediately acknowledge more traditional literary concepts such as fiction, authorship, literarity, etc., Cayley's framework does leave room for these concepts to be taken into account. Particularly, we will see that the concept of authorship—who writes to whom—will prove to be useful.

Finally, our intepretation of the macrostructures described by Kintsch and Van Dijk will rely extensively on the work done by metaphors as the conceptual level, rather than at the strictly linguistic one. Lakoff and Johsnon's seminal work develops a theory of conceptual metaphors by highlighting their essential dependence on pre-existing cognitive structures, which we associate with already-understood concepts. The metaphor maps a source domain (made up of cognitive structure(s)) to a target domain. In the process, they extend the field of applicability of metaphors from the strictly literary to the broadly cultural: metaphors work because each of us has some conception of those domains involved in the metaphorical process. Metaphors rely in part on a static understanding, resulting in a fixed meaning from the application of a given source to a given target. Some of these sources are called *schemas*, and are de-

fined enough to not be mistaken for something else, but broad enough to allow for multiple variants of itself to be applied to various targets (Lakoff, 1980), providing both diversity and reliability. As we will see below, their approach allows us to focus not just on textual objects, but on the vast range of metaphors used specifically in computing-related environments. Given that the source of the metaphor should be grounded, with as little invariablity as possible, in order to qualify a potentially ill-defined target domain, this provides us with a first foray into the inherent elusiveness and instability of computing when presented to a broader audience.

Going beyond the role of metaphors manifested in expressions such as *the desktop*, *the mouse*, or *the cloud*, we will explore Lakoff's understanding of the specifically poetic metaphor in Chapter 3 while assessing the linguistic component of computing—source code. For now, we will pay close attention to what programmers are saying about (beautiful) source code, which metaphors they employ to support these value judgments, and why—focusing first on the metaphors *of* source code, before moving, in the next section, to the metaphors *in* source code.

The corpus studied here consists of texts ranging from textbooks and trade manuals to blog posts and online forum discussions[60]. The rationale behind such a broad approach is to constitute a lexical basis for what practicing programmers consider when assessing good code, as expressed in the everyday interactions of online forums and blog posts, but also inclusive of diverse sources of communication, beyond edited volumes. From authoritative sources, such as canonical textbooks or widely-read blog posts from well-known practicioners, to more casual forum exchanges will support the empirical dimension of our research. From this approach, this section will show how there are *specific* ways to qualify well-written code,

[60] Specifically, we have gathered 47 different online sources, from forum discussions to blog posts, 26 journal articles from the Association for Computing Machinery, 20 monographs and 1 edited volume, listed in Appendix I. These constitute our primary sources insofar as they are written by practicioners on the topic of good and beautiful code.

which are echoed both from bottom-up and from top-down perspectives, and employing recurring references.

2.2.2 Lexical Field in Programmer Discourse

In terms of existing studies of the lexical field programmers use, Erik Pineiro has done significant work in his doctoral thesis. In it, he argues that aesthetics exist from a programmers perspective, decoupled from the final, executable form of the software. While this current study draws on his work, and confirms his findings, it also builds upon it in several ways. First, Pineiro focuses on a narrower corpus, that of the Slashdot.org[61] forums (Pineiro, 2003) (p. 51). Second, he examines aesthetic judgment from a private perspective of software engineers, separate from other possible aesthetic fields which might enter in dialogue with beautiful code (Pineiro, 2003) (p.52), such as artists or hackers. Finally, his discussion of aesthetics takes place in a broader context of business management and productivity, while this current study situates itself within media studies and aesthetic philosphy, and its implications within how a certain class of communicative artefacts are considered beautiful. Still, Pineiro's work provides valuable insights in terms of identifying the manifestations and rationales for an aesthetic experience of source code. Thus, we continue his research by highlighting the main adjectives in the lexical field of programmers' discourse.

Already mentioned in Peter Naur's analysis of the practice of programming, *clean* is the first adjective which stands out as a requirement when assessing the form taken by source code. Clean code, he says, is a reference to how easy it is for readers of code to build a coherent theory of the system both described and prescribed by this source code (Naur, 1985). This purpose of cleanliness is then completed by a description of the clean form

[61] `https://slashdot.org`

in an analysis of clean code appears a couple of decades later in the title of a series of best-selling trade manuals written by Robert C. Martin and published by Prentice Hall from 2009 to 2021, the full titles of which clearly enunciate their normative aim[62]. What exactly is cleanliness, in Martin's terms, is nonetheless defined by circumlocutions, relying on contributions from experts. After asking leading programmers what clean code means to them, he carries on in the volume by providing examples of *how* to achieve clean code, while only loosely defining what it is. In general, cleanliness is mostly a definition by negation: it states that something is clean if it is free from impurities, blemish, error, etc. An alternative to this definition which trade manuals such as *Clean Code* use consists in providing examples on how to move from bad, "dirty" code, to clean code through specific, practical guidelines regarding naming, spacing, class delimitation, etc.. Starting at a high-level, some hints can be glimpsed from Ward Cunningham's answer:

> *You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem. (Martin, 2008) (p.10)*

along with Grady Brooch's:

> *Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control. (Martin, 2008) (p.11)*

Cleanliness is tied to expressiveness: it is devoid of any extraneous syntactic and semantic symbols (e.g. it does one thing, and one thing well), in

[62] *Clean Code: A Handbook of Agile Software Craftsmanship*, *The Clean Coder: A Code Of Conduct For Professional Programmers*, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, *Clean Agile: Back to Basics*, *Clean Craftsmanship: Disciplines, Standards, and Ethics*.

order to let the problem at hand appear, with all its implications. Instead, the tool (i.e. programming languages) disappear at the syntactic level, to enable expressiveness at the semantic level.

Martin echoes Hunt when he advocates for such a definition of clean as lack of additional syntactic information:

> *Don't spoil a perfectly good program by over-embellishment and over-refinement. (Hunt & Thomas, 1999)*

This advice to programmers denotes a conception of clean that is not just about removing as much syntactic form as possible, but which also implies a balance. *Overembellishment* implies excess addition, while *over-refinement* implies, on the contrary, excess removal. This normative approach finds its echo in the numerous quotations of Antoine de Saint-Exupéry's comment on aircraft design[63]:

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. (de Saint-Exupéry, 1972)*[64]

This balance between too much and too little is found in another dichotomy stated by programmers: between simple and clever. Simplicity, argues Jeremy Gibbons, is not only a restraint on the quantity of syntactic tokens (as one could achieve by keeping names short, or aligning indentations), but also a semantic equilibrium at the level of abstracted ideas (Gibbons, 2012). The balance between breadth and depth regarding the task of the code, between the precision of a use-case and its generalization, and its leveraging of external—i.e. supposedly reliable—code is summed up his quoting of Ralph Waldo Emerson at the conclusion of his column:

[63] For instance, see `https://news.ycombinator.com/item?id=1640594` and `https://twitter.com/codewisdom/status/1353651398337044481`

[64] *In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away, when a body has been stripped down to its nakedness.*, translated by Lewis Galantière in the 1939 edition of *Wind, Sand and Stars*

```
  void SomeMethod(){
    if(x != y){
        //-- stuff
    }
}

void SomeClearerMethod(){
    if(x == y) return;
    //-- do stuff
}
```

Listing 20: Example of clarity differences between two methods.

> *We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes. (Gibbons, 2012)*

In another ACM publication, Kristiina Karvonen argues for simplicity not just as a design goal, as leveraged by human-computer interface designers, but as a term with a longer history within the tradition of aesthetic philosophy, especially the work of Johann Joachim Winckelmann (Karvonen, 2000). In particular, she stresses the difficulty "to create significant, that is beautiful works of art with simple means" (Karvonen, 2000). Here, her correlation between *significance* and *pleasant appearance* hints at the semantic role of simplicity, as a means to communicate ideas (i.e. to *signify*) to an audience.

Precisely, simplicity is correlated with clarity (of meaning); if the former refers mainly to the syntactic component (fewer tokens), it enables the non-obfuscated framing of the ideas at play. One example is given in 20 by Dave Bush in a post titled *15 Ways to Write Beautiful Code*.

Here, the strive for simplicity leads to removing the brackets, and flipping the boolean check in the if-statement to add a `return`. Even though

it is, strictly speaking, more characters than the brackets and newline (six characters compared to four), the program becomes cleaner, and thus clearer, he argues, by separating the two branching cases inherent to the use of conditional logic, under the form of an if-statement. In the second version, it is made clear that, if a condition *is*, the execution should stop, and any subsequent statement can entirely disregard the existence of the if-statement; in the first version, the condition that *is not* is entangled with code that should be executed, since the existence of the if-statement has to be kept in mind until the closing bracket (Bush, 2015).

As a corollary to clarity stands obfuscation. It is the act, either intentional or un-intentional, to complicate the understanding of what a program does by leading the reader astray through a combination of syntactic techniques, a process we've already seen in the works of the IOCCC above. In its most widely applied sense, obfuscation is used for practical production purposes: reducing the size of code, and preventing the leak of proprietary information regarding how a system behaves. For instance, the JavaScript source code in 21 is obfuscated through a process called *minification* into the source code in 22

This process of obfuscation has very clear, quantitative assessment criterias, such as the size of the source code file and cryptographic complexity (Pellet-Mary, 2020). Nonetheless. obfuscation can also be valued as a positive aesthetic standard, of which the IOCCC is the most institutionalized guarantor. These kinds of obfuscations, as Mateas and Montfort analyze, involve the playful exploration of the intertwinings of syntax and semantics, seeing how much one can bend the former without affecting the latter. These textual manipulations, they argue, possess an inherently literary quality:

> *Obfuscation and weird languages invite us to join programming contexts to the literary contexts that must obviously be considered when evaluating literary code. They also suggest that cod-*

```
import { ref, onMounted, reactive } from 'vue';

const msg = ref("")
const HOST = import.meta.env.DEV ? "http://localhost:3046" : ""
const syllabi = new Array<SyllabusType>()

let start = () => {
    window.location.href = '/cartridge.html'
}

onMounted(() => {
    fetch(`${HOST}/syllabi/`,
        {
            method: 'GET'
        })
        .then(res => {
            return res.json()
        })
        .then(data => {
            Object.assign(syllabi, JSON.parse(data))
            console.log(syllabi);
            if (syllabi.length == 0)
                msg.value = "No syllabi :("
            else
                msg.value = `There are ${syllabi.length} syllabi.`

        })
        .catch(err => {
            console.error(err)
            msg.value = "Network error :|"
        })
})
```

Listing 21: home.js (before minification)

```
import{_ as p,g as f,o as l,c as n,a as c,h as e,t as r,b as u,i as b,u as _,F as y
```

Listing 22: home.js (after minification)

> *ing can resist clarity and elegance to strive instead for complexity, can make the familiar unfamiliar, and can wrestle with the language in which it is written, just as much contemporary literature does. (Mateas & Montfort, 2005)*

Such literary connection can also be seen in Noël Arnaud's work *Poèmes Algol* (Arnaud, 1968), in which he uses the constructs of the language Algol 68 in order to evoke in the reader something different than what the program actually does (i.e. fail to execute anything meaningful). Here, obfuscation is can be considered a literary value, as opposed to other domains, such as the scientific or the architectural, where it is both considered exclusively negatively.

Another insight on simplicity and programming regarding the communication of ideas is hinted at by Richard P. Gabriel in his use of the concept of *compression* in both poetry and programming, there is a desire to increase the semantic charge (or significance, in Karvonen's terms) all the while reducing the syntactic load (or the quantitity of formal tokens). One of those extraneous loads is explanation, as pointed out online by user Mason Wheeler:

> *When it requires a lot of explanation like that, it's not "beautiful code," but "a clever hack." (how, 2013)*

This answer, posted on the software engineering *Stack Exchange* forum, in response to the question "How can you explain "beautiful code" to a non-programmer?" (how, 2013), not only highlights the need to be self-explanatory, but also points at a quality departing form simplicity—cleverness.

```
def is_unique(_list):
  return len(set(_list)) == len(_list)
```

Listing 23: Method to check for the uniqueness of array elements

Cleverness is often found, and sometimes derided, in examples of code written by hackers, since it unsettles this balance between precision and generality. Clever code would tend towards exploiting particularities of knowledge of the medium (the code) rather than the goal (the problem). Hillel Wayne presents the snippet of Python code in 23 as an example of bad clever code:

Here, the knowledge of how the `set()` function in Python behaves, is required in order to understand that the `is_unique()` function returns whether all the elements of the given list are unique. A programmer without familiarity with Python would be unable to do so without consulting the Python documentation (i.e. requiring extraneous explanation).

Hillel elaborates on the difference between "bad" clever code[65], which is essentiall read-only due to its idiosyncracy and reliance on tacit knowledge, and "good" clever code, and such distinction corroborates our previous observations regarding beautiful code as a means for expression of the problem domain. His example is that the problem of sorting the roughly 300 million U.S. american citizens by birthdate can be made considerably more efficient by cleverly considering that no U.S. american citizen is older than 120 years, whereby radically reducing the computation space.

Meanwhile, cleverness is a valued attribute in the context of hacker code, putting more emphasis on the technical solution than on the problem domain, as we've seen above. A salient is example was the 1994 `smr.c` entry to the IOCCC, which aimed at being the smallest self-reproducing

[65]See, for instance, Duff's device, an idiosyncratic and language-specific way to speed up loop unrolling in C. The author himself feels "a combination of pride and revulsion at this discovery" (Duff, 1983)

Listing 24: smr.c

program (Kanakarakis, 2022a). An exact reproduction of the source code can be found in 24

Consisting of a file weighing zero bytes, `smr.c` provides both a clever reduction of the problem domain, and a clever understanding of what C compilers would effectively accept or not as a valid program text (Kanakarakis, 2022b). Because it has since been banned under the rules of the IOCCC, this source code entirely renounces any claim to a more general application, and finds its aesthetic value only within a specific community.

Coming back to simplicity, we can define more precisely such a qualifier of source code to be *an exact fit to the problem*: without being too precise, or too generic, displaying an understanding of and a focus on the application domain, rather than the applied tools. William J. Mitchell sums it up in his introductory textbook for graphics programming:

> *Complex statements have a zen-like reverence for perfect simplicity of expression. (Mitchell, 1987)*

Programmers hold the idea of reaching a conceptual revelation through the reduction of complex styntactical assemblages. This strive towards attaining an inverse relationship between the complexity of an idea and the means to express it is contiguous to another related criteria for beautiful source code present in programmers' discourse: *elegance*. Leslie Valiant, recipient of the Turing Award in 2010, considers elegance as the explanatory power of simple principles, which might only appear *a posteriori*—a solution can only be qualified as elegant once it has been found, and very rarely during the process of its development(Anthes, 2011). Chad Perrin, in his article *ITLOG Import: Elegance*, first approaches the concept as a negation of the gratuitous, a means to reduce as much as possible the syntactic footprint while keeping the conceptual footprint intact:

> *In pursuing elegance, it is more important to be concise than merely brief. In a general sense, however, brevity of code does account for a decent quick and dirty measure of the potential elegance that can be eked out of a programming language, with length measured in number of distinct syntactic elements rather than the number of bytes of code: don't confuse the number of keystrokes in a variable assignment with the syntactic elements required to accomplish a variable assignment. (Perrin, 2006)*

Perrin also hints at the additional meaningfulness of elegance, as he compares it to other aesthetic properties, such as simplicity, complexity or symmetry. If simplicity inhabits a range between too specific and too general, he describes an elegant system as exactly appropriate for the task at hand, echoing others' definition of clean or simple source code. Elegance, he says, relies on strong, underlying principles, but is nonetheless subject to its manifestation through a particular, linguistic interface. While he touches at length on the influence of progamming languages in the possibility to write elegant source code, we will only address this question in Chapter 4.

Donald Knuth adds another component required to achieve elegance in software: along with leanness of code and the suitability of the language, he adds that elegance necessitates a clear definition of the problem domain (Fuller, 2008). Along with the appropriateness of the linguistic tooling, one can see here that the representation of the data which is then going to be processed by the executed source code also matters. Indeed, source code is not only about expressing dynamic processes, but also about translating the problem domain into formal static representations which will then be easy to operate on.

This aspect of implying underlying principles is also present in Bruce McLennan's discussion of the concept. As he approaches it through the dual lens of structural engineering, this indicates that he also considers

```
int factorial(int n)
{
  return n==0 ? 1 : n * factorial(n-1);
}
```

Listing 25: Use of recursivity in the computation of a factorial

elegance as a more profound concept which can manifest itself across disciplines, both as a way of making, and as a way of thinking (McLennan, 1997). He defines his *Elegance Principle* as:

> *Confine your attention to designs that* look *good because they* are *good. (McLennan, 1997)*

Such a definition relies heavily on the sensual component of elegance: while an underlying property of, at least, human activities, it must nonetheless be manifested in some perceptible way. On *Stackexchange*, user *asoundmove* corroborates this conception of achieving a simple and clean system where any subsequent modification would lead to a decrease in quality:

> *However to me beautiful code must not only be necessary, sufficient and self-explanatory, but it must also subjectively feel perfect & light. (sta)*

Once again connecting simplicity (under the guise of necessity and sufficiency), the perception of elegance is also related to a subjective feeling of adequacy, of fitting. Including some of the definitions of simplicity we've seen so far, Paul DiLascia, writing in the Microsoft Developer Network Magazine, illustrates his conception of elegance—as a combination of simplicity, efficiency and brilliance—with recursion (DiLascia, 2019), as seen in 25.

Recursion, or the technique of defining something in terms of itself, is a positively valued feature of programming (Abelson et al., 1979), which

we've seen an example of in 14. In so doing, it minimizes the number of elements at play and constrains the problem domain into a smaller set of moveable pieces. Another example, provided in the same *Stackexchange* discussion is the `quicksort` algorithm, which can be implemented recursively or iteratively, with the former being significantly shorter (see 26)

Concluding this survey of how programmers perceive, define and exemplify elegance in source code, we can follow Mahmoud Efatmaneshik and Michael J. Ryan who, in the IEEE Systems journal, offer a definition of elegance which relies both on a romantic perception—including subjective perception, "gracefulness", "appropriateness" and "usability"—and practical assessment with terms such as "simple", "neat", "parsimonious" or "efficient" (Efatmaneshnik & Ryan, 2019). In doing so, they ground source code aesthetics as a resolutely dualistic norm, between subjectivity and objectivity, qualitative and quantitative[66].

And yet, rather than subjectivity and objectivity being opposites, one could also consider them as contingent. Due to the interchangeability in the use of the some of the terms we've seen by programmers, both qualitative—in terms of the language used—and quantitative—in terms of the syntax/semantics ratio—assessments of source seem to be complementary in considering it elegant. If *clean*, *simple*, *elegant* seem to overlap, it is because they all seem to point at this maximization of meaning while appropriately minimizing syntax.

A complementary approach to understand what programmers mean when they talk about beautiful code is to look beyond the positive terms used to qualify it, and shift our attention to how other terms are used negatively. We have already touched upon qualifiers such as clever, or obfuscated, which have ambiguous statuses depending on the community that they're being used in—specifically hackers and literary artists. Further ex-

[66] A duality we will investigate further through the prism of human and machine understanding in Chapter 2

```
  public static void recursiveQsort(int[] arr,Integer start, Integer end) {
    if (end - start < 2) return; //stop clause
    int p = start + ((end-start)/2);
    p = partition(arr,p,start,end);
    recursiveQsort(arr, start, p);
    recursiveQsort(arr, p+1, end);
}

public static void iterativeQsort(int[] arr) {
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(0);
    stack.push(arr.length);
    while (!stack.isEmpty()) {
        int end = stack.pop();
        int start = stack.pop();
        if (end - start < 2) continue;
        int p = start + ((end-start)/2);
        p = partition(arr,p,start,end);

        stack.push(p+1);
        stack.push(end);

        stack.push(start);
        stack.push(p);
    }
}
```

Listing 26: Comparison two functions, one using recursion, the other one using iteration, https://stackoverflow.com/a/12553314/4665412

```
    openParen = (slash + asterix) / equals;
```

Listing 27: Choose variable names that masquerade as mathematical operators

```
for(j=0; j<array_len; j+ =8)
{
  total += array[j+0 ];
  total += array[j+1 ];
  total += array[j+2 ]; /* Main body of
  total += array[j+3]; * loop is unrolled
  total += array[j+4]; * for greater speed.
  total += array[j+5]; */
  total += array[j+6 ];
  total += array[j+7 ];
}
```

Listing 28: Code That Masquerades As Comments and Vice Versa

amination of negative qualifiers will enrich of understanding of what constitutes good code.

One of those hints comes from satirical accounts of how to write bad code. For instance, Green's post on *How To Write Unmaintainable Code* suggests new kinds of obfuscation, such as double-naming in 27 or semantic interactions in 28. The core ideas presented here revolve around creating as much friction to understanding as possible, by making it "as hard as possible for [the reader] to find the code he is looking for" and "as awkward as possible for [the reader] to safely ignore anything." (Green, 2006).

By looking at it from the opposite perspective of highly-confusing code, we see best how carefully chosen aesthetics, under the values of simplicity, clarity, cleanliness and elegance intend first and foremost to help alleviate

human cognitive friction and facilitate understanding of what the program is doing. The opposite amounts to playing misleading tricks.

Along with this mental property, programmers have another way to refer to code that does not meet aesthetic criteria, which is also relates primarily to a non-cognitive aspect of source code, by refferring to material properties.

Spaghetti code refers to a property of source code where the syntax is written in such a way that the order of reading and understanding is akin to disentangling a plate of spaghetti pasta. While still linear in execution, this linearity loses its cognitive benefits due to its extreme convolution, making it unclear what starts and ends where, both in the declaration and the execution of source code. Rather than using a synonym such as *convoluted*, the image evoked by spaghetti is particularly vivid on a sensual level, as a slimy, vaguely structured mass, even if the actual processes at play remain eminently formal (Steele, 1977). Such a material metaphor can also be in Foote and Yoder's description of code as a "big ball of mud":

> *A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. (Foote & Yoder, 1997)*

A broader approach to these sensual perceptions of code involve the reference to *code smells*. These smells are described by Martin Fowler as "surface indications that usually corresponds to a deeper problem in the system" (Fowler et al., 1999). They are aspects of source code which, by their syntax, might indicate deeper semantic problems, without being actual bugs. The name code smell evokes the fact that their recognition happens through intuition and experience of the programmer reading the

code, rather than through careful empirical analysis[67]. This points to a practice-based value system to evaluate the quality of source code, rather than to an evidence-based system, itself circling back to the qualifications of elegance discussed above, evaluated both as quantitative metric and as qualitative one.

In this section, we clarified the main terms used in programmers' discourse when discussing aesthetically pleasant code. Basing our interpretation of the gathered sources through discourse analysis, we specifically assumed a cooperative principle, in which all participants in the discourse intend to achieve writing the best source code possible. This analysis has confirmed and updated the findings of Piñeiro's earlier study. Across textbooks, blog posts, forums posts and trade books, the aesthetic properties of code are widely acknowledged and, to a certain extent, consistent in the adjectives used to qualify it (clean, elegant, simple, clear, but also clever, obscure, or dirty).

While there is a consistency in describing the means of beautiful code, by examining a lexical field with clear identifiers, this analysis also opens up additional pathways for inquiry. First, we see that there is a relationship between formal manifestations and cognitive burden, insofar as aesthetics help alleviate such a burden. Beautiful code makes the ideas which are embedded in it, and the world in which the code aims to translate and operate on, more accessible to the reader. Additionally, the negative adjectives mentioned when referring to the formal aspects of code (smelly, muddy, entangled) are eminently *materialistic*, indicating some interesting tension between the ideas of code, and the sensuality of its manifestation.

Moving beyond strict lexical tokens, we've seen in the breadth of responses in a programmer's question of "How can you explain "beautiful

[67] It should be noted that more recent computer science research has recently also focused on developing such empirical techniques (Rasool & Arshad, 2015), even though their practical usefulness is still debated (Santos et al., 2018)

code" to a non-programmer?" (how, 2013), programmers also rely multiple aesthetic domains to which they refer: from engineering and literature to architecture and mathematics. We now turn to an investigation of each of these domains, and of how they can be related to code.

2.3 Aesthetic domains

This examination of which qualifiers programmers use when they relate to the aesthetic qualities of source code (the way it looks) or the aesthetic experience that it elicits (the way they feel) has shown both a certain degree of coherence, and a certain degree of elusiveness. Subjectively, programmers associate their experience of encountering well-written code as an aesthetic one. However, on a normative level, things become a little more complicated to define: as we've seen in the previous section's discussion of forum exchanges, beauty in source code is not and explicited in and of itself, but is best illustrated by referring to to other domains.

The next step that we propose is to inquire into these specific domains, to examine in which capacity they are being summoned in relation to code, and how they help us further delineate the aesthetic qualities which belong to source code. The assumption here is that, following Neil Postman, a medium is a means of expression, and different mediums can support different qualities of expression (Postman, 1985). Since there seems to be some specific ways in which code can be considered beautiful, these contingent domains, and the specific parts of these domains which create this contingency, will prepare our work of defining source code-specific aesthetic standards.

To do so, then, we will look at the three domains most often conjured by programmers when they mention the sensual qualities of, or the aesthetic experiences elicited by, source code: literature, mathematics and architecture. While there are accounts of parallels between programming and painting (Graham, 2003) or programming and musicMcLean (2004), these refer rather to the painter or musician as an individual, rather to the specific medium, and there are, to the best of our knowledge, no account of code being like sculpture or film, for instance.

2.3.1 Literary Beauty

The most striking, and obvious similarity between code and another medium of expression is that of literature: perhaps because they both require, fundamentally, the use of alphanumeric characters laid out on a two-dimensional plane. Similarly, syntax and semantics interplay in order to convey meaning to a reader. *Code as literature*, then, focuses on this intertwining of natural language and computer language, on its narrative, rhetorical and informative properties, and even on its ability to mimick the traditional forms of poetry.

Code as a linguistic practice

In *Geek Sublime*, Vikram Chandra, novelist and programmer, lays out the deep parallels he sees between code and human language, specifically sanskrit. While stopping short of claiming that code is literature, he nonetheless makes the claim that sanskrit is, as a set of generative linguistic rules to compose meaning, a distant ancestor to computer code (Chandra, 2014), a fact corroborated by Agathe Keller in her studies of the Āryabha□a (Keller, 2021). Sanskrit, like computer code, relies on context-free rules and exhibits similar properties as in code, such as recursion and inheritance.

With a similar syntactic structure between sanskrit and code, the former also exhibits a "*search for clear, unambiguous understanding*" through careful study, a goal equally shared by the writers of source code, as we've seen above. Specifically, the complexity of the linguistic system presented both in sanskrit and in machine language implies that enjoyment of works in either medium happens not through spontaneous, subjective appreciation, but through "conoisseurship", resulting from education, experience and temperament (Chandra, 2014).

Similarly, in *Words Made Flesh: Code and Cultural Imagination*, Florian Cramer touches upon code's ability to *do* things, in order to inscribe it differently in a historical development of linguistics, connecting it to the

symbolical works of the kabbalah and Lebniz's *Ars Combinatoria*. Code, according to Cramer, is linguistic, not just because it is made up of words, but because it *acts upon* words, influencing what we consider literature and human-language writing:

> *The step from writing to action is no longer metaphorical, as it would be with a semantic text such as a political speech or a manifesto. It is concrete and physical because the very code is thought to materially contain its own activation; as permutations, recursions or viral infections. (Cramer, 2003)*

Those permutations and recursions are used in the different ways: natural language writers have attempted to apply formulas, or algorithms, to their works, from the Oulipo's *Poèmes Algol* to Cornelia Sollfrank's *Net.Art Generator*. The properties that Cramer identifies in machine languages, tensions between totality and fragmentation, rationalization and occultism, hardware and software, syntax and semantics, artificial and natural, are ascribed to the newest development of the interaction between program and expression, for instance through the shape of those combinatorial poetics (Cramer, 2003). This resemblance, or *Familienähnlichkeit*, to other forms of linguistic expression, is explored further by Katherine Hayles' work on speech, writing and code. Specifically, she sees the linguistic practices of humans and intelligence machines as influencing and interpenetrating each other, considering code as language's partner[68].

Specifically, Hayles looks at how both literature and code can be expressive in a syntagmatic manner as well as in a paradigmatic manner. In the former, the meaning spread across the words of a sentence is considered fixed in literature, while it is dynamically generated in source code, depending of the execution state and the problem domain. In the latter,

[68] Hayles's work on posthumanism should be acknowledged here, as she envisions the human brain as a platform on which code runs, both natural codes such as sanskrit, or more computational forms of codes.

the meaning across synonyms in a (program) text is always potential in literature, but always present in code (Hayles, 2004)—highlighting different levels of interpretation. If code is a form of linguistic system, then it is a dynamic one in which the semantic charge is at least as volatile as in literature, but which possesses an additional *dimension*, as orality, literacy and digitality succeed each other by bringing the specificity of their media.

Code is thus considered a linguistic system in the technical sense, having a syntactic ruleset operating on words, it seems to also be a linguistic system in the cultural sense. As such, it deals with the occult, the magical and the obscure, but also exhibits a desire to communicate and execute unambiguous meaning.

This desire for explicit communication led literacy scholars to investigate source code's relationship to rhetoric. While digital systems seem to exhibit persuasive means of their own (Bogost, 2007) (Frasca, 2013), the code that underpins them also presents rhetorical affordances. The work of Kevin Brock and Annette Vee in this domain has shown that source code isn't just a normative discourse to the machine, but also an argumentative one with respect to the audience: it tries to persuade fellow programmers of what it is doing. From points being made in large-scale software such as Mozilla's Firefox web browser, to more specific styles in job interviews, source code presents worldviews in its own specific syntax (Brock, 2019).

The connections of code to linguistics happens thus at the technical, media, and cultural levels, insofar as it can allow for the expression of ideas and arguments, straddling the line between the rational and the evocative. We now turn more specifically to two instances of program code being considered a literary text, by leading programmers in the field: Yukihiro 'Matz' Matsumoto and Donald Knuth.

Code as text

Perhaps the most famous reference to code as a literary object is to be found in Donald Knuth's *literate programming*. In his eponymous 1984 article in *The Computer Journal*, Knuth advocates for a practice of programming in which a tight coupling of documentation with source code can allow one to consider programs as "works of literature" (Knuth, 1984). It is unclear, however, what Knuth entails when he refers to a work of literature[69].

Literate programming, a direct response to *structured programming*, enables the weaving of natural language blocks with machine language blocks, in order to be able to comile a single source into either a typeset documentation of the program, using the TeX engine[70], or into a source file for a Pascal compiler. The literary, here, is only a new set of tools and practices of writing which result in a *publishable work*, rather than a *literary work*, in which the program is described in natural language, with source code being interspersed as snippets throughout. While this approach fits aptly within Knuth's interest in typesetting and workflows of scientific publications, it does not explicitly address the relationship between literature and programming beyond this.

Still, his aim remains to support a clear understanding of a program by its reader, particularly emphasizing the complexity of such tasks. If he proposes something with regards to literature, it is in the process of meaning-making through reading, and its cognitive implications:

> *This feature of WEB is perhaps its greatest asset; it makes a WEB-written program much more readable than the same program written purely in PASCAL, even if the latter program is well commented. [...] a programmer can now view a large program as a*

[69] For instance, he refers in the rest of the article as "constructing" programs, rather than "writing" them.

[70] In which this current thesis is written.

> *web, to be explored in a psychologically correct order is perhaps the greatest lesson I have learned from my recent experiences. (Knuth, 1984)*

For Knuth, then, code is a text: both in the traditional, publisher-friendly way, but also in a new, non-linear way. This attention to the materiality of the program—layout, typesetting—foresees subsequent technological solutions to allow natural language and machine language to coexist[71].

This new way to approach (digital, hybrid) texts is expressed by Yukihiro Matsumoto, the creator of the Ruby programming language, in his notion of *Code as an Essay* (Oram & Wilson, 2007). While he doesn't deal directly with questions of eloquence and rhetoric, as opposed to Brock and Vee, it does however start from the premise that code is a kind of text, insofar as it has an a message being conveyed in a written form to an audience. However, it is not a kind of text which has a specific author, or a specific finite state:

> *Most programs are not write-once. They are reworked and rewritten again and again in their lived. Bugs must be debugged. Changing requirements and the need for increased functionality mean the program itself may be modified on an ongoing basis. During this process, human beings must be able to read and understand the original code.*

This conception, in which a text remains open to being modified further by subsequent voices, thus minimizing the aura of the original version, and possibly diluting the intent of the original author, echoes the distinction made by Roland Barthes between a *text lisible* (readerly text) and *texte scriptible* (writerly text). While the former aligns with classical conceptions of literature, with a clear author and life span for the literary work, the latter remains open to subsequent, subjective appropriations.

[71] See JavaDocs, Go docs, Jupyter Notebooks

This appropriation is such that a modified program text does not result in a finite program text either; due to its very low barrier to modification and diffusion, program texts can act almost as a dialogue between two programmers. As Jesse Li puts it, building the linguistic theory of Volonishov and Bakhtin:

> *The malware author is in dialogue with the malware analyst. The software engineer is in dialogue with their teammates. The user of a piece of software is in dialogue with its creator. A web application is in dialogue with the language and framework it is written in, and its structure is mediated by the characteristics of TCP/IP and HTTP. And in the physical act of writing code, we are in dialogue with our computer and development environment. (Li, 2020)*

It is to support this act of dialogue, supported by code's affordance of rapid modification and redistribution, that Matusmoto highlights simplicity, brevity—his term for elegance— and balance—insofar as no one dimension alone is enough—as means to achieve writing beautiful code. His last criteria, lightness, applies not to the code being written, but to the language being used to write such code, adding one more dimension to the dialogue: between the writer(s), the reader(s) and the language designer(s), an additional aspect we will return to in chapter 3.

These two examples show us that source code can be considered a text, if not a literary work which needs to accomodate a hybrid of natural and machine languages, new modes of diffusion, and countless possibilities for being rewritten. In this technological environment of programming languages (from WEB to Ruby), the aim is to facilitate the understanding of what the program does, and of what it should do.

Beyond these theoretical and functional conceptions of code's textuality, a last approach to the literariness of source code can be found in the works of code poetry, in which this ambiguity is embraced.

Code poetry

Daniel Temkin, in his *Sentences of Code Art*[72], suggests the ways in which code art (encompassing code poetry, esoteric languages and obfuscated code, among others) touches on code's linguistic features mentioned by Chandra and Cramer, while coming at it from a non-functional perspective, radically opposed to Knuth and Matsumoto.

> *The ambiguity of human language is present in code, which never fully escapes its status as human writing, even when machine-generated. We bring to code our excesses of language, and an ambiguity of semantics, as discerned by the human reader. (Temkin, 12/28/2017 12:00:00 PM)*

The artists whose main medium is source code explore the possibilities of meaning-making through mechanisms usually associated with poetry. For instance, code art is focused on the evokative possibilities of machine languages, and away from its exactness. This is a step further in a direction of semantic possibilities hinted at by Richard P. Gabriel when he mentions the parallels between writing code and writing poetry. In an interview with Janice J. Jeiss, he states:

> *I'm thinking about things like simplicity – how easy is it going to be for someone to look at it later? How well is it fulfilling the overall design that I have in mind? How well does it fit into the architecture? If I were writing a very long poem with many parts, I would be thinking, "Okay, how does this piece fit in with the other pieces? How is it part of the bigger picture?". When coding, I'm doing similar things, and if you look at the source code of extremely talented programmers, there's beauty in it. There's a lot of attention to compression, using the underlying programming language*

[72] A direct reference to Sol Lewitt's *Sentences on Conceptual Art*.

```
print"a"x++$...$"x$.,$,=_;redo
```

Listing 29: All The Names of God, Nick Montfort, 2010, source

> *in a way that's easy to penetrate. Yes, writing code and writing poetry are similar. (Jeiss, 2002)*

Further exploring the semantic possibilities of considering source code as a possible medium for poetic expression, one can turn to the analyses of code poems in publications such as Ishaac Bertram's edited volume, *code {poems}* and Nick Montfort's collected poems in *#!*.

In the former's foreword, Jamie Allen develops further this ability to express oneself via machine languages, considering that programmers can have "*passionate conversations in Python*" or "*with a line in a text file [...] speak directly to function, material action, and agency*" (Bertram, 2012). This is done, not by relying on the computer as a generative device, but by harnessing from the form and subject matter of those very machine languages which subsequently can exhibit those generative properties. Focusing on the language part of the machine allows for an interplay between human and machine meanings.

Still, machine semantics are nonetheless considered as an essential device in writing code poetry, and exploring concepts that are not easily grasped in natural languages—e.g. hoisting or destructuring assignment. Additionally, the contrast between the source representation of the poem and its execution can add to the poetic tension, as we see in Nick Montfort's *All The Names of God* (2010) (source in 29, and output in 30).

This poem is the object of close literary critical examination by Maria Aquilina, who notes that *[t]he contrast between the economical minimalism of the program and the ordered but infinite series of letter combinations it produces is one of the aspects that make the poem striking* (Aquilina, 2015). Building on philosophy and literary theorists, Aquilina situates the expres-

```
_atk_atl_atm_atn_ato_atp_atq_atr_ats_att_atu_atv_atw_atx
↪  _aty_atz_aua_aub_auc_aud_aue_auf_aug_auh_aui_auj_auk
↪  _aul_aum_aun_auo_aup_auq_aur_aus_aut_auu_auv_auw_aux
↪  _auy_auz_ava_avb_avc_avd_ave_avf_avg_avh_avi_avj_avk
↪  _avl_avm_avn_avo_avp_avq_avr_avs_avt_avu_avv_avw_avx
↪  _avy_avz_awa_awb_awc_awd_awe_awf_awg_awh_awi_awj_awk
↪  _awl_awm_awn_awo_awp_awq_awr_aws_awt_awu_awv_aww_awx
↪  _awy_awz_axa_axb_axc_axd_axe_axf_axg_axh_axi_axj_axk
↪  _axl_axm_axn_axo_axp_axq_axr_axs_axt_axu_axv_axw_axx
↪  _axy_axz_aya_ayb_ayc_ayd_aye_ayf_ayg_ayh_ayi_ayj_ayk
↪  _ayl_aym_ayn_ayo_ayp_ayq_ayr_ays_ayt_ayu_ayv_ayw_ayx
↪  _ayy_ayz_aza_azb_azc_azd_aze_azf_azg_azh_azi_azj_azk
↪  _azl_azm_azn_azo_azp_azq_azr_azs_azt_azu_azv_azw_azx
↪  _azy_azz_baa_bab_bac_bad_bae_baf_bag_bah_bai_baj_bak
↪  _bal_bam_ban_bao_bap_baq_bar_bas_bat_bau_bav_baw_bax
↪  _bay_baz_bba_bbb_bbc_bbd_bbe_bbf_bbg_bbh_bbi_bbj_bbk
↪  _bbl_bbm_bbn_bbo_bbp_bbq_bbr_bbs_bbt_bbu_bbv_bbw_bbx
↪  _bby_bbz_bca_bcb_bcc_bcd_bce_bcf_bcg_bch_bci_bcj_bck
↪  _bcl_bcm_bcn_bco_bcp_bcq_bcr_bcs_bct_bcu_bcv_bcw_bcx
↪  _bcy_bcz_bda_bdb_bdc_bdd_bde_bdf_bdg_bdh_bdi_bdj_bdk
↪  _bdl_bdm_bdn_bdo_bdp_bdq_bdr_bds_bdt_bdu_bdv_bdw_bdx
↪  _bdy_bdz_bea_beb_bec_bed_bee_bef_beg_beh_bei_bej_bek
↪  _bel_bem_ben_beo_bep_beq_ber_bes_bet_beu_bev_bew_bex
↪  _bey_bez_bfa_bfb_bfc_bfd_bfe_bff_bfg_bfh_bfi_bfj_bfk
↪  _bfl_bfm_bfn_bfo_bfp_bfq_bfr_bfs_bft_bfu_bfv_bfw_bfx
↪  _bfy_bfz_bga_bgb_bgc_bgd_bge_bgf_bgg_bgh_bgi_bgj_bgk
↪  _bgl_bgm_bgn_bgo_bgp_bgq_bgr_bgs_bgt_bgu_bgv_bgw_bgx
↪  _bgy_bgz_bha_bhb_bhc_bhd_bhe_bhf_bhg_bhh_bhi_bhj_bhk
↪  _bhl_bhm_bhn_bho_bhp_bhq_bhr_bhs_bht_bhu_bhv_bhw_bhx
↪  _bhy_bhz_bia_bib_bic_bid_bie_bif_big_bih_bii_bij_bik
↪  _bil_bim_bin_bio_bip_biq_bir_bis_bit_biu_biv_biw_bix
↪  _biy_biz_bja_bjb_bjc_bjd_bje_bjf_bjg_bjh_bji_bjj_bjk
↪  _bjl_bjm_bjn_bjo_bjp_
```

Listing 30: All The Names of God, Nick Montfort, 2010, Selected output

sive power of the poem in its engagement with the concept of *eventualization*, locating the semantic load of the poem in its existence both in a human-perception of the non-human (e.g. computer time) and the dialogue between source, output and title (Aquilina, 2015).

Not only is there an aesthetic of minimalism present in the source (to which Perl lends itself perfectly), the output also represents the *depth* (in Hayles's sense) of the medium of writing. As we've seen above, a literary conception of source code aesthetic can, under certain circumstances, find its quality not just the limit in characters, but in the relationship between such a limit and the expansiveness of the ideas expressed.

From software developers to artists, different kinds of writers seem to equate code as a text, bringing forth multiple reasons to justify such a connection. Beyond the fact that source code is made up of textual characters, we see that these conceptions of code as literature are multiple''. On the one side, it is focused on its need to communicate explicit concepts related to its function (Knuth, Matsumoto, Brock), all the while embracing the semantic ambiguity which exists in the use of natural language tokens, backed-up by the potential executable semantics enabled by its machine nature (Cramer, Hayles, Montfort, Temkin). The last example, the tension between *All The Names of God*'s source and output, between notation and ideas, suggests us to look at this kind of elegance not through a literary lens, but through a scientific one—another domain conjured by code writers and readers.

2.3.2 Scientific beauty

Having strong roots in scientific thought and practice, via the connection of computer science, the aesthetic experiences of source code are also related to the scientific domain. Nonetheless, it seems to exist in two distinct ways: whether code is beautiful in a similar way that mathematics

is, or wether code is beautiful according to equivalent principles at play in engineering.

Mathematics

A recurring point in programmers' discussions of beauty in programming is oftentimes the duality of the object of discussion: is one talking about an algorithm, or about a particular implementation of an algorithm? While this thesis is concerned with the latter, we now turn to how this relationship between algorithm and implementation presents a similar tension as the relationship between theorem and proof in mathematics.

On the one side, there exists few discussions of a direct relation between code and beauty from a mathematical perspective beyond Edsger Dijkstra's discussion of the implementation of programming languages. In it, he starts from computer science's strong origin in mathematics (e.g. lambda calculus), to show that this relation exists in part through, again, the concept of *elegance*. Theorems and subroutines are compared as being similar essential building blocks in the construction of a correct system. Correctness as the ultimate aim of both mathematics and programming takes place, he writes, by the use of a limited, efficient amount of those building blocks, resulting in a set of small, general and systemic concepts, in an elegant structure (Dijkstra, 1963).

Gian-Carlo Rota, in his investigation into mathematical beauty, distinguishes between mathematical beauty, a property which in turn triggers an aesthetic experience, and mathematical elegance, the concrete implementation thereof.

> *Although one cannot strive for mathematical beauty, one can achieve elegance in the presentation of mathematics. In preparing to deliver a mathematics lecture, mathematicians often choose to stress elegance and succeed in recasting the material in a fashion that everyone will agree is elegant. Mathematical el-*

> *egance has to do with the presentation of mathematics, and only tangentially does it relate to its content. (Rota, 1997)*

This separation between the beauty of a mathematical concept (theorem) and its presentation (proof) is reflected in the separation between algorithm and computer program, as McAllister notes. According to him, the beauty of source code is considered closer to the beauty in mathematical proofs, and as such abides by norms of exactness (over approximation) and transparency (over cumbersoneness) (McAllister, 2005).

Specifically, mathematical proofs are supposed to fulfill the requirement of what McAllister calls *graspability*, that is, the tendency for a proof to have the theorem it depends on grasped in a single act of mental apprehension, which, in turn, provides genuine understanding of the reasons for the truths of the theorem. When seen as a form a mathematical beauty, code is therefore praised in being to convey its function through concrete syntax; and linking aesthetic satisfaction with an *economy of thought*.

The first to employ such an expression, the mathematician Henri Poincaré describes the rigor of a mathematical process as subsequently obtained by combining this economy of thought, a form of cognitive elegance, with the concept of *harmony* (Poincaré, 1908). By virtue of mathematics being based on formal languages, this linguistic component introduces a certain kind of structure, and the complexity of the problem domain is made more harmonious by the reliance on such an invariant structure (i.e. the syntax of the formal language used). Source code as mathematics can thus be seen as a cognitive structure, which the elements, based on formal linguistics, can exhibit elegant aspects in their communication of a broader concept.

Engineering

As we've seen in our discussion of the relationship between computer science and programming as a relationship between the abstract and the con-

crete, one can see in these two activities a parallel in mathematics and engineering. Engineering is, like programming, the concrete implementation backed by deliberate and careful planning, often with the help of formal notations, of a solution to a given problem. Mathematics, from this perspective, can be considered as one of the languages of engineering, among sketches, diagrams, techniques, tools, etc.

One of the central concepts in the practice of mathematics, elegance, can also be found, along with its connection to source code, in engineering. Bruce McLennan examines such a connection from a more holistic angle than that of a single act of mental apprehension, when looking at a proof. He suggests that,

> *Since aesthetic judgment is a highly integrative cognitive process, combining perception of subtle relationships with conscious and unconscious intellectual and emotional interpretation, it can be used to guide the design process by forming an overall assessment of the myriad interactions in a complex software system. (Schummer et al., 2009)*

His point is that software is too complex to be easily verified, and that tools to help us do so are still limited. This complexity sets our intuition adrift and analytical resources do not help. In order to handle this, he proposes to shift the attention from an analytical to phenomenological one, from the details to the general impression. Engineering, like mathematics, ultimately aim at being correct. While the latter can rely on succint formal propositions and representations to achieve this purpose, engineering composes too many moving parts of different nature. The specificity lies in the nature of software engineering's materials:

> *All arts have their formal and material characteristics, but software engineering is exceptional in the degree to which formal considerations dominate material ones. (Schummer et al., 2009)*

And yet, the development of his arguments remains on the phenomenological side, distant from the standards of mathematic abstraction. In engineering, he argues, the design looks unbalanced if the forces are unbalanced, and the design looks stable if it is stable. By restricting our attention to designs in which the interaction of features is manifest—in which good interactions look good, and bad interactions look bad—we can let our aesthetic sense guide our design, relying on concepts of efficiency, economy and elegance (McLennan, 1997).

The sciences, and specifically mathematics and engineering, have their own set of aesthetics standards, to which source code seems to be connected to, even though they might not always align. Still, the idea of elegance remains central to both mathematical and engineering approaches, as it measures the number and conciseness of the theory's basic principles, or of the structure's basic components, while keeping the need for an overall effect, whether as enlightenment (Rota) for mathematics, in which larger implications are gained from a particular way a proof of a theorem is presented, or as an ecompassing *gestalt* impression in engineering, in which a program that looks correct, would most likely be correct.

Another concept touched upon by both approaches is that of structure: mathematics deal with formal structures to represent and frame the complexity of the world, while engineering deal with concrete structures offered as solutions to a specific problem. The last of the main domains to which source code is referred to is at the intersection of concrete and abstract structures, of function and form—architecture.

2.3.3 Architectural beauty

Formal organization

Software architecture emerged as a consequence of the structured revolution (Dijkstra, 1972), which was concerned more with the higher-level or-

ganization of code in order to ensure the quality of the software produced. Such an assurance was suggested by Djikstra in two ways: by ensuring the provability of programs in a rigorously mathematic approach, and by ensuring that programs remained as readable as possible for the programmers. Structure has therefore been an essential component of the intelligibility of software since the 1970s. It's only in the late 1990s that software architecture as a discipline has been recognized as such, stemming form a bottom-up approach of recognizing that some ways in which code is organized is better than others.

Top-down software architecture

Today,

> *software architectural models are intended to describe the structure and behavior of a system in terms of computational entities, their interactions and its composition patterns, so to reason about systems at more abstract level, disregarding implementation details. (Garland, 2000)*

When Mary Shaw and David Garland publish their 1996 book *Software architecture : perspectives on an emerging discipline*, they represent the beginning of a trend of adoption of so-called architectural practices within the field of software development. This relation of the former with the latter has its origin in the structured programming shift in the late 1960s: the idea was to bring in a more normative approach to writing code, in the hope that this structure would support correctness and efficiency. Building on this need for structure, software architecture has thus developed into an approach to software patterns, modelling and documentation, through the overall processes, communications, inputs and outputs of a system can be formally described and verified.

This connection between the two fields is therefore established through the need for reliability in the structures being built, once programmers

realized that the artefacts resulting from their work—software, in the form of source code—can exhibit certain structural properties, in a similar way that a surface of a particular material exhibits particular structural properties. Concepts such as modularity, spatial organization or interdependence, it turns out, could be applied to both fields.

Good source code, from a software architecture perspective, is code which is clearly organized, respecting a blueprint which, not being source code *per se* as written in machine language and allowing for dynamic behaviour, is nonetheless formalized (see, for instance the Universal Modelling Language). These domain-specific architectures are then related to a given problem-domain (a university, a hospital, many-to-many secure communications, etc.). As Robin K. Hill mentions in her essay on software elegance:

> Brevity by itself can't be enough; the C loop control `while( i++ < 10)` may be terse, excelling in brevity, but its elegance is debatable. I would call it, in the architectural sense, brutalism. Architecture provides nice analogues because it also strives to construct artifacts that meet specifications under material constraints, prizing especially those artifacts that manifest beauty as well. (Hill, 2016)

Another parallel between the two fields can be found in Eric Raymond's *The Cathedral and the Bazaar*. This essay describes the Linux project, the open-source philosophy it propelled into the limelight, and how the quantity of self-motivated workers without rigid working structures (which is not to say without clear designs) can result in better work than if made by a few, select, highly-skilled individuals (Raymond, 2001). While the cathedral is traditionally considered more aesthetically pleasing than the bazaar, in terms of architectural canon, Raymond sides with a bazaar-like model of organization, in which all development is done in public, with a very loose, horizontal contribution structure at any stage of the software lifecycle—as

opposed to a tightly guided software project whose development is done by a restricted number of developers. While he doesn't mention specific aesthetic standards in his essay, he does highlight parallels in practices and processes, laying foundations on which to build such standards. Architecture is thus both a model for the planning of the construction of artefacts, and a model for the organization of the persons constructing these artefacts.

Simon Yuill, in the volume edited by Olga Goriunova and Alexei Shulgin, develop a parallel between code art and the brutalist style of architecture. A style characterized by its foregrounding of the raw materials constituting the building, Brutalism foregoes decoration or ornament to focus on direct utility. Yuill, building on the *HAKMEM* document circulated at MIT's computer science department in 1972, equates this approach to a coding close to the "bare metal" of the computer, using the Assembly language. Contrary to higher-level languages such as C or Java, Assembly engages directly with the intricacies of specific machines, and underlines the fundamental necessity of the hardware and the need to acknowledge such a primacy. Beyond this materiality, he also equates other architectural values such as modularity present in the work of architects such as Le Corbusier or Kunio Mayekawa, as well as in programs such as the UNIX operating system (Yuill, 2004). What we see here is yet another reference from software to architecture, focusing this time on the reality of hardware, and on some theoretical principles similar in postwar Western architecture.

Finally, the trade litterature has also been contributing to this overlap between between architecture and software as formal, high-level organization of inter-connected components. For instance, Robert Martin, in the influential *Clean Code* mentions that the standards of software architecture are based on the *5S* japanese workplace organization method, namely:

- seiri (□□) - naming and sorting all components used
- seiton (□□) - placing things where they belong

- seisō (□□) - cleanliness
- seiketsu (□□) - standardization and consistency in use
- shitsuke (□) - self-discipline

We recognize here a focus on efficiency, organization and *proper* use, along with the requirement of cleanliness of the tools, workbench and workplace, as a virtue of a good organization. While originally applied to manufacture, Martin makes the case that this can also apply to the knowledge economy—as in our case, with correct naming, correct placement, correct appearance and correct use.

Vernacular constructions

However, considering architecture as a strictly organizational practice does not show the whole picture. As Hill's quote above hints at, there is another side to architecture, concerned with details rather than with plans, feeling rather than rationalizing.

This is particularly salient in the development of software patterns, and their reference to a particular branch of architectural theory: Christopher Alexander's *A Pattern Language*.

A Pattern Language kickstarted a whole field of research based around this idea of distinct, self-contained but nevertheless composable components. In Alexandrian terms, they are a triad, *which expresses a relation between a certain context, a problem, and a solution.*. Similarly to architectural patterns, these emerged in a bottom-up fashion: individual software developers found that particular ways of writing and organizing code were in fact extensible and reusable solutions to common problems which could be formalized and shared with others.

Besides the theoretical similarities between software and architecture mentioned above, it is the lack of learning from practical successes and failures in the field which prompted interest in Alexander's work, along

with the development of Object-Oriented Programming, first through the Smalltalk language, then with C++, [73]. The similarity between a pattern and an object, and their promise of using them which would lead to better results on multiple dimensions, made it very attractive to software developers. Writing in *Patterns of Software* (with a foreword by Alexander), Richard P. Gabriel illustrates that point:

> *The promise of object-oriented programming—and of programming languages themselves—has yet to be fulfilled. That promise is to make plain to computers and to other programmers the communication of the computational intentions of a programmer or a team of programmers, throughout the long and change-plagued life of the program. The failure of programming languages to do this is the result of a variety of failures of some of us as researchers and the rest of us as practitioners to take seriously the needs of people in programming rather than the needs of the computer and the compiler writer. (Gabriel, 1998)*

Throughout his work, Gabriel weaves parallels between his experience as a software developer and as a poetry writer, drawing concepts from the latter field into the former, and inspecting it through the lens of pattern languages. Two concepts in particular are worth examining a bit further: *compression* and *habitability*.

Compression, in narrative and poetic text, is the process through which a word is given additional meaning through the rest of the sentence. In a sentence such as *"Last night I dreamt I went to Manderley again."*[74], the reader is unlikely to be familiar with the exact meaning of *Manderley*, since this is the first sentence of the novel. However, we can infer some of the properties of Manderley from the rest of the sentence: it is most likely a place, and it most likely had something to do with the narrator's past,

[73] Today most of the programming languages allow for some object-oriented paradigm
[74] From Daphne DuMaurier's *Rebecca*.

since it is being returned to. A similar phenomenon happens in source code, in which the meaning of a particular expression or statement can be derived from itself, or from a larger context. In object-oriented programming, the process of inheritance across classes allows for the meaning of a particular subclass to be mostly defined in terms of the fields and methods if its subclasses—its meaning is compressed by relying on a semantic environment, which might or not be immediately visible. This, Gabriel says, induces a tension between extendability (to create a new subclass, one must only extend the parent, and only add the differentiating aspects) and context-awareness (one has to keep in mind the whole chain of properties in order to know exactly what the definition of an interface that is being extended really is). Resolving such a tension, by including enough information to hint at the context, while not over-reaching into verbosity, is a thin line of being self-explanatory without being verbose.

Finally, Gabriel, writing in 1998, mentions that compression isn't so much a problem in poetry since, ultimately, the definitions of each words aren't quite limited to the poet's own mind but exist as well in the broad conceptual structures which readers hold. However, since all aspects of a program is always by definition explicitly defined, programmers thus have the ultimate say on the definition of most of the data and functions described in code. Compression doesn't work as well because the reader cannot assume anything that is being mentioned in the code (and defined elsewhere), without risking the (error-raising) consequence of being wrong.

His particular assumption that others will want to modify and extend source code is one that is influenced by his background as a commercial developer. Other pieces of code might just be satisfying in being read or deciphered (as we've seen in source code poetry) but this assumption of interaction with the code brings in another concept, that of *habitability*. In his terms, it is

> *the characteristic of source code that enables programmers,*

> *coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently. (Gabriel, 1998)*

In a sense, then, beautiful code is also code that is clear enough to inform action and, well-organized enough to warrant actually taking that action. It relates to Alexander's property of *comfort*, by affording involvement instead of estrangement: one is at ease when making changes to a program text, rather than fearing unknown consequences due to a lack of a clear mental model.

Dual aesthetics

Architecture, when referenced by software, includes at least two distinct approaches: one tending to the objective, and one tending to the subjective. This distinction, operated by Roger Scruton, equates these to a tendency to, respectively, emotionality or rationality (Scruton, Sun, 04/21/2013 - 12:00). This pair is reflected in how software also refers to architecture: as top-down planning or as bottom-up construction, both holding different, but overlapping aesthetic standards. One the one side, we have cleanliness, orderliness and appropriateness, following interpersonal conventions and style; on the other side, we have familiarity, compression and appropriateness of community patterns.

At its most common denominator, architecture is then concerned with the gross structure of a system. At its best, architecture can support the understanding of a system by addressing the same problem as cognitive mapping does: simplifying our ability to grasp large system.This phrase appears in Kevin Lynch's work on *The Image of the City*, in which he highlighted that our understanding of an urban environment relies on combinations of patterns (node, edge, area, limit, landmark) to which personal, imagined identities are ascribed. The process is once again that of abstraction, but goes beyond that. In parallel, Garland notes that, in its most ef-

fective cases, software architecture can expose the high-level constraints on the design of a system, as well as *the rationale for making specific architectural choices*. The rationale, the deeper meaning of a program, seems to matter as much as the formal description of its system.

Architecture is indeed a field that exists at the intersection of multiple other fields: engineering, art, design, physics and politics. As the organization of space, one can project it onto non-physical spaces, such as software, and the way that it takes shape within the medium of source code will be more thoroughly explored in Chapter XXX. As such, it provides another peek into the relationship between function and form, and how it is mediated by the materials in which a certain structure is built, whether it is a physical structure, or a mental structure which only exists in a written form.

Buildings, like most things, are more than their surface, consisting of much that is not revealed. Particularly, their sensuous surfaces sometimes tend to obscure their main function[75]. Carlson and Parsons, in their discussion of Functional Beauty, arguee that there is an ethical problem in this mystification; while art mystifies, assuming us to suspend our disbelief to enter into an artwork's appreciation, the things that are functional can have a much bigger and much worse consequence when they mystify, potentially rendering the whole construction useless, and its value null.

In summary, we've seen that the specific topic of the aesthetics of the aesthetics of software refers to three main, different domains. By referring to code as text, its linguistic nature is highlighted, along with its problematic relationship to natural languages—problematic insofar as its ambiguity can play against its desire to be clear and understood, or can play in favor of poetic undertones. The standards expressed here touch upon the specific tokens used to write programming, and thus involve the programming languages in the process of aesthetic judgement.

[75]Think of a building housing a stock-exchange disguided as a Greek temple.

Also considering the formal nature of source code, scientific references equate source code as having the potential to exhibit similar properties as mathematical proofs and theorems, in which the elegance of the proof isn't a tight coupling with the theorem to be proved, but in which an elegant proof can (and, according to some, should) enlighten the reader to deeper truths. Conversely, these scientific references also include engineering, in which the applicability, its correctness and efficiency are of prime importance: the conception of elegance, accompanied by economy and efficiency, becomes a more holistic one, tending to the general feeling of the structure at hand, rather than to its specific formalisms.

These references to engineering then lead us to the last of the domains: architecture. Presented as both relevant from a top-down perspective (with formal modelling languages and descriptions, among others) or from a bottom-up (including software patterns and familiarity and appropriateness within a given context). These similarities between software in architecture, both in planning, in process and in outlook, touch upon another subject: the place of formal and informal knowledges in the construction, maintenance and transmission of those (software) structures.

All of these three domains of references have in common the concept of elegance, and the concept of understanding, we will turn to the relationship between code and understanding next. However, we conclude this chapter by a last reference domain which, while not tied to a particular field, is a way of doing with which programmers have long associated themselves with. The next section thus examines the relationship between craftsmanship and knowledge in writing and reading source code.

2.4 Craft, knowledge and beauty

In their introduction to the field of sofware architecture, Shaw and Garland summon the need to formalise the practice as the practice moves from craft to engineering (Shaw & Garlan, 1996). Already mentioned by Dijkstra, Knuth and other early software practitioners, the conception of programming as a craft has become more and more popular amongst source code writers and readers[76]. For instance, Paul Graham, LISP programmer, co-founder of the Y Combinator startup accelerator and widely-read blogger[77], highlights the status of programming languages as a medium and craft as a way to approach it, in his essay *Hackers and Painters* (Graham, 2003). Particularly, he stresses the materiality of code, depicting hackers and craftsmen as people who:

> *are trying to write interesting software, and for whom computers are just a medium of expression, as concrete is for architects or paint for painters.*

So, while links between craftsmanship and programming have existed as self-proclaimed ones by programmers themselves, as well as by academics and writers (Sennett, 2009; Chandra, 2014), 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* (de Certeau et al., 1990), 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.

[76] See, for instance, Joel Spolsky (Spolosky, 2003; Seibel, 2009)

[77] And even achieving god-like status among certain circles (Eadicicco, 2014)

2.4.1 Craftsmen and architects

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? By examining formal and informal texts, we focus on the fact that software developers ground their practice in passion, know-how and myths. Particularly, 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.

Starting from programmers' self-identification with craftstmanship, we analyze how those references by programmers enter into a productive dialogue with our historical and cultural conception of craftsmanship. First, the focus is set on the historical unfoldings of craftsmanship and software development, showing parallels. After inquiring into the modes of organization and the economical development of both craftsmanship and software development, this section focuses on the particular comparison with building, and its involvement with knowledge management and aesthetic judgments. Building on discussions of beautiful craft and beautiful code, the focus is not on the standards which allow practicioners to ascribe specific beauty to a specific piece of software; instead, we start a discussion into the materiality of code not in terms of bits, bytes and languages, but rather as a conceptual one which is to be worked with and worked through.

Craftsmanship through the ages

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 (Thompson, 1934). 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 (Zhang et al., 2015) and Japan (Jordan & Weston, 2003); however, these lie beyond the immediate scope of this article. Following Sennett, we will use his definition of craftsmanship as *hand-held, tool-based, intrinsically-motivated work which produces functional artefacts, and in the process of which is held the possibility for unique mistakes* (Sennett, 2009).

Late Middle-Ages craftsmanship stands out as such for a couple of reasons: their socio-economic organization, and their relationship to knowledge. First and foremost, craftspeople were indissociable from the guilds they belonged to (Black, 1984). 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 (Thompson, 1956).

A key 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 (Bassett et al., 2008), 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* (Cennini, 2012). Further work, at the eve of the Industrial Revolution, continued on this intent to formalize the practice of craftsmanship (Pannabecker, 1994). In this sense, quality work is rooted in implicit knowledge: a good craftsman knows a quality work when they see it (Sennett, 2009).

Another component of craftsmanship is its alleged incompatibility with manufacture (Ruskin, 1920; Sturt, 1963). However, studies have shown that the craftsmanship, rather than standing at the strict opposite of the industrial (Jones, 2016), has been integrated into the process of modern industrialization. The practice of the craftsman, then, integrates into the design and operation of machines and industrial-scale organizations, informing ways of making in our contemporary world (Gordon, 1988; McGee, 1999).

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 of craft.

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) (Ensmenger, 2012). 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) (Raymond, 2001), emphasis was put on engagement with tools, inquiring into peers' work (Levy, 2010) and later formalized into bottom-up archives[78]. 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 (Seibel, 2009; Raymond, 2001).

As computer science solidified as a distinct field in the 1960s (Tedre, 2006), there was a process of formalizing the hitherto *ad hoc* techniques of programming computers. As a response to the myth of carefully handmade code[79] and unconstrained approaches to writing code came the structured programming approach, initally proposed by E. W. Djisktra (Dijkstra, 1972). 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, then through the Web. Inquiries into the relationship of craftsmanship with programming started to take place in the mid-1970s from an educational perspective (Dijkstra, 1982), from an organizational perspective (Brooks & Jr, 1975) and an inter-personal perspective (Weinberg, 1998), and culminated with the publication of several trade books (Martin, 2008; Hendrickson & McBreen, 2002), 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 (Hoover & Oshineye, 2009; Goodliffe, 2007).

Comparisons of software development with craftsmanship are abundant, and relate then mostly to the relationship between unstructured

[78]The most famous of which is the Jargon File, later to be published as the The New Hacker's Dictionary: `http://www.catb.org/jargon/html/`

[79]See The Story of Mel, A Real Programmer, a folktale of early programmers: `https://www.cs.utah.edu/~elb/folklore/mel.html`

practice and formalized theory; as such, it is used to self-categorize programmers as skilled makers rather than passive thinkers[80].

Craft in architecture

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 (Pevsner, 1942), 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 self-motivated workers without rigid working structures (which is not to say without clear designs) can result in better work than if made by a few, select, highly-skilled individuals (Raymond, 2001; Henningsen & Larsen, 2020).

What we see, then, is a similar result: individuals can cooperate on a

[80] See code monkey, as a derogative term for such passive, detached practice: `http://www.techopedia.com/definition/31469/code-monkey`

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 (Brooks & Jr, 1975). The further sections will investigate how such a phenomena of building complex structures through horizontal networks is possible, from both epistemological and aesthetic perspectives.

2.4.2 Knowledge acquisition and production

The problem of knowledge in software development can be examplified by the "bus factor"[81]. It describes the risk of crucial information being lost due to the disappearance or incapacity of one of the programmers of the project, and aims at the problem of *essential complexity*[82]. 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, 1976), the gap between design and implementation—the domain of the craftsman—often relies on tacit knowledge (Collins, 2010).

Explicit knowledge, in programming as in most disciplines, is carried through books, academic programs and, more recently, web-based content that is either structured (e.g. MOOCs, Codeacademy, Khan Academy) or unstructured (e.g. blog posts, forums, IRC channels), but both seem to be insufficient to reach an expert level (Davies, 1993). As demonstrated by a popular comic[83], 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—, the learning process must include implicit knowledge.

[81] `https://en.wikipedia.org/w/index.php?title=Bus_factor`
[82] See the *No Silver Bullet* essay in *The Mythical Man-Month*, op.cit.
[83] See Randall Munroe, `https://xkcd.com/844/`

Apprentices and masters

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 imitating 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. Sometimes, this relationship to a master is implemented explicitly through practices such as pair programming (Williams & Kessler, 2003) or corporate mentorship programmings (IBM's Master programmer initiative). Other times, it is re-interpretated through fictional accounts designed to impart wisdom on the readers, and taking inspiration from Taoism and Zen (James, 1987; Raymond & Steele, 1996). From higher-level programming wisdom featuring 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[84]. 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 various machines provides a counterpoint to the argument of software as craftsmanship in terms of knowledge transmission. Traditionally, since craftsmanship has been understood as that which is done by hand, and since craftsmen were working with unique artefacts (i.e. no artefact can be perfectly copied), copying someone else's realization was physically inconceivable. The realm of soft-

[84] See, for instance: `http://rubykoans.com/`

ware, on the opposite, relies heavily on the technical affordance of code to be duplicated, uploaded, downloaded and executed on multiple platforms through source code files (Manovich, 2001). 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[85]), and on a vernacular level (as enabled by Web 2.0 platforms such as StackOverflow and GitHub). 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 indeed blindly copy external code, or write their own, inspired by the external code[86].

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 (Man, 2015), an approach equally present in Japanese crafts histor (Jordan & Weston, 2003). Here again, manually copying from established quality work to seize their elusive essence is an essential aspect to craftsmanship.

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, a sort of *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* (Treude & Robillard, 2017). Code as such is freely and easily accessible as piecemeals, but often lacks the essential context.

[85] See: `https:gnu.org`

[86] See the discussions on `https://softwareengineering.stackexchange.com/questions/36978`

So while programmers are to acquire implicit knowledge through a process of learning by doing (realizing koans, coding small projects, re-using copied code), we now need to assess how much of it happens through observing. Implied in the apprentice-master relationship is that what is observed should be of *good quality*; one learns through ones own mistakes, and through ones presentation with exmaples of good work. Coming back to the relationship between architecture and software development, Christopher Alexander asks, in the preface of Richard P. Gabriel's *Patterns of Software* (Gabriel, 1998),

> *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?*

If a craftsman learns their trade by comparing their work with work of a higher quality (either their master's, or publicly available works, assembled as a canon (Taylor, 2001)), the programmer is faced with a different problem: a lot of examples, but a few good ones. Copyright stands in the way of pedagogical copying. With software becoming protectable under copyright laws in the 1980s (Oman, 2018), great works of programming craft became unacessible to programmers, despite the value they would bring in knowledge acquisition (Gabriel & Goldman, 2001). 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, which was circulated illegaly in classrooms for twenty years before its official publication was authorized by the copyright owners (Lions, 1996).

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, uploaded code snippets or by copyrighted protection on works, leading to a lack in an established canon of great works. Nonetheless, the other advice given to begin-

ner programmers—practice[87]—hints at another aspect: direct engagement with code.

2.4.3 Material aesthetics

At the heart of knowledge transmission and acquisition stands the *practice*, and inherent in the practice is the *good practice*, the one leading to a beautiful result. The existence of an aesthetic experience of code, and the adjectives used to qualify it (smelly, spaghetti, muddy), already pointed at in the previous section, already points at an appreciation of code beyond its formalisms and rationalisms. We now turn to the aesthetic experience of code within the broader context of the aesthetics of craftsmanship, highlighting code's specificity as a material.

The beauty of a thing well-made

A traditional perspective is that motor skills, with dexterity, care and experience, are an essential feature of a craftsman's ability to realize something beautiful (Osborne, 1977), along with self-assigned standards of quality (Pye, 2008; Sennett, 2009). These qualitative standards which, when pushed to their extreme, result in a craftsperson's *style*, are to be gained through practice and experience, rather than by explicit measurements (Pye, 2008) [88]. Two things are concerned here, supporting the final result: tools and materials (Pye, 2008). According to Pye, a craftsperson should have a deep, implicit knowledge of both, what they use to manipulate (chisels, hammers, ovens, etc.) as well as what they manipulate (stone, wood, steel, etc).

[87] `https://quora.com/What-are-some-of-the-best-ways-to-learn-programming`

[88] See Pye's account of craftsmanship, and his intent to make explicit the question of quality craftsmanship and *"answer factually rather than with a series of emotive noises such as protagonists of craftsmanship have too often made instead of answering it."*

This relationship to tools and materials is expected to have a relationship to *the hand*, and at first seems to exclude the keyboard-based practice of programming. But even within a world in which automated machines have replaced hand-held tools, Osborne writes:

> *In modern machine production judgement, experience, ingenuity, dexterity, artistry, skill are all concentrated in the programming before actual production starts. (Osborne, 1977)*

He opens here up a solution to the paradox of the hand-made and the computer-automated, as programming emerges from the latter as a new skill. This very rise of automation has been criticized for the rise of a Osborne's "soulless society" (Osborne, 1977), and has triggered debates about authorship, creativity and humanity at the cross-roads between artificial intelligence and artistic practice (Mazzone & Elgammal, 2019). One avenue out of this debate is human-machine cooperation, first envisioned by Licklider and proposed throughout the development of Human-Computer Interaction (Licklider, 1960; Grudin, 2016). If machines, more and more driven by computing systems, have replaced traditional craftsmanship's skills and dexterity, this replacement can nonetheless suggest programming as a distinctly 21st-century craftsmanship, as well as other forms of cratsmanship-based work in an information economy.

Code as material

Beautiful code, code well-written, is indeed an integral part of software craftsmanship (Oram & Wilson, 2007). More than just function for itself, code among programmers is held to certain standards which turn out to hold another relationship with traditional craftsmanship—specifically, form following function.

A craftsman's material consciousness is recognized by the anthropomorphic qualities ascribed by the craftsman to the material (Sennett,

2009), the personalities and qualities that are being ascribed to it beyond the immediate one it posseses. Clean code, elegant code, are indicators not just of the awareness of code as a raw material that should be worked on, but also of the necessities for code to exist in a social world. As software craftsmen assemble in loose hierarchies to construct software, the aesthetic standard is *the respect of others* (Abelson et al., 1979).

Another unique feature of software craftsmanship is its blending between tools and material: code, indeed, is both. This is, for instance, represented at its extreme by languages like LISP, in which functions and data are treated in the same way (McCarthy et al., 1965). In that sense, code is a material which can be almost seamlessly converted from information to information-*processing*, and vice-versa; code as a material is perhaps the only non-finite material that craftspeople can work with—along with words[89].

Code, from the perspective of craft, is not just an overarching, theoretical concept which can only be reckoned with in the abstract, but also the very material foundation from which the reality of software craftsmanship evolves. An analysis of computing phenomena, from software studies to platform studies, should therefore take into account the close relationship to their material that software developers can have. As Fred Brooks put it,

> *The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. (Brooks & Jr, 1975)*

So while there are arguments for developing a more rigorous, engineering conception of software development (Ensmenger, 2012), a crafts ethos based on a materiality of code holds some implications both for programmers and for society at large: engagement with code-as-material opens up

[89] Disregarding for now the very real impact of computing on the environment (Kurp, 2008)

possibilities for the acknowledgement of a different moral standard. As Pye puts it,

> *[...] the quality of the result is clear evidence of competence and assurance, and it is an ingredient of civilization to be continually faced with that evidence, even if it is taken for granted and unremarked. (Pye, 2008)*

Code well-done is a display of excellence, in a discipline in which excellence has not been made explicit. If most commentators on the history of craftsmanship, following Ruskin, lament the disappearance of a better, long-gone way of doing things, before computers came to automate everything, locating software as a contemporary iteration of the age-old ethos of craftsmanship nonetheless situates it in a longer tradition of intuitive, concrete creation.

We covered, in this first chapter, several aspects of an aesthetic conception of code. First, we've established who writes code: far from a homogeneous crowd which would reflect an abstraction conception of "code", code writers include inviduals which can be categorized as engineers, hackers, scientists or poets. While these categories do not have rigid boundaries and can easily overlap, they do allow us to establish more clearly the contexts and purposes within which code can be read and written. Hacker code and engineer code look different from each other, achieve different purposes than poetic code, abide by different requirements than scientific code, and so on. Within each of these conceptions, a judgment of what looks good will therefore be different. A conception of the aesthetics of code seems then, at first, to possess some degree of relativity.

Second, we've built on Erik Pineiro's work to complete a survey of the lexical fields that programmers use when they describe or refer to beauti-

ful code. In so doing, we've highlighted certain desired properties, such as clarity, cleanliness, and elegance—as opposed to, say, thrilling, moving, or delicate. This survey involved an analysis of textual instances of programmers' discourses: through blog posts, forum discussions, journal articles or textbooks, showing a steadiness in the expression of an aesthetic posture since the beginning of the trade. Additionally, the study of our negative terms pointed further to sensual metaphors of code, using parallels with smell and texture. As a a "big ball of mud", a "pile of spaghetti" or full of "smelly corners", ugly code is something where its sensual appearance prevents the reader or writer to grasp its true purpose—what it actually does.

While those terms are being recurrrently used to qualify aesthetically pleasing code, our survey has also noticed specific domains which programmers refer to, consciously or unconsciously, as they try to build up metaphors to communicate the nature of their aesthetic appreciation: by referring to science, literature and architecture. The summoning of each of these domains, sometimes simultaneously, does, however, only select specific parts in order to adapt it to the felt potential of source code. Literature brings linguistics, but not narrative; science brings formalism and engineering, but not numbers nor universality; architecture brings planning and construction, but not socio-economic purposes or end-usage. These domains are thus better understood as the different parts of a Venn diagram, as practictionners attempt to define what it means to do what they do well. This was confirmed by our investigation into the connections between craft and code, looking specifically at the knowledge transmission in both non-code and code craftsmanship, the place that informal knowledge holds in such transmission, and in the value judgment produced by the craftsperson as to whether a work is a good one.

The region of overlap of these different domains has to do, it turns out, with cognitive clarity. Whether wrangling with the linguistic tokens in literary exercises, structuring various pieces of code such that their organization is robust and communicated to others such that it allows for future

maintenance and expansion, or writing lines of code in a certain way in order to hint at some larger concepts and ideas beyond their immediate execution result, these domains are all mentioned in their ability to vehiculate ideas from one individual to another—as opposed to, say, elicit self-reflection or sublime physical pleasure. It seems that beautiful code is then both functional code and understandable code.

The next chapter explores this idea futher, building on the emerging development in aesthetics of integrating both function and cognition, two fundamental aspects of software, as a criteria for an aesthetic experience.

Chapter 3

Understanding source code

Aesthetics in source code are thus primarily related to cognitive load. In the previous chapter, we've highlighted a focus on understanding when it comes to aesthetic standards: whether obfuscating or illuminating, the process of acquiring a mental model of a given object is a key determinant in the value judgment as applied to source code. In this chapter, we focus on the reason for which such a cognitive load exists in the first place, before surveying the means—both linguistic and mechanistic—that programmers deploy in order to relieve such a load.

This is related to one of the essential features of software: it must be *functional*. As mentioned in our discussion of the differences between source code and software in the introduction, source code is the latent description of what the software will ultimately *do*. Similarly to sheet music, or to cooking recipes[1], they require to be put into action in order for their users (musicians and cooks, respectively) to assess their value. Therefore, buggy or dysfunctional software is going to be of less value than correct software (Hill, 2016), regardless of how aesthetically pleasing the source is.

The assessment of whether a piece of software functions correctly is

[1]Recipes are a recurring example taken to communicate the concept of an algorithm to non-experts (Zeller, 2020)

essentially an assessment of whether what the software does is what the software is supposed to do, which in turn entails knowing what it does, what it is supposed to do, and being able to tell whether these two things are aligned. Any value judgment regarding the aesthetics of the source code at hand would be subject to whether or not the software functions correctly, and such judgment is rendered moot if that software does not work.

After deciding on a benchmark to assess the functionality of the source code at hand (understanding what it should be doing), one must then determine the actual behavior of the source code at hand once it is executed (understanding what it is actually doing). This chapter examines what goes into understanding source code. The first part will lay out our definition of understanding, presenting it as a dual phenomenon, between formalism and contextualism. Starting with 20tĥ century epistemology, we will see that a dominantly rational, cognitivist perspective on the nature of understanding, as it has been hailed by theoretical computer science research, shows its limits when confronted with practice. Having highlighted this tension, we then turn to how understanding the phenomenon of computation specifically, both on an ontological level, and on a psychological level. The ontological approach will show some of the features of software give it the status on an "abstract artifact", making it a complex object to grasp; the psychological approach will show how such a comprehension takes place for a varity of programmers. Finally, we will conclude with the means that programmers deploy to grasp the concepts at play with software: starting from metaphors used by the general public, we will go down this ladder of abstraction in order to reach the technical apparatuses used in the development and inspection of source code.

The main questions that this chapter addresses are the following: given a certain nature of knowledge acquisition, what are some of the features of computers that make them hard to grasp, and what kind of techniques are deployed in order to address these hurdles and in order to understand what

the code is actually doing. This will have us investigate the relationship of knowing and doing, the nature of computation (what is software?) and its relationship to the world as it appears to us (how does modelling and abstraction translate a problem domain into software?), and the cognitive scaffoldings set up to facilitate that task.

3.1 Formal and contextual understandings

This section focuses on our definition of understanding—the process of acquiring a working knowledge of an object. Such definition relies on two main aspects: a formal, abstract understanding, and a more subjective, empirical one. We will see how the former had some traction in computer sciences circles, while the second gained traction in programming circles. To support those two approaches, we first trace back the genealogy of understanding in theoretical computer science, before outlining how concrete experience and situatedness outline an alternative tradition.

3.1.1 Between formal and informal

Theoretical foundations of formal understanding

To start this inquiry, we go back to the early 20th century in Cambridge, when the theoretical roots of modern computation were being laid by both philosophers of logic and mathematicians, such as Bertand Russell, Ludwig Wittgenstein, and Alan Turing, as they worked on the formalization of thinking.

Wittgenstein, in particular, bases his argumentation in his *Tractatus Logico-philosophicus* on the fact that much of the problems in philosophy are rather problems of understanding between philosophers—if one were to express oneself clearly, and to articulate one's through clear, unambiguous language, a common conclusion could be reached without much effort:

> *Most questions and propositions of the philosophers result from the fact that we do not understand the logic of our language. (Wittgenstein, 2010)*

Language and logic are, as we see here, closely connected. Articulated in separate points and sub-points, his work conjugates aphorisms with logical propositions depending on one another, developing from broader statements into more specific precisions. Wittgenstein hints at the intertwining of language as a form of logic, and as logic as a form of language. In this, he follows in the footsteps of Gottfried Leibniz's *Ars Combinatoria*, insofar Leibniz views reasoning and inter-subjective understanding as a formal problem. A universal, and universally-understandable language, called a *characteristica universalis* could resolve any misunderstanding issues. Quoted by Russell, Leibniz notes that:

> *If we had it [a characteristica universalis], we should be able to reason in metaphysics and morals in much the same way as in geometry and analysis... If controversies were to arise, there would be no more need of disputation between two philosophers than between two accountants [...] Let us calculate. (Russell, 1950)*

Centuries after Leibniz's declaration, Wittgenstein presents a coherent, articulated theory of meaning through the use of mathematical philosophy, and logic, and his work fits with that of Russell[2] and Frege[3]; even though these are different theories, they are part of a similar endeavour to find a basis od formal propositions through which one could establish truth-values. Such attempt was a direct influence in the work on mathematician Alan Turing—who studied at Cambridge and followed some of Wittgenstein's lectures—, as he developed his own formal system for solv-

[2] In his *Principia Mathematica*, he lays out a theory of logical expression

[3] The *Begriffschrift* similarly attempts to constitute a language in which all scientific statements could be evaluated (Korte, 2010), while *Über Sinn und Bedeutung* clarifies the semantic uncertainties between a specific sentence and how it means, or refers to a concept

ing complex, abstract mathematical problems, manifested as a symbolic machine (Turing, 1936).

The design of the Turing machine is a subsequent step engagement with the question of understanding in the philosophical sense, as well as in the practical sense—a formal proof to the *Entscheidungsproblem* solved mechanically. Indeed, it is a response to the questions of translation (of a problem) and of implementation (of a solution). This formal approach to instructing machines to operate on logic statements then prompted Turing to investigate the question of intelligence and comprehension in *Computing Machinery and Intelligence*. In it, he translates the hazy term of "thinking" machines into that of "conversing" machines, conversation being a practical human activity which involves listening, understanding and answering (i.e. input, process and output) (Turing, 2009). This conversational test, which has become a benchmark for machine intelligence, does rely on the need for a machine to *understand* what is being said. Throughout the article, however, Turing does not yet address the need for a purely formal approach of whether or not a problem can be translated, as Leibniz would have it, into atomistic symbols which would be provided as an input to a digital computer. Such a process of translation would rely on a formal approach, similar to that laid out in the *Tractatus Logico-philosophicus*, or on Frege's formal language described in the *Begriffschrift*. Following a cartesian approach, the idea in both authors is to break down a concept, or a proposition, into sub-propositions, in order to recursively[4] establish the truth of each of these sub-propositions, and then re-assembled to deduce the truth-value of the original proposition. While Turing focuses on the philosophical and moral arguments to the possibility for machines to think, he does address the issue of artificial intelligence.

With these sophisticated syntactic systems developed a certain approach to cognition, as Turing clearly establishes parallels between the digital computer and the human brain. We now turn to the form of these

[4]although it was not called as such at the time

systems, looking at how their form addresses the problem of clearly understanding and operating on mathematical and logical statements.

Logical calculus, as the integration of the symbol into relationships of many symbols formally takes place through two stylistic mechanisms, the *symbol* and the *list*. Each of the works by Frege, Russell and Wittgenstein quoted above are structured in terms of lists and sub-lists, representing the stylistic pendant to the epistemological approach of related, atomistic propositions and sub-propositions. A list, far from being an innate way of organizing information in humans, is a particular approach to language: extracting elements from their original, situated existence, and reconnecting ways in very rigorous, strictly-defined ways. As Jack Goody writes in *The Domestication of the Savage Mind*,

> *[List-making] [...] is an example of the kind of decontextualization that writing promotes, and one that gives the mind a special kind of lever on 'reality'. (Goody, 1977)*

As inventories, early textbooks, administrative documents as public mnemotechnique, the list is a way of taking symbols, pictorial language elements in order to re-assemble them to reconstitute the world, then re-assemble it from blocks, following an assumption that the world can always be decomposed into smaller, discreete and *conceptually coherent* units (i.e. symbols). The list, Goody continues, establishes clear-cut boundaries, they are simple, they are abstract and discontinuous.

Being based on some singular, symbolical entity, applying logical calculus to lists and their symbols, and doing so in a computing environment, becomes the next step in exploring these tools for thinking. Indeed, the engineering development of digital computers in post-war United States as described in 2.1.1, allowed for the putting into practice of these languages, in the budding field of artificial intelligence (AI).

Practical attempts at implementing formal understanding

This putting into practice took the form of subsequent programming languages, relying on a certain conception of human cognition—abstract, logical, as shown above.

IPL, the Information Processing Language, was created by Allen Newell, Cliff Shaw and Herbert A. Simon. The idea was to make programs understand and solve problems, through "the simulation of cognitive processes" (Newell et al., 1964). IPL achieves this with the symbol as its fundamental construct, which at the time was still largely mapped to physical addresses and cells in the computer's memory, and not yet decoupled from hardware.

A link between the ideas exposed in the writing of the mathematical logicians and the actual design and construction of electrical machines activating these ideas, IPL was originally designed to demonstrate the theorems of Russell's *Principia Mathematica*, along with a couple of early AI programs, such as the *Logic Theorist*, the *General Problem Solver*. More a proof of concept than a versatile language, IPL was then quickly replaced by LISP as the linguistic means to express intelligence in digital computers.

LISP (*LIst Processor*) was developed in 1956 was Joseph McCarthy[5]. The base structural elements of LISP are not symbols, but lists (of symbols, of lists, of nothing), and they themselves act as symbols (e.g. the empty list) (McCarthy, 1978). By manipulating those lists recursively—that it, processing something in terms of itself—Lisp shows a tendency for a formal system to separate itself from the problem domain. This is facilitated by its atomistic and relational structure: in order to solve what it has do, it evaluates each symbol and traverses a tree-structure in order to find a terminal symbol, without the explicit need to refer to an external table, for instance.

This sort of heuristic is quite similar to the approach suggested by Noam Chomsky in his *Syntactic Structures*, where he posits the tree structure of language, as a decomposition of sentences until the smallest con-

[5]McCarthy coined the phrase *Artificial Intelligence* during the 1956 Dartmouth workshop

ceptually coherent parts (e.g. Phrase -> Noun-Phrase + Verb-Phrase -> Article + Substantive + Verb-Phrase). The style is similar, insofar as it proposes a general ruleset (or the at least the existence of one) in order to construct complex structures through simple parts.

Through its direct manipulation of conceptual units upon which logic operations can be executed, LISP became the language of AI, an intelligence conceived first and foremost as logical understanding. The use of LISP as a research tool culminated in the *SHRDLU* program, a natural language understanding program built in 1968-1970 by Terry Winograd which aimed at tackling the issue of situatedness—AI can understand things abtractly through logical mathematics, but can it apply these rules within a given context? The program had the particularity of functioning with a "blocks world" a highly simplified version of a physical environment—bringing the primary qualities of abstraction into solid grasp. The computer system was expected to take into account the rest of the world and interact in natural language with a human, about this world (*Where is the red cube? Pick up the blue ball*, etc.). While incredibly impressive at the time, *SHDRLU*'s success was nonetheless relative. It could only succeed at giving barely acceptable results within highly symbolic environments, devoid of any noise. In 2004, Terry Winograd writes:

> *There are fundamental gulfs between the way that SHRDLU and its kin operate, and whatever it is that goes on in our brains. I don't think that current research has made much progress in crossing that gulf, and the relevant science may take decades or more to get to the point where the initial ambitions become realistic. (Nilsson, 2009)*

This attempt, since the beginning of the century, to enable thinking, clarify understanding and implement it in machines, had first hit an obstacle. The world, also known as the problem domain, exhibits a certain complexity which did not seem to be easily translated into singular, atom-

istic symbols. Around the same time, however, was developed another approach to formalizing the intricacies of cognition.

Warren McCullough's seminal paper, *A logical calculus of the ideas immanent in nervous activity*, co-written with Walter Pitts, offers an alternative based on the embodiment of cognition. They present a connection between the systematic, input-output procedures dear to cybernetics with the predicate logic writing style of Russell and others (McCulloch & Pitts, 1990). This attachment to input and output, to their existence in complex, inter-related ways, rather than self-contained propositions is, interestingly, rooted in his activy as a literary critic[6].

Going further in the processes of the brain, he indeed finds out, in another paper with Letvinn and Pitts (Lettvin et al., 1959), that the organs through which the world excites the brain *are themselves* agents of process, activating a series of probabilistic techniques, such as noise reduction and softmax, to provide a signal to the brain which isn't the untouched, unary, *symbolical* version of the signal input by the external stimuli, and nor does it seem to turn it into such.

We see here the development of a theory for a situated, embodied stance towards cognition, which would ultimately resurface through the rise of machine learning via convoluted neural networks in the 2000s (Nilsson, 2009). In it, the senses are as essential as the brain for an understanding—that is, for the acquisition, through translation, of a conceptual model which then enable deliberate and successful action. It seems, then, that there are other ways to know things than to rely on description through formal propositions.

A couple of decades later, Abelson and Sussman still note, in their in-

[6] Even at the Chicago Literary book club, he argues for a more sensuous approach to cognition: *"In the world of physics, if we are to have any knowledge of that world, there must be nervous impulses in our heads which happen only if the worlds excites our eyes, ears, nose or skin."* (McCulloch, 1953)

troductory textbook to computer science, the difficulty to convey meaning mechanically:

> *Understanding internal definitions well enough to be sure a program means what we intend it to mean requires a more elaborate model of the evaluation process than we have presented in this chapter. (Abelson et al., 1979)*

While formal notation is able to enable digital computation, it nonetheless proved to be limited when it came to accurately and expressively conveying meaning. This limitation, of being able to express formally what we understand intuitively (e.g. *what is a chair?*[7]) appeared as computers applications left the domain of logic and arithmetic, and were applied to more social domains.

After having seen the possibilities and limitations of making machines understand through the use of formal languages, and the shift offered by taking into account sensory perception as a possible locus of cognitive processes, we now turn to these ways of knowing that exist in humans in a more embodied capacity.

3.1.2 Knowing-what and knowing-how

1953 saw a radical posture change from one of the logicians whose work underpinned AI research, briefly before the start of these attempts to implement artificial intelligence in digital computers. This was the publication of Wittgenstein's *Philosophical Investigations*. In his second work, he disown his previous approach to language as seen in the *Tractatus Logico-philosophicus*, and favors a more contextual, use-centered frame. Rather than what knowledge is, he looks at how knowledge is acquired and used; while (formal) lanuage was previously defined as the exclusive means to translation concepts in clearly understandable terms, he broadens his perspective in the *Inquiries* by stating that language is *"the totality of language*

[7] A question addressed by Joseph Kosuth in his artwork *One and Three Chairs*, 1965

and the activities with which it is intertwined" and that *"the meaning of a word is its use within language"* (Wittgenstein, 2004), noting context and situatedness as a important factors in the understanding process.

At first, then, it seemed possible to make machines understand through the use of formal languages. The end of the first wave of AI development, as a branch of computation specifically focused on cognition, have shown some limits to this approach. We now turn to theories which support this approach of an embodied and contextualized knowing, complemented by a constructivist approach to building an understanding.

Knoweldge and situation

As hinted at by the studies of McCullough and Levitt, understanding a situation doesn't rely exclusively on abstract logical processes, but also on the processes involved in grasping this situation, such as, in their case, peripheral vision processing. It is not just what things are, but how they are, and how they are *perceived*, which matters. Different means of inscription and description do tend to have an impact on the ideas communicated and understood.

In his book *Making Sense: Cognition, Computing, Art and Embodiment*, Simon Penny refutes the so-called unversality of formulating cognition as a formal problem, and develops an alternative history of cognition, akin to Michel Foucault's archeology of knowledge. Drawing on the works of authors such as William James, Jakob von Uexküll and Gilbert Ryle, he refutes the Cartesian dualism thesis which acts as the foundation of AI research (Penny, 2019). A particular example of the fallacy of dualism, is the use of the phrase *implementation details*, which he recurringly finds in the AI literature, such as Herbert Simon's *The Sciences of the Artificial* (Simon, 1996). The phrase refers to the gap existing between the statement of an idea, of an algorithm, and a procedure, and its concrete, effective and functional manifestation.

```
recognition = false
do until recognition
wait until mousedown
    if no bounding box, initialize bounding box
    do until mouseup
    update image
    update bounding box
    rescale the material that's been added inside
    if we recognize the material:
        delete image from canvas
        add the appropriate iconic representation
        recognition = true
```

Listing 31: Example of pseudo-code attempting to reverse-engineer a software system, ignoring any of the actual implementation details, taken from (Nielsen, 2017)

For instance, pseudo-code is a way to sketch out an algorithmic procedure, which might be considered agnostic when it comes to implementation details. One can consider the pseudo-code in 31, which describes a procedure to recognize a free-hand drawing and transform it into a known, formalized glyph. Disregarding the implementation details means disregarding any reality of an actual system: the operating system (e.g. UNIX or MSDOS), the input mechanism (e.g. mouse, joystick, touch or stylus), the rendering procedure (e.g. raster or vector), or the programming language (e.g. JavaScript or Python).

Refuting the idea that pseudo-code is all that is necessary to communicate and act upon a concept, Penny argues on the contrary that information is relativistic and relational; relative to other pieces of information (intra-relation) and related to contents and forms of presenting this relation (extra-relation). Pseudo-code will only ever make sense in a particular

implementation context, which then affects the product.

He then follows Philip Agre's statement that a theory of cognition based on formal reason works only with objects of cognition whose attributes and relationships can be completely characterized in formal terms; and yet a formalist approach to cognition does not prove that such objects exist or, if they exist, that they can be useful. Uses of formal systems in artificial intelligence in specific, and in cognitive matters in general, is yet another instance of the map and the territory problem—it only goes so far.

Beyond the syntax of formal logic, there are different ways to transmit cognition in actionable form, depending on the form, the audience and the purpose. In terms of form, a symbol system of formal logic is only one of many possibilities for systems of forms. In his *Languages of Art*, Nelson Goodman elaborates a theory of symbol systems, which he defines as formal languages composed of syntactic and semantic rules (Goodman, 1976). Logical notation exists along with music, painting, poetry and prose. What follows, argues Goodman, is that all these formal languages involve an act of *reference*. Through different means (exemplification, denotation, resemblance, representation), formal systems act as sets of symbols which can denote or exemplify or refer to in more complex and indirect ways, yet always between a sender and a receiver[8].

Communication, as the transfer of meaning from one individual to one or more other individuals, does not exclusively rely on the use of mathematical based use of formal languages. From Goodman to Goody, the format of representation also affords differences in what can be thought and imagined. Something that was always implicit in the arts—that representation is a complex and ever-fleeting topic—is shown more recently in Marchand-Zañartu and Lauxerois's work on pictural representations made by philosophers, visual artists and novelists (such as Claude Simon's sketches for the structure of his novel *La Route des Flandres*, shown in 3.1)

[8] Understood as the eponymous entities in Jakobsen's model of communication functions.

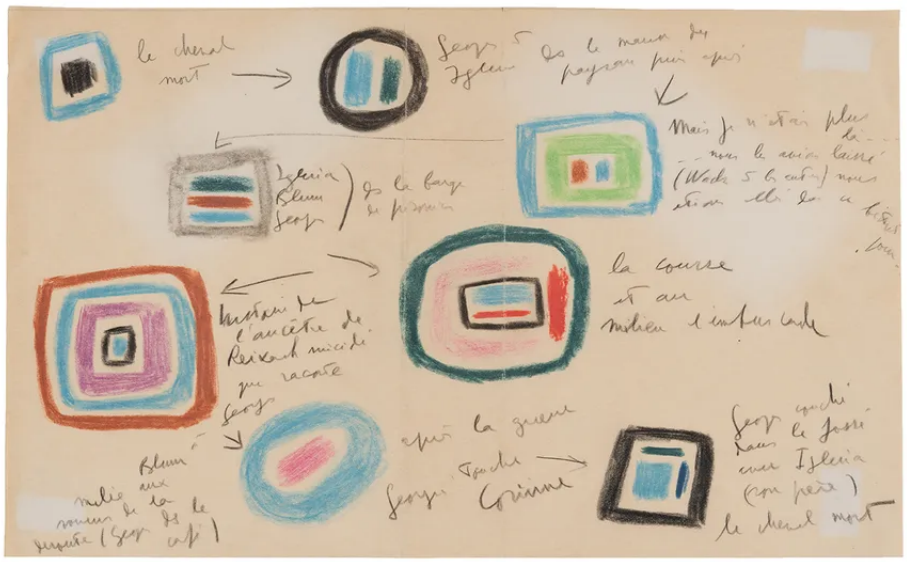


Figure 3.1: Tentative d'organisation visuelle pour le roman La Route des Flandres, années 1960 - Claude Simon, écrivain

(Marchand-Zañartu & Lauxerois, 2022). How specific domains, from mathematics to visual arts and construction, engage in the relation between form and cognition is further adressed in CHAPTER_4.

Going beyond logical notation, we have seen that there are other conceptions of knowledge which take into account the physical, social and linguistic context of an action. Nonetheless, these can still be systematized, as Goodman highlights in his investagation of referential systems, with a focus on reference of the ideas at hand.

Constructing knowledge

There are multiple ways to express an idea: on can use formal notation or draft a rough sketch with different colors. These all highlight different degrees of expression, but one particular way can be considered problematic in its ambition. Formal languages rely on the assumption, that all which

can be known can ultimately be expressed in unambiguous terms. First shown by Wittgenstein in the two main eras various eras of his work, we know focus on the ways of knowing which cannot be explicited.

First of all, there is a separation between *knowing-how* and *knowing-that*; the latter, propositional knowledge, does not cover the former, practical knowledge, as shown by Ryle (Ryle, 1951). Perhaps one of the most obvious example of this duality is in the failure of Leibniz to construct a calculating machine, as told by Matthew L. Jones in his book *Reckoning with Matter*. In it, he traces the history of philosophers to solve the problem of constructing a calculating machine, a problem which would ultimately be solved by Charles Babbage, with the consequences that we know (Jones, 2016).

Jones depicts Leibniz in his written correspondence with watchmaker Ollivier, in their fruitless attempt to construct Leibniz's design; the implementations details seem to elude the German philosopher as he refers to the "confused" knowledge of the nonetheless highly-skilled Parisian watchmaker. The (theoretical) plans of Leibniz do not match the (concrete) plans of Ollivier.

These are two complementary approaches to the knoweldge of something: to know *what* constructing calculating machine entails and knowing *how* to construct such a machne. In the fact that Ollivier could not communicate clearly to Leibniz what his technical difficulties, we can see an instance of something which would be theorized centuries later by Michael Polanyi as *tacit knowledge*, knowledge which cannot be entirely made explicit.

Polanyi, as a scientist himself, starts from another assumption: we know more than we can tell. In his eponymous work, he argues against a positivist approach to knowledge, in which empirical and factual deductions are sufficient to achieve satisfying epistemological work. What he proposes, derived from *gestalt* psychology, is to consider some knowledge of an object as the knowledge of an integrated set of particulars, of which

we already know some features, by virtue of the object existing in an external approach. This integrated set, in turn, displays more properties than the sum of its parts. While formal notation suggests that the combination of formal symbols does not result in additional knowledge, Polanyi rather argues, against Descartes, that relations and perceptions do result in additional knowledge.

> *The knowledge of a problem is, therefore, like the knowing of unspecifiables, a knowing of more than you can tell. (Polanyi & Grene, 1969)*

Rooted in psychology, and therefore in the assumption of the embodimed of the human mind, Polanyi posits that all thought is incarnate, that it lives by the body and by the favour of society, hence giving it a physio-social dimension. This confrontation with the real-world, rather than being a strict hurdle that has to be avoided or overcome, as in the case of SHRDLU above, becomes one of the two poles of cognitive action. Knowledge finds its roots and evaluation in concrete situations, as much as in abstract thinking. In the words of Cecil Wright Mills, writing about his practice as a social scientist research,

> *Thinking is a continuous struggle between conceptual order and empirical comprehensiveness. (Mills, 2000)*

Polanyi's presentation of a form of knowledge following the movement of a pendulum, between dismemberment and integration of concepts finds an echo in the sociological work of Mills: a knowledge of some objects in the world happens not exclusively through formal descriptions in logical symbol systems, but involves imagination and phenomenological experience—wondering and seeing. This reliance on vision—starting by recognizing shapes, as Polanyi states—directly implies the notion of aesthetic assessment, such as a judgement of typical or non-typical shapes. He does not, however, immediately elucidate how aesthetics support the

formation of mental models at the basis of understanding, only that this morphology is at the basis of higher order of represenations.

Seeing, though, is not passive seeing, simply noticing. It is an active engagement with what is being seen. Mills's quote above also contains this other aspect of Polanyi's investigation of knowledge, and already present in Ollivier's relation with Leibniz: knowing through doing.

This approach has been touched upon from a practical programmer's perspective in section 2.4.2, through a historical lens but it does also posses theoretical grounding. Specifically, Harry Collins offers a deconstruction of the Polanyi's notion by breaking it down into *relational*, *somatic* and *collective* tacit knowledges (Collins, 2010). While he lays out a strong approach to tacitness of knowledge (i.e. it cannot be communicated at all), his distinction between relational and somatic is useful here[9]. It is possible to think about knowledge as a social construct, acquired through social relations: learning the linguo of a particular technical domain, exchanging with peers at conferences, imitating an expert or explaining to a novice. Collective, unspoken agreements and implicit statements of folk wisdom, or implicit demonstrations of expert action are all means of communication through which knowledge gets replicated across subjects.

Concurrently, somatic tacit knowledge tackles the physiological perspective as already pointed out by Polanyi. Rather than knowledge that exists in one's interactions with others, somatic tacit knowledge exists within one's physical perceptions and actions. For instance, one might base one's typing of one's password strictly on one's muscle memory, without thinking about the actual letters being typed, through repetition of the task. Or one might be spotting a cache bug which simply requires a machine reboot, due to experience machine lifecycles, package updates, networking behaviour. Not completely distinct from its relational pendant, somatic knowledge is acquired through experience, repetition and

[9]His definition of collective tacit knowledge touches on the knowledge present in any living species and is impossible to ever be explicited, and is therefore out of scope here.

mimeomorphism—replicating actions and behaviours, or the instructions, often under the guidance of someone more experienced.

We started our discussion of understanding by defining it as the acquisition of the knowledge of a object—be it a concept, a situation, an individual or an artfefact„ which is accurate enough that it allows us to predict the behaviour and to interact with such object. Within this defintion, one could take the example of a human discussion as a demonstration of advanced understanding, as something that is both situated, and formalized.

Theories of how individuals acquire understanding (how they come to know things, and know conceptual representations of things), have been approached from an explicit perspective, and an implicit one. In the rationalist, logical philosophical tradition, we have seen that the belief that meaning can be rendered unambiguous through the use of specific notation. This has led to the development of logic and computer science, as this meaning got mechanized. Explicit understanding is therefore the theoretical lineage of computation.

However, as we've seen in the first hurdles of artificial intelligence research, explicit specification of meaning falls short of handling everyday tasks which humans would consider to be menial. This has led us to consider a more implicit approach to understanding, in which it is acquired by tacit means. Particularly, we've identified this tacit knowledge as relying on a social component, as well as on a somatic component.

Source code, as a formal system with a high dependence of context, intent and implementation, mobilizes both approaches to understanding. Before we dive deeper about how these two modes of understanding are mobilized at the end of this chapter, we now turn to what makes computation a cognitively complex object, and what are some cognitive reactions that humans display in situations where they have to understand,

and work with, software.

3.2 Understanding computation

In the previous chapter, we've laid out the groundwork by showing that there are multiple ways to understand something. We now turn to the thing we want to understand. What makes it challenging to understand computation?

First, we will inquire into some distinguishing features of software, by looking at some of its particularities, in terms of levels of abstractions, and in terms of types of complexity. This will highlight some of the theoretical properties that make it hard to understand, such as its relation to hardware, its relation to a specification, and its relation to time and space.

Then, we will shift our perspective to a more abstract point of view, investigating the ontological status of software. This will highlight how software exists as an *abstract artifact*, a term which encapsulates software's intrinsic and extrinsic tensions.

Finally, we will conclude this section by looking specifically at the source code component of software, and how programmers deploy strategies to understand it. Approaching it from a cognitive and psychological perspective, this will give us a better picture of the concrete process of understanding source code—a process which aesthetics are primarily intended to affect.

3.2.1 Software complexity

Software exhibits several particularities, which result in a compex whole—meaning that software possesses several independent components which interact with each other in non-trivial, and non-obvious ways. This subsection focuses on these particularies; that is, on the *properties* of software, and hence how these properties manifest themselves concretely, leading to those complex interactions. We will start by looking at the different levels at which software exists, before turning to the different kinds of complex-

ity which make software hard to grasp.

Levels of software

One of the essential aspects of software is that of *implementation*. Implementation is the realization of a plan, the concrete manifestation of an idea, and therefore hints at a first tension in software's multiple facets. It can happen through individuation, instantation, exemplification and reduction (Rapaport, 2005). On the one side, there is what we will call here *ideal* software, often existing only as a shared mental representation by humans (not limited to programmers), or as printed documentation, as a series of specifications, etc. On the other side, we have *actual* software, which is manifested into lines of code, written in one or more particular languages, and running with more or less bugs.

The relationship between the *ideal* and the *actual* versions of the same software is not straightforward. Ideal software only provides an intent, a guidance towards a goal, assuming, but not guaranteeing, that this goal will be reached. A popular engineering saying is that complements this approach by stating that:

> *In theory, there is no difference between theory and practice. In practice, there is.*[10]

Actual software, as most programmers know, differs greatly from its ideal version, largely due to the process of implementation, translating the purpose of the software from natural and diagrammatic languages, into programming languages, from what it should do, into what it actually does.

Writing on the myths of computer science, James Moor (Moor, 1978) allows us to think through this distinction between ideal and practical along the lines of the separation between a theory and a model. The difference between a model and a theory is that both can exist independently of one

[10] Sometimes mis-attributed to Richard P. Feynman or Albert Einstein, but traced to Benjamin Brewster, writing in the Yale Literary Magazine of 1882.

another—one can have a theory for a system without being able to model it, while one can also model a system using *ad hoc* programming techniques, instead of a coherent theory.

Most of the practice of programmers (writing and reading code for the purposes of creating, maintaining and learning software) depends on closing this gap between the ideal and the practical existences of software.

The third level at which software exists is that of hardware. While the ideal version of software is presented in natural language, diagrams or pseudo-code, and while the practical version of software exists as executable source code, software also exists at a very physical level—that of transistors and integrated circuits.

The distinction between software and hardware has been examined thoroughly (Kittler, 1997; Chun, 2008; Rapaport, 2005), but never strictly defined. Rather, the distinction between what is hardware and what is software is relative to where one draws the line: to a front-end web developer writing JavaScript, the browser, operating system and motherboard might all be considered hardware. For a RISC-V assembly programmer, only the specific CPU chip might be considered hardware, while the operating system being implemented in C, itself compiled through Assembly, would be considered software. A common definition of hardware, as the physical elements making up the computer system, overlooks the fact that software itself is, ultimately, physical changes in the electrical charge of the components of the computer.

Software can be characterized the dynamic evolution of logical processes, described as an ideal specification in natural languages, as a practical realization in programming languages, and in specific states of hardware components. Furthermore, the relations between each of these levels is not straightforward: the ideal and the practical can exist independently of each other, while the practical cannot exist independently of a machine[11].

[11]Even if that machine is a *virtual machine*, further complicating the boundary between

Types of complexity

Along with different levels of existence needed to be taken into account by the programmer, software also exhibits specific kinds of complexity. Our definition of complexity will be the one proposed by Warren Weaver. He defines problems of (organized) complexity as those which involve dealing simultaneously with a sizable number of factors which are interrelated into an organic whole (Weaver, 1948)[12]. Specifically, there are four different types of software complexity that we look at: conceptual complexity, modeling complexity, temporal complexity and spatial complexity.

Conceptual complexity, as refered to by Lando et. al. in their ontology of computer programs, addresses the necessity to model complex objects at different abstraction levels (Lando et al., 2007). As mentioned above, software exists at least on three somewhat distinct levels: the ideal, the practical, and the physical. This means that software, in its source code representation, should be able to provide the programmer with tools to engage with its intended function and its actual function at each of these levels. For instance, comments might relate to the ideal behaviour of the software, function declarations to its practical behaviour, while macro definitions can point to the specific hardware on which the code is run.

Additionally, conceptual complexity involves the distinction of the different elements of a computer program at the source code level, and keep track of their ontological status, sometimes independently from its level of existence. This distinction should be made notwithstanding the naming conventions assigned by different platforms, vendors, or programming languages, and rather based on ontological properties of the elements at hand. For instance, the distinction between *endurants* and *perdurants* by Lando et. al. focuses on the temporal dimension of software components (i.e. a data structure declaration has a different temporal property than

hardware and software

[12] As opposed to disorganized complexity, which are dealt with statistical tools.

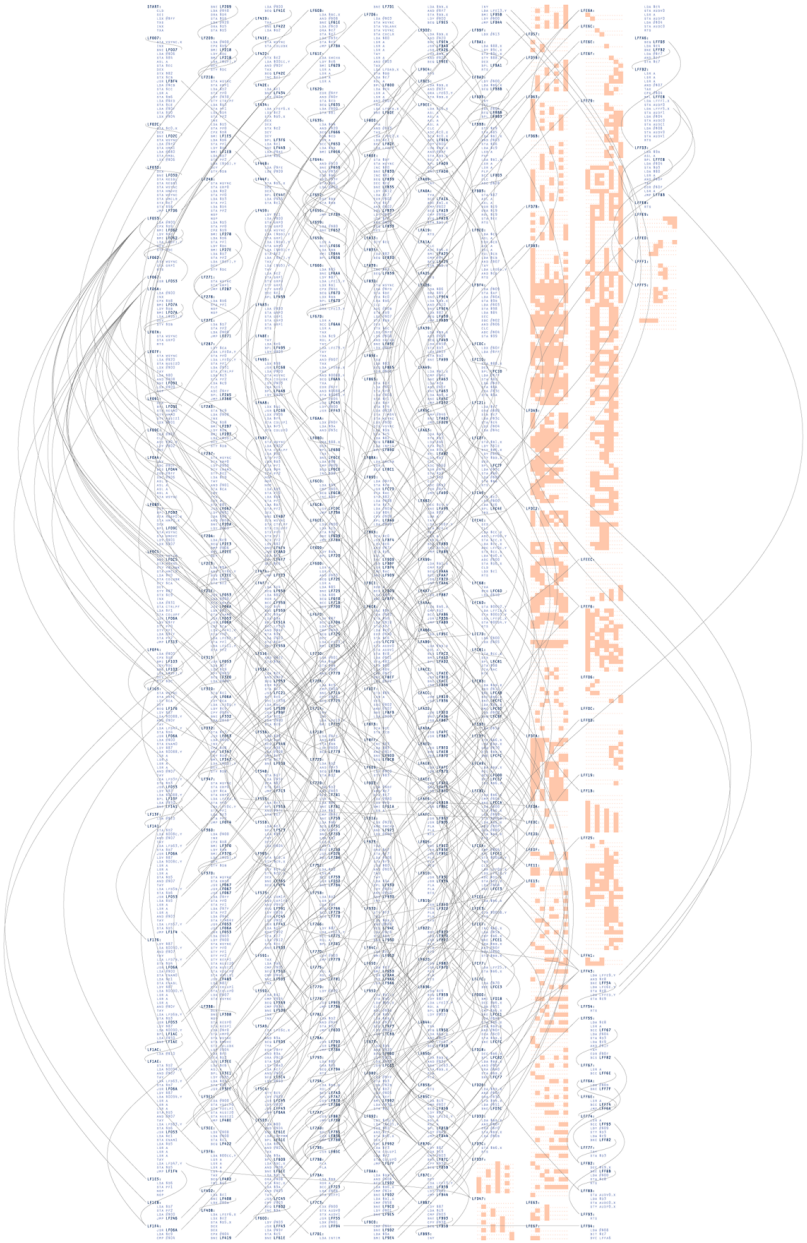
a function call). Here we have a second, complementary frame to that of the levels of existence, through which we can analyze source code; being somewhat related but an exact mapping of one another.

Modeling complexity addresses the hurdles in translating a non-discrete, non-logical object, event, or action, into a discreete, logical software description through source code. Indeed, the history of software development is also the history of the extension of the application of software, and the hurdles to be overcome in the process. From translation of natural languages (Poibeau, 2017), to education (Watters, 2021) or psychological treatment (Weizenbaum, 1976), it seems that problems that seem somehat straightforward from a human perspective become more intricate once the time for implementation has come.

We've seen so far the complexities tied to the conceptualization and modelization of the components of software. Two other significant contributions to the cognitive load of understanding software happen are more closely related to the concrete execution of the software—temporal and spatial complexity.

Temporal complexity relates to the discrepancy between the original conceptualization of the computer as a Turing machine—i.e. a machine which operates linearly, on a one-dimensional tape—and further technological developments. Originally, the Turing machine would involve the ability for the head of the machine to jump at different locations. This meant that the execution and reading of a program would be non-linear, jumping from one routine to another across the source code. Such an entanglement is particulary obvious in Ben Fry's Distellamap series of visualizations of source code (32 represents the execution of the source code for the arcade game Pac-Man), and led to Edsger W. Dijkstra's statement on the harmfulness of such jumps on the cognitive abilities of programmers.

Later on, the introduction of multi-core architecture for central processing units in the late 2000s has enabled the broad adoption of multi-threading and threaded programming. As a result, source code has trans-



Listing 32: Visualization of the execution of Pac-Man's source code

formed from a single non-linear execution to a multiple non-linear process, in which several of these non-linear executions are happening in parallel. Keep tracking of what is executing when on which resource is involved in problems such as *race conditions*, when understanding the scheduling of events (each event every e.g. 1/180000000th of a second on a 3.0 Ghz CPU machine) becomes crucial to ensuring the correct behaviour of the software.

Conversely, the locus of the execution of software creates contributes to those issues. Software can be (dis-)located across multiple files on a single machine, on multiple processes on a single machine, or on multiple processes on multiple machines (on a local-area or wide-area network) (Berry, 2011). This further fragments the interface to the object of understanding. For instance, the asynchronicity of requesting and processing information from distinct processes is a spatial separation of code which has temporal implications (e.g. due to network latency).

Software, then, exhibits certain properties which make it difficult to understand, from conception to application: software in the real-world involves implementing concepts that lose in translation, interfacing the world through discreete representations, and following the execution of these representations through space and time, as software executes more and more in a distributed and parallel fashion.

3.2.2 Software ontology

Now that we've clarified some difficulties in the understanding of software, we can frame these difficulties in a philosophical context. It turns out that not only is the nature of software a complex philosophical object, but so is that of computer science. Computer science itself inherits this

complexity from the nature of technology itself. Before moving back to practical inquiries into how specific individuals engage with this nature, this section will help provide a theoretical backdrop for further empirical inquiries. This conceptual framework will first by an investigation into the denomination of software as an *abstract artifact*, followed by an analysis of the ambiguous status of computer science between a science and a technology.

Software as abstract artifact

When he coins the phrase *abstract artifact*, Nurbay Irmak addresses software as an abstract object, similar to Platonic entities, but have nonetheless have spatio-temporal properties (Irmak, 2012). As mentioned previous, software does need to exist as an implementation, in the form of source code (Suber, 1988); this implies that software has a begin and an end, all the while representing ideas of structure and procedure which go beyond this beginning and end. Typically, the physical aspects of software (its manifestation as source code) can be changed[13] without changing any of the ideas expressed by the software—a phenomenon one can also observe in natural languages, in which one can radically change a syntax without drastically changing the semantics of a sentence.

Colburn's phrasing is that of a *concrete abstraction*. He does not explicit this dichotomy further, but we can see how its oxymoronic nature resonates with the concept of an abstract artifact.

He grounds this tension in the distinction between a medium of execution (a—potentially virtual—machine) and a medium of description (source code). He considers that while, ultimately, any high-level language in which the programmer writes is already the result of layers of abstraction, such high-level language gets reduced to the zeroes and ones input into the central processing unit (Colburn, 2000). And yet, if we carry on

[13] A process called *refactoring*.

this reasoning further, these representation of voltage changes into zeroes and ones are themselves abstractions over a concrete, physical event.

Writing on computational artefacts, of which software is a subset, Raymond Turner formalizes this apparent contradiction in a three-way relationship. Namely, artefact A is an implementation in medium M of the definition F. For instance, concerning the medium:

> *Instead of properties such as* made from carbon fiber*, we have properties such as* constructed from arrays in the Pascal programming language, implemented in Java. *(Turner, 2018)*

This metaphor provides an accurate account of the place of source code within the definition of software: source code is that which gives the ideas in software shape—through a process of concretization—and which hides away the details of the hardware—through abstraction.

The concept of *abstract artifact* therefore helps to clarify the tensions within software, and to locate the specific role of source code within the different moving parts of definition, medium and model.

Software, as a part of the family of technical artefacts, has *functional* properties (i.e. purpose that are intended to be achieved through their use) and *structural* ones (physical configuration which are involved in the fullfilment of the functional purpose) (Turner, 2018). As it turns out, these tensions exist throughout computer science and technology as a whole.

Philosophical investigations into software

The fact that software exists between the state of being conceptually designed and materially implemented predates software itself, and can be observed in modern technological developments at large.

As we've seen, software exists at the intersection of various tensions, domains and perspectives. According to Gilbert Simondon, the technical object is a relation between multiple structures and a complex operation

of various knowledges (Simondon, 1958). The technical object is indeed a scientific object, but also a social object and an artistic object at the same time.

From there, one could complement Bernard Stiegler's statement that technology is a dynamic of organized, but inorganic matter (Stiegler, 1998), if we are to consider inorganized matter to include social influences, personal practices, and forms of tacit knowledge. That is, the ambiguity of the technical object is that it extends beyond itself as an object, entering into a relation with its surrounding environment, including the human individual which makes use of it.

A particularly interesting status is that which it holds between magic, religion and philosophy. Tracing back the genesis of the technological object, Simondon writes that the technical mode of existence is another mode through which the human can relate to the world, similar to the aesthetic mode, or the religious mode. Technology, along with religion, is thus one of the two primary ways humans established a relationship to the world s(Simondon, 1958). While both are based on theoretical knowledge, technology is manifested in a practice, and in an ethics in religion. Technology results in a theory of knowledge and action, echoing Brian Cantell-Smith qualification of computers as *meaning mechanically realized*, as opposed to a theory of being, without being autonomous from one another.

Another stance, beyond the technical and the religious, is the aesthetic stance. It counter-balances the apparent split between technics and religion by striving for unity and totality, for the balance between the objective and the subjective. Yet, rather than being a monadic unity of a single principle, Simondon considers as unifying a network of relationships, a form of organizing Stiegler's inorganized matter. An aesthetic object therefore acquires the property of being beautiful by virtue of its relationships, of its connections between the subject and the obective, one's history and one's perceptions, between the various elements of the world, and the actions of the individual. Finally, the aesthetic thought when related to the tech-

nical object consists in preparing the communication between different communities of users, between different perspectives on the world, and different modes of action upon this world.

Specifically regarding our object of study, David M. Berry manifests this intersection of relations as such:

> *The computational device is, in some senses, a container of a universe (as a digital space) which is itself a container for the basic primordial structures which allow further complexification and abstraction towards a notion of world presented to the user. (Berry, 2011)*

Software is therefore a manifestation of technology as both knowledge and action. Furthermore, it also enables ways to act mechanically on knowledge and ideas, an affordance named *epistemic action* by David Kirsh and Paul Maglio (Kirsh & Maglio, 1994). They define epistemic actions as actions which facilitate thinking through a particular situation or environment, rather than having an immediate functional effect on the state of the world. As technology changes the individual's relationship to the world, software does so by being the dynamic notion of a state of a process (Rapaport, 2005), and by changing the conceptual understanding of said world—e.g. the social environment in which the software exists, or the environment which has been recreated within software.

Because software is the material implementation of a proposed model, itself derived from a theory, it primarily acts at the level of *episteme*, sometimes even limiting itself to it[14]. It is only through peripherals that software can act as a mechanical technology in the industrial sense of the word.

Due to software's ambiguous material nature (i.e. in contemporary digital computers, it consists of electrons, copper and silicium), one variable

[14]Functional programming languages take pride in the fact that they have no effect on the world around them, being composed exclusively of so-called *pure functions*, and no external side-effects, or input/output considerations

remains—that of the intent of the humans programming such software. Since software can be so many things, it might not be exactly clear what a specific piece of software, manifested in source code, is exactly about.

Thinking through the function of computational artefacts, Turner states that it is agency which determines what the function is: agency resolves the difference between the specification (intent-free, external to the program) and semantic interpretation (intent-rich, internal to the programmer) (Turner, 2018). In order to understand a computer program, we need to give it meaning and make sense of it, such that the question "what does a Turing machine do?" has n+1 answers. 1 syntactic, and n semantic (e.g. however many interpretations as there can be interpreters) (Rapaport, 2005). In his investigation into what software is, Suber provides a hint in stating that:

> *This suggests that, to understand software, we must understand intentions, purposes, goals, or will, which enlarges the problem far more than we originally anticipated. [...] We should not be surprised if human compositions that are meant to make machines do useful work should require us to posit and understand human purposiveness. After all, to distinguish* literature *from noise requires a similar undertaking. (Suber, 1988)*

In conclusion, we've seen that while software can be given the unique status of an *abstract artifact*, these tensions are shared across technological objects. Technology, as a combination of a theory of knowledge and a theory of action, as an interface to the world and a recreation of the world, is related to other modes of existence—and in particular the aesthetic mode. All of these tensions and paradoxes, from the various levels of existence, the various types of complexities, the different kinds of ac-

tions and interpretations that it allows, contribute to the cognitive hurdles encountered when attemption understanding software.

And yet, programmers, have been understanding software (some of them quite well, so called *rockstar* programmers, or virtuosos). We now turn to understanding how programmers understand software, through a psychological perspective.

3.2.3 The psychology of programming

How programmers deal with such a complex object as software has been a research topic which appeared much later on in the history of software development. How do they build up their understanding, in order to afford appropriate modification, re-use or maintenance of the software? What cognitive abilities do they summon, and what kind of technical apparatuses play a role in this process?

In their work on computer-enabled cognitive skills, Kirsh and Maglio develop on the use of epistemic actions:

> *More precisely, we use the term epistemic action to designate a physical action whose primary function is to improve cognition by:*
>
> 1. *reducing the memory involved in mental computation, that is, space complexity;*
> 2. *reducing the number of steps involved in mental computation, that is, time complexity;*
> 3. *reducing the probability of error of mental computation, that is, unreliability.*
>
> *(Kirsh & Maglio, 1994)*

Epistemic actions thus contribute to reducing the kinds of complexities involved with software. Concretely, this involves the creation of mental models of the software system with which the indvidual is interacting.

Psychological studies of programming practices have been focused on the identification and analysis of these mental models, as well as on their optimization through experimental devices, and the social behaviour of programmers within a context of practice (Weinberg, 1998).

Mental models, as kinds of internal symbolic representation of an external reality, is a rigorous and personal conceptual structure. They are related to knowledge, since the construction of accurate and useful mental models through the process of understanding underpins knowledge acquisition. However, mental models need not be correlated with empirical truth, due to their personal nature, but are extensive enough to be described by formal (logical or diagrammatical) means. Mental models can be informed, constructed or further qualified by the use of metaphors, but they are nonetheless more precise than other cognitive structures such as metaphors—a mental model can be seen as a more specific instance of a conceptual structure.

The earliest studies of how computer programmers understand the code they are presented with consisted mostly in pointing out the methodological difficulties in doing so (Sheil, 1981; Shneiderman, 1977). This is mainly due to three parameters. First, programming is an intertwined combination of notation, practices, tasks and management, each of which have their own impact on the extent to which a piece of source code is correctly understood. Second, program comprehension is strongly influenced by practice—the skill level of the programmer therefore also influences experimental conditions. Third, these early studies have found that programmers have organized knowledge bases, if informal. This means that, while programmers demonstrate epistemic mastery, they are limited in their ability to explain the workings of such ability—that is, the constitution and use of their own epistemic models, which she calls knowledge maps.

In their 1992 study, Marian Petre and Alan Blackwell attempt to identify these mental models and their uses. They asked 10 expert programmers

from North America and Europe to describe the thought process in source code-related problem-solving and design solutions in code. The main conclusion of their study is that, beyond the fact that each programmer had slightly different descriptions of their mental process, there a some commonalities to what is happening in someone's thoughts as they start to design software. The behaviour is dynamic, but controlled; the resolution of that behaviour was also dynamic, with some aspects coming in and out of focus that the will of the programmer, providing more or less uncertainty, level of details and fuzziness on-demand; and those images co-existed with other images, such that one representation could be compared with another representation of a different nature (Petre & Blackwell, 1997). Finally, while most imagery was non-verbal, all programmers talked about the need to have elements of this imagery labelled at all times, hinting at a relationship between syntax and semantics to be translated into source code. While this study was an investigation into the design of code, before any writing happens, one of the limitations is that it did not investigate the understanding of code, which takes places once the writing has been done (by oneself, or someone else), and the code now needs to be read.

Francoise Détienne, in her study of how computer programmers design and understand programs (Detienne, 2012), defines the activity of designing and understanding programs in activating *schemas*, mental representations that are abstract enough to encompass a wide use (web servers all share a common schema in terms of dealing with requests and responses), but nonetheless specific enough to be useful (requests and responses are qualitatively different subsets of the broader concept of inputs and outputs). An added complexity to the task of programming comes with the one of the dual nature of the mental models needing to be activated: the computer's actions and responses are comprised of the prescriptive (what the computer should do) to the effective (what the computer actually does). In order to be appropriately dealt with, then, programmers must activate and refine mental models of a program which resolves this tension.

In programming, within a given context—which include goals and heuristics—, elements are being perceived, processed through existing knowledge schemas in order to extract meaning. Starting from Kintsch and Van Dijk's approach of understanding text (Kintsch & van Dijk, 1978), she nonetheless highlights some differences. In program texts, there is an entanglement of the plan, of the arc, of the tension, which does not happen so often in most of the traditional narrative text. A programmer can jump between lines and files in a non-linear, explorative manner, following the features of computation, rather than textuality. Program texts are also dynamic, procedural texts, which exhibit complex causal relations between states and events, which need to be kept track of in order to resolve the prescriptive/effective discrepancies. Finally, the understanding of program text is first a general one, which only subsequently applies to a particular situation (a fix or an extension needing to be written), while narrative texts tend to focus on specific instances of protagonists, scenes and descriptions, leading to broad thematic appreciation.

Conversely, a similarity in understanding program texts and narrative texts is that the sources of information for understanding either are: the text itself, the individual experience and the broader environment in which the text is located (e.g. technical, social). Building on Chomsky's concepts, the activity of understanding in programming can be seen as understanding the *deep structure* of a text through its *surface structure* (Chomsky, 1965). One of the heuristics deployed to achieve such a goal is looking out for what she calls *beacons*, as thematic organizers which structure the reading and understanding process. For instance, in traditional narrative texts, beacons might be represented by section headings, or the beginning or end of paragraphs. However, one of the questions that her study hasn't answered specificall, how the specific surface structure in programming results in the understanding of the deep structure—in other terms, what is the connection between source code syntax, programmer semantics and program behavior.

Due to its relation to text, syntax and semantics, it has often been assumed by programmers and researchers that reading and writing code is akin to reading and writing natural languag prose. Additional recent research in the cognitive responses to programming tasks, conducted by Ivanova et. al., do not appear to settle the question of whether programming is rather dependent on language processing brain functions, or on functions related to mathematics (which do not rely on the language part of the brain) (Ivanova et al., 2020), but contributes empirical evidence to that debate. They conclude that, while language processing might not be one of the essential ways that we process code—excluding the *code is language* hypothesis—, it also does not rely on exclusively mathematical functions. Stimulating in particular the so-called multi-demand system, it seems that programming is a polymorphous activity involving multiple exchanges between different brain functions. What this implies, though, is that neither literature, linguistics nor mathematics should be the only lens through which we look at code.

In a way, then, programming is a sort of fiction, in that the pinpointing of its source of existence is difficult, and in that it affords the experience of imagining contents of which one is not the source, and of which the certainty of isn't defined. Both programming and fiction suggest surface-level guiding points helping the process of constructing mental models as a sort of conceptual representation. It is also something else than fiction, in that it deals with concrete issues and rational problems [15], and that it provides a pragmatic frame for processing representations, in which assumptions stemming from burgeoning mental models can be easily verified or falsified, through the taking of epistemic actions. It might then be appropriate to treat it as such, simultaneously fiction and non-fiction, as knowledge and action, mathetmatic and artistic. Indeed, it is also an artistic activity which, in Goodman's terms, might be seen as *an analysis of [artistic] behavior as a sequence of problem-solving and planning activities.*" (Goodman &

[15] more often than not, a pestering bug

Others, 1972).

Remains the interpretation issue mentioned above: the interpretation of the machine is different from the interpretation of the human, of which there are many, and therefore what also needs to be intepreted is the intent of the author(s). Reading is then akin to constructing a *cognitive cartography*, allowing for an experience to be made intelligible, sensible, and verifiable, and when an experience is made sensible is when it enters the realm of the aesthetic. In a very immediate understanding of aesthetics, we conclude on the role of form in cognition in those psychological studies. One of the focus was on demonstrating the impact that formal arrangement has on program comprehension(Oman & Cook, 1990; Oliveira et al., 2022). Spacing, alignment, color-coding and casing are all parameters which have an impact on the readability, and therefore understandability of code. The next section therefore looks at two ways in which meaning can be extracted from intention and source code; this means how individuals use metaphors to communicate complex ideas, and how they use computational tools in order to navigate program texts.

3.3 Means of understanding

As we've seen in the previous sequence, there are specific problems to understanding computation, but there are also ways of achieving a good understanding of a program, specifically metaphors and tools.

We'll spend most of this section looking at the ubiquity of metaphors in computing, as a cognitive mechanism, relying on the work of Lakoff and Johnson, both from users and programmers. We will also investigate the concrete use of extensions of mind through software tools, particularly on the role of *Integrated Development Environments* (IDEs).

3.3.1 Metaphors of computation

Here we include the section in the 4th report about metaphors in computation.

The implication of spatial and visual components in mental models allows us to turn to metaphors as an architecture of thought (Forsythe, 1986).

But we will also need more practical examples.

computers can't be proved/assumed to be machines, because their terminology comes from logic/textual/discursive traditions (e.g. reference, statement, names, recursion, etc.) *yet they are still built*. (cantell smith)

- sally wyatt
- chun, on the persistence of visual knowledge
- a first pass on interfaces as metaphors (nielsen, galloway)
- cramer and fuller in *Software studies*
- lakoff (ricoeur will come when we talk about PLs)
- example about the clipboard

3.3.2 Tools as an extension

Essentially talking about HCI

tracing and chunking: https://www.sciencedirect.com/science/article/abs/pii/095058499591491

- extended mind hypothesis of clark and chalmers (1998)
- fishwick aesthetic progamming
- allamanis, using ML for code generation and analysis, and mattt (as we may code) highlights the need for such a thing (quoting: What if, instead of lowering source code down for the purpose of execution, we raised source code for the purpose of understanding?)

- barker writing software documentation

- wilken card index

> *Code is therefore technical and social, and material and symbolic simultaneously. Rather, code needs to be approached in its multiplicity, that is, as a literature, a mechanism, a spatial form (organization), and as a repository of social norms, values, patterns and processes. (Berry, 2011)*

Conclusion: we circle back to aesthetics, particularly relying on (Dexter et al., 2011) and we look at the things that are both tools and metaphors: programming languages. We will see what roles metaphors play; and, if linguistics is a key component in the writing of clear source code, we should also look at programming languages.

Bibliography

(????). Stack Overflow Developer Survey 2021.

(2009). EARLY EXPLORATIONS: 1950S AND 1960S. In N. J. Nilsson (Ed.) *The Quest for Artificial Intelligence*, (pp. 47–48). Cambridge: Cambridge University Press.

(2013). How can you explain "beautiful code" to a non-programmer?

Abelson, H., Sussman, G. J., & Sussman, J. (1979). *Structure and Interpretation of Computer Programs - 2nd Edition*. Justin Kelly.

Akesson, L. (2017). A Mind Is Born.

Anthes, G. (2011). Beauty and elegance. *Communications of the ACM*, *54*(6), 14–15.

Aquilina, M. (2015). The Computational Sublime in Nick Montfort's 'Round' and 'All the Names of God'. *CounterText*, *1*, 348–365.

Arendt, H. (1998). *The Human Condition*. Chicago: University of Chicago Press, 2nd ed. / introduction by margaret canovan. ed.

Arnaud, N. (1968). *Poèmes ALGOL*. Temps mélées.

Arns, I. (2005). Code as performative speech act. *Artnodes*, *0*(4).

Barthes, R. (1984). *Le bruissement de la langue: essais critiques IV*. Paris: Seuil.

Bassett, A. C. D. o. D. A. a. S. C. J., Bassett, J. L., Fogelman, P., Scott, D. A., Calif.), G. C. I. L. A., & Schmidtling, R. C. (2008). *The Craftsman Revealed: Adriaen de Vries*. Getty Publications.

Beardsley, M. C. (1970). The Aesthetic Point of View*. *Metaphilosophy*, *1*(1), 39–58.

Berry, D. M. (2011). *The Philosophy of Software: Code and Mediation in the Digital Age*. Palgrave-Macmillan.

Bertram, I. (2012). *Code {poems}*. Barcelona: Impremta Badia.

Black, A. (1984). *Guilds and Civil Society in European Political Thought from the Twelfth Century to the Present*. Methuen.

Black, M. J. (2002). The art of code. *Dissertations available from ProQuest*, (pp. 1–228).

Bogost, I. (2007). *The Rhetoric of Video Games*.

Bourque, P., & Fairley, R. E. (Eds.) (2014). *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Los Alamitos, CA: IEEE Computer Society, version 3.0 ed.

Brock, K. (2019). *Rhetorical Code Studies: Discovering Arguments in and around Code*. Open Research Library.

Brooks, D. (2019). Finally, a historical marker that talks about something important - Granite Geek. https://web.archive.org/web/20190611180750/https://granitegeek.concordmonitor.com/2019/06/1 a-historical-marker-that-talks-about-something-important/.

Brooks, F. P., & Jr, F. P. B. (1975). *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Publishing Company.

Bush, D. (2015). 15 Ways to Write Beautiful Code - DZone DevOps. https://dzone.com/articles/15-ways-to-write-beautiful-code.

Byrd, W. (2017). William Byrd on The Most Beautiful Program Ever Written.

Campbell-Kelly, M. (2003). *From Airline Reservations to Sonic the Hedgehog : A History of the Software Industry*. Cambridge, Mass. : MIT Press.

Cayley, J. (2012). The Code is not the Text (Unless It Is the Text) › electronic book review.

Cennini, C. (2012). *The Craftsman's Handbook*. Courier Corporation.

Ceruzzi, P. E. (2003). *A History of Modern Computing*. History of Computing. Cambridge, MA, USA: MIT Press, second ed.

Chandra, V. (2014). *Geek Sublime: The Beauty of Code, the Code of Beauty*. Graywolf Press.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. Cambridge, M.I.T. Press.

Chun, W. H. K. (2005). On Software, or the Persistence of Visual Knowledge. *Grey Room*, *18*, 26–51.

Chun, W. H. K. (2008). On "Sourcery," or Code as Fetish. *Configurations*, *16*(3), 299–324.

Cohendet, P., Créplet, F., & Dupouët, O. (2001). Organisational Innovation, Communities of Practice and Epistemic Communities: The Case of Linux.

Colburn, T. R. (2000). *Philosophy and Computer Science*. M.E. Sharpe.

Coleman, R. (2018). Aesthetics Versus Readability of Source Code. *International Journal of Advanced Computer Science and Applications*, *9*(9).

Collins, H. (2010). *Tacit and Explicit Knowledge*. University of Chicago Press.

Committee, G. B. P. H. o. C. S. a. T. (2010). *The Disclosure of Climate Data from the Climatic Research Unit at the University of East Anglia: Eighth Report of Session 2009-10, Vol. 2: Oral and Written Evidence*. The Stationery Office.

Cox, G., & McLean, C. A. (2013). *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press.

Cramer, F. (2003). *Words Made Flesh*. Piet Zwart Institute.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*(2), 237–267.

de Certeau, M., Giard, L., & Mayol, P. (1990). *L'invention du quotidien*. Gallimard.

de Saint-Exupéry, A. (1972). *Terre des Hommes*. Gallimard.

Defense Technical Information Center (1970). *DTIC AD0715513: An Empirical Study of Fortran Programs*.

Detienne, F. (2012). *Software Design – Cognitive Aspect*. Springer Science & Business Media.

Dexter, S., Dolese, M., Seidel, A., & Kozbelt, A. (2011). On the Embodied Aesthetics of Code. *Culture Machine*, *12*(1).

Dijk, T. A., & Kintsch, W. (1983). Strategies of discourse comprehension.

Dijkstra, E. (1963). On the design of machine independent programming languages. *Annual Review in Automatic Programming*, *3*, 27–42.

Dijkstra, E. W. (1972). Chapter I: Notes on structured programming. In *Structured Programming*, (pp. 1–82). Academic Press Ltd.

Dijkstra, E. W. (1982). "Craftsman or Scientist?". In E. W. Dijkstra (Ed.) *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, (pp. 104–109). New York, NY: Springer.

Dijkstra, E. W. (2007). The humble programmer. In *ACM Turing Award Lectures*, (p. 1972). New York, NY, USA: Association for Computing Machinery.

DiLascia, P. (2019). { End Bracket }: What Makes Good Code Good? https://docs.microsoft.com/en-us/archive/msdn-magazine/2004/july/%7b-end-bracket-%7d-what-makes-good-code-good.

Dourish, P. (1988). The Original Hacker's Dictionary.

Duff, T. (1983). Tom Duff on Duff's Device. http://www.lysator.liu.se/c/duffs-device.html.

Eadicicco, L. (2014). Startup God Paul Graham Reveals The Single Most Important Quality To Look For In A Company. https://finance.yahoo.com/news/startup-god-paul-graham-reveals-165445488.html?guccounter=1.

Efatmaneshnik, M., & Ryan, M. J. (2019). On the Definitions of Sufficiency and Elegance in Systems Design. *IEEE Systems Journal*, *13*(3), 2077–2088.

Ensmenger, N. L. (2012). *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, Mass.: The MIT Press.

Fakhoury, S., Roy, D., Hassan, S. A., & Arnaoudova, V. (2019). Improving source code readability: Theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, (pp. 2–12). Montreal, Quebec, Canada: IEEE Press.

Fishwick, P. (2001). Aesthetic Programming.

FLUSSER, V., & Novaes, R. M. (2014). *On Doubt*. University of Minnesota Press.

Foote, B., & Yoder, J. (1997). Big Ball of Mud. http://www.laputan.org/mud/mud.html#BigBallOfMud.

Forsythe, K. (1986). Cathedrals in the Mind: The Architecture of Metaphor in Understanding Learning. In R. Trappl (Ed.) *Cybernetics and Systems*

'86: Proceedings of the Eighth European Meeting on Cybernetics and Systems Research, Organized by the Austrian Society for Cybernetic Studies, Held at the University of Vienna, Austria, 1–4 April 1986, (pp. 285–292). Dordrecht: Springer Netherlands.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., & Gamma, E. (1999). *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley Professional, 1st edition ed.

Frasca, G. (2013). *Simulation versus Narrative : Introduction to Ludology*. Routledge.

Fuller, M. (Ed.) (2008). *Software Studies: A Lexicon*. Cambridge, Mass: The MIT Press.

Gabriel, R. P. (1998). *Patterns of Software: Tales from the Software Community*. Oxford University Press.

Gabriel, R. P., & Goldman, R. (2001). *Mob Software: The Erotic Life of Code*.

Garland, D. (2000). Software Architecture: A Roadmap. In *The Future of Software Engineering*. ACM Press, anthony finkelstein (ed.) ed.

Gefen, A., & Perez, C. P. (2019). Extension du domaine de la littérature Extension du domaine de la littérature. *Elfe XX-XXI Études de la littérature française des XXe et XXIe siècles*.

Gelernter, D. H. (1998). *Machine Beauty : Elegance and the Heart of Technology*. New York : Basic Books.

Genette, G. (1993). *Fiction & Diction*. Cornell University Press.

Gibbons, J. (2012). The beauty of simplicity. *Communications of the ACM*, *55*(4), 6–7.

Goodliffe, P. (2007). *Code Craft: The Practice of Writing Excellent Code*. No Starch Press.

Goodman, N. (1976). *Languages of Art*. Indianapolis, Ind.: Hackett Publishing Company, Inc., 2nd edition ed.

Goodman, N. (1978). *Ways Of Worldmaking*.

Goodman, N., & Others, A. (1972). Basic Abilities Required for Understanding and Creation in the Arts. Final Report.

Goody, J. (1977). *The Domestication of the Savage Mind*. Cambridge University Press.

Gordon, R. B. (1988). Who Turned the Mechanical Ideal into Mechanical Reality? *Technology and Culture*, *29*(4), 744–778.

Goriunova, O., & Shulgin, A. (2005). *Read Me: Software Art & Cultures*. Aarhus: Aarhus University Press, 2004th edition ed.

Graham, P. (2003). Hackers and Painters. http://www.paulgraham.com/hp.html.

Green, R. (2006). How To Write Unmaintainable Code. https://archive.ph/Pn5hH.

Grudin, J. (2016). From Tool to Partner: The Evolution of Human-Computer Interaction. *Synthesis Lectures on Human-Centered Informatics*, *10*(1), i–183.

Guerrouj, L. (2013). Normalizing source code vocabulary to support program comprehension and software quality. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (pp. 1385–1388). San Francisco, CA, USA: IEEE Press.

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., & Wilson, G. (2009). How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, (pp. 1–8).

Hansen, M. B. N. (2006). *Bodies in Code: Interfaces with Digital Media*. New York: Routledge.

Harlow, F., & Fromm, J. (1965). Computer Experiments in Fluid Dynamics.

Harman, M. (2010). Why Source Code Analysis and Manipulation Will Always be Important. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, (pp. 7–19).

Hatton, L., & Roberts, A. (1994). How accurate is scientific software? *IEEE Transactions on Software Engineering*, *20*(10), 785–797.

Hayes, B. (2017). *Cultures of Code*.

Hayles, N. K. (2004). Print Is Flat, Code Is Deep: The Importance of Media-Specific Analysis. *Poetics Today*, *25*(1), 67–90.

Hayles, N. K. (2010). *My Mother Was a Computer: Digital Subjects and Literary Texts*. University of Chicago Press.

Hendrickson, M., & McBreen, P. (2002). *Software Craftsmanship: The New Imperative*. Addison-Wesley Professional.

Henningsen, E., & Larsen, H. (2020). The Joys of Wiki Work: Craftsmanship, Flow and Self-externalization in a Digital Environment. In R. Audunson, H. Andresen, & C. Fagerlid (Eds.) *Libraries, Archives and Museums as Democratic Spaces in a Digital Age*, (pp. 345–362). De Gruyter Saur.

Hill, R. K. (2016). What Makes a Program Elegant? https://cacm.acm.org/blogs/blog-cacm/208547-what-makes-a-program-elegant/fulltext.

Hoare, C. A. R. (1972). Chapter II: Notes on data structuring. In *Structured Programming*, (pp. 83–174). GBR: Academic Press Ltd.

Holden, M. D., & Kerr, M. C. (2016). *./Code –Poetry*. S.L: CreateSpace Independent Publishing Platform, 1.0 edition ed.

Hoover, D., & Oshineye, A. (2009). *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman*. "O'Reilly Media, Inc.".

Hopkins, S. (1992). Camels and Needles: Computer Poetry Meets the Perl Programming Language. *Usenix Technical Conference*.

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Reading, Mass: Addison-Wesley Professional, 1st edition ed.

Ifrah, G. (2001). *The Universal History of Computing : From the Abacus to the Quantum Computer*. New York : John Wiley.

Irmak, N. (2012). Software is an Abstract Artifact. *Grazer Philosophische Studien*, *86*(1), 55–72.

Ivanova, A. A., Srikant, S., Sueoka, Y., Kean, H. H., Dhamala, R., O'Reilly, U.-M., Bers, M. U., & Fedorenko, E. (2020). Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife*, *9*, e58906.

James, G. (1987). *The Tao of Programming*. InfoBooks.

Jeiss, J. (2002). The Poetry of Programming. https://www.dreamsongs.com/PoetryOfProgramming.html.

Jones, M. L. (2016). *Reckoning with Matter: Calculating Machines, Innovation, and Thinking about Thinking from Pascal to Babbage*. Chicago ; London: University of Chicago Press, 1st edition ed.

Jordan, B. G., & Weston, V. L. (2003). *Copying the Master and Stealing His Secrets: Talent and Training in Japanese Painting*. University of Hawaii Press.

Kanakarakis, I. (2022a). The International Obfuscated C Code Contest.

Kanakarakis, I. (2022b). The International Obfuscated C Code Contest.

Karvonen, K. (2000). The beauty of simplicity. In *Proceedings on the 2000 Conference on Universal Usability*, CUU '00, (pp. 85–90). New York, NY, USA: Association for Computing Machinery.

Kay, A. (2004). A Conversation with Alan Kay - ACM Queue. *ACM Queue*, *2*(9).

Keller, A. (2021). Des textes d'algorithmes concis, quelques exemples tirés de sūtras d'Āryabhaṭa et de son commentaire par Bhāskara.

Kelly, D. F. (2007). A Software Chasm: Software Engineering and Scientific Computing. *IEEE Software*, *24*(6), 120–119.

Kernighan, B. W., & Plauger, P. J. (1978). *The Elements of Programming Style, 2nd Edition*. New York: McGraw-Hill, 2nd edition ed.

Kintsch, W., & van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, *85*(5), 363–394.

Kirsh, D., & Maglio, P. (1994). On Distinguishing Epistemic From Pragmatic Action. *Cognitive Science*, *18*(4), 513–49.

Kitchin, R., & Dodge, M. (2011). *Code/Space: Software and Everyday Life*. The MIT Press.

Kittler, F. A. (1997). There Is No Software. In *Literature, Media, Information Systems: Essays*, (pp. 147–155). Amsterdam: Amsterdam Overseas Publishers Association, john johnston ed.

Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM*, *17*(12), 667–673.

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*(2), 97–111.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc.

Korte, T. (2010). Frege's Begriffsschrift as a lingua characteristica. *Synthese*, *174*(2), 283–294.

Kurp, P. (2008). Green computing. *Communications of the ACM*, *51*(10), 11–13.

Lakoff, G. (1980). *Metaphors We Live By*. University of Chicago Press.

Lammers, S. M. (1986). *Programmers at Work : Interviews*. Redmond, Wash. : Microsoft Press ; [New York] : Distributed in the U.S. by Harper and Row.

Landau, R. H., Páez, J., & Bordeianu, C. C. (2011). *A Survey of Computational Physics: Introductory Computational Science*. Princeton University Press.

Lando, P., Lapujade, A., Kassel, G., & Fürst, F. (2007). Towards a General Ontology of Computer Programs. (pp. 163–170).

Laurel, B. (1993). *Computers as Theatre*. Addison-Wesley.

Lettvin, J. Y., Maturana, H. R., McCulloch, W. S., & Pitts, W. H. (1959). What the Frog's Eye Tells the Frog's Brain. *Proceedings of the IRE*, *47*(11), 1940–1951.

LeVeque, R. J., Mitchell, I. M., & Stodden, V. (2012). Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science Engineering*, *14*(4), 13–17.

Lévy, P. (1992). *De la programmation considérée comme un des beaux-arts*. Textes à l'appui. Anthropologie des sciences et des techniques. Paris: Éd. la Découverte.

Levy, S. (2010). *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*. "O'Reilly Media, Inc.".

Li, J. (2020). Where Did Software Go Wrong? https://blog.jse.li/posts/software/.

Licklider, J. C. R. (1960). Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics*, *HFE-1*(1), 4–11.

Light, J. S. (1999). When Computers Were Women. *Technology and Culture*, *40*(3), 455–483.

Lions, J. (1996). *Lions' Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications.

Mackenzie, A. (2006). *Cutting Code: Software and Sociality*. Peter Lang.

Man, E. K. W. (2015). Influence of Global Aesthetics on Chinese Aesthetics: The Adaptation of Moxie and the Case of Dafen Cun. In E. K. W. Man (Ed.) *Issues of Contemporary Art and Aesthetics in Chinese Context*, Chinese Contemporary Art Series, (pp. 95–103). Berlin, Heidelberg: Springer.

Manovich, L. (2001). *The Language of New Media*. Cambridge, MA: MIT Press.

Marchand-Zañartu, N., & Lauxerois, J. (2022). *32 grammes de pensée, essai sur l'imagination graphique*. Médiapop Éditions.

Marcus, A., & Baecker, R. (1982). On The Graphic Design of Program Text.

Marino, M. C. (2020). *Critical Code Studies*. Software Studies. Cambridge, MA, USA: MIT Press.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

Mateas, M., & Montfort, N. (2005). A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics. *undefined*.

Mazzone, M., & Elgammal, A. (2019). Art, Creativity, and the Potential of Artificial Intelligence. *Arts*, *8*(1), 26.

McAllister, J. (2005). Mathematical Beauty and the Evolution of the Standards of Mathematical Proof. *undefined*.

McCarthy, J. (1978). History of LISP. *ACM SIGPLAN Notices*, *13*(8), 217–223.

McCarthy, J., Levin, M. I., Abrahams, P. W., Center, M. I. o. T. C., & Edwards, D. J. (1965). *LISP 1.5 Programmer's Manual*. MIT Press.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Redmond, Wash: Microsoft Press, 2nd edition ed.

McCulloch, W. (1953). The Past of a Delusion. *Chicago Literary Club*.

McCulloch, W. S., & Pitts, W. (1990). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, *52*(1), 99–115.

McGee, D. (1999). From Craftsmanship to Draftsmanship: Naval Architecture and the Three Traditions of Early Modern Design. *Technology and Culture*, *40*(2), 209–236.

McLean, A. (2004). Hacking Perl in Nightclubs. https://perl.com/pub/2004/08/31/livecode.html/.

McLennan, B. J. (1997). "Who Care About Elegance?": The Role of Aesthetics in Programming Language Design. Technical Report UT-CS-97-344, University of Tennessee.

Melik, D. (2012). PC demos FAQ.

Mentor+++, T. (1986). \textbackslash/\textbackslashThe Conscience of a Hacker/\textbackslash/.

Merali, Z. (2010). Computational science: ...Error. *Nature*, *467*(7317), 775–777.

Millman, K., & Aivazis, M. (2011). Python for Scientists and Engineers. *Computing in Science & Engineering*, *13*(02), 9–12.

Mills, C. W. (2000). *The Sociological Imagination*. Oxford; New York: Oxford University Press.

Mindell, D. A. (2011). *Digital Apollo: Human and Machine in Spaceflight*. MIT Press.

Mitchell, W. J. W. J. (1987). *The Art of Computer Graphics Programming : A Structured Introduction for Architects and Designers*. New York : Van Nostrand Reinhold.

Moler, C., & Little, J. (2020). A history of MATLAB. *Proceedings of the ACM on Programming Languages*, *4*(HOPL), 81:1–81:67.

Molzberger, P. (1983). Aesthetics and programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, (pp. 247–250). New York, NY, USA: Association for Computing Machinery.

Montfort, N., Baudoin, P., Bell, J., Bogost, I., & Douglass, J. (2014). *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. The MIT Press, illustrated edition ed.

Moor, J. H. (1978). Three Myths of Computer Science. *British Journal for the Philosophy of Science*, *29*(3), 213–222.

Morgan, C. (1982). Of IBM, Operating Systems and Rosetta Stones. *Byte Magazine Volume 07 Number 01 - The IBM Personal Computer*.

Moss, R. (2022). BeautifulAlgorithms.jl.

Mullet, D. R. (2018). A General Critical Discourse Analysis Framework for Educational Research. *Journal of Advanced Academics*, *29*(2), 116–142.

Murray, J. H. (1998). *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. Cambridge, MA, USA: MIT Press.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, *15*(5), 253–261.

Newell, A., Tonge, F. M., Feigenbaum, E. A., Green Jr., B. F., & Mealy, G. H. (1964). *Information Processing Language-V Manual*. Prentice-Hall, the rand corporation ed.

Nielsen, M. (2012). Lisp as the Maxwell's equations of software \textbar DDI.

Nielsen, M. (2017). Working Notes on Chalktalk. https://cognitivemedium.com/interfaces-1/index.html.

Nilsson, N. J. (2009). *The Quest for Artificial Intelligence*. Cambridge: Cambridge University Press.

Nolte, A., Pe-Than, E. P. P., Filippova, A., Bird, C., Scallen, S., & Herbsleb, J. D. (2018). You Hacked and Now What? - Exploring Outcomes of a Corporate Hackathon. *Proceedings of the ACM on Human-Computer Interaction*, *2*(CSCW), 129:1–129:23.

Norick, B., Krohn, J., Howard, E., Welna, B., & Izurieta, C. (2010). Effects of the number of developers on code quality in open source software: A case study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, (p. 1). New York, NY, USA: Association for Computing Machinery.

Oberkampf, W. L., & Roy, C. J. (2010). *Verification and Validation in Scientific Computing*. Cambridge University Press.

Oliveira, D., Bruno, R., Madeiral, F., Masuhara, H., & Filho, F. C. (2022). A Systematic Literature Review on the Impact of Formatting Elements on Program Understandability. *undefined*.

Oman, P. W., & Cook, C. R. (1990). Typographic style is more than cosmetic. *Communications of the ACM*, *33*(5), 506–520.

Oman, R. (2018). Computer Software as Copyrightable Subject Matter: Oracle V. Google, Legislative Intent, and the Scope of Rights in Digital Works. *Harvard Journal of Law and Technology*, *31*(Special Issue Spring 2018), 639–652.

O'Neil, S. T. (2019). *A Primer for Computational Biology*. Oregon State University.

Ong, W. J. (2012). *Orality and Literacy: 30th Anniversary Edition*. London: Routledge, third ed.

Oram, A., & Wilson, G. (Eds.) (2007). *Beautiful Code: Leading Programmers Explain How They Think*. Beijing ; Sebastapol, Calif: O'Reilly Media, 1st edition ed.

Osborne, H. (1977). The Aesthetic Concept of Craftsmanship. *British Journal of Aesthetics*, *17*(2), 138.

Ousterhout, J. K. (1998). Scripting: Higher-Level Programming for the 21st Century. *Computer*, *31*(3), 23–30.

Paloque-Bergès, C. (2009). *Poétique des codes sur le réseau informatique*. Archives contemporaines.

Pannabecker, J. R. (1994). Diderot, the Mechanical Arts, and the Encyclopdie: In Search of the Heritage of Technology Education. *Journal of Technology Education*, *6*, 45–57.

Pattis, R. E. (1988). Textbook errors in binary searching. In *SIGCSE '88*.

Pellet-Mary, A. (2020). C06GC: Program Obfuscation | COSIC.

Penny, S. (2019). *Making Sense: Cognition, Computing, Art and Embodiment*. MIT Press.

Perrin, C. (2006). ITLOG Import: Elegance. https://web.archive.org/web/20200730232944/http://sob.apotheon.org/?p=113.

Petre, M., & Blackwell, A. F. (1997). A glimpse of expert programmers' mental imagery. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ESP '97, (pp. 109–123). New York, NY, USA: Association for Computing Machinery.

Pevsner, N. (1942). The Term 'Architect' in the Middle Ages. *Speculum*, *17*(4), 549–562.

Pineiro, E. (2003). *The Aesthetics of Code : On Excellence in Instrumental Action*. Ph.D. thesis, KTH, Superseded Departments, Industrial Economics and Management.

Poibeau, T. (2017). *Machine Translation*. MIT Press.

Poincaré, H. (1908). *Science et méthode*. Paris: E. Flammarion.

Polanyi, M., & Grene, M. (1969). *Knowing and Being; Essays*. [Chicago] University of Chicago Press.

Postman, N. (1985). *Amusing Ourselves to Death: Public Discourse in the Age of Show Business*. New York: Viking Penguin, 1st edition ed.

Prabhu, P., Kim, H., Oh, T., Jablin, T. B., Johnson, N. P., Zoufaly, M., Raman, A., Liu, F., Walker, D., Zhang, Y., Ghosh, S., August, D. I., Huang, J., & Beard, S. (2011). A survey of the practice of computational science. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, (pp. 1–12).

Project, L. I. (2004). Source code definition by The Linux Information Project.

Pye, D. (2008). *The Nature and Art of Workmanship*. Herbert Press, illustrated edition ed.

Ranciere, J. (2013). *Aisthesis: Scenes from the Aesthetic Regime of Art*. London ; New York: Verso, 1st edition ed.

Randell, B. (1996). NATO Software Engineering Conference 1968. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/.

Rapaport, W. J. (2005). Philosophy of Computer Science: An Introductory Course. *Teaching Philosophy*, *28*(4), 319–341.

Rasool, G., & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, *27*(11), 867–895.

Raymond, E. (2003). *The Art of UNIX Programming*. Boston: Addison-Wesley, 1st edition ed.

Raymond, E. S. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. "O'Reilly Media, Inc.".

Raymond, E. S., & Steele, G. L. (1996). *The New Hacker's Dictionary*. MIT Press.

Reed, D. (2010). Sometimes style really does matter. *Journal of Computing Sciences in Colleges*, *25*(5), 180–187.

Reunanen, M. (2010). *Computer Demos - What Makes Them Tick?; Tietokonedemot - Mikä Saa Ne Hyrräämään?*. G3 Lisensiaatintyö, Aalto-yliopisto; Aalto University.

Richards, I. A. (1930). *Practical Criticism*. Kegan Paul Trench Trubner And Company Limited.

Rosenbaum, R. (2004). Secrets of the Little Blue Box.

Ross, D. (1986). A personal view of the personal work station: Some firsts in the Fifties. In *Proceedings of the ACM Conference on The History of Personal Workstations*, HPW '86, (pp. 19–48). New York, NY, USA: Association for Computing Machinery.

Rota, G.-C. (1997). The Phenomenology of Mathematical Beauty. *Synthese*, *111*(2), 171–182.

Ruskin, J. (1920). *The Seven Lamps of Architecture. With Illustrations Drawn by the Author*. London Waverley Book Co.

Russell, B. (1950). Logical positivism. *Revue Internationale de Philosophie*, *4*(11), 3–19.

Ryle, G. (1951). *The Concept Of Mind*. Hutchinsons University Library.

Sack, W. (2019). *The Software Arts*. The MIT Press.

Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., do Nascimento, R. S., Freitas, M. F., & de Mendonça, M. G. (2018). A systematic review on the code smell effect. *Journal of Systems and Software*, *144*, 450–477.

@Scale (2015). Why Google Stores Billions of Lines of Code in a Single Repository.

Schiffrin, D. (1994). *Approaches to Discourse*. Oxford, UK ; Cambridge, Mass., USA : B. Blackwell.

Schummer, J., MacLennan, B., & Taylor, N. (2009). Aesthetic Values in Technology and Engineering Design. In A. Meijers (Ed.) *Philosophy of Technology and Engineering Sciences*, Handbook of the Philosophy of Science, (pp. 1031–1068). Amsterdam: North-Holland.

Scopatz, A., & Huff, K. D. (2015). *Effective Computation in Physics*. O'Reilly Media.

Scruton, R. (Sun, 04/21/2013 - 12:00). *The Aesthetics of Architecture*. Princeton: Princeton University Press.

Segal, J. (2005). When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering*, *10*(4), 517–536.

Seibel, P. (2009). *Coders at Work: Reflections on the Craft of Programming*. Apress.

Sennett, R. (2009). *The Craftsman*. Yale University Press.

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, N.J: Pearson.

Sheil, B. A. (1981). The Psychological Study of Programming. *ACM Computing Surveys*, *13*(1), 101–120.

Shneiderman, B. (1977). Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, *9*(4), 465–478.

Simon, H. (1996). *The Sciences of the Artificial*. MIT Press.

Simondon, G. (1958). *Du mode d'existence des objets techniques*. Ph.D. thesis, Aubier et Montaigne, Paris.

Smith, B. C. (1998). *On the Origin of Objects*. Cambridge, Mass.: A Bradford Book, reprint edition ed.

Sommerville, I. (2010). *Software Engineering*. Boston: Pearson, 9th edition ed.

Sondheim, A. (2001). Introduction: Codework. *American Book Review*, *22*(6).

Spolosky, J. (2003). Craftsmanship. https://www.joelonsoftware.com/2003/12/01/craftsmanship-2/.

Stallman, R., & Free Software Foundation (Cambridge, M. . (2002). *Free Software, Free Society : Selected Essays of Richard M. Stallman*. Boston, MA : Free Software Foundation.

Steele, G. L. (1977). Macaroni is better than spaghetti. *ACM SIGPLAN Notices*, *12*(8), 60–66.

Stiegler, B. (1998). *Technics and Time, 1: The Fault of Epimetheus*. Stanford University Press.

Sturt, G. (1963). *The Wheelwright's Shop*. Cambridge ; New York: Cambridge University Press, revised ed. edition ed.

Suber, P. (1988). What is Software? *Journal of Speculative Philosophy*, *2*(2), 89–119.

Taylor, P. (2001). Patterns as Software Design Canon. *ACIS 2001 Proceedings*.

Team, O. E. (2021). *Low-Code and the Democratization of Programming*. O'Reilly Media.

Tedre, M. (2006). The development of computer science: A sociocultural perspective. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, (pp. 21–24). New York, NY, USA: Association for Computing Machinery.

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. CRC Press.

Temkin, D. (12/28/2017 12:00:00 PM). Sentences on Code Art. https://esoteric.codes/blog/sentences-on-code-art.

Thompson, D. V. (1934). The Study of Medieval Craftsmanship. *Bulletin of the Fogg Art Museum*, *3*, 3–8.

Thompson, D. V. (1956). *The Materials and Techniques of Medieval Painting*. Courier Corporation.

Treude, C., & Robillard, M. P. (2017). Understanding Stack Overflow Code Fragments. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (pp. 509–513).

Turing, A. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *42*(1), 230–265.

Turing, A. M. (2009). Computing Machinery and Intelligence. In R. Epstein, G. Roberts, & G. Beber (Eds.) *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*, (pp. 23–65). Dordrecht: Springer Netherlands.

Turner, R. (2018). Computational Artifacts. In R. Turner (Ed.) *Computational Artifacts: Towards a Philosophy of Computer Science*, (pp. 25–29). Berlin, Heidelberg: Springer.

Vardi, M. Y. (2010). Science has only two legs. *Communications of the ACM*, *53*(9), 5.

Voloshinov, V. N., & Bachtin, M. M. (1986). *Marxism and the Philosophy of Language*. Harvard University Press.

Waldron, R. (2020). Idiomatic.js/readme.md at master · rwaldron/idiomatic.js.

Warren, T. (2020). Windows XP source code leaks online.

Watters, A. (2021). *Teaching Machines: The History of Personalized Learning*. MIT Press.

Weaver, W. (1948). Science and Complexity. *American Scientist*, *36*(4), 536–544.

Wegenstein, B. (2010). Bodies. In *Critical Terms for Media Studies*. University of Chicago Press.

Weinberg, G. M. (1998). *The Psychology of Computer Programming*. Dorset House Pub.

Weizenbaum, J. (1976). *Computer Power and Human Reason: From Judgment to Calculation*. San Francisco: W H Freeman & Co, 1st edition ed.

Wikipedia (2021). Linux kernel.

Williams, L., & Kessler, R. R. (2003). *Pair Programming Illuminated*. Addison-Wesley Professional.

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., & Wilson, P. (2014). Best Practices for Scientific Computing. *PLOS Biology*, *12*(1), e1001745.

Wirth, N. (1976). *Algorithms + Data Structures*. Prentice-Hall.

Wirth, N. (2008). A Brief History of Software Engineering. *IEEE Annals of the History of Computing*.

Wittgenstein, L. (2004). *Recherches philosophiques*. Paris: Gallimard.

Wittgenstein, L. (2010). *Tractatus Logico-Philosophicus*.

Woolston, C. (2022). Why science needs more research software engineers. *Nature*.

Yuill, S. (2004). Code Art Brutalism: Low-level systems and simple programs. In *Read_me: Software Arts and Culture*. Aarhus: Digital Aesthetics Research Center.

Zeller, J. (2020). Algorithms are like recipes. https://www.goethe.de/en/kul/ges/21877729.html.

Zhang, G., Cheng, Z., & Wang, Q. (2015). Jingdezhen's Ceramic Civilization: The Past and Today. In *2015 International Conference on Humanities and Social Science Research*, (pp. 9–14). Atlantis Press.